Intermediate Quantitative Economics with Python

Thomas J. Sargent & John Stachurski

Feb 19, 2026

CONTENTS

This website presents a set of lectures on quantitative economic modeling.

Part I

Tools and Techniques

CHAPTER

ONE

MODELING COVID 19

Contents

1.1 Overview

This is a Python version of the code for analyzing the COVID-19 pandemic provided by Andrew Atkeson.

See, in particular

- NBER Working Paper No. 26867
- COVID-19 Working papers and code

The purpose of his notes is to introduce economists to quantitative modeling of infectious disease dynamics.

Dynamics are modeled using a standard SIR (Susceptible-Infected-Removed) model of disease spread.

The model dynamics are represented by a system of ordinary differential equations.

The main objective is to study the impact of suppression through social distancing on the spread of the infection.

The focus is on US outcomes but the parameters can be adjusted to study other countries.

We will use the following standard imports:

```
import matplotlib.pyplot as plt
import numpy as np
from numpy import exp
```

We will also use SciPy's numerical routine odeint for solving differential equations.

```
from scipy.integrate import odeint
```

This routine calls into compiled code from the FORTRAN library odepack.

1.2 The SIR Model

In the version of the SIR model we will analyze there are four states.

All individuals in the population are assumed to be in one of these four states.

The states are: susceptible (S), exposed (E), infected (I) and removed ®.

Comments:

- Those in state R have been infected and either recovered or died.
- Those who have recovered are assumed to have acquired immunity.
- Those in the exposed group are not yet infectious.

1.2.1 Time Path

The flow across states follows the path $S \to E \to I \to R$.

All individuals in the population are eventually infected when the transmission rate is positive and $i(0) > 0$.

The interest is primarily in

- the number of infections at a given time (which determines whether or not the health care system is overwhelmed) and
- how long the caseload can be deferred (hopefully until a vaccine arrives)

Using lower case letters for the fraction of the population in each state, the dynamics are

$$\begin{aligned}
\dot{s}(t) &= -\beta(t)\, s(t)\, i(t) \\
\dot{e}(t) &= \beta(t)\, s(t)\, i(t) - \sigma e(t) \\
\dot{i}(t) &= \sigma e(t) - \gamma i(t)
\end{aligned} \tag{1.1}$$

In these equations,

- $\beta(t)$ is called the **transmission rate** (the rate at which individuals bump into others and expose them to the virus).
- σ is called the **infection rate** (the rate at which those who are exposed become infected)
- γ is called the **recovery rate** (the rate at which infected people recover or die).
- the dot symbol $\dot{y}$ represents the time derivative dy/dt.

We do not need to model the fraction r of the population in state R separately because the states form a partition.

In particular, the "removed" fraction of the population is $r = 1 - s - e - i$.

We will also track $c = i + r$, which is the cumulative caseload (i.e., all those who have or have had the infection).

The system (1.1) can be written in vector form as

$$\dot{x} = F(x, t), \qquad x := (s, e, i) \tag{1.2}$$

for suitable definition of F (see the code below).

1.2.2 Parameters

Both σ and γ are thought of as fixed, biologically determined parameters.

As in Atkeson's note, we set

- $\sigma = 1/5.2$ to reflect an average incubation period of 5.2 days.
- $\gamma = 1/18$ to match an average illness duration of 18 days.

The transmission rate is modeled as

- $\beta(t) := R(t)\gamma$ where $R(t)$ is the **effective reproduction number** at time t.

(The notation is slightly confusing, since $R(t)$ is different to R, the symbol that represents the removed state.)

1.3 Implementation

First we set the population size to match the US.

```
pop_size = 3.3e8
```

Next we fix parameters as described above.

```
γ = 1 / 18
σ = 1 / 5.2
```

Now we construct a function that represents F in (1.2)

```
def F(x, t, R0=1.6):
    """
    Time derivative of the state vector.

        * x is the state vector (array_like)
        * t is time (scalar)
        * R0 is the effective transmission rate, defaulting to a constant

    """
    s, e, i = x

    # New exposure of susceptibles
    β = R0(t) * γ if callable(R0) else R0 * γ
    ne = β * s * i

    # Time derivatives
    ds = - ne
    de = ne - σ * e
    di = σ * e - γ * i

    return ds, de, di
```

Note that `R0` can be either constant or a given function of time.

The initial conditions are set to

```
# initial conditions of s, e, i
i_0 = 1e-7
```

(continues on next page)

(continued from previous page)

```
e_0 = 4 * i_0
s_0 = 1 - i_0 - e_0
```

In vector form the initial condition is

```
x_0 = s_0, e_0, i_0
```

We solve for the time path numerically using odeint, at a sequence of dates `t_vec`.

```
def solve_path(R0, t_vec, x_init=x_0):
    """
    Solve for i(t) and c(t) via numerical integration,
    given the time path for R0.

    """
    G = lambda x, t: F(x, t, R0)
    s_path, e_path, i_path = odeint(G, x_init, t_vec).transpose()

    c_path = 1 - s_path - e_path       # cumulative cases
    return i_path, c_path
```

1.4 Experiments

Let's run some experiments using this code.

The time period we investigate will be 550 days, or around 18 months:

```
t_length = 550
grid_size = 1000
t_vec = np.linspace(0, t_length, grid_size)
```

1.4.1 Experiment 1: Constant R0 Case

Let's start with the case where `R0` is constant.

We calculate the time path of infected people under different assumptions for `R0`:

```
R0_vals = np.linspace(1.6, 3.0, 6)
labels = [f'$R0 = {r:.2f}$' for r in R0_vals]
i_paths, c_paths = [], []

for r in R0_vals:
    i_path, c_path = solve_path(r, t_vec)
    i_paths.append(i_path)
    c_paths.append(c_path)
```

Here's some code to plot the time paths.

```
def plot_paths(paths, labels, times=t_vec):

    fig, ax = plt.subplots()

    for path, label in zip(paths, labels):
```

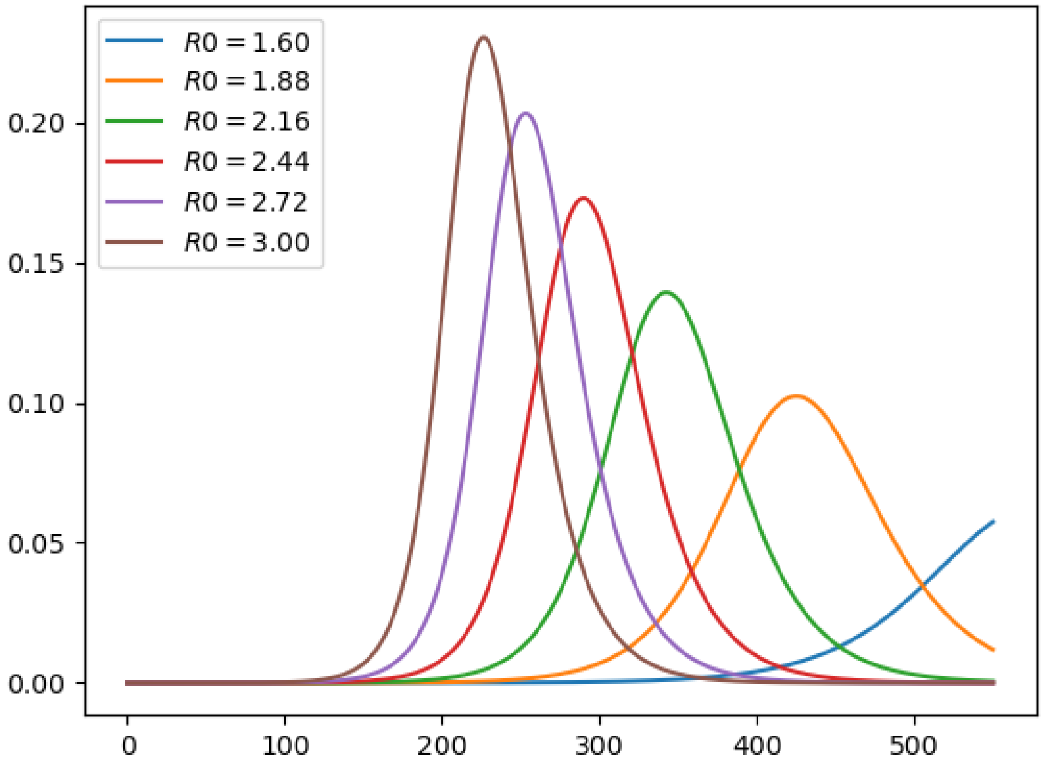
(continues on next page)

(continued from previous page)

```
        ax.plot(times, path, label=label)

    ax.legend(loc='upper left')

    plt.show()
```

Let's plot current cases as a fraction of the population.

```
plot_paths(i_paths, labels)
```

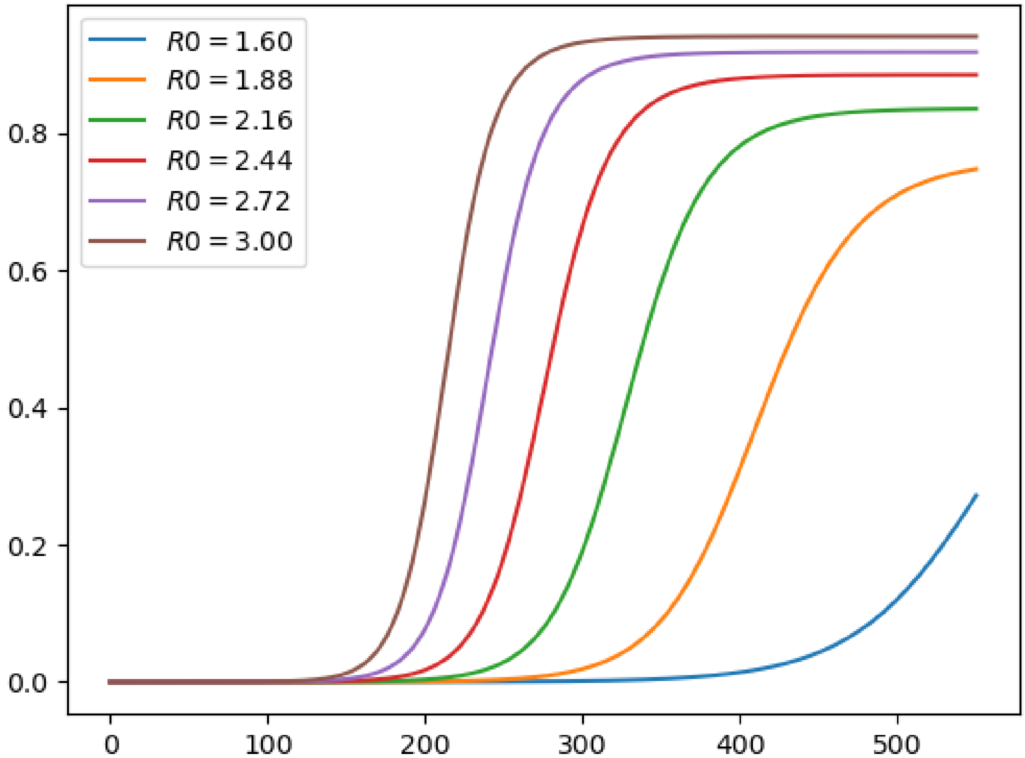


As expected, lower effective transmission rates defer the peak of infections.

They also lead to a lower peak in current cases.

Here are cumulative cases, as a fraction of population:

```
plot_paths(c_paths, labels)
```



1.4.2 Experiment 2: Changing Mitigation

Let's look at a scenario where mitigation (e.g., social distancing) is successively imposed.

Here's a specification for `R0` as a function of time.

```
def R0_mitigating(t, r0=3, η=1, r_bar=1.6):
    R0 = r0 * exp(- η * t) + (1 - exp(- η * t)) * r_bar
    return R0
```

The idea is that `R0` starts off at 3 and falls to 1.6.

This is due to progressive adoption of stricter mitigation measures.

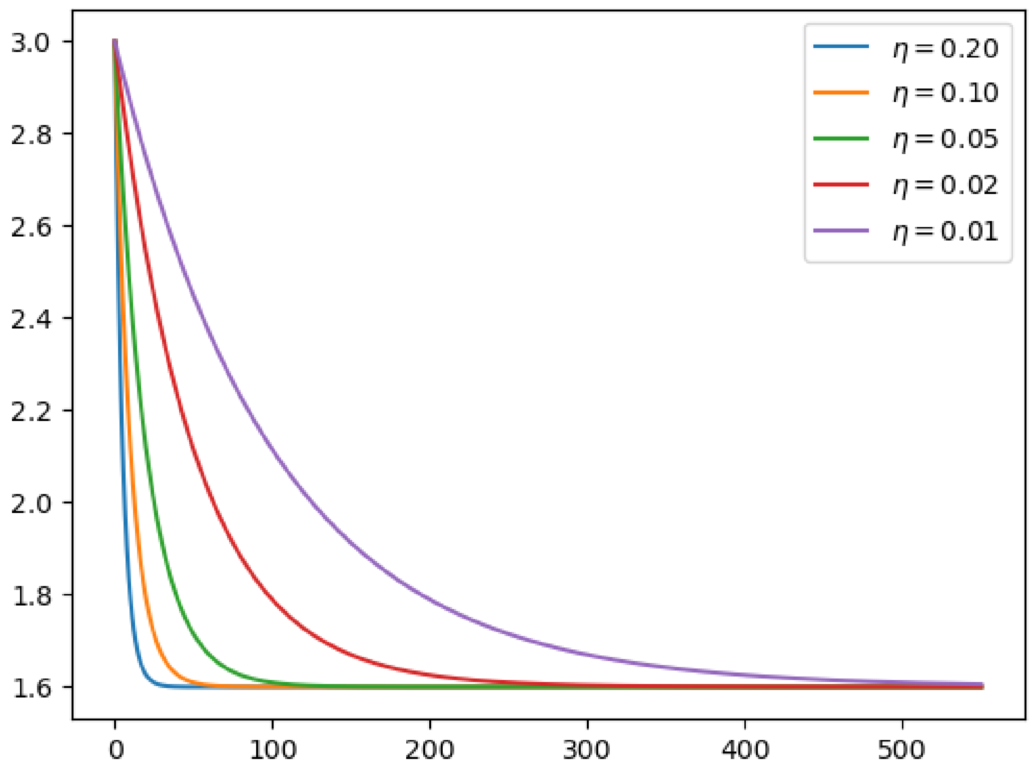
The parameter `η` controls the rate, or the speed at which restrictions are imposed.

We consider several different rates:

```
η_vals = 1/5, 1/10, 1/20, 1/50, 1/100
labels = [fr'$\eta = {η:.2f}$' for η in η_vals]
```

This is what the time path of `R0` looks like at these alternative rates:

```
fig, ax = plt.subplots()

for η, label in zip(η_vals, labels):
    ax.plot(t_vec, R0_mitigating(t_vec, η=η), label=label)

ax.legend()
plt.show()
```

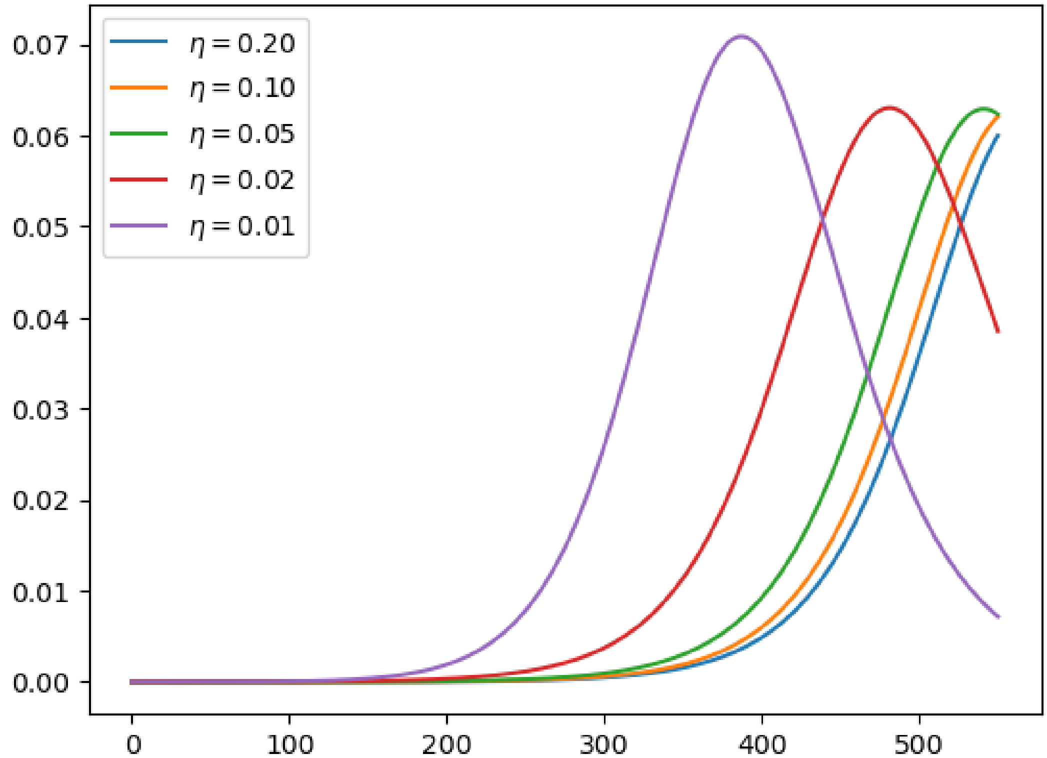


Let's calculate the time path of infected people:

```
i_paths, c_paths = [], []

for η in η_vals:
    R0 = lambda t: R0_mitigating(t, η=η)
    i_path, c_path = solve_path(R0, t_vec)
    i_paths.append(i_path)
    c_paths.append(c_path)
```

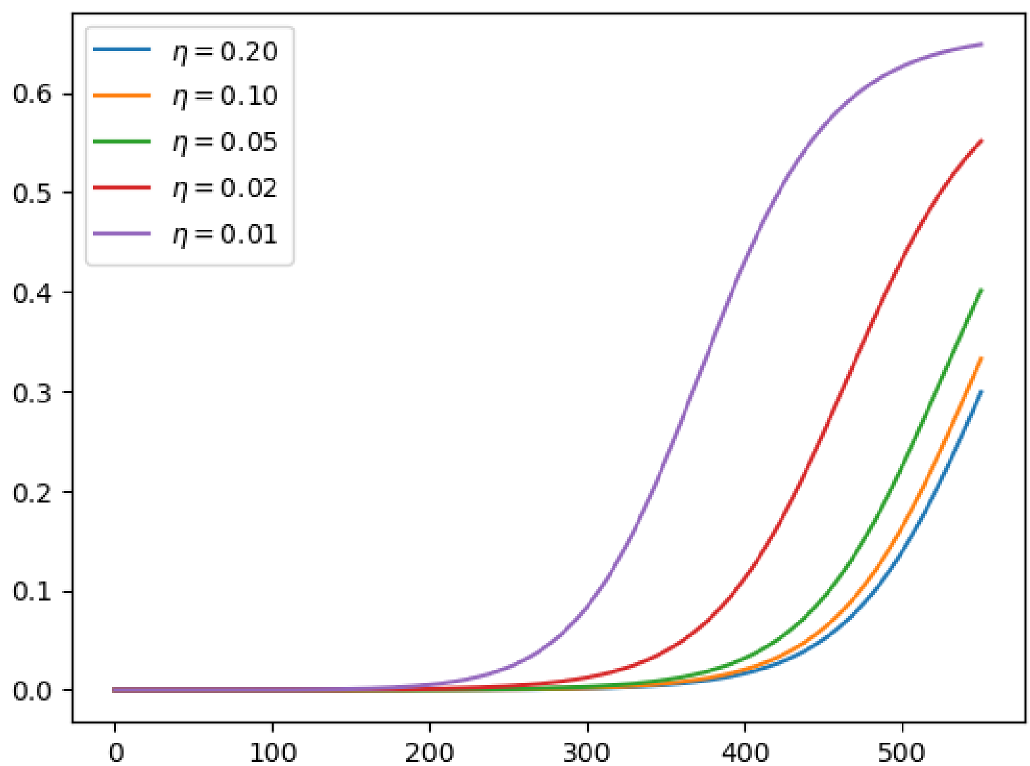
These are current cases under the different scenarios:

```
plot_paths(i_paths, labels)
```



Here are cumulative cases, as a fraction of population:

```
plot_paths(c_paths, labels)
```



1.5 Ending Lockdown

The following replicates additional results by Andrew Atkeson on the timing of lifting lockdown.

Consider these two mitigation scenarios:

1. $R_t = 0.5$ for 30 days and then $R_t = 2$ for the remaining 17 months. This corresponds to lifting lockdown in 30 days.
2. $R_t = 0.5$ for 120 days and then $R_t = 2$ for the remaining 14 months. This corresponds to lifting lockdown in 4 months.

The parameters considered here start the model with 25,000 active infections and 75,000 agents already exposed to the virus and thus soon to be contagious.

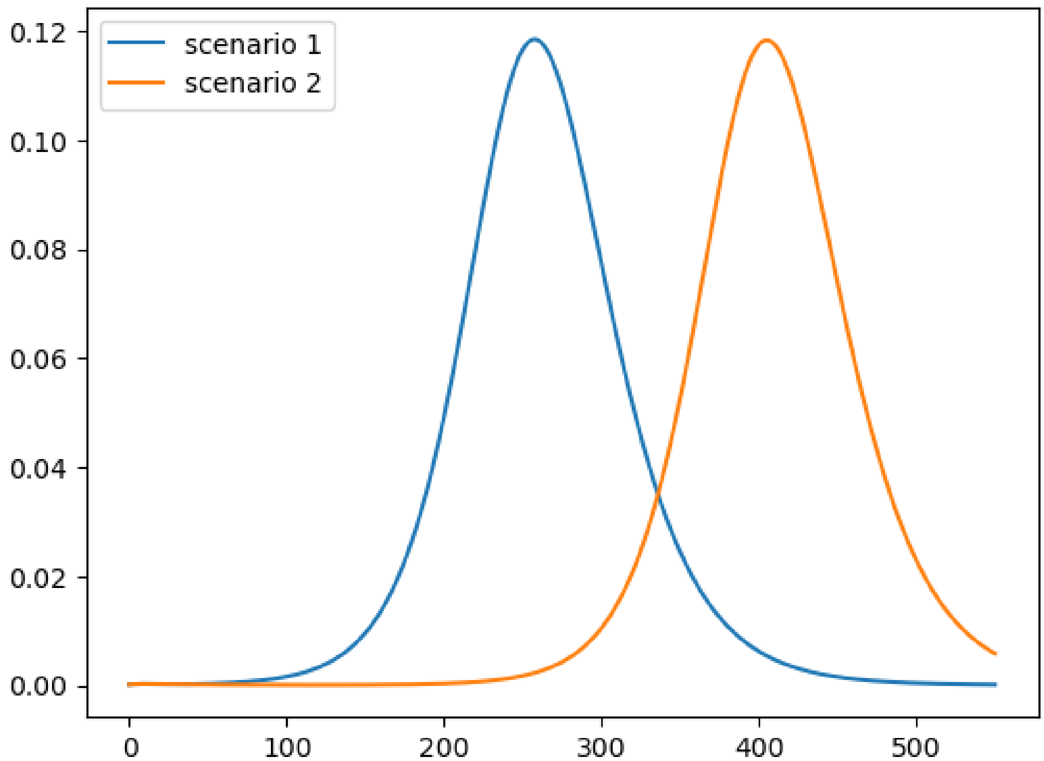
```
# initial conditions
i_0 = 25_000 / pop_size
e_0 = 75_000 / pop_size
s_0 = 1 - i_0 - e_0
x_0 = s_0, e_0, i_0
```

Let's calculate the paths:

```
R0_paths = (lambda t: 0.5 if t < 30 else 2,
            lambda t: 0.5 if t < 120 else 2)

labels = [f'scenario {i}' for i in (1, 2)]

i_paths, c_paths = [], []

for R0 in R0_paths:
    i_path, c_path = solve_path(R0, t_vec, x_init=x_0)
    i_paths.append(i_path)
    c_paths.append(c_path)
```

Here is the number of active infections:

```
plot_paths(i_paths, labels)
```



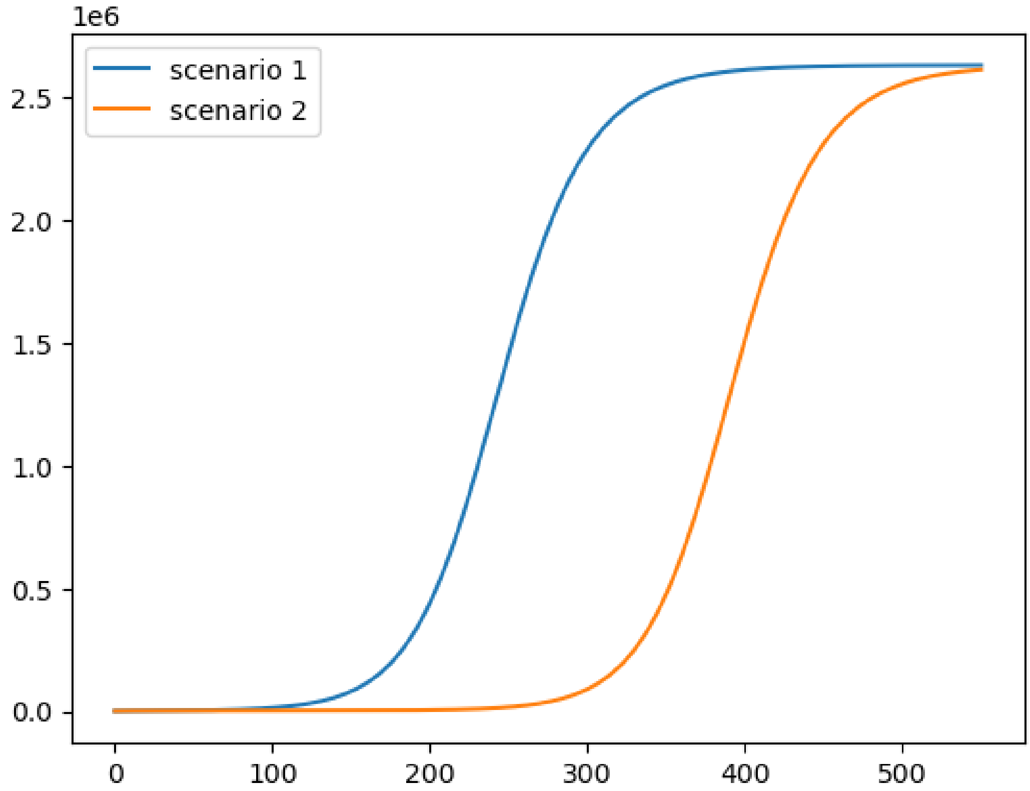
What kind of mortality can we expect under these scenarios?

Suppose that 1% of cases result in death

```
ν = 0.01
```

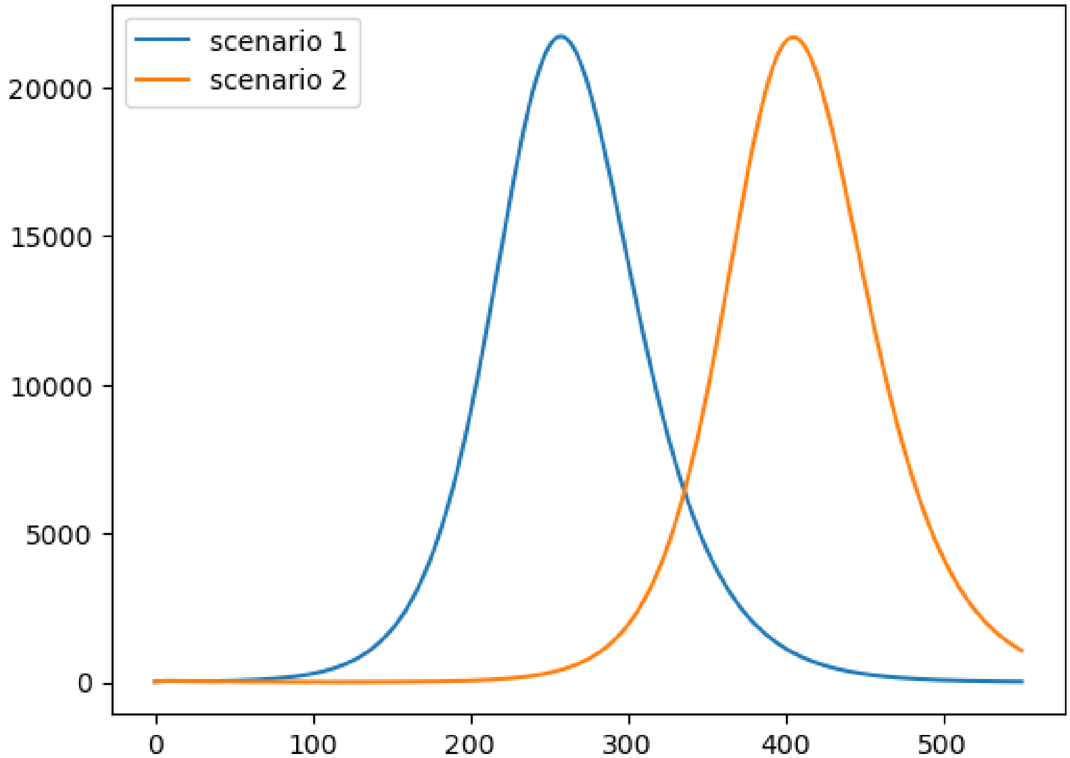
This is the cumulative number of deaths:

```
paths = [path * ν * pop_size for path in c_paths]
plot_paths(paths, labels)
```



This is the daily death rate:

```
paths = [path * ν * γ * pop_size for path in i_paths]
plot_paths(paths, labels)
```



Pushing the peak of curve further into the future may reduce cumulative deaths if a vaccine is found.

CHAPTER

TWO

LINEAR ALGEBRA

Contents

2.1 Overview

Linear algebra is one of the most useful branches of applied mathematics for economists to invest in.

For example, many applied problems in economics and finance require the solution of a linear system of equations, such as

$$
\begin{aligned}
y_1 &= a x_1 + b x_2 \\
y_2 &= c x_1 + d x_2
\end{aligned}
$$

or, more generally,

$$
\begin{aligned}
y_1 &= a_{11} x_1 + a_{12} x_2 + \cdots + a_{1k} x_k \\
&\vdots \\
y_n &= a_{n1} x_1 + a_{n2} x_2 + \cdots + a_{nk} x_k
\end{aligned}
\tag{2.1}
$$

The objective here is to solve for the "unknowns" $x_1, \ldots, x_k$ given $a_{11}, \ldots, a_{nk}$ and $y_1, \ldots, y_n$.

When considering such problems, it is essential that we first consider at least some of the following questions

- Does a solution actually exist?
- Are there in fact many solutions, and if so how should we interpret them?
- If no solution exists, is there a best "approximate" solution?

- If a solution exists, how should we compute it?

These are the kinds of topics addressed by linear algebra.

In this lecture we will cover the basics of linear and matrix algebra, treating both theory and computation.

We admit some overlap with this lecture, where operations on NumPy arrays were first explained.

Note that this lecture is more theoretical than most, and contains background material that will be used in applications as we go along.

Let's start with some imports:

```
import matplotlib.pyplot as plt
import numpy as np
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D
from scipy.linalg import inv, solve, det, eig
```

2.2 Vectors

A **vector** of length n is just a sequence (or array, or tuple) of n numbers, which we write as $x = (x_1, \ldots, x_n)$ or $x = [x_1, \ldots, x_n]$.

We will write these sequences either horizontally or vertically as we please.

(Later, when we wish to perform certain matrix operations, it will become necessary to distinguish between the two)

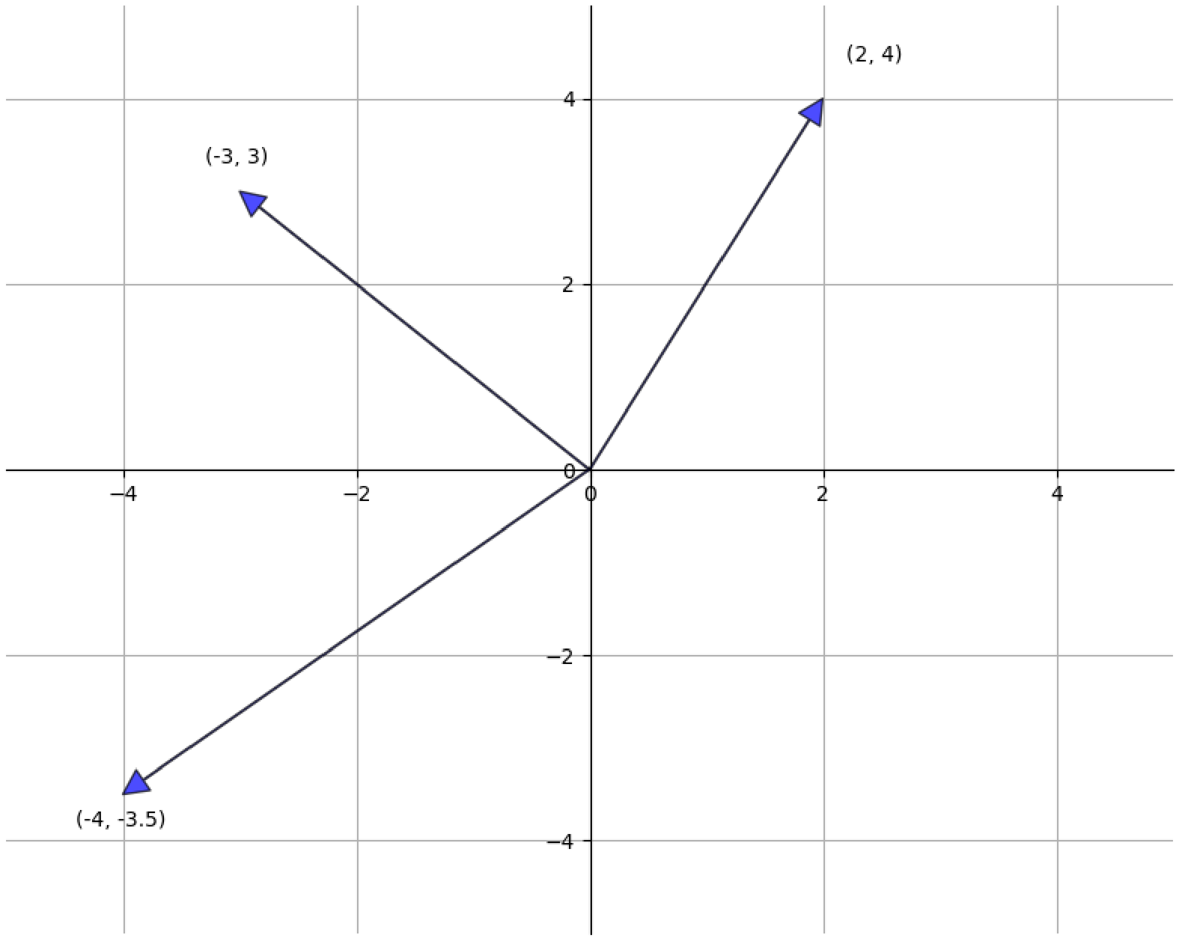
The set of all n-vectors is denoted by $\mathbb{R}^n$.

For example, $\mathbb{R}^2$ is the plane, and a vector in $\mathbb{R}^2$ is just a point in the plane.

Traditionally, vectors are represented visually as arrows from the origin to the point.

The following figure represents three vectors in this manner

```
fig, ax = plt.subplots(figsize=(10, 8))
# Set the axes through the origin
for spine in ['left', 'bottom']:
    ax.spines[spine].set_position('zero')
for spine in ['right', 'top']:
    ax.spines[spine].set_color('none')

ax.set(xlim=(-5, 5), ylim=(-5, 5))
ax.grid()
vecs = ((2, 4), (-3, 3), (-4, -3.5))
for v in vecs:
    ax.annotate('', xy=v, xytext=(0, 0),
                arrowprops=dict(facecolor='blue',
                shrink=0,
                alpha=0.7,
                width=0.5))
    ax.text(1.1 * v[0], 1.1 * v[1], str(v))
plt.show()
```



2.2.1 Vector Operations

The two most common operators for vectors are addition and scalar multiplication, which we now describe.

As a matter of definition, when we add two vectors, we add them element-by-element

$$x + y = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} := \begin{bmatrix} x_1 + y_1 \\ x_2 + y_2 \\ \vdots \\ x_n + y_n \end{bmatrix}$$

Scalar multiplication is an operation that takes a number γ and a vector x and produces

$$\gamma x := \begin{bmatrix} \gamma x_1 \\ \gamma x_2 \\ \vdots \\ \gamma x_n \end{bmatrix}$$

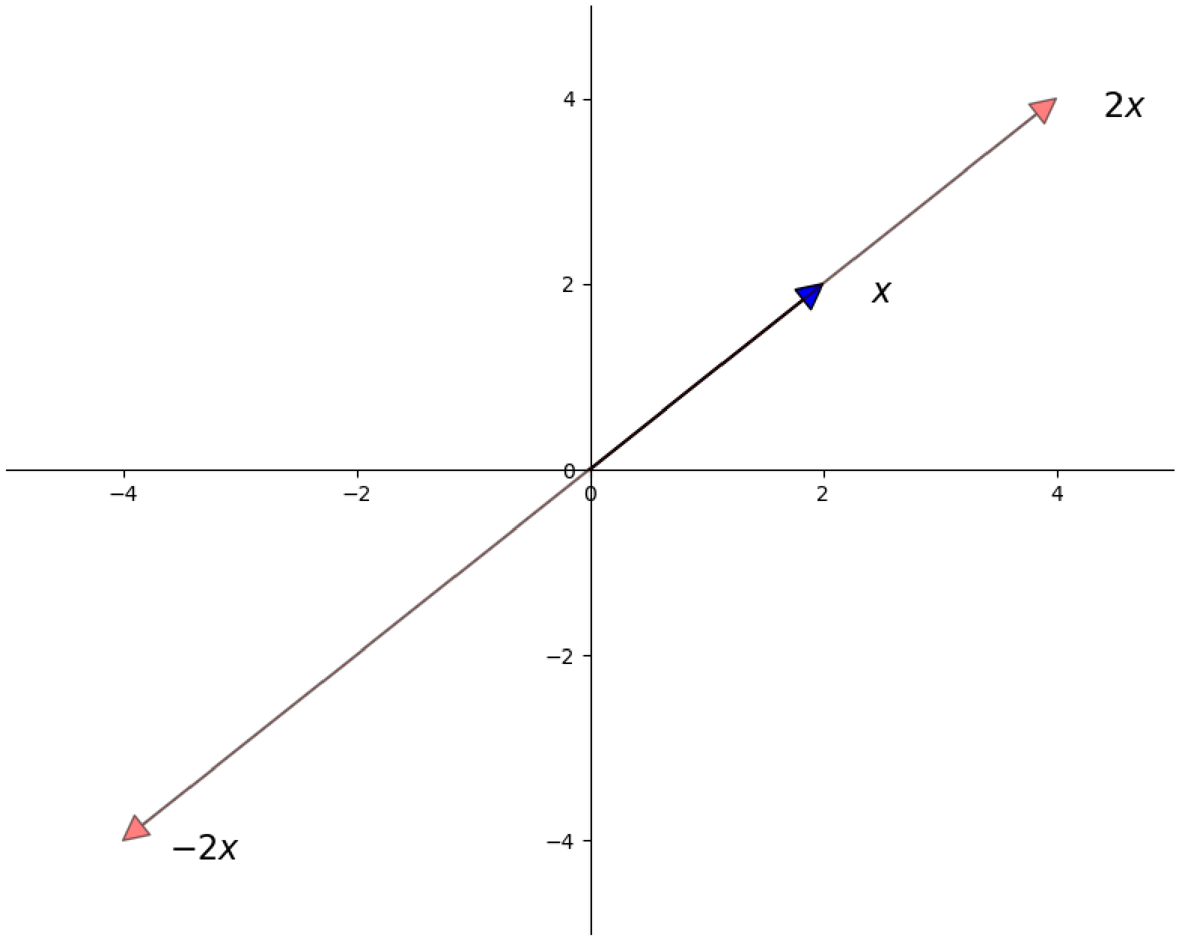
Scalar multiplication is illustrated in the next figure

```
fig, ax = plt.subplots(figsize=(10, 8))
# Set the axes through the origin
for spine in ['left', 'bottom']:
```

(continues on next page)

(continued from previous page)

```
    ax.spines[spine].set_position('zero')
for spine in ['right', 'top']:
    ax.spines[spine].set_color('none')

ax.set(xlim=(-5, 5), ylim=(-5, 5))
x = (2, 2)
ax.annotate('', xy=x, xytext=(0, 0),
            arrowprops=dict(facecolor='blue',
            shrink=0,
            alpha=1,
            width=0.5))
ax.text(x[0] + 0.4, x[1] - 0.2, '$x$', fontsize='16')


scalars = (-2, 2)
x = np.array(x)

for s in scalars:
    v = s * x
    ax.annotate('', xy=v, xytext=(0, 0),
                arrowprops=dict(facecolor='red',
                shrink=0,
                alpha=0.5,
                width=0.5))
    ax.text(v[0] + 0.4, v[1] - 0.2, f'${s} x$', fontsize='16')
plt.show()
```



In Python, a vector can be represented as a list or tuple, such as `x = (2, 4, 6)`, but is more commonly represented as a NumPy array.

One advantage of NumPy arrays is that scalar multiplication and addition have very natural syntax

```
x = np.ones(3)            # Vector of three ones
y = np.array((2, 4, 6))   # Converts tuple (2, 4, 6) into array
x + y
```

```
array([3., 5., 7.])
```

```
4 * x
```

```
array([4., 4., 4.])
```

2.2.2 Inner Product and Norm

The **inner product** of vectors $x, y \in \mathbb{R}^n$ is defined as

$$x'y := \sum_{i=1}^{n} x_i y_i$$

Two vectors are called **orthogonal** if their inner product is zero.

The **norm** of a vector x represents its "length" (i.e., its distance from the zero vector) and is defined as

$$\|x\| := \sqrt{x'x} := \left(\sum_{i=1}^{n} x_i^2 \right)^{1/2}$$

The expression $\|x - y\|$ is thought of as the distance between x and y.

Continuing on from the previous example, the inner product and norm can be computed as follows

```
np.sum(x * y)          # Inner product of x and y, method 1
```

```
np.float64(12.0)
```

```
x @ y                  # Inner product of x and y, method 2 (preferred)
```

```
np.float64(12.0)
```

The `@` operator is preferred because it uses optimized BLAS libraries that implement fused multiply-add operations, providing better performance and numerical accuracy compared to the separate multiply and sum operations.

```
np.sqrt(np.sum(x**2))  # Norm of x, take one
```

```
np.float64(1.7320508075688772)
```

```
np.sqrt(x @ x)         # Norm of x, take two (preferred)
```

```
np.float64(1.7320508075688772)
```

```
np.linalg.norm(x)      # Norm of x, take three
```

```
np.float64(1.7320508075688772)
```

2.2.3 Span

Given a set of vectors $A := \{a_1, \dots, a_k\}$ in $\mathbb{R}^n$, it's natural to think about the new vectors we can create by performing linear operations.

New vectors created in this manner are called **linear combinations** of A.

In particular, $y \in \mathbb{R}^n$ is a linear combination of $A := \{a_1, \dots, a_k\}$ if

$$y = \beta_1 a_1 + \cdots + \beta_k a_k \text{ for some scalars } \beta_1, \dots, \beta_k$$

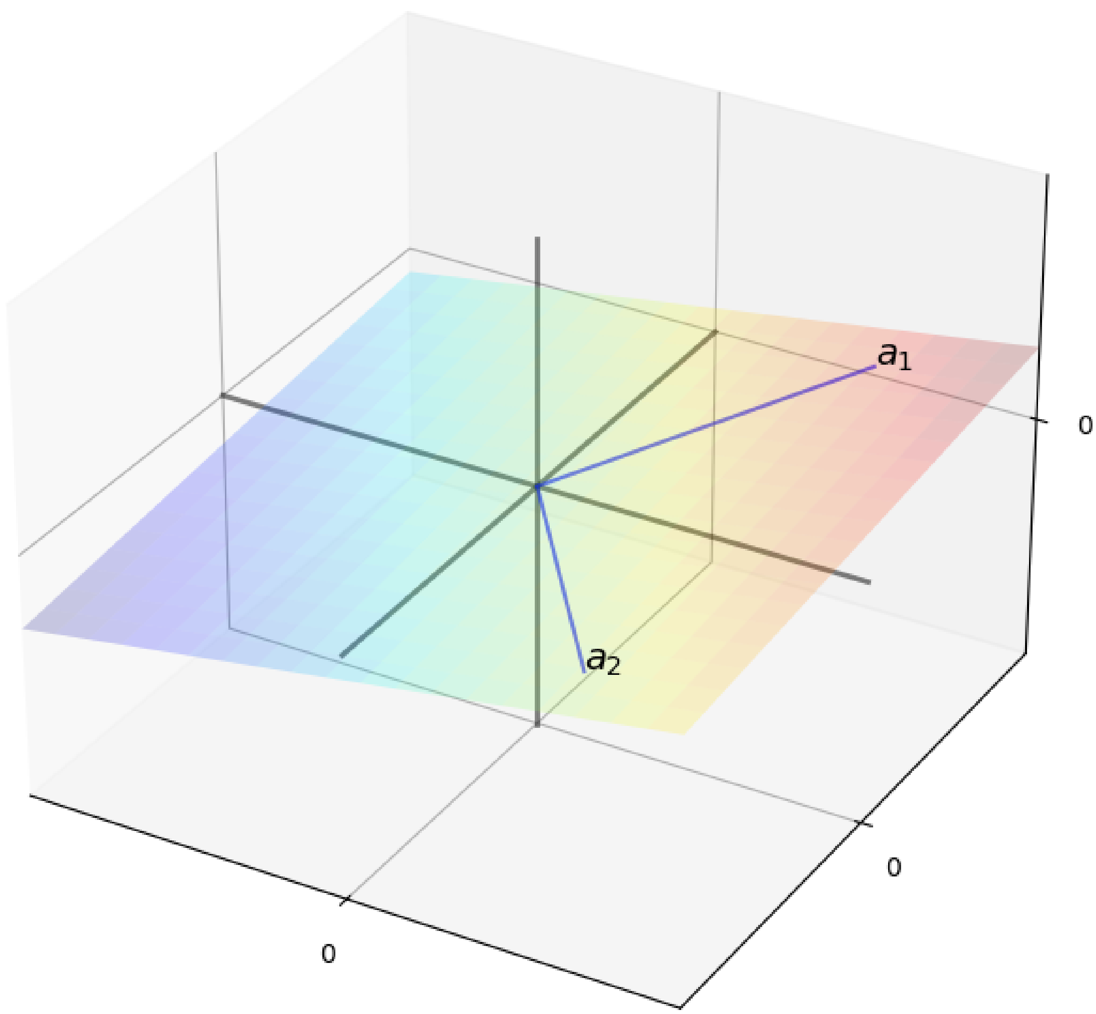
In this context, the values $\beta_1, \dots, \beta_k$ are called the **coefficients** of the linear combination.

The set of linear combinations of A is called the **span** of A.

The next figure shows the span of $A = \{a_1, a_2\}$ in $\mathbb{R}^3$.

The span is a two-dimensional plane passing through these two points and the origin.

```
ax = plt.figure(figsize=(10, 8)).add_subplot(projection='3d')

x_min, x_max = -5, 5
y_min, y_max = -5, 5

α, β = 0.2, 0.1

ax.set(xlim=(x_min, x_max), ylim=(x_min, x_max), zlim=(x_min, x_max),
       xticks=(0,), yticks=(0,), zticks=(0,))

gs = 3
z = np.linspace(x_min, x_max, gs)
x = np.zeros(gs)
y = np.zeros(gs)
ax.plot(x, y, z, 'k-', lw=2, alpha=0.5)
ax.plot(z, x, y, 'k-', lw=2, alpha=0.5)
ax.plot(y, z, x, 'k-', lw=2, alpha=0.5)


# Fixed linear function, to generate a plane
def f(x, y):
    return α * x + β * y

# Vector locations, by coordinate
x_coords = np.array((3, 3))
y_coords = np.array((4, -4))
z = f(x_coords, y_coords)
for i in (0, 1):
    ax.text(x_coords[i], y_coords[i], z[i], f'$a_{i+1}$', fontsize=14)

# Lines to vectors
for i in (0, 1):
    x = (0, x_coords[i])
    y = (0, y_coords[i])
    z = (0, f(x_coords[i], y_coords[i]))
    ax.plot(x, y, z, 'b-', lw=1.5, alpha=0.6)


# Draw the plane
grid_size = 20
xr2 = np.linspace(x_min, x_max, grid_size)
yr2 = np.linspace(y_min, y_max, grid_size)
x2, y2 = np.meshgrid(xr2, yr2)
z2 = f(x2, y2)
ax.plot_surface(x2, y2, z2, rstride=1, cstride=1, cmap=cm.jet,
                linewidth=0, antialiased=True, alpha=0.2)
plt.show()
```



Examples

If A contains only one vector $a_1 \in \mathbb{R}^2$, then its span is just the scalar multiples of a_1, which is the unique line passing through both a_1 and the origin.

If $A = \{e_1, e_2, e_3\}$ consists of the **canonical basis vectors** of $\mathbb{R}^3$, that is

$$e_1 := \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad e_2 := \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad e_3 := \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

then the span of A is all of $\mathbb{R}^3$, because, for any $x = (x_1, x_2, x_3) \in \mathbb{R}^3$, we can write

$$x = x_1 e_1 + x_2 e_2 + x_3 e_3$$

Now consider $A_0 = \{e_1, e_2, e_1 + e_2\}$.

If $y = (y_1, y_2, y_3)$ is any linear combination of these vectors, then $y_3 = 0$ (check it).

Hence A_0 fails to span all of $\mathbb{R}^3$.

2.2.4 Linear Independence

As we'll see, it's often desirable to find families of vectors with relatively large span, so that many vectors can be described by linear operators on a few vectors.

The condition we need for a set of vectors to have a large span is what's called linear independence.

In particular, a collection of vectors $A := \{a_1, \ldots, a_k\}$ in $\mathbb{R}^n$ is said to be

- **linearly dependent** if some strict subset of A has the same span as A.
- **linearly independent** if it is not linearly dependent.

Put differently, a set of vectors is linearly independent if no vector is redundant to the span and linearly dependent otherwise.

To illustrate the idea, recall *the figure* that showed the span of vectors $\{a_1, a_2\}$ in $\mathbb{R}^3$ as a plane through the origin.

If we take a third vector a_3 and form the set $\{a_1, a_2, a_3\}$, this set will be

- linearly dependent if a_3 lies in the plane
- linearly independent otherwise

As another illustration of the concept, since $\mathbb{R}^n$ can be spanned by n vectors (see the discussion of canonical basis vectors above), any collection of $m > n$ vectors in $\mathbb{R}^n$ must be linearly dependent.

The following statements are equivalent to linear independence of $A := \{a_1, \ldots, a_k\} \subset \mathbb{R}^n$

1. No vector in A can be formed as a linear combination of the other elements.
2. If $\beta_1 a_1 + \cdots \beta_k a_k = 0$ for scalars $\beta_1, \ldots, \beta_k$, then $\beta_1 = \cdots = \beta_k = 0$.

(The zero in the first expression is the origin of $\mathbb{R}^n$)

2.2.5 Unique Representations

Another nice thing about sets of linearly independent vectors is that each element in the span has a unique representation as a linear combination of these vectors.

In other words, if $A := \{a_1, \ldots, a_k\} \subset \mathbb{R}^n$ is linearly independent and

$$y = \beta_1 a_1 + \cdots \beta_k a_k$$

then no other coefficient sequence $\gamma_1, \ldots, \gamma_k$ will produce the same vector y.

Indeed, if we also have $y = \gamma_1 a_1 + \cdots \gamma_k a_k$, then

$$(\beta_1 - \gamma_1) a_1 + \cdots + (\beta_k - \gamma_k) a_k = 0$$

Linear independence now implies $\gamma_i = \beta_i$ for all i.

2.3 Matrices

Matrices are a neat way of organizing data for use in linear operations.

An $n \times k$ matrix is a rectangular array A of numbers with n rows and k columns:

$$
A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1k} \\ a_{21} & a_{22} & \cdots & a_{2k} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nk} \end{bmatrix}
$$

Often, the numbers in the matrix represent coefficients in a system of linear equations, as discussed at the start of this lecture.

For obvious reasons, the matrix A is also called a vector if either $n = 1$ or $k = 1$.

In the former case, A is called a **row vector**, while in the latter it is called a **column vector**.

If $n = k$, then A is called **square**.

The matrix formed by replacing a_{ij} by a_{ji} for every i and j is called the **transpose** of A and denoted A' or $A^\top$.

If $A = A'$, then A is called **symmetric**.

For a square matrix A, the i elements of the form a_{ii} for $i = 1, \ldots, n$ are called the **principal diagonal**.

A is called **diagonal** if the only nonzero entries are on the principal diagonal.

If, in addition to being diagonal, each element along the principal diagonal is equal to 1, then A is called the **identity matrix** and denoted by I.

2.3.1 Matrix Operations

Just as was the case for vectors, a number of algebraic operations are defined for matrices.

Scalar multiplication and addition are immediate generalizations of the vector case:

$$
\gamma A = \gamma \begin{bmatrix} a_{11} & \cdots & a_{1k} \\ \vdots & \vdots & \vdots \\ a_{n1} & \cdots & a_{nk} \end{bmatrix} := \begin{bmatrix} \gamma a_{11} & \cdots & \gamma a_{1k} \\ \vdots & \vdots & \vdots \\ \gamma a_{n1} & \cdots & \gamma a_{nk} \end{bmatrix}
$$

and

$$
A + B = \begin{bmatrix} a_{11} & \cdots & a_{1k} \\ \vdots & \vdots & \vdots \\ a_{n1} & \cdots & a_{nk} \end{bmatrix} + \begin{bmatrix} b_{11} & \cdots & b_{1k} \\ \vdots & \vdots & \vdots \\ b_{n1} & \cdots & b_{nk} \end{bmatrix} := \begin{bmatrix} a_{11} + b_{11} & \cdots & a_{1k} + b_{1k} \\ \vdots & \vdots & \vdots \\ a_{n1} + b_{n1} & \cdots & a_{nk} + b_{nk} \end{bmatrix}
$$

In the latter case, the matrices must have the same shape in order for the definition to make sense.

We also have a convention for *multiplying* two matrices.

The rule for matrix multiplication generalizes the idea of inner products discussed above and is designed to make multiplication play well with basic linear operations.

If A and B are two matrices, then their product AB is formed by taking as its i, j-th element the inner product of the i-th row of A and the j-th column of B.

There are many tutorials to help you visualize this operation, such as this one, or the discussion on the Wikipedia page.

If A is $n \times k$ and B is $j \times m$, then to multiply A and B we require $k = j$, and the resulting matrix AB is $n \times m$.

As perhaps the most important special case, consider multiplying $n \times k$ matrix A and $k \times 1$ column vector x.

According to the preceding rule, this gives us an $n \times 1$ column vector

$$Ax = \begin{bmatrix} a_{11} & \cdots & a_{1k} \\ \vdots & \vdots & \vdots \\ a_{n1} & \cdots & a_{nk} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_k \end{bmatrix} := \begin{bmatrix} a_{11}x_1 + \cdots + a_{1k}x_k \\ \vdots \\ a_{n1}x_1 + \cdots + a_{nk}x_k \end{bmatrix} \tag{2.2}$$

Note

AB and BA are not generally the same thing.

Another important special case is the identity matrix.

You should check that if A is $n \times k$ and I is the $k \times k$ identity matrix, then $AI = A$.

If I is the $n \times n$ identity matrix, then $IA = A$.

2.3.2 Matrices in NumPy

NumPy arrays are also used as matrices, and have fast, efficient functions and methods for all the standard matrix operations[1].

You can create them manually from tuples of tuples (or lists of lists) as follows

```
A = ((1, 2),
     (3, 4))

type(A)
```

```
tuple
```

```
A = np.array(A)

type(A)
```

```
numpy.ndarray
```

```
A.shape
```

```
(2, 2)
```

The `shape` attribute is a tuple giving the number of rows and columns — see here for more discussion.

To get the transpose of `A`, use `A.transpose()` or, more simply, `A.T`.

There are many convenient functions for creating common matrices (matrices of zeros, ones, etc.) — see here.

Since operations are performed elementwise by default, scalar multiplication and addition have very natural syntax

```
A = np.identity(3)
B = np.ones((3, 3))
2 * A
```

[1] Although there is a specialized matrix data type defined in NumPy, it's more standard to work with ordinary NumPy arrays. See this discussion.

```
array([[2., 0., 0.],
       [0., 2., 0.],
       [0., 0., 2.]])
```

```
A + B
```

```
array([[2., 1., 1.],
       [1., 2., 1.],
       [1., 1., 2.]])
```

To multiply matrices we use the `@` symbol.

In particular, `A @ B` is matrix multiplication, whereas `A * B` is element-by-element multiplication.

See here for more discussion.

2.3.3 Matrices as Maps

Each $n \times k$ matrix A can be identified with a function $f(x) = Ax$ that maps $x \in \mathbb{R}^k$ into $y = Ax \in \mathbb{R}^n$.

These kinds of functions have a special property: they are **linear**.

A function $f: \mathbb{R}^k \to \mathbb{R}^n$ is called **linear** if, for all $x, y \in \mathbb{R}^k$ and all scalars α, β, we have

$$f(\alpha x + \beta y) = \alpha f(x) + \beta f(y)$$

You can check that this holds for the function $f(x) = Ax + b$ when b is the zero vector and fails when b is nonzero.

In fact, it's known that f is linear if and *only if* there exists a matrix A such that $f(x) = Ax$ for all x.

2.4 Solving Systems of Equations

Recall again the system of equations (2.1).

If we compare (2.1) and (2.2), we see that (2.1) can now be written more conveniently as

$$y = Ax \tag{2.3}$$

The problem we face is to determine a vector $x \in \mathbb{R}^k$ that solves (2.3), taking y and A as given.

This is a special case of a more general problem: Find an x such that $y = f(x)$.

Given an arbitrary function f and a y, is there always an x such that $y = f(x)$?

If so, is it always unique?

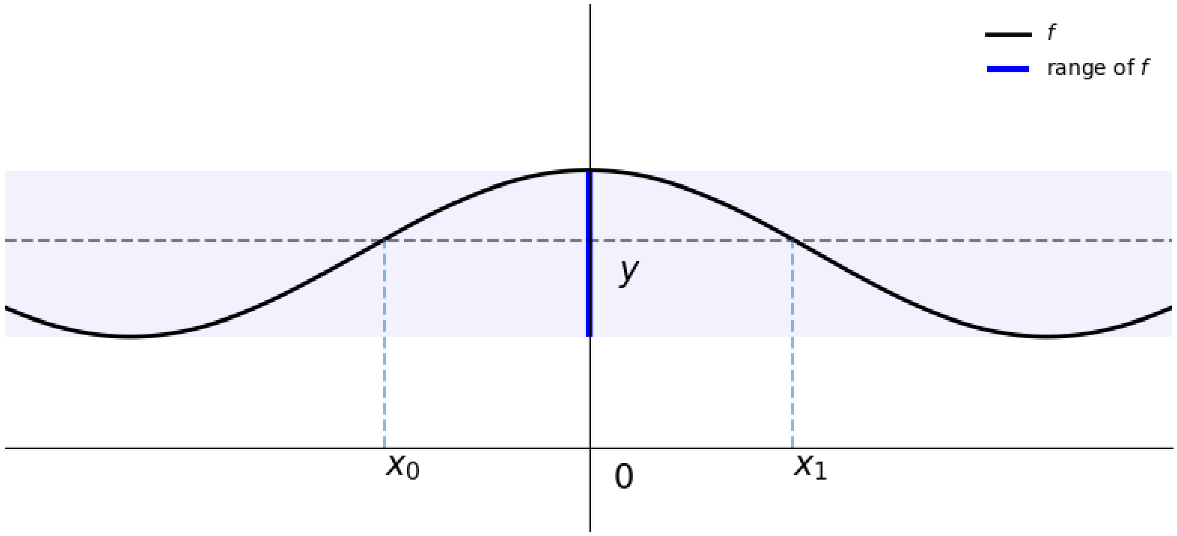
The answer to both these questions is negative, as the next figure shows

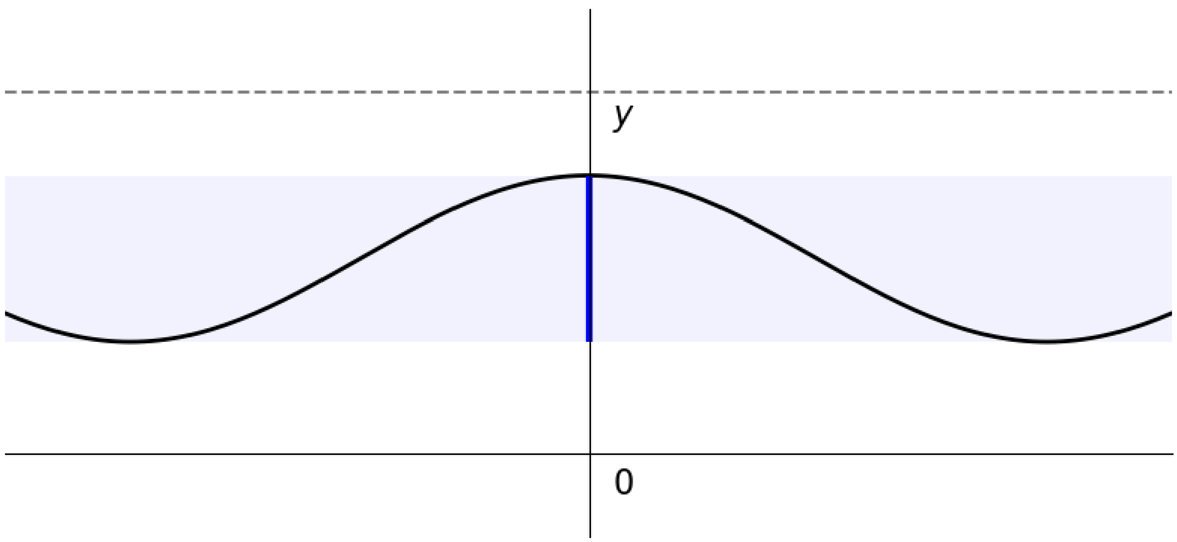
```
def f(x):
    return 0.6 * np.cos(4 * x) + 1.4


xmin, xmax = -1, 1
x = np.linspace(xmin, xmax, 160)
y = f(x)
ya, yb = np.min(y), np.max(y)
```

(continues on next page)

(continued from previous page)

```
fig, axes = plt.subplots(2, 1, figsize=(10, 10))

for ax in axes:
    # Set the axes through the origin
    for spine in ['left', 'bottom']:
        ax.spines[spine].set_position('zero')
    for spine in ['right', 'top']:
        ax.spines[spine].set_color('none')

    ax.set(ylim=(-0.6, 3.2), xlim=(xmin, xmax),
           yticks=(), xticks=())

    ax.plot(x, y, 'k-', lw=2, label='$f$')
    ax.fill_between(x, ya, yb, facecolor='blue', alpha=0.05)
    ax.vlines([0], ya, yb, lw=3, color='blue', label='range of $f$')
    ax.text(0.04, -0.3, '$0$', fontsize=16)

ax = axes[0]

ax.legend(loc='upper right', frameon=False)
ybar = 1.5
ax.plot(x, x * 0 + ybar, 'k--', alpha=0.5)
ax.text(0.05, 0.8 * ybar, '$y$', fontsize=16)
for i, z in enumerate((-0.35, 0.35)):
    ax.vlines(z, 0, f(z), linestyle='--', alpha=0.5)
    ax.text(z, -0.2, f'$x_{i}$', fontsize=16)

ax = axes[1]

ybar = 2.6
ax.plot(x, x * 0 + ybar, 'k--', alpha=0.5)
ax.text(0.04, 0.91 * ybar, '$y$', fontsize=16)

plt.show()
```





In the first plot, there are multiple solutions, as the function is not one-to-one, while in the second there are no solutions, since y lies outside the range of f.

Can we impose conditions on A in (2.3) that rule out these problems?

In this context, the most important thing to recognize about the expression Ax is that it corresponds to a linear combination of the columns of A.

In particular, if $a_1, \ldots, a_k$ are the columns of A, then

$$Ax = x_1 a_1 + \cdots + x_k a_k$$

Hence the range of $f(x) = Ax$ is exactly the span of the columns of A.

We want the range to be large so that it contains arbitrary y.

As you might recall, the condition that we want for the span to be large is *linear independence*.

A happy fact is that linear independence of the columns of A also gives us uniqueness.

Indeed, it follows from our *earlier discussion* that if $\{a_1, \ldots, a_k\}$ are linearly independent and $y = Ax = x_1 a_1 + \cdots + x_k a_k$, then no $z \neq x$ satisfies $y = Az$.

2.4.1 The Square Matrix Case

Let's discuss some more details, starting with the case where A is $n \times n$.

This is the familiar case where the number of unknowns equals the number of equations.

For arbitrary $y \in \mathbb{R}^n$, we hope to find a unique $x \in \mathbb{R}^n$ such that $y = Ax$.

In view of the observations immediately above, if the columns of A are linearly independent, then their span, and hence the range of $f(x) = Ax$, is all of $\mathbb{R}^n$.

Hence there always exists an x such that $y = Ax$.

Moreover, the solution is unique.

In particular, the following are equivalent

1. The columns of A are linearly independent.
2. For any $y \in \mathbb{R}^n$, the equation $y = Ax$ has a unique solution.

The property of having linearly independent columns is sometimes expressed as having **full column rank**.

Inverse Matrices

Can we give some sort of expression for the solution?

If y and A are scalar with $A \neq 0$, then the solution is $x = A^{-1} y$.

A similar expression is available in the matrix case.

In particular, if square matrix A has full column rank, then it possesses a multiplicative **inverse matrix** A^{-1}, with the property that $AA^{-1} = A^{-1}A = I$.

As a consequence, if we pre-multiply both sides of $y = Ax$ by A^{-1}, we get $x = A^{-1} y$.

This is the solution that we're looking for.

Determinants

Another quick comment about square matrices is that to every such matrix we assign a unique number called the **determinant** of the matrix — you can find the expression for it here.

If the determinant of A is not zero, then we say that A is **nonsingular**.

Perhaps the most important fact about determinants is that A is nonsingular if and only if A is of full column rank.

This gives us a useful one-number summary of whether or not a square matrix can be inverted.

2.4.2 More Rows than Columns

This is the $n \times k$ case with $n > k$.

This case is very important in many settings, not least in the setting of linear regression (where n is the number of observations, and k is the number of explanatory variables).

Given arbitrary $y \in \mathbb{R}^n$, we seek an $x \in \mathbb{R}^k$ such that $y = Ax$.

In this setting, the existence of a solution is highly unlikely.

Without much loss of generality, let's go over the intuition focusing on the case where the columns of A are linearly independent.

It follows that the span of the columns of A is a k-dimensional subspace of $\mathbb{R}^n$.

This span is very "unlikely" to contain arbitrary $y \in \mathbb{R}^n$.

To see why, recall the *figure above*, where $k = 2$ and $n = 3$.

Imagine an arbitrarily chosen $y \in \mathbb{R}^3$, located somewhere in that three-dimensional space.

What's the likelihood that y lies in the span of $\{a_1, a_2\}$ (i.e., the two dimensional plane through these points)?

In a sense, it must be very small, since this plane has zero "thickness".

As a result, in the $n > k$ case we usually give up on existence.

However, we can still seek the best approximation, for example, an x that makes the distance $\|y - Ax\|$ as small as possible.

To solve this problem, one can use either calculus or the theory of orthogonal projections.

The solution is known to be $\hat{x} = (A'A)^{-1}A'y$ — see for example chapter 3 of these notes.

2.4.3 More Columns than Rows

This is the $n \times k$ case with $n < k$, so there are fewer equations than unknowns.

In this case there are either no solutions or infinitely many — in other words, uniqueness never holds.

For example, consider the case where $k = 3$ and $n = 2$.

Thus, the columns of A consists of 3 vectors in $\mathbb{R}^2$.

This set can never be linearly independent, since it is possible to find two vectors that span $\mathbb{R}^2$.

(For example, use the canonical basis vectors)

It follows that one column is a linear combination of the other two.

For example, let's say that $a_1 = \alpha a_2 + \beta a_3$.

Then if $y = Ax = x_1 a_1 + x_2 a_2 + x_3 a_3$, we can also write

$$y = x_1(\alpha a_2 + \beta a_3) + x_2 a_2 + x_3 a_3 = (x_1 \alpha + x_2) a_2 + (x_1 \beta + x_3) a_3$$

In other words, uniqueness fails.

2.4.4 Linear Equations with SciPy

Here's an illustration of how to solve linear equations with SciPy's `linalg` submodule.

All of these routines are Python front ends to time-tested and highly optimized FORTRAN code

```
A = ((1, 2), (3, 4))
A = np.array(A)
y = np.ones((2, 1))  # Column vector
det(A)  # Check that A is nonsingular, and hence invertible
```

```
np.float64(-2.0)
```

```
A_inv = inv(A)  # Compute the inverse
A_inv
```

```
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
```

```
x = A_inv @ y  # Solution
A @ x          # Should equal y
```

```
array([[1.],
       [1.]])
```

```
solve(A, y)  # Produces the same solution
```

```
array([[-1.],
       [ 1.]])
```

Observe how we can solve for $x = A^{-1}y$ by either via `inv(A) @ y`, or using `solve(A, y)`.

The latter method uses a different algorithm (LU decomposition) that is numerically more stable, and hence should almost always be preferred.

To obtain the least-squares solution $\hat{x} = (A'A)^{-1}A'y$, use `scipy.linalg.lstsq(A, y)`.

2.5 Eigenvalues and Eigenvectors

Let A be an $n \times n$ square matrix.

If λ is scalar and v is a non-zero vector in $\mathbb{R}^n$ such that

$$Av = \lambda v$$

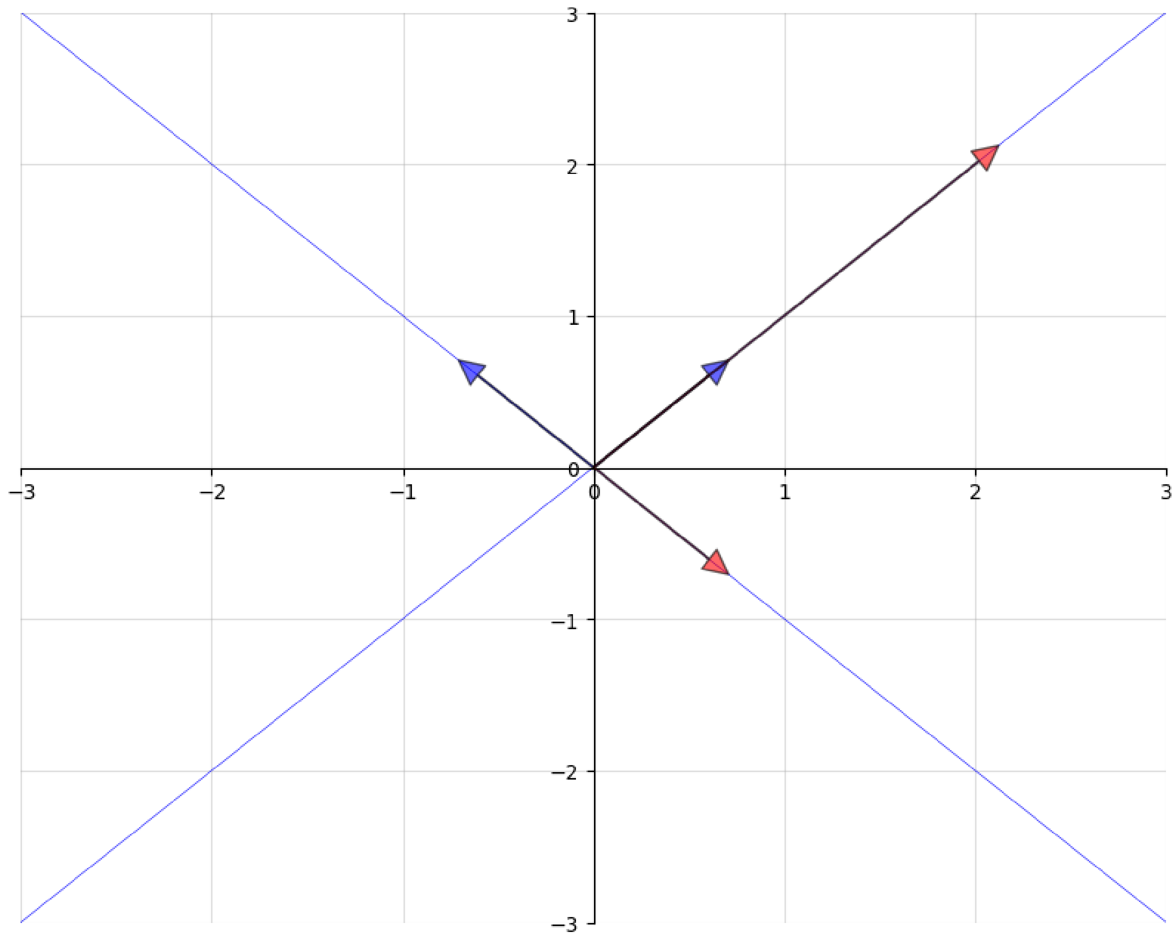
then we say that λ is an **eigenvalue** of A, and v is an **eigenvector**.

Thus, an eigenvector of A is a vector such that when the map $f(x) = Ax$ is applied, v is merely scaled.

The next figure shows two eigenvectors (blue arrows) and their images under A (red arrows).

As expected, the image Av of each v is just a scaled version of the original

```python
A = ((1, 2),
     (2, 1))
A = np.array(A)
evals, evecs = eig(A)
evecs = evecs[:, 0], evecs[:, 1]

fig, ax = plt.subplots(figsize=(10, 8))
# Set the axes through the origin
for spine in ['left', 'bottom']:
    ax.spines[spine].set_position('zero')
for spine in ['right', 'top']:
    ax.spines[spine].set_color('none')
ax.grid(alpha=0.4)

xmin, xmax = -3, 3
ymin, ymax = -3, 3
ax.set(xlim=(xmin, xmax), ylim=(ymin, ymax))

# Plot each eigenvector
for v in evecs:
    ax.annotate('', xy=v, xytext=(0, 0),
                arrowprops=dict(facecolor='blue',
                shrink=0,
                alpha=0.6,
                width=0.5))

# Plot the image of each eigenvector
for v in evecs:
    v = A @ v
    ax.annotate('', xy=v, xytext=(0, 0),
                arrowprops=dict(facecolor='red',
                shrink=0,
                alpha=0.6,
                width=0.5))

# Plot the lines they run through
x = np.linspace(xmin, xmax, 3)
for v in evecs:
    a = v[1] / v[0]
    ax.plot(x, a * x, 'b-', lw=0.4)

plt.show()
```



The eigenvalue equation is equivalent to $(A - \lambda I)v = 0$, and this has a nonzero solution v only when the columns of $A - \lambda I$ are linearly dependent.

This in turn is equivalent to stating that the determinant is zero.

Hence to find all eigenvalues, we can look for λ such that the determinant of $A - \lambda I$ is zero.

This problem can be expressed as one of solving for the roots of a polynomial in λ of degree n.

This in turn implies the existence of n solutions in the complex plane, although some might be repeated.

Some nice facts about the eigenvalues of a square matrix A are as follows

1. The determinant of A equals the product of the eigenvalues.
2. The trace of A (the sum of the elements on the principal diagonal) equals the sum of the eigenvalues.
3. If A is symmetric, then all of its eigenvalues are real.
4. If A is invertible and $\lambda_1, \ldots, \lambda_n$ are its eigenvalues, then the eigenvalues of A^{-1} are $1/\lambda_1, \ldots, 1/\lambda_n$.

A corollary of the first statement is that a matrix is invertible if and only if all its eigenvalues are nonzero.

Using SciPy, we can solve for the eigenvalues and eigenvectors of a matrix as follows

```
A = ((1, 2),
     (2, 1))
```

(continues on next page)

(continued from previous page)

```
A = np.array(A)
evals, evecs = eig(A)
evals
```

```
array([ 3.+0.j, -1.+0.j])
```

```
evecs
```

```
array([[ 0.70710678, -0.70710678],
       [ 0.70710678,  0.70710678]])
```

Note that the *columns* of `evecs` are the eigenvectors.

Since any scalar multiple of an eigenvector is an eigenvector with the same eigenvalue (check it), the eig routine normalizes the length of each eigenvector to one.

2.5.1 Generalized Eigenvalues

It is sometimes useful to consider the **generalized eigenvalue problem**, which, for given matrices A and B, seeks generalized eigenvalues λ and eigenvectors v such that

$$Av = \lambda Bv$$

This can be solved in SciPy via `scipy.linalg.eig(A, B)`.

Of course, if B is square and invertible, then we can treat the generalized eigenvalue problem as an ordinary eigenvalue problem $B^{-1}Av = \lambda v$, but this is not always the case.

2.6 Further Topics

We round out our discussion by briefly mentioning several other important topics.

2.6.1 Series Expansions

Recall the usual summation formula for a geometric progression, which states that if $|a| < 1$, then $\sum_{k=0}^{\infty} a^k = (1-a)^{-1}$.

A generalization of this idea exists in the matrix setting.

Matrix Norms

Let A be a square matrix, and let

$$\|A\| := \max_{\|x\|=1} \|Ax\|$$

The norms on the right-hand side are ordinary vector norms, while the norm on the left-hand side is a **matrix norm** — in this case, the so-called **spectral norm**.

For example, for a square matrix S, the condition $\|S\| < 1$ means that S is **contractive**, in the sense that it pulls all vectors towards the origin[2].

[2] Suppose that $\|S\| < 1$. Take any nonzero vector x, and let $r := \|x\|$. We have $\|Sx\| = r\|S(x/r)\| \leq r\|S\| < r = \|x\|$. Hence every point is pulled towards the origin.

Neumann's Theorem

Let A be a square matrix and let $A^k := AA^{k-1}$ with $A^1 := A$.

In other words, A^k is the k-th power of A.

Neumann's theorem states the following: If $\|A^k\| < 1$ for some $k \in \mathbb{N}$, then $I - A$ is invertible, and

$$(I - A)^{-1} = \sum_{k=0}^{\infty} A^k \tag{2.4}$$

Spectral Radius

A result known as Gelfand's formula tells us that, for any square matrix A,

$$\rho(A) = \lim_{k \to \infty} \|A^k\|^{1/k}$$

Here $\rho(A)$ is the **spectral radius**, defined as $\max_i |\lambda_i|$, where $\{\lambda_i\}_i$ is the set of eigenvalues of A.

As a consequence of Gelfand's formula, if all eigenvalues are strictly less than one in modulus, there exists a k with $\|A^k\| < 1$.

In which case (2.4) is valid.

2.6.2 Positive Definite Matrices

Let A be a symmetric $n \times n$ matrix.

We say that A is

1. **positive definite** if $x' A x > 0$ for every $x \in \mathbb{R}^n \setminus \{0\}$
2. **positive semi-definite** or **nonnegative definite** if $x' A x \geq 0$ for every $x \in \mathbb{R}^n$

Analogous definitions exist for negative definite and negative semi-definite matrices.

It is notable that if A is positive definite, then all of its eigenvalues are strictly positive, and hence A is invertible (with positive definite inverse).

2.6.3 Differentiating Linear and Quadratic Forms

The following formulas are useful in many economic contexts. Let

- z, x and a all be $n \times 1$ vectors
- A be an $n \times n$ matrix
- B be an $m \times n$ matrix and y be an $m \times 1$ vector

Then

1. $\frac{\partial a' x}{\partial x} = a$
2. $\frac{\partial A x}{\partial x} = A'$
3. $\frac{\partial x' A x}{\partial x} = (A + A') x$
4. $\frac{\partial y' B z}{\partial y} = B z$

5. $\frac{\partial y' B z}{\partial B} = y z'$

Exercise 2.7.1 below asks you to apply these formulas.

2.6.4 Further Reading

The documentation of the `scipy.linalg` submodule can be found here.

Chapters 2 and 3 of the Econometric Theory contains a discussion of linear algebra along the same lines as above, with solved exercises.

If you don't mind a slightly abstract approach, a nice intermediate-level text on linear algebra is [Jänich, 1994].

2.7 Exercises

Exercise 2.7.1

Let x be a given $n \times 1$ vector and consider the problem

$$v(x) = \max_{y,u} \{-y'Py - u'Qu\}$$

subject to the linear constraint

$$y = Ax + Bu$$

Here

- P is an $n \times n$ matrix and Q is an $m \times m$ matrix
- A is an $n \times n$ matrix and B is an $n \times m$ matrix
- both P and Q are symmetric and positive semidefinite

(What must the dimensions of y and u be to make this a well-posed problem?)

One way to solve the problem is to form the Lagrangian

$$\mathcal{L} = -y'Py - u'Qu + \lambda' [Ax + Bu - y]$$

where λ is an $n \times 1$ vector of Lagrange multipliers.

Try applying the formulas given above for differentiating quadratic and linear forms to obtain the first-order conditions for maximizing $\mathcal{L}$ with respect to y, u and minimizing it with respect to λ.

Show that these conditions imply that

1. $\lambda = -2Py$.
2. The optimizing choice of u satisfies $u = -(Q + B'PB)^{-1}B'PAx$.
3. The function v satisfies $v(x) = -x'\tilde{P}x$ where $\tilde{P} = A'PA - A'PB(Q + B'PB)^{-1}B'PA$.

As we will see, in economic contexts Lagrange multipliers often are shadow prices.

Note

If we don't care about the Lagrange multipliers, we can substitute the constraint into the objective function, and then just maximize $-(Ax + Bu)'P(Ax + Bu) - u'Qu$ with respect to u. You can verify that this leads to the same maximizer.

Solution

We have an optimization problem:

$$v(x) = \max_{y,u}\{-y'Py - u'Qu\}$$

s.t.

$$y = Ax + Bu$$

with primitives

- P be a symmetric and positive semidefinite $n \times n$ matrix
- Q be a symmetric and positive semidefinite $m \times m$ matrix

- A an $n \times n$ matrix
- B an $n \times m$ matrix

The associated Lagrangian is:

$$L = -y'Py - u'Qu + \lambda'[Ax + Bu - y]$$

Step 1.

Differentiating Lagrangian equation w.r.t y and setting its derivative equal to zero yields

$$\frac{\partial L}{\partial y} = -(P + P')y - \lambda = -2Py - \lambda = 0 \; ,$$

since P is symmetric.

Accordingly, the first-order condition for maximizing L w.r.t. y implies

$$\lambda = -2Py$$

Step 2.

Differentiating Lagrangian equation w.r.t. u and setting its derivative equal to zero yields

$$\frac{\partial L}{\partial u} = -(Q + Q')u - B'\lambda = -2Qu + B'\lambda = 0$$

Substituting $\lambda = -2Py$ gives

$$Qu + B'Py = 0$$

Substituting the linear constraint $y = Ax + Bu$ into above equation gives

$$Qu + B'P(Ax + Bu) = 0$$

$$(Q + B'PB)u + B'PAx = 0$$

which is the first-order condition for maximizing L w.r.t. u.

Thus, the optimal choice of u must satisfy

$$u = -(Q + B'PB)^{-1}B'PAx \; ,$$

which follows from the definition of the first-order conditions for Lagrangian equation.

Step 3.

Rewriting our problem by substituting the constraint into the objective function, we get

$$v(x) = \max_u \{-(Ax + Bu)'P(Ax + Bu) - u'Qu\}$$

Since we know the optimal choice of u satisfies $u = -(Q + B'PB)^{-1}B'PAx$, then

$$v(x) = -(Ax + Bu)'P(Ax + Bu) - u'Qu \quad with \quad u = -(Q + B'PB)^{-1}B'PAx$$

To evaluate the function

$$\begin{aligned}
v(x) &= -(Ax + Bu)'P(Ax + Bu) - u'Qu \\
&= -(x'A' + u'B')P(Ax + Bu) - u'Qu \\
&= -x'A'PAx - u'B'PAx - x'A'PBu - u'B'PBu - u'Qu \\
&= -x'A'PAx - 2u'B'PAx - u'(Q + B'PB)u
\end{aligned}$$

For simplicity, denote by $S := (Q + B'PB)^{-1}B'PA$, then $u = -Sx$.

Regarding the second term $-2u'B'PAx$,

$$
\begin{aligned}
-2u'B'PAx &= -2x'S'B'PAx \\
&= 2x'A'PB(Q + B'PB)^{-1}B'PAx
\end{aligned}
$$

Notice that the term $(Q + B'PB)^{-1}$ is symmetric as both P and Q are symmetric.

Regarding the third term $-u'(Q + B'PB)u$,

$$
\begin{aligned}
-u'(Q + B'PB)u &= -x'S'(Q + B'PB)Sx \\
&= -x'A'PB(Q + B'PB)^{-1}B'PAx
\end{aligned}
$$

Hence, the summation of second and third terms is $x'A'PB(Q + B'PB)^{-1}B'PAx$.

This implies that

$$
\begin{aligned}
v(x) &= -x'A'PAx - 2u'B'PAx - u'(Q + B'PB)u \\
&= -x'A'PAx + x'A'PB(Q + B'PB)^{-1}B'PAx \\
&= -x'[A'PA - A'PB(Q + B'PB)^{-1}B'PA]x
\end{aligned}
$$

Therefore, the solution to the optimization problem $v(x) = -x'\tilde{P}x$ follows the above result by denoting $\tilde{P} := A'PA - A'PB(Q + B'PB)^{-1}B'PA$

CHAPTER

THREE

QR DECOMPOSITION

3.1 Overview

This lecture describes the QR decomposition and how it relates to

- Orthogonal projection and least squares
- A Gram-Schmidt process
- Eigenvalues and eigenvectors

We'll write some Python code to help consolidate our understandings.

3.2 Matrix Factorization

The QR decomposition (also called the QR factorization) of a matrix is a decomposition of a matrix into the product of an orthogonal matrix and a triangular matrix.

A QR decomposition of a real matrix A takes the form

$$A = QR$$

where

- Q is an orthogonal matrix (so that $Q^T Q = I$)
- R is an upper triangular matrix

We'll use a **Gram-Schmidt process** to compute a QR decomposition

Because doing so is so educational, we'll write our own Python code to do the job

3.3 Gram-Schmidt process

We'll start with a **square** matrix A.

If a square matrix A is nonsingular, then a QR factorization is unique.

We'll deal with a rectangular matrix A later.

Actually, our algorithm will work with a rectangular A that is not square.

3.3.1 Gram-Schmidt process for square A

Here we apply a Gram-Schmidt process to the **columns** of matrix A.

In particular, let

$$A = \left[\begin{array}{c|c|c|c} a_1 & a_2 & \cdots & a_n \end{array} \right]$$

Let $||\cdot||$ denote the L2 norm.

The Gram-Schmidt algorithm repeatedly combines the following two steps in a particular order

- **normalize** a vector to have unit norm
- **orthogonalize** the next vector

To begin, we set $u_1 = a_1$ and then **normalize**:

$$u_1 = a_1, \quad e_1 = \frac{u_1}{||u_1||}$$

We **orthogonalize** first to compute u_2 and then **normalize** to create e_2:

$$u_2 = a_2 - (a_2 \cdot e_1)e_1, \quad e_2 = \frac{u_2}{||u_2||}$$

We invite the reader to verify that e_1 is orthogonal to e_2 by checking that $e_1 \cdot e_2 = 0$.

The Gram-Schmidt procedure continues iterating.

Thus, for $k = 2, \ldots, n-1$ we construct

$$u_{k+1} = a_{k+1} - (a_{k+1} \cdot e_1)e_1 - \cdots - (a_{k+1} \cdot e_k)e_k, \quad e_{k+1} = \frac{u_{k+1}}{||u_{k+1}||}$$

Here $(a_j \cdot e_i)$ can be interpreted as the linear least squares **regression coefficient** of a_j on e_i

- it is the inner product of a_j and e_i divided by the inner product of e_i where $e_i \cdot e_i = 1$, as *normalization* has assured us.
- this regression coefficient has an interpretation as being a **covariance** divided by a **variance**

It can be verified that

$$A = \left[\begin{array}{c|c|c|c} a_1 & a_2 & \cdots & a_n \end{array} \right] = \left[\begin{array}{c|c|c|c} e_1 & e_2 & \cdots & e_n \end{array} \right] \begin{bmatrix} a_1 \cdot e_1 & a_2 \cdot e_1 & \cdots & a_n \cdot e_1 \\ 0 & a_2 \cdot e_2 & \cdots & a_n \cdot e_2 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_n \cdot e_n \end{bmatrix}$$

Thus, we have constructed the decomposision

$$A = QR$$

where

$$Q = \left[\begin{array}{c|c|c|c} a_1 & a_2 & \cdots & a_n \end{array} \right] = \left[\begin{array}{c|c|c|c} e_1 & e_2 & \cdots & e_n \end{array} \right]$$

and

$$R = \begin{bmatrix} a_1 \cdot e_1 & a_2 \cdot e_1 & \cdots & a_n \cdot e_1 \\ 0 & a_2 \cdot e_2 & \cdots & a_n \cdot e_2 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_n \cdot e_n \end{bmatrix}$$

3.3.2 A not square

Now suppose that A is an $n \times m$ matrix where $m > n$.

Then a QR decomposition is

$$A = \left[\, a_1 \mid a_2 \mid \cdots \mid a_m \,\right] = \left[\, e_1 \mid e_2 \mid \cdots \mid e_n \,\right] \begin{bmatrix} a_1 \cdot e_1 & a_2 \cdot e_1 & \cdots & a_n \cdot e_1 & a_{n+1} \cdot e_1 & \cdots & a_m \cdot e_1 \\ 0 & a_2 \cdot e_2 & \cdots & a_n \cdot e_2 & a_{n+1} \cdot e_2 & \cdots & a_m \cdot e_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_n \cdot e_n & a_{n+1} \cdot e_n & \cdots & a_m \cdot e_n \end{bmatrix}$$

which implies that

$$\begin{aligned} a_1 &= (a_1 \cdot e_1) e_1 \\ a_2 &= (a_2 \cdot e_1) e_1 + (a_2 \cdot e_2) e_2 \\ \vdots &\quad \vdots \\ a_n &= (a_n \cdot e_1) e_1 + (a_n \cdot e_2) e_2 + \cdots + (a_n \cdot e_n) e_n \\ a_{n+1} &= (a_{n+1} \cdot e_1) e_1 + (a_{n+1} \cdot e_2) e_2 + \cdots + (a_{n+1} \cdot e_n) e_n \\ \vdots &\quad \vdots \\ a_m &= (a_m \cdot e_1) e_1 + (a_m \cdot e_2) e_2 + \cdots + (a_m \cdot e_n) e_n \end{aligned}$$

3.4 Some Code

Now let's write some homemade Python code to implement a QR decomposition by deploying the Gram-Schmidt process described above.

```python
import numpy as np
from scipy.linalg import qr
```

```python
def QR_Decomposition(A):
    n, m = A.shape # get the shape of A

    Q = np.empty((n, n)) # initialize matrix Q
    u = np.empty((n, n)) # initialize matrix u

    u[:, 0] = A[:, 0]
    Q[:, 0] = u[:, 0] / np.linalg.norm(u[:, 0])

    for i in range(1, n):

        u[:, i] = A[:, i]
        for j in range(i):
            u[:, i] -= (A[:, i] @ Q[:, j]) * Q[:, j] # get each u vector

        Q[:, i] = u[:, i] / np.linalg.norm(u[:, i]) # compute each e vetor

    R = np.zeros((n, m))
    for i in range(n):
        for j in range(i, m):
            R[i, j] = A[:, j] @ Q[:, i]

    return Q, R
```

The preceding code is fine but can benefit from some further housekeeping.

We want to do this because later in this notebook we want to compare results from using our homemade code above with the code for a QR that the Python `scipy` package delivers.

There can be be sign differences between the Q and R matrices produced by different numerical algorithms.

All of these are valid QR decompositions because of how the sign differences cancel out when we compute QR.

However, to make the results from our homemade function and the QR module in `scipy` comparable, let's require that Q have positive diagonal entries.

We do this by adjusting the signs of the columns in Q and the rows in R appropriately.

To accomplish this we'll define a pair of functions.

```
def diag_sign(A):
    "Compute the signs of the diagonal of matrix A"

    D = np.diag(np.sign(np.diag(A)))

    return D

def adjust_sign(Q, R):
    """
    Adjust the signs of the columns in Q and rows in R to
    impose positive diagonal of Q
    """

    D = diag_sign(Q)

    Q[:, :] = Q @ D
    R[:, :] = D @ R

    return Q, R
```

3.5 Example

Now let's do an example.

```
A = np.array([[1.0, 1.0, 0.0], [1.0, 0.0, 1.0], [0.0, 1.0, 1.0]])
# A = np.array([[1.0, 0.5, 0.2], [0.5, 0.5, 1.0], [0.0, 1.0, 1.0]])
# A = np.array([[1.0, 0.5, 0.2], [0.5, 0.5, 1.0]])

A
```

```
array([[1., 1., 0.],
       [1., 0., 1.],
       [0., 1., 1.]])
```

```
Q, R = adjust_sign(*QR_Decomposition(A))
```

```
Q
```

```
array([[ 0.70710678, -0.40824829, -0.57735027],
       [ 0.70710678,  0.40824829,  0.57735027],
       [ 0.        , -0.81649658,  0.57735027]])
```

```
R
```

```
array([[ 1.41421356,  0.70710678,  0.70710678],
       [ 0.        , -1.22474487, -0.40824829],
       [ 0.        ,  0.        ,  1.15470054]])
```

Let's compare outcomes with what the `scipy` package produces

```
Q_scipy, R_scipy = adjust_sign(*qr(A))
```

```
print('Our Q: \n', Q)
print('\n')
print('Scipy Q: \n', Q_scipy)
```

```
Our Q: 
 [[ 0.70710678 -0.40824829 -0.57735027]
 [ 0.70710678  0.40824829  0.57735027]
 [ 0.         -0.81649658  0.57735027]]


Scipy Q: 
 [[ 0.70710678 -0.40824829 -0.57735027]
 [ 0.70710678  0.40824829  0.57735027]
 [ 0.         -0.81649658  0.57735027]]
```

```
print('Our R: \n', R)
print('\n')
print('Scipy R: \n', R_scipy)
```

```
Our R: 
 [[ 1.41421356  0.70710678  0.70710678]
 [ 0.         -1.22474487 -0.40824829]
 [ 0.          0.          1.15470054]]


Scipy R: 
 [[ 1.41421356  0.70710678  0.70710678]
 [ 0.         -1.22474487 -0.40824829]
 [ 0.          0.          1.15470054]]
```

The above outcomes give us the good news that our homemade function agrees with what scipy produces.

Now let's do a QR decomposition for a rectangular matrix A that is $n \times m$ with $m > n$.

```
A = np.array([[1, 3, 4], [2, 0, 9]])
```

```
Q, R = adjust_sign(*QR_Decomposition(A))
Q, R
```

```
(array([[ 0.4472136 , -0.89442719],
        [ 0.89442719,  0.4472136 ]]),
```

(continues on next page)

(continued from previous page)

```
array([[ 2.23606798,  1.34164079,  9.8386991 ],
       [ 0.        , -2.68328157,  0.4472136 ]]))
```

```
Q_scipy, R_scipy = adjust_sign(*qr(A))
Q_scipy, R_scipy
```

```
(array([[ 0.4472136 , -0.89442719],
        [ 0.89442719,  0.4472136 ]]),
 array([[ 2.23606798,  1.34164079,  9.8386991 ],
        [ 0.        , -2.68328157,  0.4472136 ]]))
```

3.6 Using QR Decomposition to Compute Eigenvalues

Now for a useful fact about the QR algorithm.

The following iterations on the QR decomposition can be used to compute **eigenvalues** of a **square** matrix A.

Here is the algorithm:

1. Set $A_0 = A$ and form $A_0 = Q_0 R_0$
2. Form $A_1 = R_0 Q_0$. Note that A_1 is similar to A_0 (easy to verify) and so has the same eigenvalues.
3. Form $A_1 = Q_1 R_1$ (i.e., form the QR decomposition of A_1).
4. Form $A_2 = R_1 Q_1$ and then $A_2 = Q_2 R_2$.
5. Iterate to convergence.
6. Compute eigenvalues of A and compare them to the diagonal values of the limiting A_n found from this process.

Remark: this algorithm is close to one of the most efficient ways of computing eigenvalues!

Let's write some Python code to try out the algorithm

```
def QR_eigvals(A, tol=1e-12, maxiter=1000):
    "Find the eigenvalues of A using QR decomposition."

    A_old = np.copy(A)
    A_new = np.copy(A)

    diff = np.inf
    i = 0
    while (diff > tol) and (i < maxiter):
        A_old[:, :] = A_new
        Q, R = QR_Decomposition(A_old)

        A_new[:, :] = R @ Q

        diff = np.abs(A_new - A_old).max()
        i += 1

    eigvals = np.diag(A_new)

    return eigvals
```

Now let's try the code and compare the results with what `scipy.linalg.eigvals` gives us

Here goes

```
# experiment this with one random A matrix
A = np.random.random((3, 3))
```

```
sorted(QR_eigvals(A))
```

```
[np.float64(-0.20148191414068659),
 np.float64(0.023777676110310512),
 np.float64(1.2101216897907014)]
```

Compare with the `scipy` package.

```
sorted(np.linalg.eigvals(A))
```

```
[np.complex128(-0.08885211901518819-0.26907577121214177j),
 np.complex128(-0.08885211901518819+0.26907577121214177j),
 np.complex128(1.210121689790699+0j)]
```

3.7 QR and PCA

There are interesting connections between the QR decomposition and principal components analysis (PCA).

Here are some.

1. Let X' be a $k \times n$ random matrix where the jth column is a random draw from $\mathcal{N}(\mu, \Sigma)$ where μ is $k \times 1$ vector of means and Σ is a $k \times k$ covariance matrix. We want $n >> k$ – this is an "econometrics example".
2. Form $X' = QR$ where Q is $k \times k$ and R is $k \times n$.
3. Form the eigenvalues of RR', i.e., we'll compute $RR' = \tilde{P}\Lambda\tilde{P}'$.
4. Form $X'X = Q\tilde{P}\Lambda\tilde{P}'Q'$ and compare it with the eigen decomposition $X'X = P\hat{\Lambda}P'$.
5. It will turn out that that $\Lambda = \hat{\Lambda}$ and that $P = Q\tilde{P}$.

Let's verify conjecture 5 with some Python code.

Start by simulating a random (n, k) matrix X.

```
k = 5
n = 1000

# generate some random moments
μ = np.random.random(size=k)
C = np.random.random((k, k))
Σ = C.T @ C
```

```
# X is random matrix where each column follows multivariate normal dist.
X = np.random.multivariate_normal(μ, Σ, size=n)
```

```
X.shape
```

```
(1000, 5)
```

Let's apply the QR decomposition to X'.

```
Q, R = adjust_sign(*QR_Decomposition(X.T))
```

Check the shapes of Q and R.

```
Q.shape, R.shape
```

```
((5, 5), (5, 1000))
```

Now we can construct $RR' = \tilde{P}\Lambda\tilde{P}'$ and form an eigen decomposition.

```
RR = R @ R.T

λ, P_tilde = np.linalg.eigh(RR)
Λ = np.diag(λ)
```

We can also apply the decomposition to $X'X = P\hat{\Lambda}P'$.

```
XX = X.T @ X

λ_hat, P = np.linalg.eigh(XX)
Λ_hat = np.diag(λ_hat)
```

Compare the eigenvalues that are on the diagonals of Λ and $\hat{\Lambda}$.

```
λ, λ_hat
```

```
(array([  49.15610839,  160.86464237,  326.57838575,  761.40655371,
        6480.46106859]),
 array([  49.15610839,  160.86464237,  326.57838575,  761.40655371,
        6480.46106859]))
```

Let's compare P and $Q\tilde{P}$.

Again we need to be careful about sign differences between the columns of P and $Q\tilde{P}$.

```
QP_tilde = Q @ P_tilde

np.abs(P @ diag_sign(P) - QP_tilde @ diag_sign(QP_tilde)).max()
```

```
np.float64(3.352873534367973e-14)
```

Let's verify that $X'X$ can be decomposed as $Q\tilde{P}\Lambda\tilde{P}'Q'$.

```
QPΛPQ = Q @ P_tilde @ Λ @ P_tilde.T @ Q.T
```

```
np.abs(QPΛPQ - XX).max()
```

```
np.float64(2.000888343900442e-11)
```

CHAPTER

FOUR

CIRCULANT MATRICES

4.1 Overview

This lecture describes circulant matrices and some of their properties.

Circulant matrices have a special structure that connects them to useful concepts including

- convolution
- Fourier transforms
- permutation matrices

Because of these connections, circulant matrices are widely used in machine learning, for example, in image processing.

We begin by importing some Python packages

```
import numpy as np
from numba import jit
import matplotlib.pyplot as plt
```

```
np.set_printoptions(precision=3, suppress=True)
```

4.2 Constructing a Circulant Matrix

To construct an $N \times N$ circulant matrix, we need only the first row, say,

$$\begin{bmatrix} c_0 & c_1 & c_2 & c_3 & c_4 & \cdots & c_{N-1} \end{bmatrix}.$$

After setting entries in the first row, the remaining rows of a circulant matrix are determined as follows:

$$C = \begin{bmatrix} c_0 & c_1 & c_2 & c_3 & c_4 & \cdots & c_{N-1} \\ c_{N-1} & c_0 & c_1 & c_2 & c_3 & \cdots & c_{N-2} \\ c_{N-2} & c_{N-1} & c_0 & c_1 & c_2 & \cdots & c_{N-3} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ c_3 & c_4 & c_5 & c_6 & c_7 & \cdots & c_2 \\ c_2 & c_3 & c_4 & c_5 & c_6 & \cdots & c_1 \\ c_1 & c_2 & c_3 & c_4 & c_5 & \cdots & c_0 \end{bmatrix} \tag{4.1}$$

It is also possible to construct a circulant matrix by creating the transpose of the above matrix, in which case only the first column needs to be specified.

Let's write some Python code to generate a circulant matrix.

```python
@jit
def construct_cirlulant(row):

    N = row.size

    C = np.empty((N, N))

    for i in range(N):

        C[i, i:] = row[:N-i]
        C[i, :i] = row[N-i:]

    return C
```

```python
# a simple case when N = 3
construct_cirlulant(np.array([1., 2., 3.]))
```

```
array([[1., 2., 3.],
       [3., 1., 2.],
       [2., 3., 1.]])
```

4.2.1 Some Properties of Circulant Matrices

Here are some useful properties:

Suppose that A and B are both circulant matrices. Then it can be verified that

- The transpose of a circulant matrix is a circulant matrix.
- $A + B$ is a circulant matrix
- AB is a circulant matrix
- $AB = BA$

Now consider a circulant matrix with first row

$$c = \begin{bmatrix} c_0 & c_1 & \cdots & c_{N-1} \end{bmatrix}$$

and consider a vector

$$a = \begin{bmatrix} a_0 & a_1 & \cdots & a_{N-1} \end{bmatrix}$$

The **convolution** of vectors c and a is defined as the vector $b = c * a$ with components

$$b_k = \sum_{i=0}^{n-1} c_{k-i} a_i \tag{4.2}$$

We use $*$ to denote **convolution** via the calculation described in equation (4.2).

It can be verified that the vector b satisfies

$$b = C^T a$$

where C^T is the transpose of the circulant matrix defined in equation (4.1).

4.3 Connection to Permutation Matrix

A good way to construct a circulant matrix is to use a **permutation matrix**.

Before defining a permutation **matrix**, we'll define a **permutation**.

A **permutation** of a set of the set of non-negative integers $\{0, 1, 2, ...\}$ is a one-to-one mapping of the set into itself.

A permutation of a set $\{1, 2, \ldots, n\}$ rearranges the n integers in the set.

A permutation matrix is obtained by permuting the rows of an $n \times n$ identity matrix according to a permutation of the numbers 1 to n.

Thus, every row and every column contain precisely a single 1 with 0 everywhere else.

Every permutation corresponds to a unique permutation matrix.

For example, the $N \times N$ matrix

$$P = \begin{bmatrix} 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 \\ 1 & 0 & 0 & 0 & \cdots & 0 \end{bmatrix} \tag{4.3}$$

serves as a **cyclic shift** operator that, when applied to an $N \times 1$ vector h, shifts entries in rows 2 through N up one row and shifts the entry in row 1 to row N.

Eigenvalues of the cyclic shift permutation matrix P defined in equation (4.3) can be computed by constructing

$$P - \lambda I = \begin{bmatrix} -\lambda & 1 & 0 & 0 & \cdots & 0 \\ 0 & -\lambda & 1 & 0 & \cdots & 0 \\ 0 & 0 & -\lambda & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 \\ 1 & 0 & 0 & 0 & \cdots & -\lambda \end{bmatrix}$$

and solving

$$\det(P - \lambda I) = (-1)^N \lambda^N - 1 = 0$$

Eigenvalues λ_i can be complex.

Magnitudes $\mid \lambda_i \mid$ of these eigenvalues λ_i all equal 1.

Thus, **singular values** of the permutation matrix P defined in equation (4.3) all equal 1.

It can be verified that permutation matrices are orthogonal matrices:

$$PP' = I$$

4.4 Examples with Python

Let's write some Python code to illustrate these ideas.

```
@jit
def construct_P(N):

    P = np.zeros((N, N))

    for i in range(N-1):
        P[i, i+1] = 1
    P[-1, 0] = 1

    return P
```

```
P4 = construct_P(4)
P4
```

```
array([[0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.],
       [1., 0., 0., 0.]])
```

```
# compute the eigenvalues and eigenvectors
λ, Q = np.linalg.eig(P4)
```

```
for i in range(4):
    print(f'λ{i} = {λ[i]:.1f} \nvec{i} = {Q[i, :]}\n')
```

```
λ0 = -1.0+0.0j
vec0 = [-0.5+0.j  -0. -0.5j -0. +0.5j -0.5+0.j ]

λ1 = 0.0+1.0j
vec1 = [ 0.5+0.j   0.5+0.j   0.5-0.j  -0.5+0.j]

λ2 = 0.0-1.0j
vec2 = [-0.5+0.j    0. +0.5j  0. -0.5j -0.5+0.j ]

λ3 = 1.0+0.0j
vec3 = [ 0.5+0.j -0.5-0.j -0.5+0.j -0.5+0.j]
```

In graphs below, we shall portray eigenvalues of a shift permutation matrix in the complex plane.

These eigenvalues are uniformly distributed along the unit circle.

They are the n **roots of unity**, meaning they are the n numbers z that solve $z^n = 1$, where z is a complex number.

In particular, the n roots of unity are

$$z = \exp\left(\frac{2\pi jk}{N}\right), \quad k = 0, \dots, N-1$$

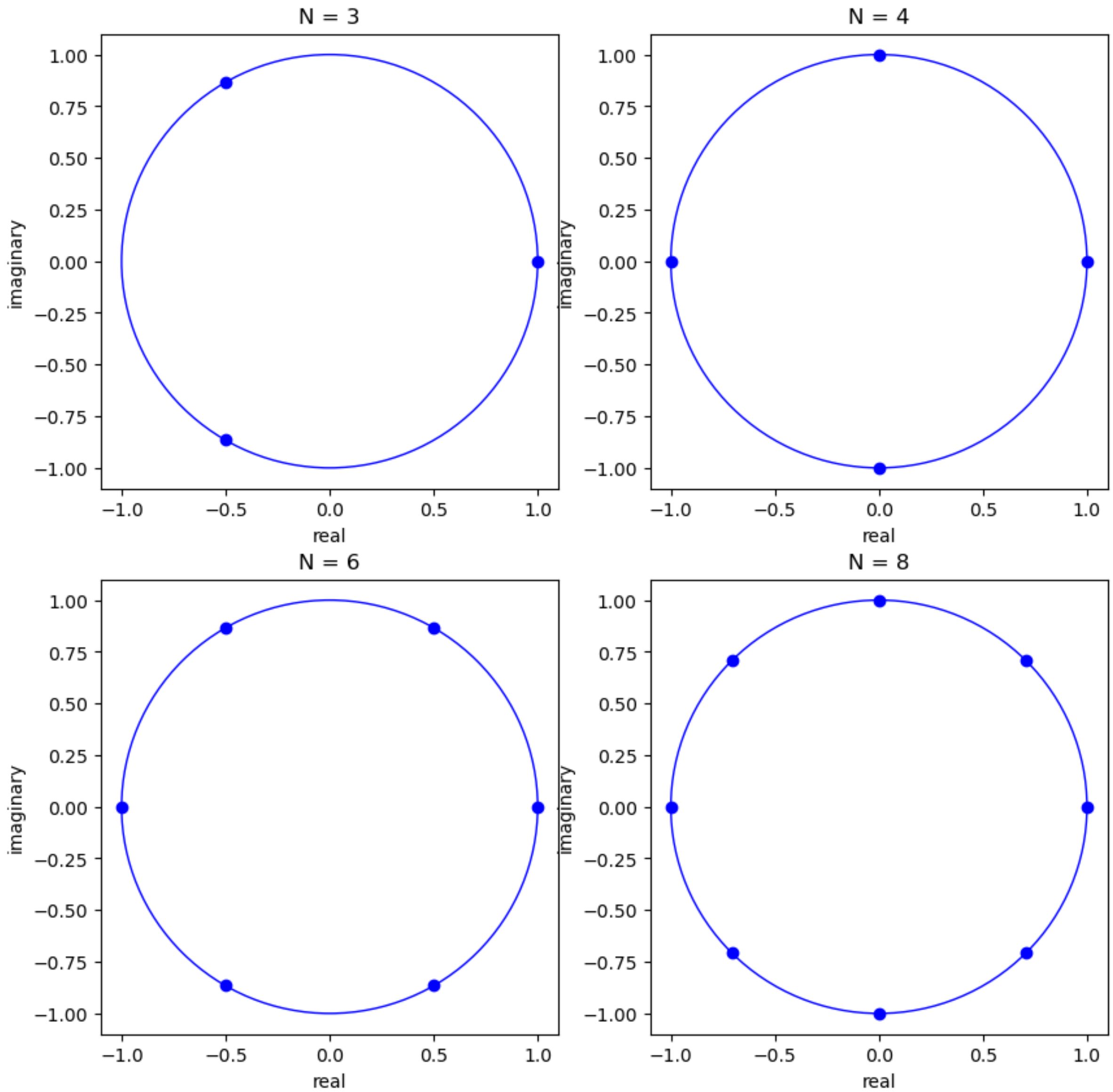
where j denotes the purely imaginary unit number.

```
fig, ax = plt.subplots(2, 2, figsize=(10, 10))
```

(continues on next page)

(continued from previous page)

```
for i, N in enumerate([3, 4, 6, 8]):

    row_i = i // 2
    col_i = i % 2

    P = construct_P(N)
    λ, Q = np.linalg.eig(P)

    circ = plt.Circle((0, 0), radius=1, edgecolor='b', facecolor='None')
    ax[row_i, col_i].add_patch(circ)

    for j in range(N):
        ax[row_i, col_i].scatter(λ[j].real, λ[j].imag, c='b')

    ax[row_i, col_i].set_title(f'N = {N}')
    ax[row_i, col_i].set_xlabel('real')
    ax[row_i, col_i].set_ylabel('imaginary')

plt.show()
```



For a vector of coefficients $\{c_i\}_{i=0}^{n-1}$, eigenvectors of P are also eigenvectors of

$$C = c_0 I + c_1 P + c_2 P^2 + \cdots + c_{N-1} P^{N-1}.$$

Consider an example in which $N = 8$ and let $w = e^{-2\pi j/N}$.

It can be verified that the matrix F_8 of eigenvectors of P_8 is

$$F_8 = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & w & w^2 & \cdots & w^7 \\ 1 & w^2 & w^4 & \cdots & w^{14} \\ 1 & w^3 & w^6 & \cdots & w^{21} \\ 1 & w^4 & w^8 & \cdots & w^{28} \\ 1 & w^5 & w^{10} & \cdots & w^{35} \\ 1 & w^6 & w^{12} & \cdots & w^{42} \\ 1 & w^7 & w^{14} & \cdots & w^{49} \end{bmatrix}$$

The matrix F_8 defines a Discete Fourier Transform.

To convert it into an orthogonal eigenvector matrix, we can simply normalize it by dividing every entry by $\sqrt{8}$.

- stare at the first column of F_8 above to convince yourself of this fact

The eigenvalues corresponding to each eigenvector are $\{w^j\}_{j=0}^7$ in order.

```
def construct_F(N):

    w = np.e ** (-complex(0, 2*np.pi/N))

    F = np.ones((N, N), dtype=complex)
    for i in range(1, N):
        F[i, 1:] = w ** (i * np.arange(1, N))

    return F, w
```

```
F8, w = construct_F(8)
```

```
w
```

```
(0.7071067811865476-0.7071067811865475j)
```

```
F8
```

```
array([[ 1.   +0.j   ,  1.   +0.j   ,  1.   +0.j   ,  1.   +0.j   ,
         1.   +0.j   ,  1.   +0.j   ,  1.   +0.j   ,  1.   +0.j   ],
       [ 1.   +0.j   ,  0.707-0.707j,  0.   -1.j   , -0.707-0.707j,
        -1.   -0.j   , -0.707+0.707j, -0.   +1.j   ,  0.707+0.707j],
       [ 1.   +0.j   ,  0.   -1.j   , -1.   -0.j   , -0.   +1.j   ,
         1.   +0.j   ,  0.   -1.j   , -1.   -0.j   , -0.   +1.j   ],
       [ 1.   +0.j   , -0.707-0.707j, -0.   +1.j   ,  0.707-0.707j,
        -1.   -0.j   ,  0.707+0.707j,  0.   -1.j   , -0.707+0.707j],
       [ 1.   +0.j   , -1.   -0.j   ,  1.   +0.j   , -1.   -0.j   ,
         1.   +0.j   , -1.   -0.j   ,  1.   +0.j   , -1.   -0.j   ],
       [ 1.   +0.j   , -0.707+0.707j,  0.   -1.j   ,  0.707+0.707j,
        -1.   -0.j   ,  0.707-0.707j, -0.   +1.j   , -0.707-0.707j],
       [ 1.   +0.j   , -0.   +1.j   , -1.   -0.j   ,  0.   -1.j   ,
         1.   +0.j   , -0.   +1.j   , -1.   -0.j   ,  0.   -1.j   ],
       [ 1.   +0.j   ,  0.707+0.707j, -0.   +1.j   , -0.707+0.707j,
        -1.   -0.j   , -0.707-0.707j,  0.   -1.j   ,  0.707-0.707j]])
```

```
# normalize
Q8 = F8 / np.sqrt(8)
```

```
# verify the orthogonality (unitarity)
Q8 @ np.conjugate(Q8)
```

```
array([[ 1.+0.j, -0.+0.j, -0.+0.j, -0.+0.j, -0.+0.j,  0.+0.j,  0.+0.j,
         0.+0.j],
       [-0.-0.j,  1.+0.j, -0.+0.j, -0.+0.j, -0.+0.j, -0.+0.j,  0.+0.j,
         0.+0.j],
       [-0.-0.j, -0.-0.j,  1.+0.j, -0.+0.j, -0.+0.j, -0.+0.j,  0.+0.j,
         0.+0.j],
       [-0.-0.j, -0.-0.j, -0.-0.j,  1.+0.j, -0.+0.j, -0.+0.j, -0.+0.j,
        -0.+0.j],
       [-0.-0.j, -0.-0.j, -0.-0.j, -0.-0.j,  1.+0.j, -0.+0.j, -0.+0.j,
```

(continues on next page)

(continued from previous page)

```
        -0.+0.j],
       [ 0.-0.j, -0.-0.j, -0.-0.j, -0.-0.j, -0.-0.j,  1.+0.j, -0.+0.j,
        -0.+0.j],
       [ 0.-0.j,  0.-0.j,  0.-0.j, -0.-0.j, -0.-0.j, -0.-0.j,  1.+0.j,
        -0.+0.j],
       [ 0.-0.j,  0.-0.j,  0.-0.j, -0.-0.j, -0.-0.j, -0.-0.j, -0.-0.j,
         1.+0.j]])
```

Let's verify that kth column of Q_8 is an eigenvector of P_8 with an eigenvalue w^k.

```
P8 = construct_P(8)
```

```
diff_arr = np.empty(8, dtype=complex)
for j in range(8):
    diff = P8 @ Q8[:, j] - w ** j * Q8[:, j]
    diff_arr[j] = diff @ diff.T
```

```
diff_arr
```

```
array([ 0.+0.j, -0.+0.j, -0.+0.j, -0.+0.j, -0.+0.j, -0.+0.j, -0.+0.j,
       -0.+0.j])
```

4.5 Associated Permutation Matrix

Next, we execute calculations to verify that the circulant matrix C defined in equation (4.1) can be written as

$$C = c_0 I + c_1 P + \cdots + c_{n-1} P^{n-1}$$

and that every eigenvector of P is also an eigenvector of C.

We illustrate this for $N = 8$ case.

```
c = np.random.random(8)
```

```
c
```

```
array([0.87 , 0.275, 0.039, 0.506, 0.538, 0.953, 0.208, 0.082])
```

```
C8 = construct_cirlulant(c)
```

Compute $c_0 I + c_1 P + \cdots + c_{n-1} P^{n-1}$.

```
N = 8

C = np.zeros((N, N))
P = np.eye(N)

for i in range(N):
    C += c[i] * P
    P = P8 @ P
```

```
C
```

```
array([[0.87 , 0.275, 0.039, 0.506, 0.538, 0.953, 0.208, 0.082],
       [0.082, 0.87 , 0.275, 0.039, 0.506, 0.538, 0.953, 0.208],
       [0.208, 0.082, 0.87 , 0.275, 0.039, 0.506, 0.538, 0.953],
       [0.953, 0.208, 0.082, 0.87 , 0.275, 0.039, 0.506, 0.538],
       [0.538, 0.953, 0.208, 0.082, 0.87 , 0.275, 0.039, 0.506],
       [0.506, 0.538, 0.953, 0.208, 0.082, 0.87 , 0.275, 0.039],
       [0.039, 0.506, 0.538, 0.953, 0.208, 0.082, 0.87 , 0.275],
       [0.275, 0.039, 0.506, 0.538, 0.953, 0.208, 0.082, 0.87 ]])
```

```
C8
```

```
array([[0.87 , 0.275, 0.039, 0.506, 0.538, 0.953, 0.208, 0.082],
       [0.082, 0.87 , 0.275, 0.039, 0.506, 0.538, 0.953, 0.208],
       [0.208, 0.082, 0.87 , 0.275, 0.039, 0.506, 0.538, 0.953],
       [0.953, 0.208, 0.082, 0.87 , 0.275, 0.039, 0.506, 0.538],
       [0.538, 0.953, 0.208, 0.082, 0.87 , 0.275, 0.039, 0.506],
       [0.506, 0.538, 0.953, 0.208, 0.082, 0.87 , 0.275, 0.039],
       [0.039, 0.506, 0.538, 0.953, 0.208, 0.082, 0.87 , 0.275],
       [0.275, 0.039, 0.506, 0.538, 0.953, 0.208, 0.082, 0.87 ]])
```

Now let's compute the difference between two circulant matrices that we have constructed in two different ways.

```
np.abs(C - C8).max()
```

```
np.float64(0.0)
```

The kth column of P_8 associated with eigenvalue w^{k-1} is an eigenvector of C_8 associated with an eigenvalue $\sum_{h=0}^{7} c_j w^{hk}$.

```
λ_C8 = np.zeros(8, dtype=complex)

for j in range(8):
    for k in range(8):
        λ_C8[j] += c[k] * w ** (j * k)
```

```
λ_C8
```

```
array([ 3.47 +0.j   , -0.448+0.349j,  1.162-0.64j ,  1.112+0.011j,
       -0.161-0.j   ,  1.112-0.011j,  1.162+0.64j , -0.448-0.349j])
```

We can verify this by comparing `C8 @ Q8[:, j]` with `λ_C8[j] * Q8[:, j]`.

```
# verify
for j in range(8):
    diff = C8 @ Q8[:, j] - λ_C8[j] * Q8[:, j]
    print(diff)
```

```
[0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
[ 0.-0.j  0.-0.j  0.-0.j -0.-0.j -0.-0.j -0.-0.j -0.+0.j -0.+0.j]
[ 0.-0.j -0.-0.j -0.-0.j -0.-0.j -0.-0.j -0.+0.j  0.+0.j  0.-0.j]
[ 0.-0.j  0.+0.j -0.-0.j -0.-0.j -0.+0.j  0.-0.j -0.-0.j  0.+0.j]
[0.+0.j 0.-0.j 0.-0.j 0.-0.j 0.+0.j 0.-0.j 0.+0.j 0.-0.j]
```

(continues on next page)

(continued from previous page)

```
[ 0.+0.j -0.-0.j  0.-0.j  0.-0.j  0.+0.j -0.-0.j  0.-0.j -0.+0.j]
[ 0.-0.j  0.-0.j  0.-0.j  0.-0.j  0.-0.j  0.+0.j -0.+0.j -0.-0.j]
[-0.+0.j -0.-0.j  0.-0.j  0.-0.j  0.-0.j  0.-0.j  0.+0.j  0.+0.j]
```

4.6 Discrete Fourier Transform

The **Discrete Fourier Transform** (DFT) allows us to represent a discrete time sequence as a weighted sum of complex sinusoids.

Consider a sequence of N real number $\{x_j\}_{j=0}^{N-1}$.

The **Discrete Fourier Transform** maps $\{x_j\}_{j=0}^{N-1}$ into a sequence of complex numbers $\{X_k\}_{k=0}^{N-1}$

where

$$X_k = \sum_{n=0}^{N-1} x_n e^{-2\pi \frac{kn}{N} i}$$

```python
def DFT(x):
    "The discrete Fourier transform."

    N = len(x)
    w = np.e ** (-complex(0, 2*np.pi/N))

    X = np.zeros(N, dtype=complex)
    for k in range(N):
        for n in range(N):
            X[k] += x[n] * w ** (k * n)

    return X
```

Consider the following example.

$$x_n = \begin{cases} 1/2 & n = 0, 1 \\ 0 & \text{otherwise} \end{cases}$$

```python
x = np.zeros(10)
x[0:2] = 1/2
```

```python
x
```

```
array([0.5, 0.5, 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ])
```

Apply a discrete Fourier transform.

```python
X = DFT(x)
```

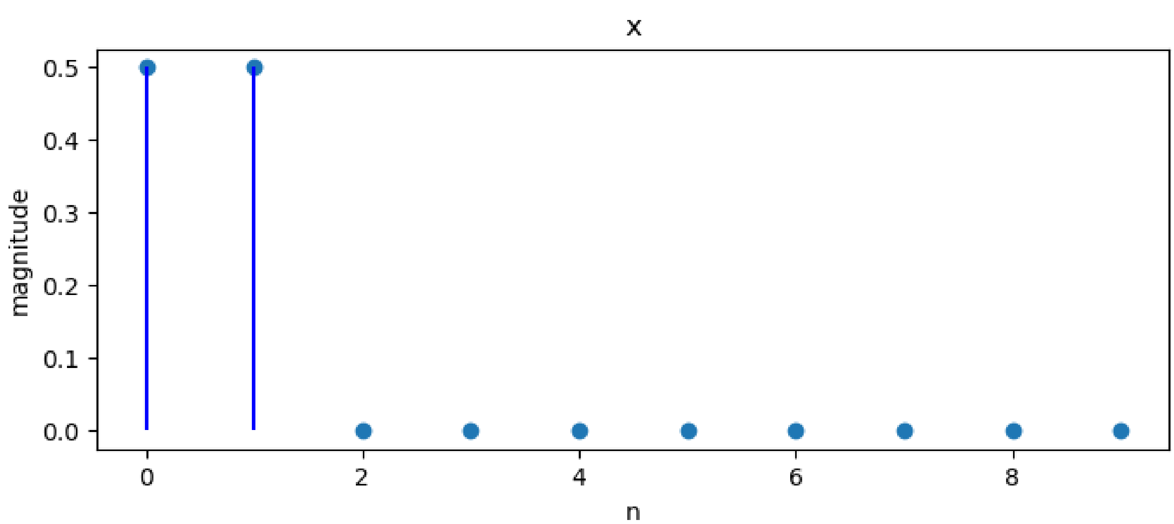
```python
X
```

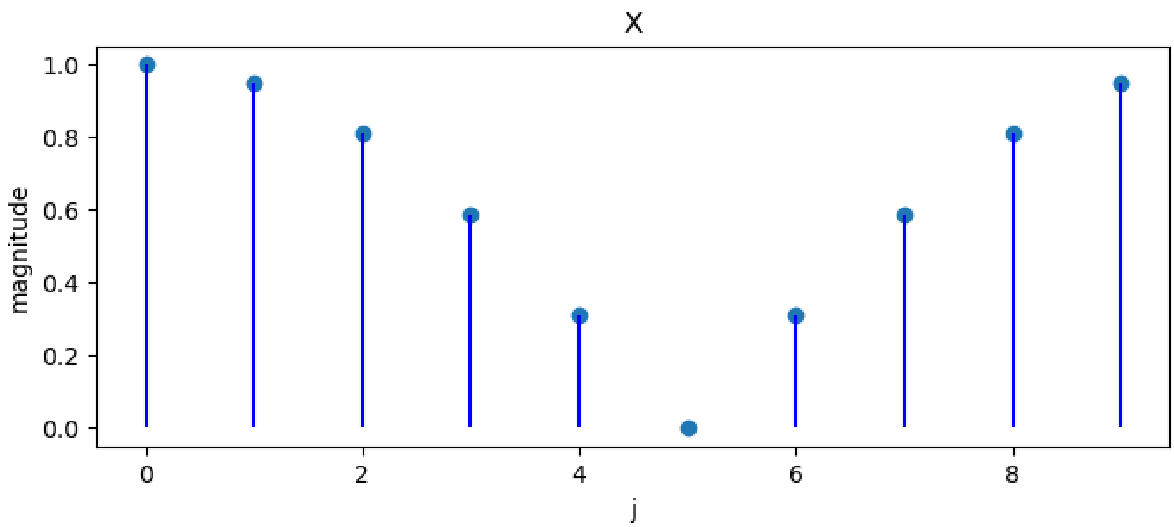
```
array([ 1.   +0.j   ,  0.905-0.294j,  0.655-0.476j,  0.345-0.476j,
        0.095-0.294j, -0.   +0.j   ,  0.095+0.294j,  0.345+0.476j,
        0.655+0.476j,  0.905+0.294j])
```

We can plot magnitudes of a sequence of numbers and the associated discrete Fourier transform.

```
def plot_magnitude(x=None, X=None):

    data = []
    names = []
    xs = []
    if (x is not None):
        data.append(x)
        names.append('x')
        xs.append('n')
    if (X is not None):
        data.append(X)
        names.append('X')
        xs.append('j')

    num = len(data)
    for i in range(num):
        n = data[i].size
        plt.figure(figsize=(8, 3))
        plt.scatter(range(n), np.abs(data[i]))
        plt.vlines(range(n), 0, np.abs(data[i]), color='b')

        plt.xlabel(xs[i])
        plt.ylabel('magnitude')
        plt.title(names[i])
        plt.show()
```

```
plot_magnitude(x=x, X=X)
```





The **inverse Fourier transform** transforms a Fourier transform X of x back to x.

The inverse Fourier transform is defined as

$$x_n = \sum_{k=0}^{N-1} \frac{1}{N} X_k e^{2\pi(\frac{kn}{N})i}, \quad n = 0, 1, \dots, N-1$$

```
def inverse_transform(X):

    N = len(X)
    w = np.e ** (complex(0, 2*np.pi/N))

    x = np.zeros(N, dtype=complex)
    for n in range(N):
        for k in range(N):
            x[n] += X[k] * w ** (k * n) / N

    return x
```

```
inverse_transform(X)
```

```
array([ 0.5+0.j,  0.5-0.j, -0. -0.j, -0. -0.j, -0. -0.j, -0. -0.j,
       -0. +0.j, -0. +0.j, -0. +0.j, -0. +0.j])
```

Another example is

$$x_n = 2\cos\left(2\pi \frac{11}{40} n\right), \ n = 0, 1, 2, \cdots 19$$

Since $N = 20$, we cannot use an integer multiple of $\frac{1}{20}$ to represent a frequency $\frac{11}{40}$.

To handle this, we shall end up using all N of the availble frequencies in the DFT.

Since $\frac{11}{40}$ is in between $\frac{10}{40}$ and $\frac{12}{40}$ (each of which is an integer multiple of $\frac{1}{20}$), the complex coefficients in the DFT have their largest magnitudes at $k = 5, 6, 15, 16$, not just at a single frequency.

```
N = 20
x = np.empty(N)
```

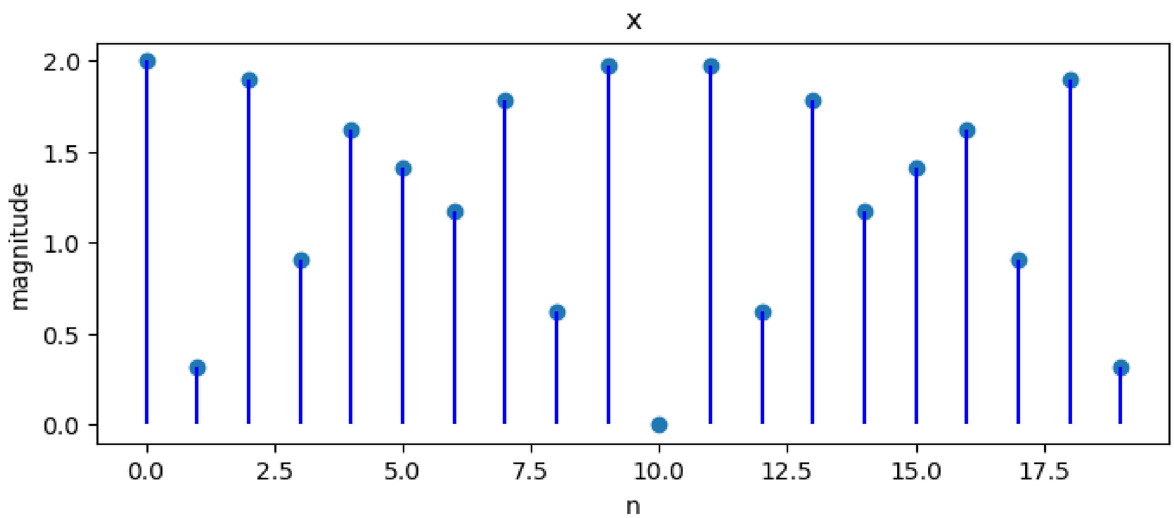
(continues on next page)

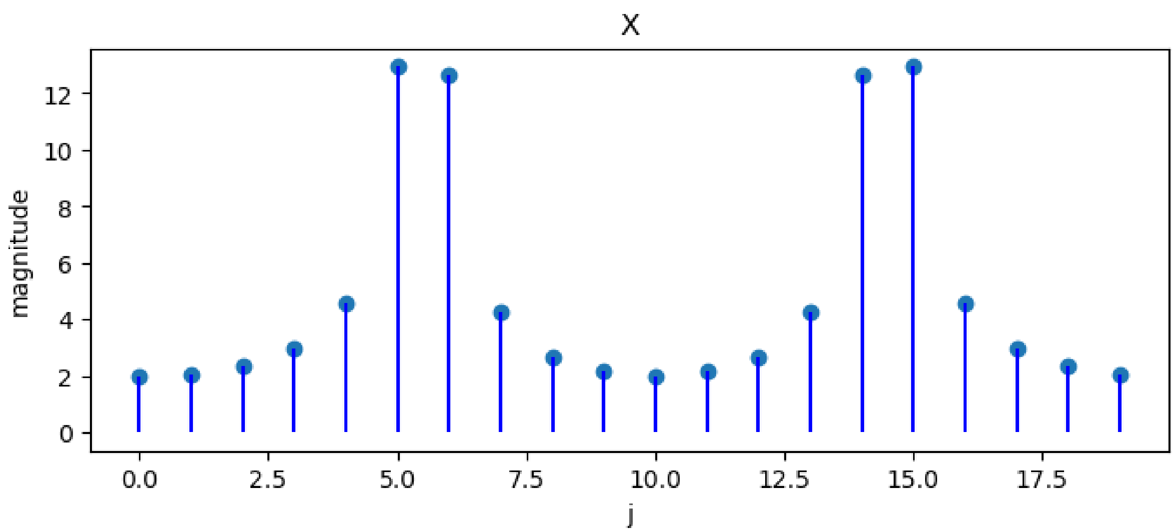
(continued from previous page)

```
for j in range(N):
    x[j] = 2 * np.cos(2 * np.pi * 11 * j / 40)
```

```
X = DFT(x)
```

```
plot_magnitude(x=x, X=X)
```





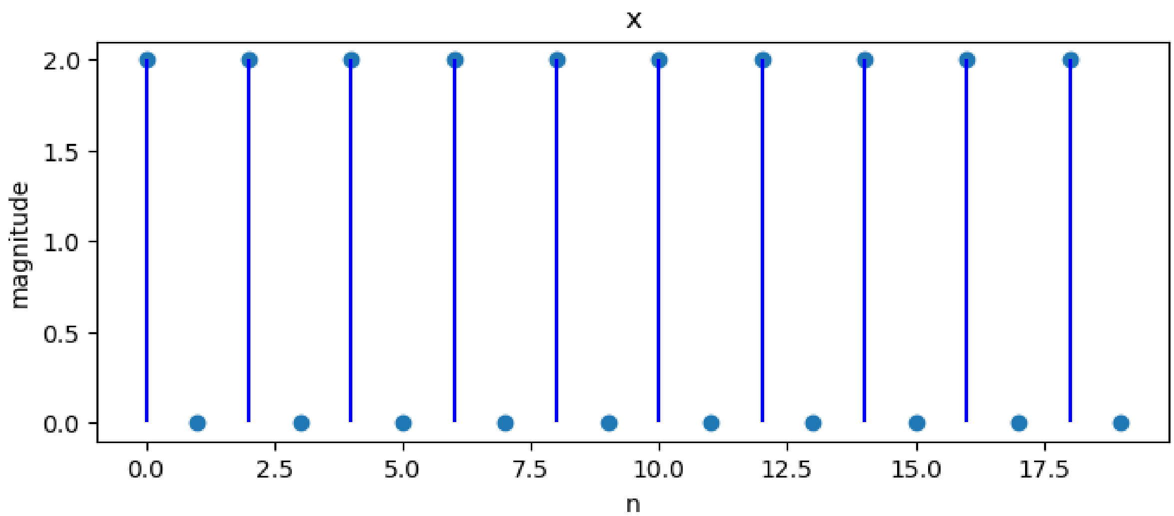
What happens if we change the last example to $x_n = 2\cos\left(2\pi \frac{10}{40} n\right)$?

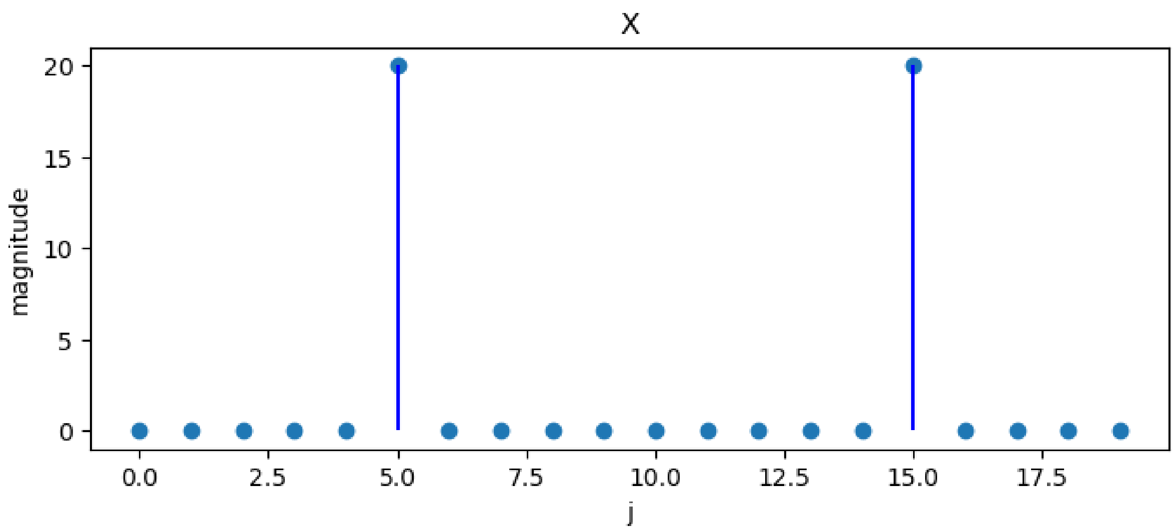
Note that $\frac{10}{40}$ is an integer multiple of $\frac{1}{20}$.

```
N = 20
x = np.empty(N)

for j in range(N):
    x[j] = 2 * np.cos(2 * np.pi * 10 * j / 40)
```

```
X = DFT(x)
```

```
plot_magnitude(x=x, X=X)
```





If we represent the discrete Fourier transform as a matrix, we discover that it equals the matrix F_N of eigenvectors of the permutation matrix P_N.

We can use the example where $x_n = 2\cos\left(2\pi\frac{11}{40}n\right),\ n = 0, 1, 2, \cdots 19$ to illustrate this.

```
N = 20
x = np.empty(N)

for j in range(N):
    x[j] = 2 * np.cos(2 * np.pi * 11 * j / 40)
```

```
x
```

```
array([ 2.   , -0.313, -1.902,  0.908,  1.618, -1.414, -1.176,  1.782,
        0.618, -1.975, -0.   ,  1.975, -0.618, -1.782,  1.176,  1.414,
       -1.618, -0.908,  1.902,  0.313])
```

First use the summation formula to transform x to X.

```
X = DFT(x)
X
```

```
array([2. +0.j   , 2. +0.558j, 2. +1.218j, 2. +2.174j, 2. +4.087j,
       2.+12.785j, 2.-12.466j, 2. -3.751j, 2. -1.801j, 2. -0.778j,
       2. -0.j   , 2. +0.778j, 2. +1.801j, 2. +3.751j, 2.+12.466j,
       2.-12.785j, 2. -4.087j, 2. -2.174j, 2. -1.218j, 2. -0.558j])
```

Now let's evaluate the outcome of postmultiplying the eigenvector matrix F_{20} by the vector x, a product that we claim should equal the Fourier tranform of the sequence $\{x_n\}_{n=0}^{N-1}$.

```
F20, _ = construct_F(20)
```

```
F20 @ x
```

```
array([2. +0.j   , 2. +0.558j, 2. +1.218j, 2. +2.174j, 2. +4.087j,
       2.+12.785j, 2.-12.466j, 2. -3.751j, 2. -1.801j, 2. -0.778j,
       2. -0.j   , 2. +0.778j, 2. +1.801j, 2. +3.751j, 2.+12.466j,
       2.-12.785j, 2. -4.087j, 2. -2.174j, 2. -1.218j, 2. -0.558j])
```

Similarly, the inverse DFT can be expressed as a inverse DFT matrix F_{20}^{-1}.

```
F20_inv = np.linalg.inv(F20)
F20_inv @ X
```

```
array([ 2.   -0.j, -0.313-0.j, -1.902+0.j,  0.908-0.j,  1.618-0.j,
       -1.414+0.j, -1.176+0.j,  1.782+0.j,  0.618-0.j, -1.975-0.j,
       -0.   +0.j,  1.975-0.j, -0.618-0.j, -1.782+0.j,  1.176+0.j,
        1.414-0.j, -1.618-0.j, -0.908+0.j,  1.902+0.j,  0.313-0.j])
```

CHAPTER

FIVE

SINGULAR VALUE DECOMPOSITION (SVD)

5.1 Overview

The **singular value decomposition** (SVD) is a work-horse in applications of least squares projection that form foundations for many statistical and machine learning methods.

After defining the SVD, we'll describe how it connects to

- **four fundamental spaces** of linear algebra
- under-determined and over-determined **least squares regressions**
- **principal components analysis** (PCA)

Like principal components analysis (PCA), DMD can be thought of as a data-reduction procedure that represents salient patterns by projecting data onto a limited set of factors.

In a sequel to this lecture about *Dynamic Mode Decompositions*, we'll describe how SVD's provide ways rapidly to compute reduced-order approximations to first-order Vector Autoregressions (VARs).

5.2 The Setting

Let X be an $m \times n$ matrix of rank p.

Necessarily, $p \leq \min(m, n)$.

In much of this lecture, we'll think of X as a matrix of **data** in which

- each column is an **individual** – a time period or person, depending on the application
- each row is a **random variable** describing an attribute of a time period or a person, depending on the application

We'll be interested in two situations

- A **short and fat** case in which $m << n$, so that there are many more columns (individuals) than rows (attributes).
- A **tall and skinny** case in which $m >> n$, so that there are many more rows (attributes) than columns (individuals).

We'll apply a **singular value decomposition** of X in both situations.

In the $m << n$ case in which there are many more individuals n than attributes m, we can calculate sample moments of a joint distribution by taking averages across observations of functions of the observations.

In this $m << n$ case, we'll look for **patterns** by using a **singular value decomposition** to do a **principal components analysis** (PCA).

In the $m >> n$ case in which there are many more attributes m than individuals n and when we are in a time-series setting in which n equals the number of time periods covered in the data set X, we'll proceed in a different way.

We'll again use a **singular value decomposition**, but now to construct a **dynamic mode decomposition** (DMD)

5.3 Singular Value Decomposition

A **singular value decomposition** of an $m \times n$ matrix X of rank $p \leq \min(m, n)$ is

$$X = U\Sigma V^\top \tag{5.1}$$

where

$$\begin{aligned} UU^\top &= I & \quad U^\top U &= I \\ VV^\top &= I & \quad V^\top V &= I \end{aligned}$$

and

- U is an $m \times m$ orthogonal matrix of **left singular vectors** of X
- Columns of U are eigenvectors of $XX^\top$
- V is an $n \times n$ orthogonal matrix of **right singular vectors** of X
- Columns of V are eigenvectors of $X^\top X$
- Σ is an $m \times n$ matrix in which the first p places on its main diagonal are positive numbers $\sigma_1, \sigma_2, \ldots, \sigma_p$ called **singular values**; remaining entries of Σ are all zero
- The p singular values are positive square roots of the eigenvalues of the $m \times m$ matrix $XX^\top$ and also of the $n \times n$ matrix $X^\top X$
- We adopt a convention that when U is a complex valued matrix, $U^\top$ denotes the **conjugate-transpose** or **Hermitian-transpose** of U, meaning that $U_{ij}^\top$ is the complex conjugate of U_{ji}.
- Similarly, when V is a complex valued matrix, $V^\top$ denotes the **conjugate-transpose** or **Hermitian-transpose** of V

The matrices U, Σ, V entail linear transformations that reshape in vectors in the following ways:

- multiplying vectors by the unitary matrices U and V **rotates** them, but leaves **angles between vectors** and **lengths of vectors** unchanged.
- multiplying vectors by the diagonal matrix Σ leaves **angles between vectors** unchanged but **rescales** vectors.

Thus, representation (5.1) asserts that multiplying an $n \times 1$ vector y by the $m \times n$ matrix X amounts to performing the following three multiplications of y sequentially:

- **rotating** y by computing $V^\top y$
- **rescaling** $V^\top y$ by multiplying it by Σ
- **rotating** $\Sigma V^\top y$ by multiplying it by U

This structure of the $m \times n$ matrix X opens the door to constructing systems of data **encoders** and **decoders**.

Thus,

- $V^\top y$ is an encoder
- Σ is an operator to be applied to the encoded data
- U is a decoder to be applied to the output from applying operator Σ to the encoded data

We'll apply this circle of ideas later in this lecture when we study Dynamic Mode Decomposition.

Road Ahead

What we have described above is called a **full** SVD.

In a **full** SVD, the shapes of U, Σ, and V are (m, m), (m, n), (n, n), respectively.

Later we'll also describe an **economy** or **reduced** SVD.

Before we study a **reduced** SVD we'll say a little more about properties of a **full** SVD.

5.4 Four Fundamental Subspaces

Let $\mathcal{C}$ denote a column space, $\mathcal{N}$ denote a null space, and $\mathcal{R}$ denote a row space.

Let's start by recalling the four fundamental subspaces of an $m \times n$ matrix X of rank p.

- The **column space** of X, denoted $\mathcal{C}(X)$, is the span of the columns of X, i.e., all vectors y that can be written as linear combinations of columns of X. Its dimension is p.
- The **null space** of X, denoted $\mathcal{N}(X)$ consists of all vectors y that satisfy $Xy = 0$. Its dimension is $n - p$.
- The **row space** of X, denoted $\mathcal{R}(X)$ is the column space of $X^\top$. It consists of all vectors z that can be written as linear combinations of rows of X. Its dimension is p.
- The **left null space** of X, denoted $\mathcal{N}(X^\top)$, consist of all vectors z such that $X^\top z = 0$. Its dimension is $m - p$.

For a full SVD of a matrix X, the matrix U of left singular vectors and the matrix V of right singular vectors contain orthogonal bases for all four subspaces.

They form two pairs of orthogonal subspaces that we'll describe now.

Let $u_i, i = 1, \ldots, m$ be the m column vectors of U and let $v_i, i = 1, \ldots, n$ be the n column vectors of V.

Let's write the full SVD of X as

$$X = \begin{bmatrix} U_L & U_R \end{bmatrix} \begin{bmatrix} \Sigma_p & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} V_L & V_R \end{bmatrix}^\top \tag{5.2}$$

where Σ_p is a $p \times p$ diagonal matrix with the p singular values on the diagonal and

$$\begin{aligned} U_L &= \begin{bmatrix} u_1 & \cdots & u_p \end{bmatrix}, \quad U_R = \begin{bmatrix} u_{p+1} & \cdots u_m \end{bmatrix} \\ V_L &= \begin{bmatrix} v_1 & \cdots & v_p \end{bmatrix}, \quad U_R = \begin{bmatrix} v_{p+1} & \cdots u_n \end{bmatrix} \end{aligned}$$

Representation (5.2) implies that

$$X \begin{bmatrix} V_L & V_R \end{bmatrix} = \begin{bmatrix} U_L & U_R \end{bmatrix} \begin{bmatrix} \Sigma_p & 0 \\ 0 & 0 \end{bmatrix}$$

or

$$\begin{aligned} XV_L &= U_L \Sigma_p \\ XV_R &= 0 \end{aligned} \tag{5.3}$$

or

$$\begin{aligned} Xv_i &= \sigma_i u_i, \quad i = 1, \ldots, p \\ Xv_i &= 0, \quad i = p + 1, \ldots, n \end{aligned} \tag{5.4}$$

Equations (5.4) tell how the transformation X maps a pair of orthonormal vectors v_i, v_j for i and j both less than or equal to the rank p of X into a pair of orthonormal vectors u_i, u_j.

Equations (5.3) assert that

$$
\begin{aligned}
\mathcal{C}(X) &= \mathcal{C}(U_L) \\
\mathcal{N}(X) &= \mathcal{C}(V_R)
\end{aligned}
$$

Taking transposes on both sides of representation (5.2) implies

$$
X^\top \begin{bmatrix} U_L & U_R \end{bmatrix} = \begin{bmatrix} V_L & V_R \end{bmatrix} \begin{bmatrix} \Sigma_p & 0 \\ 0 & 0 \end{bmatrix}
$$

or

$$
\begin{aligned}
X^\top U_L &= V_L \Sigma_p \\
X^\top U_R &= 0
\end{aligned} \tag{5.5}
$$

or

$$
\begin{aligned}
X^\top u_i &= \sigma_i v_i, \quad i = 1, \ldots, p \\
X^\top u_i &= 0 \quad i = p+1, \ldots, m
\end{aligned} \tag{5.6}
$$

Notice how equations (5.6) assert that the transformation $X^\top$ maps a pair of distinct orthonormal vectors u_i, u_j for i and j both less than or equal to the rank p of X into a pair of distinct orthonormal vectors v_i, v_j .

Equations (5.5) assert that

$$
\begin{aligned}
\mathcal{R}(X) \equiv \mathcal{C}(X^\top) &= \mathcal{C}(V_L) \\
\mathcal{N}(X^\top) &= \mathcal{C}(U_R)
\end{aligned}
$$

Thus, taken together, the systems of equations (5.3) and (5.5) describe the four fundamental subspaces of X in the following ways:

$$
\begin{aligned}
\mathcal{C}(X) &= \mathcal{C}(U_L) \\
\mathcal{N}(X^\top) &= \mathcal{C}(U_R) \\
\mathcal{R}(X) \equiv \mathcal{C}(X^\top) &= \mathcal{C}(V_L) \\
\mathcal{N}(X) &= \mathcal{C}(V_R)
\end{aligned} \tag{5.7}
$$

Since U and V are both orthonormal matrices, collection (5.7) asserts that

- U_L is an orthonormal basis for the column space of X
- U_R is an orthonormal basis for the null space of $X^\top$
- V_L is an orthonormal basis for the row space of X
- V_R is an orthonormal basis for the null space of X

We have verified the four claims in (5.7) simply by performing the multiplications called for by the right side of (5.2) and reading them.

The claims in (5.7) and the fact that U and V are both unitary (i.e, orthonormal) matrices imply that

- the column space of X is orthogonal to the null space of $X^\top$
- the null space of X is orthogonal to the row space of X

Sometimes these properties are described with the following two pairs of orthogonal complement subspaces:

- $\mathcal{C}(X)$ is the orthogonal complement of $\mathcal{N}(X^\top)$
- $\mathcal{R}(X)$ is the orthogonal complement $\mathcal{N}(X)$

Let's do an example.

```
import numpy as np
import numpy.linalg as LA
import matplotlib.pyplot as plt
```

Having imported these modules, let's do the example.

```
np.set_printoptions(precision=2)

# Define the matrix
A = np.array([[1, 2, 3, 4, 5],
              [2, 3, 4, 5, 6],
              [3, 4, 5, 6, 7],
              [4, 5, 6, 7, 8],
              [5, 6, 7, 8, 9]])

# Compute the SVD of the matrix
U, S, V = np.linalg.svd(A,full_matrices=True)

# Compute the rank of the matrix
rank = np.linalg.matrix_rank(A)

# Print the rank of the matrix
print("Rank of matrix:\n", rank)
print("S: \n", S)

# Compute the four fundamental subspaces
row_space = U[:, :rank]
col_space = V[:, :rank]
null_space = V[:, rank:]
left_null_space = U[:, rank:]


print("U:\n", U)
print("Column space:\n", col_space)
print("Left null space:\n", left_null_space)
print("V.T:\n", V.T)
print("Row space:\n", row_space.T)
print("Right null space:\n", null_space.T)
```

```
Rank of matrix:
 2
S:
 [2.69e+01 1.86e+00 1.20e-15 2.24e-16 5.82e-17]
U:
 [[-0.27 -0.73  0.63 -0.06  0.06]
 [-0.35 -0.42 -0.69 -0.45  0.12]
 [-0.43 -0.11 -0.24  0.85  0.12]
 [-0.51  0.19  0.06 -0.1  -0.83]
 [-0.59  0.5   0.25 -0.24  0.53]]
Column space:
 [[-0.27 -0.35]
 [ 0.73  0.42]
 [ 0.32 -0.65]
 [ 0.54 -0.39]
 [-0.06 -0.35]]
Left null space:
 [[ 0.63 -0.06  0.06]
```

(continues on next page)

(continued from previous page)

```
 [-0.69 -0.45  0.12]
 [-0.24  0.85  0.12]
 [ 0.06 -0.1  -0.83]
 [ 0.25 -0.24  0.53]]
V.T:
 [[-0.27  0.73  0.32  0.54 -0.06]
 [-0.35  0.42 -0.65 -0.39 -0.35]
 [-0.43  0.11  0.02 -0.29  0.85]
 [-0.51 -0.19  0.61 -0.41 -0.4 ]
 [-0.59 -0.5  -0.31  0.55 -0.04]]
Row space:
 [[-0.27 -0.35 -0.43 -0.51 -0.59]
 [-0.73 -0.42 -0.11  0.19  0.5 ]]
Right null space:
 [[-0.43  0.11  0.02 -0.29  0.85]
 [-0.51 -0.19  0.61 -0.41 -0.4 ]
 [-0.59 -0.5  -0.31  0.55 -0.04]]
```

5.5 Eckart-Young Theorem

Suppose that we want to construct the best rank r approximation of an $m \times n$ matrix X.

By best, we mean a matrix X_r of rank $r < p$ that, among all rank r matrices, minimizes

$$||X - X_r||$$

where $|| \cdot ||$ denotes a norm of a matrix X and where X_r belongs to the space of all rank r matrices of dimension $m \times n$.

Three popular **matrix norms** of an $m \times n$ matrix X can be expressed in terms of the singular values of X

- the **spectral** or l^2 norm $||X||_2 = \max_{||y|| \neq 0} \frac{||Xy||}{||y||} = \sigma_1$
- the **Frobenius** norm $||X||_F = \sqrt{\sigma_1^2 + \cdots + \sigma_p^2}$
- the **nuclear** norm $||X||_N = \sigma_1 + \cdots + \sigma_p$

The Eckart-Young theorem states that for each of these three norms, same rank r matrix is best and that it equals

$$\hat{X}_r = \sigma_1 U_1 V_1^\top + \sigma_2 U_2 V_2^\top + \cdots + \sigma_r U_r V_r^\top \tag{5.8}$$

This is a very powerful theorem that says that we can take our $m \times n$ matrix X that in not full rank, and we can best approximate it by a full rank $p \times p$ matrix through the SVD.

Moreover, if some of these p singular values carry more information than others, and if we want to have the most amount of information with the least amount of data, we can take r leading singular values ordered by magnitude.

We'll say more about this later when we present Principal Component Analysis.

You can read about the Eckart-Young theorem and some of its uses here.

We'll make use of this theorem when we discuss principal components analysis (PCA) and also dynamic mode decomposition (DMD).

5.6 Full and Reduced SVD's

Up to now we have described properties of a **full** SVD in which shapes of U, Σ, and V are (m,m), (m,n), (n,n), respectively.

There is an alternative bookkeeping convention called an **economy** or **reduced** SVD in which the shapes of U, Σ and V are different from what they are in a full SVD.

Thus, note that because we assume that X has rank p, there are only p nonzero singular values, where $p = \text{rank}(X) \leq \min(m,n)$.

A **reduced** SVD uses this fact to express U, Σ, and V as matrices with shapes (m,p), (p,p), (n,p).

You can read about reduced and full SVD here https://numpy.org/doc/stable/reference/generated/numpy.linalg.svd.html

For a full SVD,

$$\begin{aligned} UU^\top = I \qquad & U^\top U = I \\ VV^\top = I \qquad & V^\top V = I \end{aligned}$$

But not all these properties hold for a **reduced** SVD.

Which properties hold depend on whether we are in a **tall-skinny** case or a **short-fat** case.

- In a **tall-skinny** case in which $m >> n$, for a **reduced** SVD

$$\begin{aligned} UU^\top \neq I \qquad & U^\top U = I \\ VV^\top = I \qquad & V^\top V = I \end{aligned}$$

- In a **short-fat** case in which $m << n$, for a **reduced** SVD

$$\begin{aligned} UU^\top = I \qquad & U^\top U = I \\ VV^\top = I \qquad & V^\top V \neq I \end{aligned}$$

When we study Dynamic Mode Decomposition below, we shall want to remember these properties when we use a reduced SVD to compute some DMD representations.

Let's do an exercise to compare **full** and **reduced** SVD's.

To review,

- in a **full** SVD
  - U is $m \times m$
  - Σ is $m \times n$
  - V is $n \times n$
- in a **reduced** SVD
  - U is $m \times p$
  - Σ is $p \times p$
  - V is $n \times p$

First, let's study a case in which $m = 5 > n = 2$.

(This is a small example of the **tall-skinny** case that will concern us when we study **Dynamic Mode Decompositions** below.)

```python
import numpy as np
X = np.random.rand(5,2)
U, S, V = np.linalg.svd(X,full_matrices=True)  # full SVD
Uhat, Shat, Vhat = np.linalg.svd(X,full_matrices=False) # economy SVD
print('U, S, V =')
U, S, V
```

```
U, S, V =
```

```
(array([[-0.22,  0.23, -0.3 , -0.55, -0.71],
        [-0.34, -0.62, -0.69,  0.13,  0.1 ],
        [-0.58, -0.45,  0.65, -0.11, -0.16],
        [-0.41,  0.4 , -0.05,  0.76, -0.31],
        [-0.57,  0.45, -0.1 , -0.31,  0.6 ]]),
 array([1.69, 0.79]),
 array([[-0.82, -0.58],
        [ 0.58, -0.82]]))
```

```python
print('Uhat, Shat, Vhat = ')
Uhat, Shat, Vhat
```

```
Uhat, Shat, Vhat =
```

```
(array([[-0.22,  0.23],
        [-0.34, -0.62],
        [-0.58, -0.45],
        [-0.41,  0.4 ],
        [-0.57,  0.45]]),
 array([1.69, 0.79]),
 array([[-0.82, -0.58],
        [ 0.58, -0.82]]))
```

```python
rr = np.linalg.matrix_rank(X)
print(f'rank of X = {rr}')
```

```
rank of X = 2
```

Properties:

- Where U is constructed via a full SVD, $U^\top U = I_{m\times m}$ and $UU^\top = I_{m\times m}$
- Where $\hat{U}$ is constructed via a reduced SVD, although $\hat{U}^\top \hat{U} = I_{p\times p}$, it happens that $\hat{U}\hat{U}^\top \neq I_{m\times m}$

We illustrate these properties for our example with the following code cells.

```python
UTU = U.T@U
UUT = U@U.T
print('UUT, UTU = ')
UUT, UTU
```

```
UUT, UTU =
```

```
(array([[ 1.00e+00, -4.61e-17, -5.27e-17, -9.13e-17, -5.21e-17],
        [-4.61e-17,  1.00e+00, -1.24e-17,  8.30e-18, -4.87e-17],
        [-5.27e-17, -1.24e-17,  1.00e+00,  1.10e-17, -1.11e-17],
```

(continues on next page)

(continued from previous page)

```
        [-9.13e-17,  8.30e-18,  1.10e-17,  1.00e+00,  6.68e-17],
        [-5.21e-17, -4.87e-17, -1.11e-17,  6.68e-17,  1.00e+00]]),
 array([[ 1.00e+00, -3.54e-17, -2.55e-17, -1.02e-16, -1.09e-16],
        [-3.54e-17,  1.00e+00,  4.94e-17,  1.36e-17,  7.57e-17],
        [-2.55e-17,  4.94e-17,  1.00e+00, -2.50e-17, -8.97e-18],
        [-1.02e-16,  1.36e-17, -2.50e-17,  1.00e+00, -3.28e-17],
        [-1.09e-16,  7.57e-17, -8.97e-18, -3.28e-17,  1.00e+00]]))
```

```
UhatUhatT = Uhat@Uhat.T
UhatTUhat = Uhat.T@Uhat
print('UhatUhatT, UhatTUhat= ')
UhatUhatT, UhatTUhat
```

```
UhatUhatT, UhatTUhat=
```

```
(array([[ 0.1 , -0.07,  0.02,  0.18,  0.23],
        [-0.07,  0.5 ,  0.48, -0.1 , -0.08],
        [ 0.02,  0.48,  0.54,  0.06,  0.13],
        [ 0.18, -0.1 ,  0.06,  0.33,  0.42],
        [ 0.23, -0.08,  0.13,  0.42,  0.53]]),
 array([[ 1.00e+00, -3.54e-17],
        [-3.54e-17,  1.00e+00]]))
```

Remarks:

The cells above illustrate the application of the `full_matrices=True` and `full_matrices=False` options. Using `full_matrices=False` returns a reduced singular value decomposition.

The **full** and **reduced** SVD's both accurately decompose an $m \times n$ matrix X

When we study Dynamic Mode Decompositions below, it will be important for us to remember the preceding properties of full and reduced SVD's in such tall-skinny cases.

Now let's turn to a short-fat case.

To illustrate this case, we'll set $m = 2 < 5 = n$ and compute both full and reduced SVD's.

```
import numpy as np
X = np.random.rand(2,5)
U, S, V = np.linalg.svd(X,full_matrices=True)  # full SVD
Uhat, Shat, Vhat = np.linalg.svd(X,full_matrices=False) # economy SVD
print('U, S, V = ')
U, S, V
```

```
U, S, V =
```

```
(array([[ 0.64, -0.77],
        [ 0.77,  0.64]]),
 array([1.26, 0.29]),
 array([[ 0.28,  0.81,  0.2 ,  0.08,  0.47],
        [-0.77,  0.1 ,  0.61,  0.14,  0.01],
        [ 0.54, -0.38,  0.75, -0.05,  0.03],
        [ 0.11, -0.12, -0.07,  0.98, -0.  ],
        [-0.15, -0.42, -0.14, -0.04,  0.88]]))
```

```
print('Uhat, Shat, Vhat = ')
Uhat, Shat, Vhat
```

```
Uhat, Shat, Vhat =
```

```
(array([[ 0.64, -0.77],
        [ 0.77,  0.64]]),
 array([1.26, 0.29]),
 array([[ 0.28,  0.81,  0.2 ,  0.08,  0.47],
        [-0.77,  0.1 ,  0.61,  0.14,  0.01]]))
```

Let's verify that our reduced SVD accurately represents X

```
SShat=np.diag(Shat)
np.allclose(X, Uhat@SShat@Vhat)
```

```
True
```

5.7 Polar Decomposition

A **reduced** singular value decomposition (SVD) of X is related to a **polar decomposition** of X

$$X = SQ$$

where

$$\begin{aligned} S &= U\Sigma U^\top \\ Q &= UV^\top \end{aligned}$$

Here

- S is an $m \times m$ **symmetric** matrix
- Q is an $m \times n$ **orthogonal** matrix

and in our reduced SVD

- U is an $m \times p$ orthonormal matrix
- Σ is a $p \times p$ diagonal matrix
- V is an $n \times p$ orthonormal

5.8 Application: Principal Components Analysis (PCA)

Let's begin with a case in which $n >> m$, so that we have many more individuals n than attributes m.

The matrix X is **short and fat** in an $n >> m$ case as opposed to a **tall and skinny** case with $m >> n$ to be discussed later.

We regard X as an $m \times n$ matrix of **data**:

$$X = [X_1 \mid X_2 \mid \cdots \mid X_n]$$

where for $j = 1, \dots, n$ the column vector $X_j = \begin{bmatrix} x_{1j} \\ x_{2j} \\ \vdots \\ x_{mj} \end{bmatrix}$ is a vector of observations on variables $\begin{bmatrix} X_1 \\ X_2 \\ \vdots \\ X_m \end{bmatrix}$.

In a **time series** setting, we would think of columns j as indexing different **times** at which random variables are observed, while rows index different random variables.

In a **cross-section** setting, we would think of columns j as indexing different **individuals** for which random variables are observed, while rows index different **attributes**.

As we have seen before, the SVD is a way to decompose a matrix into useful components, just like polar decomposition, eigendecomposition, and many others.

PCA, on the other hand, is a method that builds on the SVD to analyze data. The goal is to apply certain steps, to help better visualize patterns in data, using statistical tools to capture the most important patterns in data.

Step 1: Standardize the data:

Because our data matrix may hold variables of different units and scales, we first need to standardize the data.

First by computing the average of each row of X.

$$\bar{X}_i = \frac{1}{n} \sum_{j=1}^{n} x_{ij}$$

We then create an average matrix out of these means:

$$\bar{X} = \begin{bmatrix} \bar{X}_1 \\ \bar{X}_2 \\ \ldots \\ \bar{X}_m \end{bmatrix} [1 \mid 1 \mid \cdots \mid 1]$$

And subtract out of the original matrix to create a mean centered matrix:

$$B = X - \bar{X}$$

Step 2: Compute the covariance matrix:

Then because we want to extract the relationships between variables rather than just their magnitude, in other words, we want to know how they can explain each other, we compute the covariance matrix of B.

$$C = \frac{1}{n} B B^\top$$

Step 3: Decompose the covariance matrix and arrange the singular values:

Since the matrix C is positive definite, we can eigendecompose it, find its eigenvalues, and rearrange the eigenvalue and eigenvector matrices in a decreasing order.

The eigendecomposition of C can be found by decomposing B instead. Since B is not a square matrix, we obtain an SVD of B:

$$\begin{aligned} B B^\top &= U \Sigma V^\top (U \Sigma V^\top)^\top \\ &= U \Sigma V^\top V \Sigma^\top U^\top \\ &= U \Sigma \Sigma^\top U^\top \end{aligned}$$

$$C = \frac{1}{n} U \Sigma \Sigma^\top U^\top$$

We can then rearrange the columns in the matrices U and Σ so that the singular values are in decreasing order.

Step 4: Select singular values, (optional) truncate the rest:

We can now decide how many singular values to pick, based on how much variance you want to retain. (e.g., retaining 95% of the total variance).

We can obtain the percentage by calculating the variance contained in the leading r factors divided by the variance in total:

$$\frac{\sum_{i=1}^{r} \sigma_i^2}{\sum_{i=1}^{p} \sigma_i^2}$$

Step 5: Create the Score Matrix:

$$\begin{aligned} T &= BV \\ &= U\Sigma V^\top V \\ &= U\Sigma \end{aligned}$$

5.9 Relationship of PCA to SVD

To relate an SVD to a PCA of data set X, first construct the SVD of the data matrix X:

Let's assume that sample means of all variables are zero, so we don't need to standardize our matrix.

$$X = U\Sigma V^\top = \sigma_1 U_1 V_1^\top + \sigma_2 U_2 V_2^\top + \cdots + \sigma_p U_p V_p^\top \tag{5.9}$$

where

$$U = [U_1 | U_2 | \ldots | U_m]$$

$$V^\top = \begin{bmatrix} V_1^\top \\ V_2^\top \\ \ldots \\ V_n^\top \end{bmatrix}$$

In equation (5.9), each of the $m \times n$ matrices $U_j V_j^\top$ is evidently of rank 1.

Thus, we have

$$X = \sigma_1 \begin{bmatrix} U_{11} V_1^\top \\ U_{21} V_1^\top \\ \cdots \\ U_{m1} V_1^\top \end{bmatrix} + \sigma_2 \begin{bmatrix} U_{12} V_2^\top \\ U_{22} V_2^\top \\ \cdots \\ U_{m2} V_2^\top \end{bmatrix} + \ldots + \sigma_p \begin{bmatrix} U_{1p} V_p^\top \\ U_{2p} V_p^\top \\ \cdots \\ U_{mp} V_p^\top \end{bmatrix} \tag{5.10}$$

Here is how we would interpret the objects in the matrix equation (5.10) in a time series context:

- for each $k = 1, \ldots, n$, the object $\{V_{kj}\}_{j=1}^n$ is a time series for the kth **principal component**
- $U_j = \begin{bmatrix} U_{1k} \\ U_{2k} \\ \ldots \\ U_{mk} \end{bmatrix}$ $k = 1, \ldots, m$ is a vector of **loadings** of variables X_i on the kth principal component, $i = 1, \ldots, m$
- σ_k for each $k = 1, \ldots, p$ is the strength of kth **principal component**, where strength means contribution to the overall covariance of X.

5.10 PCA with Eigenvalues and Eigenvectors

We now use an eigen decomposition of a sample covariance matrix to do PCA.

Let $X_{m \times n}$ be our $m \times n$ data matrix.

Let's assume that sample means of all variables are zero.

We can assure this by **pre-processing** the data by subtracting sample means.

Define a sample covariance matrix Ω as

$$\Omega = XX^\top$$

Then use an eigen decomposition to represent Ω as follows:

$$\Omega = P\Lambda P^\top$$

Here

- P is $m \times m$ matrix of eigenvectors of Ω
- Λ is a diagonal matrix of eigenvalues of Ω

We can then represent X as

$$X = P\epsilon$$

where

$$\epsilon = P^{-1}X$$

and

$$\epsilon\epsilon^\top = \Lambda.$$

We can verify that

$$XX^\top = P\Lambda P^\top. \tag{5.11}$$

It follows that we can represent the data matrix X as

$$X = [X_1|X_2|\ldots|X_m] = [P_1|P_2|\ldots|P_m] \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \ldots \\ \epsilon_m \end{bmatrix} = P_1\epsilon_1 + P_2\epsilon_2 + \ldots + P_m\epsilon_m$$

To reconcile the preceding representation with the PCA that we had obtained earlier through the SVD, we first note that $\epsilon_j^2 = \lambda_j \equiv \sigma_j^2$.

Now define $\tilde{\epsilon_j} = \frac{\epsilon_j}{\sqrt{\lambda_j}}$, which implies that $\tilde{\epsilon}_j\tilde{\epsilon}_j^\top = 1$.

Therefore

$$\begin{aligned} X &= \sqrt{\lambda_1}P_1\tilde{\epsilon_1} + \sqrt{\lambda_2}P_2\tilde{\epsilon_2} + \ldots + \sqrt{\lambda_m}P_m\tilde{\epsilon_m} \\ &= \sigma_1 P_1\tilde{\epsilon_2} + \sigma_2 P_2\tilde{\epsilon_2} + \ldots + \sigma_m P_m\tilde{\epsilon_m}, \end{aligned}$$

which agrees with

$$X = \sigma_1 U_1 {V_1}^T + \sigma_2 U_2 {V_2}^T + \ldots + \sigma_r U_r {V_r}^T$$

provided that we set

- $U_j = P_j$ (a vector of loadings of variables on principal component j)
- $V_k^T = \tilde{\epsilon_k}$ (the kth principal component)

Because there are alternative algorithms for computing P and U for given a data matrix X, depending on algorithms used, we might have sign differences or different orders of eigenvectors.

We can resolve such ambiguities about U and P by

1. sorting eigenvalues and singular values in descending order
2. imposing positive diagonals on P and U and adjusting signs in $V^\top$ accordingly

5.11 Connections

To pull things together, it is useful to assemble and compare some formulas presented above.

First, consider an SVD of an $m \times n$ matrix:

$$X = U\Sigma V^\top$$

Compute:

$$\begin{aligned} XX^\top &= U\Sigma V^\top V\Sigma^\top U^\top \\ &\equiv U\Sigma\Sigma^\top U^\top \\ &\equiv U\Lambda U^\top \end{aligned} \tag{5.12}$$

Compare representation (5.12) with equation (5.11) above.

Evidently, U in the SVD is the matrix P of eigenvectors of $XX^\top$ and $\Sigma\Sigma^\top$ is the matrix Λ of eigenvalues.

Second, let's compute

$$\begin{aligned} X^\top X &= V\Sigma^\top U^\top U\Sigma V^\top \\ &= V\Sigma^\top\Sigma V^\top \end{aligned}$$

Thus, the matrix V in the SVD is the matrix of eigenvectors of $X^\top X$

Summarizing and fitting things together, we have the eigen decomposition of the sample covariance matrix

$$XX^\top = P\Lambda P^\top$$

where P is an orthogonal matrix.

Further, from the SVD of X, we know that

$$XX^\top = U\Sigma\Sigma^\top U^\top$$

where U is an orthogonal matrix.

Thus, $P = U$ and we have the representation of X

$$X = P\epsilon = U\Sigma V^\top$$

It follows that

$$U^\top X = \Sigma V^\top = \epsilon$$

Note that the preceding implies that

$$\epsilon\epsilon^\top = \Sigma V^\top V \Sigma^\top = \Sigma\Sigma^\top = \Lambda,$$

so that everything fits together.

Below we define a class `DecomAnalysis` that wraps PCA and SVD for a given a data matrix `X`.

```python
class DecomAnalysis:
    """
    A class for conducting PCA and SVD.
    X: data matrix
    r_component: chosen rank for best approximation
    """

    def __init__(self, X, r_component=None):

        self.X = X

        self.Ω = (X @ X.T)

        self.m, self.n = X.shape
        self.r = LA.matrix_rank(X)

        if r_component:
            self.r_component = r_component
        else:
            self.r_component = self.m

    def pca(self):

        𝜆, P = LA.eigh(self.Ω)    # columns of P are eigenvectors

        ind = sorted(range(𝜆.size), key=lambda x: 𝜆[x], reverse=True)

        # sort by eigenvalues
        self.𝜆 = 𝜆[ind]
        P = P[:, ind]
        self.P = P @ diag_sign(P)

        self.Λ = np.diag(self.𝜆)

        self.explained_ratio_pca = np.cumsum(self.𝜆) / self.𝜆.sum()

        # compute the N by T matrix of principal components
        self.𝜖 = self.P.T @ self.X

        P = self.P[:, :self.r_component]
        𝜖 = self.𝜖[:self.r_component, :]

        # transform data
        self.X_pca = P @ 𝜖

    def svd(self):

        U, 𝜎, VT = LA.svd(self.X)

        ind = sorted(range(𝜎.size), key=lambda x: 𝜎[x], reverse=True)
```

(continues on next page)

(continued from previous page)

```
        # sort by eigenvalues
        d = min(self.m, self.n)

        self.σ = σ[ind]
        U = U[:, ind]
        D = diag_sign(U)
        self.U = U @ D
        VT[:d, :] = D @ VT[ind, :]
        self.VT = VT

        self.Σ = np.zeros((self.m, self.n))
        self.Σ[:d, :d] = np.diag(self.σ)

        σ_sq = self.σ ** 2
        self.explained_ratio_svd = np.cumsum(σ_sq) / σ_sq.sum()

        # slicing matrices by the number of components to use
        U = self.U[:, :self.r_component]
        Σ = self.Σ[:self.r_component, :self.r_component]
        VT = self.VT[:self.r_component, :]

        # transform data
        self.X_svd = U @ Σ @ VT

    def fit(self, r_component):

        # pca
        P = self.P[:, :r_component]
        ϵ = self.ϵ[:r_component, :]

        # transform data
        self.X_pca = P @ ϵ

        # svd
        U = self.U[:, :r_component]
        Σ = self.Σ[:r_component, :r_component]
        VT = self.VT[:r_component, :]

        # transform data
        self.X_svd = U @ Σ @ VT

def diag_sign(A):
    "Compute the signs of the diagonal of matrix A"

    D = np.diag(np.sign(np.diag(A)))

    return D
```

We also define a function that prints out information so that we can compare decompositions obtained by different algorithms.

```
def compare_pca_svd(da):
    """
    Compare the outcomes of PCA and SVD.
    """

    da.pca()
```

(continues on next page)

(continued from previous page)

```
    da.svd()

    print('Eigenvalues and Singular values\n')
    print(f'λ = {da.λ}\n')
    print(f'σ^2 = {da.σ**2}\n')
    print('\n')

    # loading matrices
    fig, axs = plt.subplots(1, 2, figsize=(14, 5))
    plt.suptitle('loadings')
    axs[0].plot(da.P.T)
    axs[0].set_title('P')
    axs[0].set_xlabel('m')
    axs[1].plot(da.U.T)
    axs[1].set_title('U')
    axs[1].set_xlabel('m')
    plt.show()

    # principal components
    fig, axs = plt.subplots(1, 2, figsize=(14, 5))
    plt.suptitle('principal components')
    axs[0].plot(da.ε.T)
    axs[0].set_title('ε')
    axs[0].set_xlabel('n')
    axs[1].plot(da.VT[:da.r, :].T * np.sqrt(da.λ))
    axs[1].set_title(r'$V^\top *\sqrt{\lambda}$')
    axs[1].set_xlabel('n')
    plt.show()
```

5.12 Exercises

Exercise 5.12.1

In Ordinary Least Squares (OLS), we learn to compute $\hat{\beta} = (X^\top X)^{-1} X^\top y$, but there are cases such as when we have colinearity or an underdetermined system: **short fat** matrix.

In these cases, the $(X^\top X)$ matrix is not not invertible (its determinant is zero) or ill-conditioned (its determinant is very close to zero).

What we can do instead is to create what is called a pseudoinverse, a full rank approximation of the inverted matrix so we can compute $\hat{\beta}$ with it.

Thinking in terms of the Eckart-Young theorem, build the pseudoinverse matrix X^+ and use it to compute $\hat{\beta}$.

Solution

We can use SVD to compute the pseudoinverse:

$$X = U\Sigma V^\top$$

inverting X, we have:

$$X^{+} = V\Sigma^{+}U^{\top}$$

where:

$$\Sigma^{+} = \begin{bmatrix} \frac{1}{\sigma_1} & 0 & \cdots & 0 & 0 \\ 0 & \frac{1}{\sigma_2} & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & \frac{1}{\sigma_p} & 0 \\ 0 & 0 & \cdots & 0 & 0 \end{bmatrix}$$

and finally:

$$\hat{\beta} = X^{+}y = V\Sigma^{+}U^{\top}y$$

For an example PCA applied to analyzing the structure of intelligence tests see this lecture *Multivariable Normal Distribution*.

Look at parts of that lecture that describe and illustrate the classic factor analysis model.

As mentioned earlier, in a sequel to this lecture about *Dynamic Mode Decompositions*, we'll describe how SVD's provide ways rapidly to compute reduced-order approximations to first-order Vector Autoregressions (VARs).

CHAPTER

SIX

VARS AND DMDS

This lecture applies computational methods that we learned about in this lecture *Singular Value Decomposition* to

- first-order vector autoregressions (VARs)
- dynamic mode decompositions (DMDs)
- connections between DMDs and first-order VARs

6.1 First-Order Vector Autoregressions

We want to fit a **first-order vector autoregression**

$$X_{t+1} = AX_t + C\epsilon_{t+1}, \quad \epsilon_{t+1} \perp X_t \tag{6.1}$$

where ϵ_{t+1} is the time $t+1$ component of a sequence of i.i.d. $m \times 1$ random vectors with mean vector zero and identity covariance matrix and where the $m \times 1$ vector X_t is

$$X_t = \begin{bmatrix} X_{1,t} & X_{2,t} & \cdots & X_{m,t} \end{bmatrix}^\top \tag{6.2}$$

and where $\cdot^\top$ again denotes complex transposition and $X_{i,t}$ is variable i at time t.

We want to fit equation (6.1).

Our data are organized in an $m \times (n+1)$ matrix $\tilde{X}$

$$\tilde{X} = [X_1 \mid X_2 \mid \cdots \mid X_n \mid X_{n+1}]$$

where for $t = 1, \ldots, n+1$, the $m \times 1$ vector X_t is given by (6.2).

Thus, we want to estimate a system (6.1) that consists of m least squares regressions of **everything** on one lagged value of **everything**.

The i'th equation of (6.1) is a regression of $X_{i,t+1}$ on the vector X_t.

We proceed as follows.

From $\tilde{X}$, we form two $m \times n$ matrices

$$X = [X_1 \mid X_2 \mid \cdots \mid X_n]$$

and

$$X' = [X_2 \mid X_3 \mid \cdots \mid X_{n+1}]$$

Here $'$ is part of the name of the matrix X' and does not indicate matrix transposition.

We use $\cdot^\top$ to denote matrix transposition or its extension to complex matrices.

In forming X and X', we have in each case dropped a column from $\tilde{X}$, the last column in the case of X, and the first column in the case of X'.

Evidently, X and X' are both $m \times n$ matrices.

We denote the rank of X as $p \leq \min(m, n)$.

Two cases that interest us are

- $n >> m$, so that we have many more time series observations n than variables m
- $m >> n$, so that we have many more variables m than time series observations n

At a general level that includes both of these special cases, a common formula describes the least squares estimator $\hat{A}$ of A.

But important details differ.

The common formula is

$$\hat{A} = X' X^+ \tag{6.3}$$

where X^+ is the pseudo-inverse of X.

To read about the **Moore-Penrose pseudo-inverse** please see Moore-Penrose pseudo-inverse

Applicable formulas for the pseudo-inverse differ for our two cases.

Short-Fat Case:

When $n >> m$, so that we have many more time series observations n than variables m and when X has linearly independent **rows**, $XX^\top$ has an inverse and the pseudo-inverse X^+ is

$$X^+ = X^\top (XX^\top)^{-1}$$

Here X^+ is a **right-inverse** that verifies $XX^+ = I_{m\times m}$.

In this case, our formula (6.3) for the least-squares estimator of the population matrix of regression coefficients A becomes

$$\hat{A} = X' X^\top (XX^\top)^{-1} \tag{6.4}$$

This formula for least-squares regression coefficients is widely used in econometrics.

It is used to estimate vector autorgressions.

The right side of formula (6.4) is proportional to the empirical cross second moment matrix of X_{t+1} and X_t times the inverse of the second moment matrix of X_t.

Tall-Skinny Case:

When $m >> n$, so that we have many more attributes m than time series observations n and when X has linearly independent **columns**, $X^\top X$ has an inverse and the pseudo-inverse X^+ is

$$X^+ = (X^\top X)^{-1} X^\top$$

Here X^+ is a **left-inverse** that verifies $X^+ X = I_{n\times n}$.

In this case, our formula (6.3) for a least-squares estimator of A becomes

$$\hat{A} = X' (X^\top X)^{-1} X^\top \tag{6.5}$$

Please compare formulas (6.4) and (6.5) for $\hat{A}$.

Here we are especially interested in formula (6.5).

The ith row of $\hat{A}$ is an $m \times 1$ vector of regression coefficients of $X_{i,t+1}$ on $X_{j,t}, j = 1, \ldots, m$.

If we use formula (6.5) to calculate $\hat{A}X$ we find that

$$\hat{A}X = X'$$

so that the regression equation **fits perfectly**.

This is a typical outcome in an **underdetermined least-squares** model.

To reiterate, in the **tall-skinny** case (described in *Singular Value Decomposition*) in which we have a number n of observations that is small relative to the number m of attributes that appear in the vector X_t, we want to fit equation (6.1).

We confront the facts that the least squares estimator is underdetermined and that the regression equation fits perfectly.

To proceed, we'll want efficiently to calculate the pseudo-inverse X^+.

The pseudo-inverse X^+ will be a component of our estimator of A.

As our estimator $\hat{A}$ of A we want to form an $m \times m$ matrix that solves the least-squares best-fit problem

$$\hat{A} = \text{argmin}_{\check{A}} ||X' - \check{A}X||_F \tag{6.6}$$

where $|| \cdot ||_F$ denotes the Frobenius (or Euclidean) norm of a matrix.

The Frobenius norm is defined as

$$||A||_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{m} |A_{ij}|^2}$$

The minimizer of the right side of equation (6.6) is

$$\hat{A} = X'X^+ \tag{6.7}$$

where the (possibly huge) $n \times m$ matrix $X^+ = (X^\top X)^{-1} X^\top$ is again a pseudo-inverse of X.

For some situations that we are interested in, $X^\top X$ can be close to singular, a situation that makes some numerical algorithms be inaccurate.

To acknowledge that possibility, we'll use efficient algorithms to constructing a **reduced-rank approximation** of $\hat{A}$ in formula (6.5).

Such an approximation to our vector autoregression will no longer fit perfectly.

The ith row of $\hat{A}$ is an $m \times 1$ vector of regression coefficients of $X_{i,t+1}$ on $X_{j,t}, j = 1, \ldots, m$.

An efficient way to compute the pseudo-inverse X^+ is to start with a singular value decomposition

$$X = U\Sigma V^\top \tag{6.8}$$

where we remind ourselves that for a **reduced** SVD, X is an $m \times n$ matrix of data, U is an $m \times p$ matrix, Σ is a $p \times p$ matrix, and V is an $n \times p$ matrix.

We can efficiently construct the pertinent pseudo-inverse X^+ by recognizing the following string of equalities.

$$\begin{aligned} X^+ &= (X^\top X)^{-1} X^\top \\ &= (V\Sigma U^\top U \Sigma V^\top)^{-1} V \Sigma U^\top \\ &= (V\Sigma\Sigma V^\top)^{-1} V \Sigma U^\top \\ &= V\Sigma^{-1}\Sigma^{-1} V^\top V \Sigma U^\top \\ &= V\Sigma^{-1} U^\top \end{aligned} \tag{6.9}$$

(Since we are in the $m >> n$ case in which $V^\top V = I_{p\times p}$ in a reduced SVD, we can use the preceding string of equalities for a reduced SVD as well as for a full SVD.)

Thus, we shall construct a pseudo-inverse X^+ of X by using a singular value decomposition of X in equation (6.8) to compute

$$X^+ = V\Sigma^{-1}U^\top \tag{6.10}$$

where the matrix Σ^{-1} is constructed by replacing each non-zero element of Σ with σ_j^{-1}.

We can use formula (6.10) together with formula (6.7) to compute the matrix $\hat{A}$ of regression coefficients.

Thus, our estimator $\hat{A} = X'X^+$ of the $m \times m$ matrix of coefficients A is

$$\hat{A} = X'V\Sigma^{-1}U^\top \tag{6.11}$$

6.2 Dynamic Mode Decomposition (DMD)

We turn to the $m >> n$ **tall and skinny** case associated with **Dynamic Mode Decomposition**.

Here an $m \times n + 1$ data matrix $\tilde{X}$ contains many more attributes (or variables) m than time periods $n + 1$.

Dynamic mode decomposition was introduced by [Schmid, 2010],

You can read about Dynamic Mode Decomposition [Kutz *et al.*, 2016] and [Brunton and Kutz, 2019] (section 7.2).

Dynamic Mode Decomposition (DMD) computes a rank $r < p$ approximation to the least squares regression coefficients $\hat{A}$ described by formula (6.11).

We'll build up gradually to a formulation that is useful in applications.

We'll do this by describing three alternative representations of our first-order linear dynamic system, i.e., our vector autoregression.

Guide to three representations: In practice, we'll mainly be interested in Representation 3.

We use the first two representations to present some useful intermediate steps that help us to appreciate what is under the hood of Representation 3.

In applications, we'll use only a small subset of **DMD modes** to approximate dynamics.

We use such a small subset of DMD modes to construct a reduced-rank approximation to A.

To do that, we'll want to use the **reduced** SVD's affiliated with representation 3, not the **full** SVD's affiliated with representations 1 and 2.

Guide to impatient reader: In our applications, we'll be using Representation 3.

You might want to skip the stage-setting representations 1 and 2 on first reading.

6.3 Representation 1

In this representation, we shall use a **full** SVD of X.

We use the m **columns** of U, and thus the m **rows** of $U^\top$, to define a $m \times 1$ vector $\tilde{b}_t$ as

$$\tilde{b}_t = U^\top X_t. \tag{6.12}$$

The original data X_t can be represented as

$$X_t = U\tilde{b}_t \tag{6.13}$$

(Here we use b to remind ourselves that we are creating a **basis** vector.)

Since we are now using a **full** SVD, $UU^\top = I_{m\times m}$.

So it follows from equation (6.12) that we can reconstruct X_t from $\tilde{b}_t$.

In particular,

- Equation (6.12) serves as an **encoder** that **rotates** the $m \times 1$ vector X_t to become an $m \times 1$ vector $\tilde{b}_t$
- Equation (6.13) serves as a **decoder** that **reconstructs** the $m \times 1$ vector X_t by rotating the $m \times 1$ vector $\tilde{b}_t$

Define a transition matrix for an $m \times 1$ basis vector $\tilde{b}_t$ by

$$\tilde{A} = U^\top \hat{A} U \tag{6.14}$$

We can recover $\hat{A}$ from

$$\hat{A} = U\tilde{A}U^\top$$

Dynamics of the $m \times 1$ basis vector $\tilde{b}_t$ are governed by

$$\tilde{b}_{t+1} = \tilde{A}\tilde{b}_t$$

To construct forecasts $\overline{X}_t$ of future values of X_t conditional on X_1, we can apply decoders (i.e., rotators) to both sides of this equation and deduce

$$\overline{X}_{t+1} = U\tilde{A}^t U^\top X_1$$

where we use $\overline{X}_{t+1}, t \geq 1$ to denote a forecast.

6.4 Representation 2

This representation is related to one originally proposed by [Schmid, 2010].

It can be regarded as an intermediate step on the way to obtaining a related representation 3 to be presented later

As with Representation 1, we continue to

- use a **full** SVD and **not** a reduced SVD

As we observed and illustrated in a lecture about the *Singular Value Decomposition*

- (a) for a full SVD $UU^\top = I_{m\times m}$ and $U^\top U = I_{p\times p}$ are both identity matrices
- (b) for a reduced SVD of X, $U^\top U$ is not an identity matrix.

As we shall see later, a full SVD is too confining for what we ultimately want to do, namely, cope with situations in which $U^\top U$ is **not** an identity matrix because we use a reduced SVD of X.

But for now, let's proceed under the assumption that we are using a full SVD so that requirements (a) and (b) are both satisfied.

Form an eigendecomposition of the $m \times m$ matrix $\tilde{A} = U^\top \hat{A} U$ defined in equation (6.14):

$$\tilde{A} = W\Lambda W^{-1} \tag{6.15}$$

where Λ is a diagonal matrix of eigenvalues and W is an $m \times m$ matrix whose columns are eigenvectors corresponding to rows (eigenvalues) in Λ.

When $UU^\top = I_{m \times m}$, as is true with a full SVD of X, it follows that

$$\hat{A} = U\tilde{A}U^\top = UW\Lambda W^{-1}U^\top \tag{6.16}$$

According to equation (6.16), the diagonal matrix Λ contains eigenvalues of $\hat{A}$ and corresponding eigenvectors of $\hat{A}$ are columns of the matrix UW.

It follows that the systematic (i.e., not random) parts of the X_t dynamics captured by our first-order vector autoregressions are described by

$$X_{t+1} = UW\Lambda W^{-1}U^\top X_t$$

Multiplying both sides of the above equation by $W^{-1}U^\top$ gives

$$W^{-1}U^\top X_{t+1} = \Lambda W^{-1}U^\top X_t$$

or

$$\hat{b}_{t+1} = \Lambda \hat{b}_t$$

where our **encoder** is

$$\hat{b}_t = W^{-1}U^\top X_t$$

and our **decoder** is

$$X_t = UW\hat{b}_t$$

We can use this representation to construct a predictor $\overline{X}_{t+1}$ of X_{t+1} conditional on X_1 via:

$$\overline{X}_{t+1} = UW\Lambda^t W^{-1}U^\top X_1 \tag{6.17}$$

In effect, [Schmid, 2010] defined an $m \times m$ matrix Φ_s as

$$\Phi_s = UW \tag{6.18}$$

and a generalized inverse

$$\Phi_s^+ = W^{-1}U^\top \tag{6.19}$$

[Schmid, 2010] then represented equation (6.17) as

$$\overline{X}_{t+1} = \Phi_s \Lambda^t \Phi_s^+ X_1 \tag{6.20}$$

Components of the basis vector $\hat{b}_t = W^{-1}U^\top X_t \equiv \Phi_s^+ X_t$ are
DMD **projected modes**.

To understand why they are called **projected modes**, notice that

$$\Phi_s^+ = (\Phi_s^\top \Phi_s)^{-1}\Phi_s^\top$$

so that the $m \times p$ matrix

$$\hat{b} = \Phi_s^+ X$$

is a matrix of regression coefficients of the $m \times n$ matrix X on the $m \times p$ matrix Φ_s.

We'll say more about this interpretation in a related context when we discuss representation 3, which was suggested by Tu et al. [Tu *et al.*, 2014].

It is more appropriate to use representation 3 when, as is often the case in practice, we want to use a reduced SVD.

6.5 Representation 3

Departing from the procedures used to construct Representations 1 and 2, each of which deployed a **full** SVD, we now use a **reduced** SVD.

Again, we let $p \leq \min(m, n)$ be the rank of X.

Construct a **reduced** SVD

$$X = \tilde{U}\tilde{\Sigma}\tilde{V}^\top,$$

where now $\tilde{U}$ is $m \times p$, $\tilde{\Sigma}$ is $p \times p$, and $\tilde{V}^\top$ is $p \times n$.

Our minimum-norm least-squares approximator of A now has representation

$$\hat{A} = X'\tilde{V}\tilde{\Sigma}^{-1}\tilde{U}^\top \tag{6.21}$$

Computing Dominant Eigenvectors of $\hat{A}$

We begin by paralleling a step used to construct Representation 1, define a transition matrix for a rotated $p \times 1$ state $\tilde{b}_t$ by

$$\tilde{A} = \tilde{U}^\top \hat{A}\tilde{U} \tag{6.22}$$

Interpretation as projection coefficients

[Brunton and Kutz, 2022] remark that $\tilde{A}$ can be interpreted in terms of a projection of $\hat{A}$ onto the p modes in $\tilde{U}$.

To verify this, first note that, because $\tilde{U}^\top\tilde{U} = I$, it follows that

$$\tilde{A} = \tilde{U}^\top \hat{A}\tilde{U} = \tilde{U}^\top X'\tilde{V}\tilde{\Sigma}^{-1}\tilde{U}^\top\tilde{U} = \tilde{U}^\top X'\tilde{V}\tilde{\Sigma}^{-1}\tilde{U}^\top \tag{6.23}$$

Next, we'll just compute the regression coefficients in a projection of $\hat{A}$ on $\tilde{U}$ using a standard least-squares formula

$$(\tilde{U}^\top\tilde{U})^{-1}\tilde{U}^\top\hat{A} = (\tilde{U}^\top\tilde{U})^{-1}\tilde{U}^\top X'\tilde{V}\tilde{\Sigma}^{-1}\tilde{U}^\top = \tilde{U}^\top X'\tilde{V}\tilde{\Sigma}^{-1}\tilde{U}^\top = \tilde{A}.$$

Thus, we have verified that $\tilde{A}$ is a least-squares projection of $\hat{A}$ onto $\tilde{U}$.

An Inverse Challenge

Because we are using a reduced SVD, $\tilde{U}\tilde{U}^\top \neq I$.

Consequently,

$$\hat{A} \neq \tilde{U}\tilde{A}\tilde{U}^\top,$$

so we can't simply recover $\hat{A}$ from $\tilde{A}$ and $\tilde{U}$.

A Blind Alley

We can start by hoping for the best and proceeding to construct an eigendecomposition of the $p \times p$ matrix $\tilde{A}$:

$$\tilde{A} = \tilde{W}\Lambda\tilde{W}^{-1} \tag{6.24}$$

where Λ is a diagonal matrix of p eigenvalues and the columns of $\tilde{W}$ are corresponding eigenvectors.

Mimicking our procedure in Representation 2, we cross our fingers and compute an $m \times p$ matrix

$$\tilde{\Phi}_s = \tilde{U}\tilde{W} \tag{6.25}$$

that corresponds to (6.18) for a full SVD.

At this point, where $\hat{A}$ is given by formula (6.21) it is interesting to compute $\hat{A}\tilde{\Phi}_s$:

$$\begin{aligned}\hat{A}\tilde{\Phi}_s &= (X'\tilde{V}\tilde{\Sigma}^{-1}\tilde{U}^\top)(\tilde{U}\tilde{W}) \\ &= X'\tilde{V}\tilde{\Sigma}^{-1}\tilde{W} \\ &\neq (\tilde{U}\tilde{W})\Lambda \\ &= \tilde{\Phi}_s\Lambda\end{aligned}$$

That $\hat{A}\tilde{\Phi}_s \neq \tilde{\Phi}_s\Lambda$ means that, unlike the corresponding situation in Representation 2, columns of $\tilde{\Phi}_s = \tilde{U}\tilde{W}$ are **not** eigenvectors of $\hat{A}$ corresponding to eigenvalues on the diagonal of matix Λ.

An Approach That Works

Continuing our quest for eigenvectors of $\hat{A}$ that we **can** compute with a reduced SVD, let's define an $m \times p$ matrix Φ as

$$\Phi \equiv \hat{A}\tilde{\Phi}_s = X'\tilde{V}\tilde{\Sigma}^{-1}\tilde{W} \tag{6.26}$$

It turns out that columns of Φ **are** eigenvectors of $\hat{A}$.

This is a consequence of a result established by Tu et al. [Tu *et al.*, 2014] that we now present.

Proposition The p columns of Φ are eigenvectors of $\hat{A}$.

Proof: From formula (6.26) we have

$$\begin{aligned}\hat{A}\Phi &= (X'\tilde{V}\tilde{\Sigma}^{-1}\tilde{U}^\top)(X'\tilde{V}\Sigma^{-1}\tilde{W}) \\ &= X'\tilde{V}\tilde{\Sigma}^{-1}\tilde{A}\tilde{W} \\ &= X'\tilde{V}\tilde{\Sigma}^{-1}\tilde{W}\Lambda \\ &= \Phi\Lambda\end{aligned}$$

so that

$$\hat{A}\Phi = \Phi\Lambda. \tag{6.27}$$

Let ϕ_i be the ith column of Φ and λ_i be the corresponding i eigenvalue of $\tilde{A}$ from decomposition (6.24).

Equating the $m \times 1$ vectors that appear on the two sides of equation (6.27) gives

$$\hat{A}\phi_i = \lambda_i\phi_i.$$

This equation confirms that ϕ_i is an eigenvector of $\hat{A}$ that corresponds to eigenvalue λ_i of both $\tilde{A}$ and $\hat{A}$.

This concludes the proof.

Also see [Brunton and Kutz, 2022] (p. 238)

6.5.1 Decoder of $\check{b}$ as a linear projection

From eigendecomposition (6.27) we can represent $\hat{A}$ as

$$\hat{A} = \Phi\Lambda\Phi^+. \tag{6.28}$$

From formula (6.28) we can deduce dynamics of the $p \times 1$ vector $\check{b}_t$:

$$\check{b}_{t+1} = \Lambda\check{b}_t$$

where

$$\check{b}_t = \Phi^+ X_t \tag{6.29}$$

Since the $m \times p$ matrix Φ has p linearly independent columns, the generalized inverse of Φ is

$$\Phi^+ = (\Phi^\top \Phi)^{-1} \Phi^\top$$

and so

$$\check{b} = (\Phi^\top \Phi)^{-1} \Phi^\top X \tag{6.30}$$

The $p \times n$ matrix $\check{b}$ is recognizable as a matrix of least squares regression coefficients of the $m \times n$ matrix X on the $m \times p$ matrix Φ and consequently

$$\check{X} = \Phi \check{b} \tag{6.31}$$

is an $m \times n$ matrix of least squares projections of X on Φ.

Variance Decomposition of X

By virtue of the least-squares projection theory discussed in this quantecon lecture https://python-advanced.quantecon.org/orth_proj.html, we can represent X as the sum of the projection $\check{X}$ of X on Φ plus a matrix of errors.

To verify this, note that the least squares projection $\check{X}$ is related to X by

$$X = \check{X} + \epsilon$$

or

$$X = \Phi \check{b} + \epsilon \tag{6.32}$$

where ϵ is an $m \times n$ matrix of least squares errors satisfying the least squares orthogonality conditions $\epsilon^\top \Phi = 0$ or

$$(X - \Phi \check{b})^\top \Phi = 0_{m \times p} \tag{6.33}$$

Rearranging the orthogonality conditions (6.33) gives $X^\top \Phi = \check{b} \Phi^\top \Phi$, which implies formula (6.30).

6.5.2 An Approximation

We now describe a way to approximate the $p \times 1$ vector $\check{b}_t$ instead of using formula (6.29).

In particular, the following argument adapted from [Brunton and Kutz, 2022] (page 240) provides a computationally efficient way to approximate $\check{b}_t$.

For convenience, we'll apply the method at time $t = 1$.

For $t = 1$, from equation (6.32) we have

$$\check{X}_1 = \Phi \check{b}_1 \tag{6.34}$$

where $\check{b}_1$ is a $p \times 1$ vector.

Recall from representation 1 above that $X_1 = U \tilde{b}_1$, where $\tilde{b}_1$ is a time 1 basis vector for representation 1 and U is from the full SVD $X = U \Sigma V^\top$.

It then follows from equation (6.32) that

$$U \tilde{b}_1 = X' \tilde{V} \tilde{\Sigma}^{-1} \tilde{W} \check{b}_1 + \epsilon_1$$

where ϵ_1 is a least-squares error vector from equation (6.32).

It follows that

$$\tilde{b}_1 = U^\top X' V \tilde{\Sigma}^{-1} \tilde{W} \check{b}_1 + U^\top \epsilon_1$$

Replacing the error term $U^\top \epsilon_1$ by zero, and replacing U from a **full** SVD of X with $\tilde{U}$ from a **reduced** SVD, we obtain an approximation $\hat{b}_1$ to $\tilde{b}_1$:

$$\hat{b}_1 = \tilde{U}^\top X' \tilde{V} \tilde{\Sigma}^{-1} \tilde{W} \check{b}_1$$

Recall that from equation (6.23), $\tilde{A} = \tilde{U}^\top X' \tilde{V} \tilde{\Sigma}^{-1}$.

It then follows that

$$\hat{b}_1 = \tilde{A} \tilde{W} \check{b}_1$$

and therefore, by the eigendecomposition (6.24) of $\tilde{A}$, we have

$$\hat{b}_1 = \tilde{W} \Lambda \check{b}_1$$

Consequently,

$$\hat{b}_1 = (\tilde{W} \Lambda)^{-1} \tilde{b}_1$$

or

$$\hat{b}_1 = (\tilde{W} \Lambda)^{-1} \tilde{U}^\top X_1, \tag{6.35}$$

which is a computationally efficient approximation to the following instance of equation (6.29) for the initial vector $\check{b}_1$:

$$\check{b}_1 = \Phi^+ X_1 \tag{6.36}$$

(To highlight that (6.35) is an approximation, users of DMD sometimes call components of basis vector $\check{b}_t = \Phi^+ X_t$ the **exact** DMD modes and components of $\hat{b}_t = (\tilde{W} \Lambda)^{-1} \tilde{U}^\top X_t$ the **approximate** modes.)

Conditional on X_t, we can compute a decoded $\check{X}_{t+j}, j = 1, 2, \ldots$ from the exact modes via

$$\check{X}_{t+j} = \Phi \Lambda^j \Phi^+ X_t \tag{6.37}$$

or use compute a decoded $\hat{X}_{t+j}$ from approximate modes via

$$\hat{X}_{t+j} = \Phi \Lambda^j (\tilde{W} \Lambda)^{-1} \tilde{U}^\top X_t. \tag{6.38}$$

We can then use a decoded $\check{X}_{t+j}$ or $\hat{X}_{t+j}$ to forecast X_{t+j}.

6.5.3 Using Fewer Modes

In applications, we'll actually use only a few modes, often three or less.

Some of the preceding formulas assume that we have retained all p modes associated with singular values of X.

We can adjust our formulas to describe a situation in which we instead retain only the $r < p$ largest singular values.

In that case, we simply replace $\tilde{\Sigma}$ with the appropriate $r \times r$ matrix of singular values, $\tilde{U}$ with the $m \times r$ matrix whose columns correspond to the r largest singular values, and $\tilde{V}$ with the $n \times r$ matrix whose columns correspond to the r largest singular values.

Counterparts of all of the salient formulas above then apply.

6.6 Source for Some Python Code

You can find a Python implementation of DMD here:

https://mathlab.sissa.it/pydmd

CHAPTER

SEVEN

USING NEWTON'S METHOD TO SOLVE ECONOMIC MODELS

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

Contents

7.1 Overview

Many economic problems involve finding fixed points or zeros (also called "roots") of functions.

For example, in a simple supply and demand model, an equilibrium price is one that makes excess demand zero.

In other words, an equilibrium is a zero of the excess demand function.

There are various computational techniques for solving for fixed points and zeros.

In this lecture we study an important gradient-based technique called Newton's method.

Newton's method does not always work but, in situations where it does, convergence is often fast when compared to other methods.

The lecture will apply Newton's method in one-dimensional and multidimensional settings to solve fixed-point and zero-finding problems.

- When finding the fixed point of a function f, Newton's method updates an existing guess of the fixed point by solving for the fixed point of a linear approximation to the function f.
- When finding the zero of a function f, Newton's method updates an existing guess by solving for the zero of a linear approximation to the function f.

To build intuition, we first consider an easy, one-dimensional fixed point problem where we know the solution and solve it using both successive approximation and Newton's method.

Then we apply Newton's method to multidimensional settings to solve for market equilibria with multiple goods.

At the end of the lecture, we leverage the power of automatic differentiation in `jax` to solve a very high-dimensional equilibrium problem.

We use the following imports in this lecture

```
import matplotlib.pyplot as plt
from typing import NamedTuple
from scipy.optimize import root
import jax.numpy as jnp
import jax

# Enable 64-bit precision
jax.config.update("jax_enable_x64", True)
```

7.2 Fixed point computation using Newton's method

In this section we solve the fixed point of the law of motion for capital in the setting of the Solow growth model.

We will inspect the fixed point visually, solve it by successive approximation, and then apply Newton's method to achieve faster convergence.

7.2.1 The Solow model

In the Solow growth model, assuming Cobb-Douglas production technology and zero population growth, the law of motion for capital is

$$k_{t+1} = g(k_t) \quad \text{where} \quad g(k) := sAk^{\alpha} + (1-\delta)k \tag{7.1}$$

Here

- k_t is capital stock per worker,
- $A, \alpha > 0$ are production parameters with $\alpha < 1$
- $s > 0$ is a savings rate, and
- $\delta \in (0, 1)$ is a rate of depreciation

In this example, we wish to calculate the unique strictly positive fixed point of g, the law of motion for capital.

In other words, we seek a $k^* > 0$ such that $g(k^*) = k^*$.

- Such a k^* is called a steady state, since $k_t = k^*$ implies $k_{t+1} = k^*$.

Using pencil and paper to solve $g(k) = k$, you will be able to confirm that

$$k^* = \left(\frac{sA}{\delta}\right)^{1/(1-\alpha)}$$

7.2.2 Implementation

Let's store our parameters in `NamedTuple` to help us keep our code clean and concise.

```
class SolowParameters(NamedTuple):
    A: float
    s: float
    α: float
    δ: float
```

This function creates a suitable `SolowParameters` with default parameter values.

```
def create_solow_params(A=2.0, s=0.3, α=0.3, δ=0.4):
    """Creates a Solow model parameterization with default values."""
    return SolowParameters(A=A, s=s, α=α, δ=δ)
```

The next two functions implement the law of motion *(7.2.1)* and store the true fixed point k^*.

```
def g(k, params):
    A, s, α, δ = params
    return A * s * k**α + (1 - δ) * k


def exact_fixed_point(params):
    A, s, α, δ = params
    return ((s * A) / δ) ** (1 / (1 - α))
```

Here is a function to provide a 45 degree plot of the dynamics.

```
def plot_45(params, ax, fontsize=14):

    k_min, k_max = 0.0, 3.0
    k_grid = jnp.linspace(k_min, k_max, 1200)

    # Plot the functions
    lb = r"$g(k) = sAk^{\alpha} + (1 - \delta)k$"
    ax.plot(k_grid, g(k_grid, params), lw=2, alpha=0.6, label=lb)
    ax.plot(k_grid, k_grid, "k--", lw=1, alpha=0.7, label="45")

    # Show and annotate the fixed point
    kstar = exact_fixed_point(params)
    fps = (kstar,)
    ax.plot(fps, fps, "go", ms=10, alpha=0.6)
    ax.annotate(
        r"$k^* = (sA / \delta)^{\frac{1}{1-\alpha}}$",
        xy=(kstar, kstar),
        xycoords="data",
        xytext=(20, -20),
        textcoords="offset points",
        fontsize=fontsize,
    )

    ax.legend(loc="upper left", frameon=False, fontsize=fontsize)

    ax.set_yticks((0, 1, 2, 3))
    ax.set_yticklabels((0.0, 1.0, 2.0, 3.0), fontsize=fontsize)
    ax.set_ylim(0, 3)
```

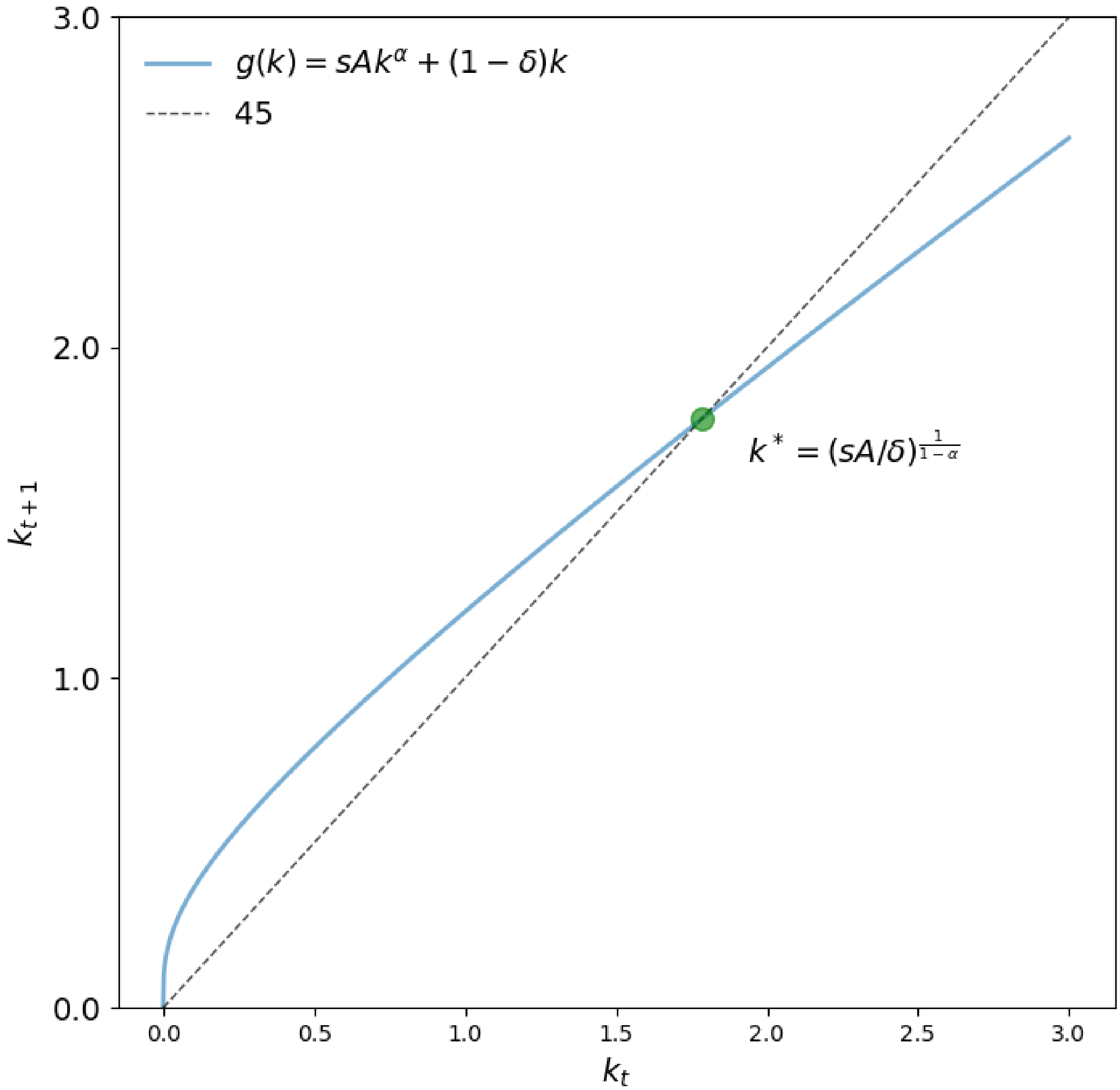
(continues on next page)

(continued from previous page)

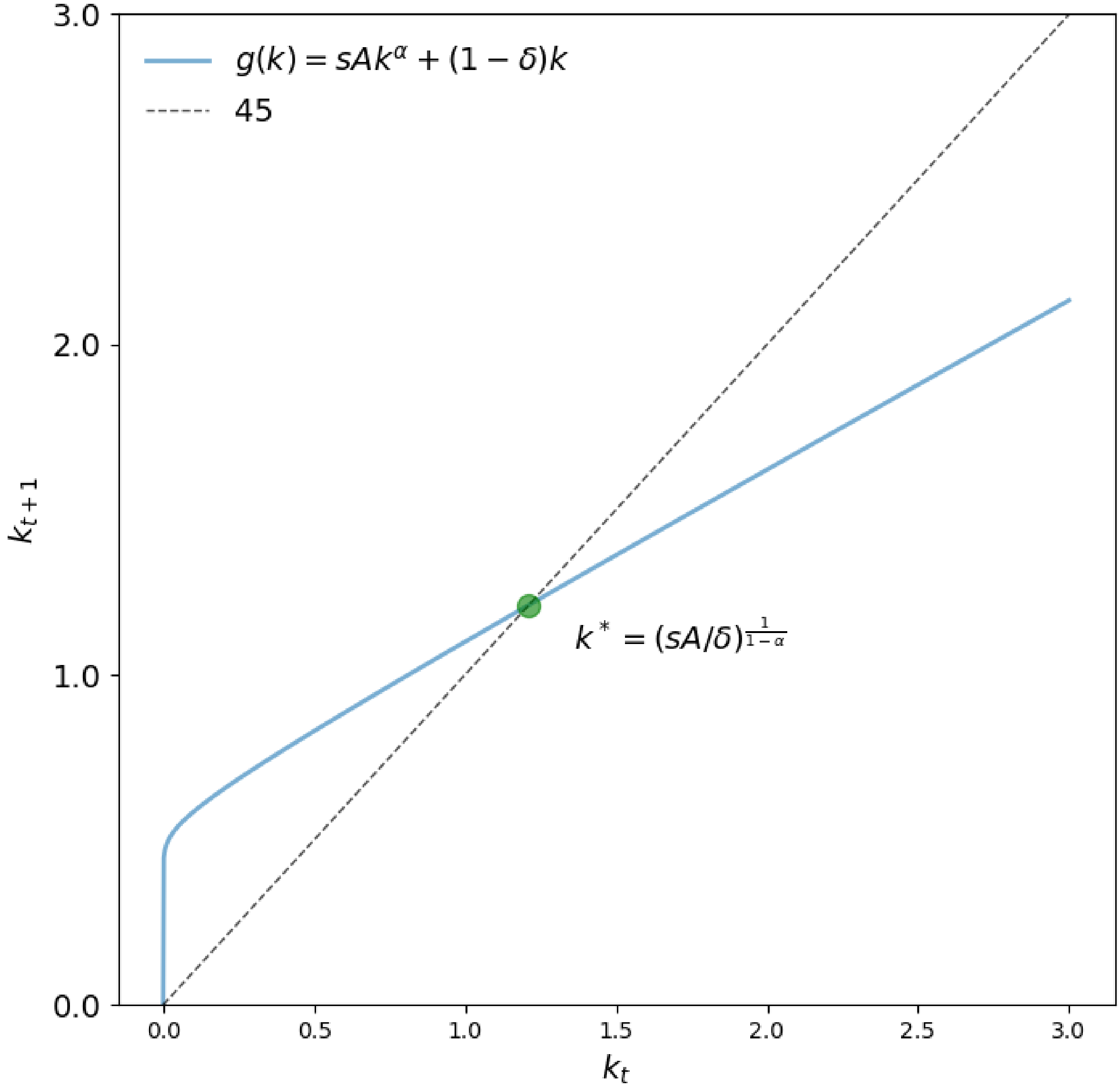
```
    ax.set_xlabel("$k_t$", fontsize=fontsize)
    ax.set_ylabel("$k_{t+1}$", fontsize=fontsize)
```

Let's look at the 45 degree diagram for two parameterizations.

```
params = create_solow_params()
fig, ax = plt.subplots(figsize=(8, 8))
plot_45(params, ax)
plt.show()
```




```
params = create_solow_params(α=0.05, δ=0.5)
fig, ax = plt.subplots(figsize=(8, 8))
plot_45(params, ax)
plt.show()
```



We see that k^* is indeed the unique positive fixed point.

Successive approximation

First let's compute the fixed point using successive approximation.

In this case, successive approximation means repeatedly updating capital from some initial state k_0 using the law of motion.

Here's a time series from a particular choice of k_0.

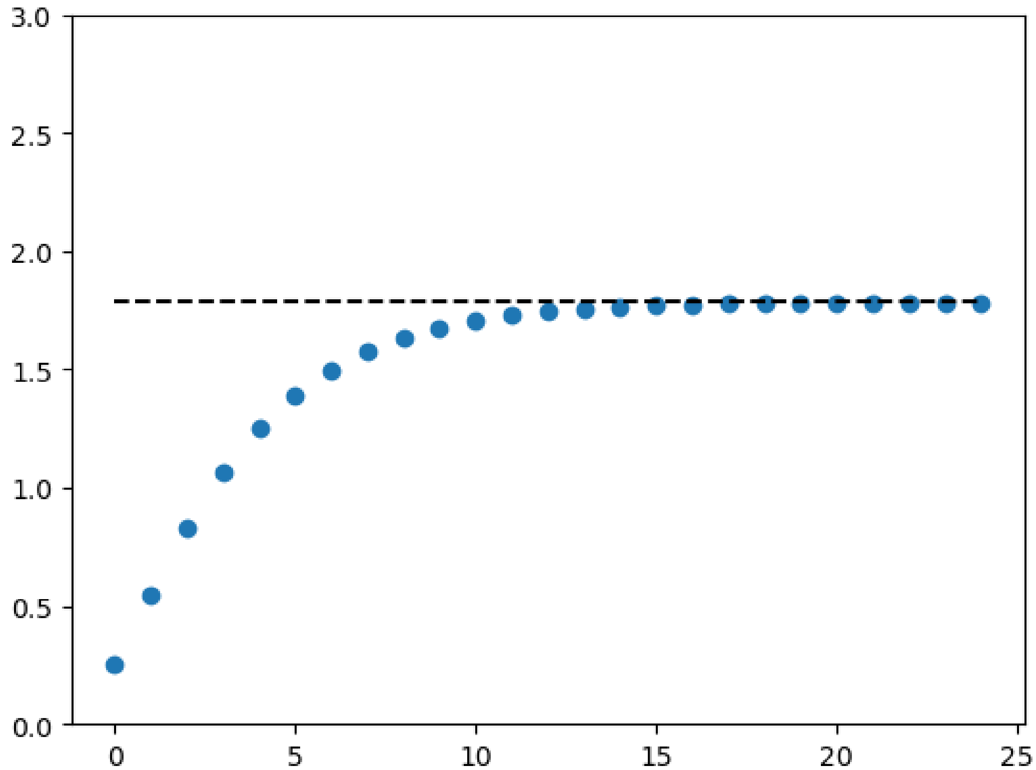
```
def compute_iterates(k_0, f, params, n=25):
    """Compute time series of length n generated by function f."""
    k = k_0
    k_iterates = []
    for t in range(n):
        k_iterates.append(k)
```

(continues on next page)

(continued from previous page)

```
        k = f(k, params)
    return k_iterates
```

```
params = create_solow_params()
k_0 = 0.25
k_series = compute_iterates(k_0, g, params)
k_star = exact_fixed_point(params)

fig, ax = plt.subplots()
ax.plot(k_series, "o")
ax.plot([k_star] * len(k_series), "k--")
ax.set_ylim(0, 3)
plt.show()
```



Let's see the output for a long time series.

```
k_series = compute_iterates(k_0, g, params, n=10_000)
k_star_approx = k_series[-1]
k_star_approx
```

```
1.7846741842265788
```

This is close to the true value.

```
k_star
```

```
1.7846741842265788
```

Newton's method

In general, when applying Newton's fixed point method to some function g, we start with a guess x_0 of the fixed point and then update by solving for the fixed point of a tangent line at x_0.

To begin with, we recall that the first-order approximation of g at x_0 (i.e., the first order Taylor approximation of g at x_0) is the function

$$\hat{g}(x) \approx g(x_0) + g'(x_0)(x - x_0) \tag{7.2}$$

We solve for the fixed point of $\hat{g}$ by calculating the x_1 that solves

$$x_1 = \frac{g(x_0) - g'(x_0)x_0}{1 - g'(x_0)}$$

Generalising the process above, Newton's fixed point method iterates on

$$x_{t+1} = \frac{g(x_t) - g'(x_t)x_t}{1 - g'(x_t)}, \quad x_0 \text{ given} \tag{7.3}$$

To implement Newton's method we observe that the derivative of the law of motion for capital *(7.2.1)* is

$$g'(k) = \alpha s A k^{\alpha - 1} + (1 - \delta) \tag{7.4}$$

Let's define this:

```
def Dg(k, params):
    A, s, α, δ = params
    return α * A * s * k ** (α - 1) + (1 - δ)
```

Here's a function q representing *(7.2.3)*.

```
def q(k, params):
    return (g(k, params) - Dg(k, params) * k) / (1 - Dg(k, params))
```

Now let's plot some trajectories.

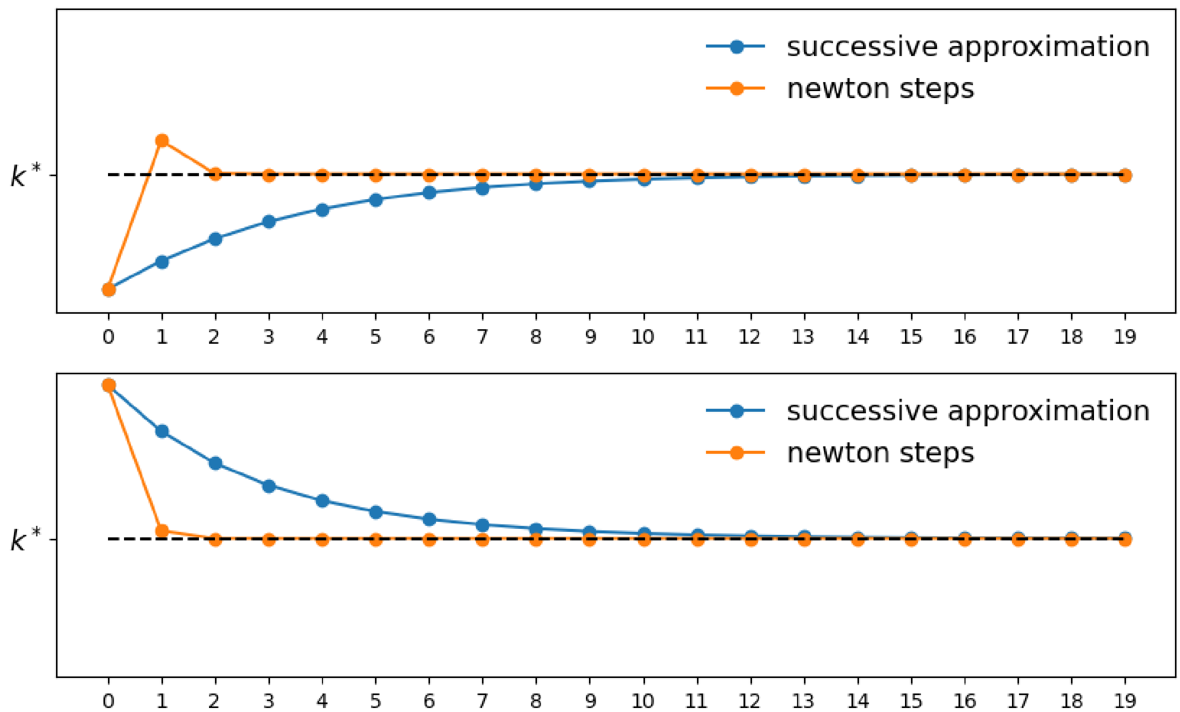
```
def plot_trajectories(
    params,
    k0_a=0.8,  # first initial condition
    k0_b=3.1,  # second initial condition
    n=20,      # length of time series
    fs=14,     # fontsize
):

    fig, axes = plt.subplots(2, 1, figsize=(10, 6))
    ax1, ax2 = axes

    ks1 = compute_iterates(k0_a, g, params, n)
    ax1.plot(ks1, "-o", label="successive approximation")

    ks2 = compute_iterates(k0_b, g, params, n)
    ax2.plot(ks2, "-o", label="successive approximation")

    ks3 = compute_iterates(k0_a, q, params, n)
    ax1.plot(ks3, "-o", label="newton steps")

    ks4 = compute_iterates(k0_b, q, params, n)
```

(continues on next page)

(continued from previous page)

```
    ax2.plot(ks4, "-o", label="newton steps")

    for ax in axes:
        ax.plot(k_star * jnp.ones(n), "k--")
        ax.legend(fontsize=fs, frameon=False)
        ax.set_ylim(0.6, 3.2)
        ax.set_yticks((k_star,))
        ax.set_yticklabels(("$k^*$",), fontsize=fs)
        ax.set_xticks(jnp.linspace(0, 19, 20))

    plt.show()
```

```
params = create_solow_params()
plot_trajectories(params)
```



We can see that Newton's method converges faster than successive approximation.

7.3 Root-Finding in one dimension

In the previous section we computed fixed points.

In fact Newton's method is more commonly associated with the problem of finding zeros of functions.

Let's discuss this "root-finding" problem and then show how it is connected to the problem of finding fixed points.

7.3.1 Newton's method for zeros

Let's suppose we want to find an x such that $f(x) = 0$ for some smooth function f mapping real numbers to real numbers.

Suppose we have a guess x_0 and we want to update it to a new point x_1.

As a first step, we take the first-order approximation of f around x_0:

$$\hat{f}(x) \approx f\left(x_0\right) + f'\left(x_0\right)\left(x - x_0\right)$$

Now we solve for the zero of $\hat{f}$.

In particular, we set $\hat{f}(x_1) = 0$ and solve for x_1 to get

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}, \quad x_0 \text{ given}$$

Generalizing the formula above, for one-dimensional zero-finding problems, Newton's method iterates on

$$x_{t+1} = x_t - \frac{f(x_t)}{f'(x_t)}, \quad x_0 \text{ given} \tag{7.5}$$

The following code implements the iteration *(7.3.1)*

```
def newton(f, x_0, tol=1e-7, max_iter=100_000):
    x = x_0
    Df = jax.grad(f)

    # Implement the zero-finding formula
    @jax.jit
    def q(x):
        return x - f(x) / Df(x)

    error = tol + 1
    n = 0
    while error > tol:
        n += 1
        if n > max_iter:
            raise Exception("Max iteration reached without convergence")
        y = q(x)
        error = jnp.abs(x - y)
        x = y
        print(f"iteration {n}, error = {error:.5f}")
    return x.item()
```

Numerous libraries implement Newton's method in one dimension, including SciPy, so the code is just for illustrative purposes.

(That said, when we want to apply Newton's method using techniques such as automatic differentiation or GPU acceleration, it will be helpful to know how to implement Newton's method ourselves.)

7.3.2 Application to finding fixed points

Now consider again the Solow fixed-point calculation, where we solve for k satisfying $g(k) = k$.

We can convert to this to a zero-finding problem by setting $f(x) := g(x) - x$.

Any zero of f is clearly a fixed point of g.

Let's apply this idea to the Solow problem

```
params = create_solow_params()
k_star_approx_newton = newton(f = lambda x: g(x, params) - x, x_0=0.8)
```

```
iteration 1, error = 1.27209
iteration 2, error = 0.28180
iteration 3, error = 0.00561
iteration 4, error = 0.00000
iteration 5, error = 0.00000
```

```
k_star_approx_newton
```

```
1.7846741842265788
```

The result confirms convergence we saw in the graphs above: a very accurate result is reached with only 5 iterations.

7.4 Multivariate Newton's method

In this section, we introduce a two-good problem, present a visualization of the problem, and solve for the equilibrium of the two-good market using both a zero finder in `SciPy` and Newton's method.

We then expand the idea to a larger market with 5,000 goods and compare the performance of the two methods again.

We will see a significant performance gain when using Newton's method.

7.4.1 A two-goods market equilibrium

Let's start by computing the market equilibrium of a two-good problem.

We consider a market for two related products, good 0 and good 1, with price vector $p = (p_0, p_1)$

Supply of good i at price p is

$$q_i^s(p) = b_i \sqrt{p_i}$$

Demand of good i at price p is

$$q_i^d(p) = \exp(-(a_{i0} p_0 + a_{i1} p_1)) + c_i$$

Here c_i, b_i and a_{ij} are parameters.

For example, the two goods might be computer components that are typically used together, in which case they are complements. Hence demand depends on the price of both components.

The excess demand function is

$$e_i(p) = q_i^d(p) - q_i^s(p), \quad i = 0, 1$$

An equilibrium price vector p^* satisfies $e_i(p^*) = 0$.

We set

$$A = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix}, \qquad b = \begin{bmatrix} b_0 \\ b_1 \end{bmatrix} \qquad \text{and} \qquad c = \begin{bmatrix} c_0 \\ c_1 \end{bmatrix}$$

for this particular question.

A graphical exploration

Since our problem is only two-dimensional, we can use graphical analysis to visualize and help understand the problem.

Our first step is to define the excess demand function

$$e(p) = \begin{bmatrix} e_0(p) \\ e_1(p) \end{bmatrix}$$

The function below calculates the excess demand for given parameters

```
@jax.jit
def e(p, A, b, c):
    return jnp.exp(-A @ p) + c - b * jnp.sqrt(p)
```

Our default parameter values will be

$$A = \begin{bmatrix} 0.5 & 0.4 \\ 0.8 & 0.2 \end{bmatrix}, \qquad b = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \qquad \text{and} \qquad c = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

```
A = jnp.array([[0.5, 0.4], [0.8, 0.2]])
b = jnp.ones(2)
c = jnp.ones(2)
```

At a price level of $p = (1, 0.5)$, the excess demand is

```
p = jnp.array([1, 0.5])
ex_demand = e(p, A, b, c)

print(
    f"The excess demand for good 0 is {ex_demand[0]:.3f} \n"
    f"The excess demand for good 1 is {ex_demand[1]:.3f}"
)
```

```
The excess demand for good 0 is 0.497
The excess demand for good 1 is 0.699
```

To increase the efficiency of computation, we will use the power of vectorization using `jax.vmap`. This is much faster than the python loops.

```
# Create vectorization on the first axis of p.
e_vectorized_p_1 = jax.vmap(e, in_axes=(0, None, None, None))

# Create vectorization on the second axis of p.
e_vectorized = jax.vmap(e_vectorized_p_1, in_axes=(0, None, None, None))
```

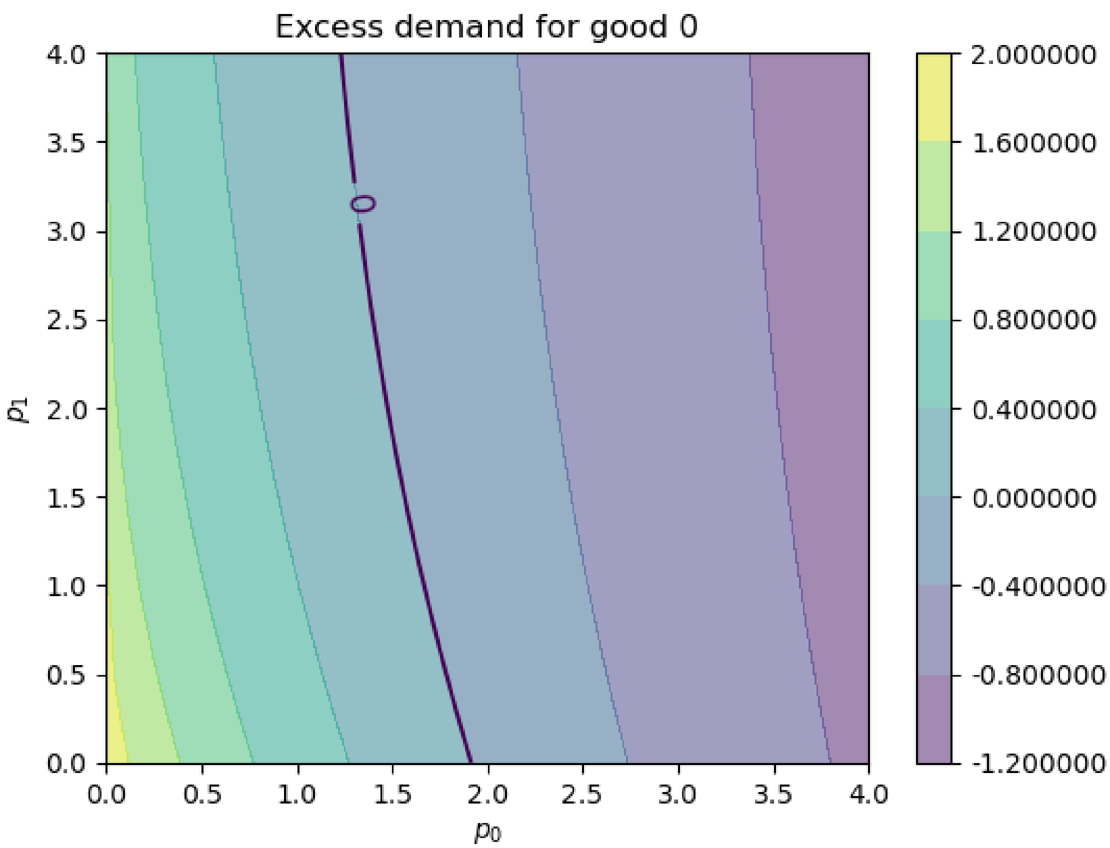
Next we plot the two functions e_0 and e_1 on a grid of (p_0, p_1) values, using contour surfaces and lines.

We will use the following function to build the contour plots

```python
def plot_excess_demand(ax, good=0, grid_size=100, grid_max=4, surface=True):
    p_grid = jnp.linspace(0, grid_max, grid_size)

    # Create meshgrid for all combinations of p_1 and p_2
    P1, P2 = jnp.meshgrid(p_grid, p_grid, indexing="ij")

    # Stack to create array of shape (grid_size, grid_size, 2)
    P = jnp.stack([P1, P2], axis=-1)

    # Compute all values at once using vectorized function
    z_full = e_vectorized(P, A, b, c)
    z = z_full[:, :, good]

    if surface:
        cs1 = ax.contourf(p_grid, p_grid, z.T, alpha=0.5)
        plt.colorbar(cs1, ax=ax, format="%.6f")

    ctr1 = ax.contour(p_grid, p_grid, z.T, levels=[0.0])
    ax.set_xlabel("$p_0$")
    ax.set_ylabel("$p_1$")
    ax.set_title(f"Excess demand for good {good}")
    plt.clabel(ctr1, inline=1, fontsize=13)
```

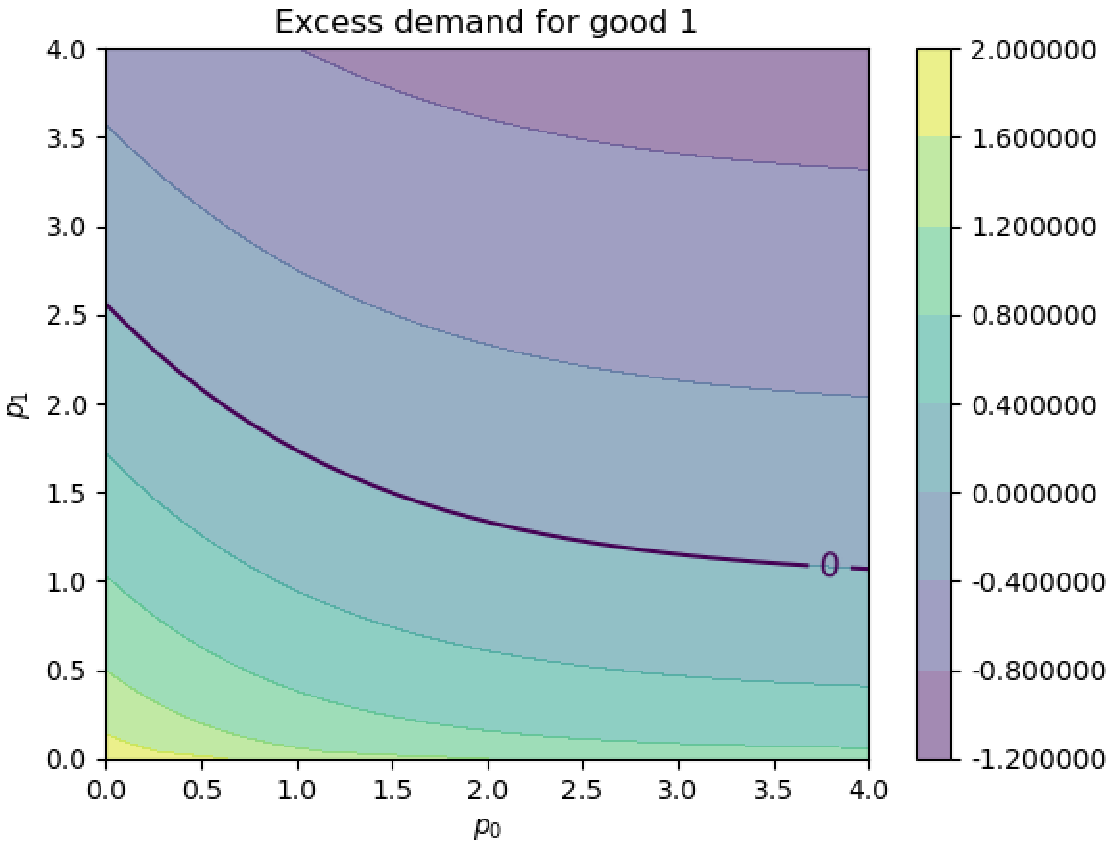
Here's our plot of e_0:

```python
fig, ax = plt.subplots()
plot_excess_demand(ax, good=0)
plt.show()
```



Here's our plot of e_1:

```
fig, ax = plt.subplots()
plot_excess_demand(ax, good=1)
plt.show()
```

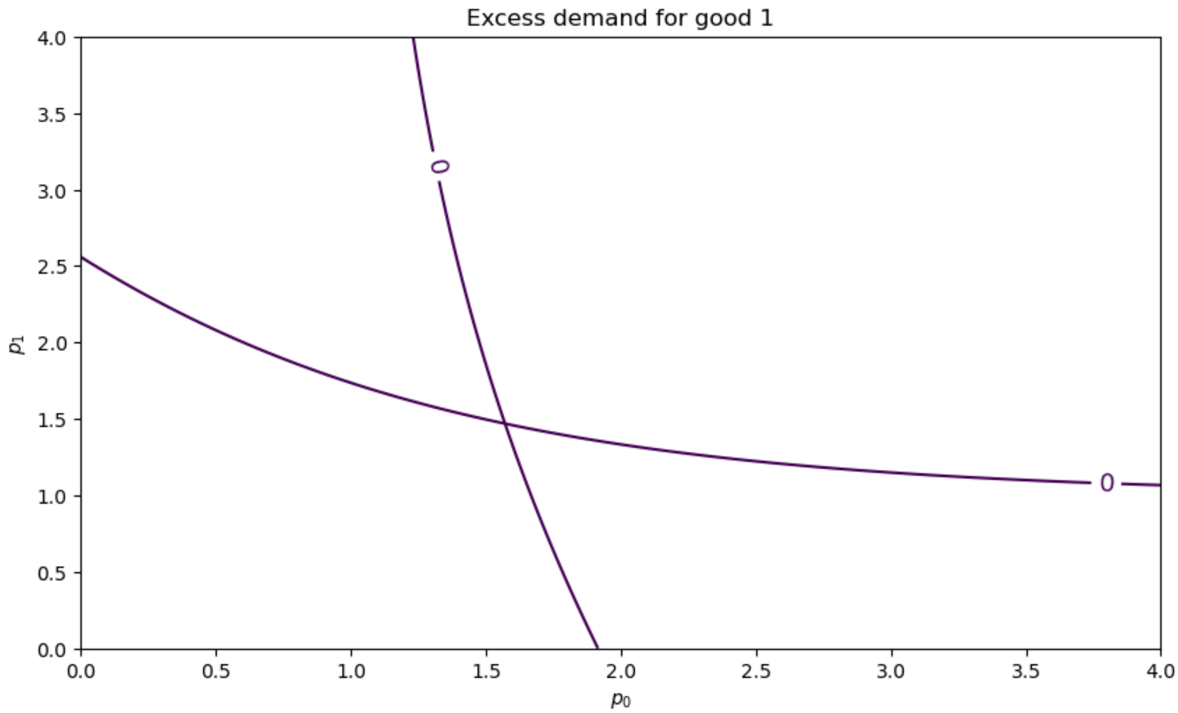


We see the black contour line of zero, which tells us when $e_i(p) = 0$.

For a price vector p such that $e_i(p) = 0$ we know that good i is in equilibrium (demand equals supply).

If these two contour lines cross at some price vector p^*, then p^* is an equilibrium price vector.

```
fig, ax = plt.subplots(figsize=(10, 5.7))
for good in (0, 1):
    plot_excess_demand(ax, good=good, surface=False)
plt.show()
```



It seems there is an equilibrium close to $p = (1.6, 1.5)$.

Using a multidimensional root finder

To solve for p^* more precisely, we use a zero-finding algorithm from `scipy.optimize`.

We supply $p = (1, 1)$ as our initial guess.

```
init_p = jnp.ones(2)
```

This uses the modified Powell method to find the zero

```
%%time
solution = root(lambda p: e(p, A, b, c), init_p, method="hybr")
```

```
CPU times: user 7.06 ms, sys: 626 µs, total: 7.68 ms
Wall time: 4.37 ms
```

Here's the resulting value:

```
p = solution.x
p
```

```
array([1.57080182, 1.46928838])
```

This looks close to our guess from observing the figure. We can plug it back into e to test that $e(p) \approx 0$:

```
e_p = jnp.max(jnp.abs(e(p, A, b, c)))
e_p.item()
```

```
2.0383694732117874e-13
```

This is indeed a very small error.

Adding gradient information

In many cases, for zero-finding algorithms applied to smooth functions, supplying the Jacobian of the function leads to better convergence properties.

Here, we manually calculate the elements of the Jacobian

$$J(p) = \begin{bmatrix} \frac{\partial e_0}{\partial p_0}(p) & \frac{\partial e_0}{\partial p_1}(p) \\ \frac{\partial e_1}{\partial p_0}(p) & \frac{\partial e_1}{\partial p_1}(p) \end{bmatrix}$$

```
def jacobian_e(p, A, b, c):
    p_0, p_1 = p
    a_00, a_01 = A[0, :]
    a_10, a_11 = A[1, :]
    j_00 = -a_00 * jnp.exp(-a_00 * p_0) - (b[0] / 2) * p_0 ** (-1 / 2)
    j_01 = -a_01 * jnp.exp(-a_01 * p_1)
    j_10 = -a_10 * jnp.exp(-a_10 * p_0)
    j_11 = -a_11 * jnp.exp(-a_11 * p_1) - (b[1] / 2) * p_1 ** (-1 / 2)
    J = [[j_00, j_01], [j_10, j_11]]
    return jnp.array(J)
```

```
%%time
solution = root(
    lambda p: e(p, A, b, c),
    init_p,
    jac = lambda p: jacobian_e(p, A, b, c),
    method="hybr",
)
```

```
CPU times: user 260 ms, sys: 17.4 ms, total: 277 ms
Wall time: 391 ms
```

Now the solution is even more accurate (although, in this low-dimensional problem, the difference is quite small):

```
p = solution.x
e_p = jnp.max(jnp.abs(e(p, A, b, c)))
e_p.item()
```

```
1.3322676295501878e-15
```

Using Newton's method

Now let's use Newton's method to compute the equilibrium price using the multivariate version of Newton's method

$$p_{n+1} = p_n - J_e(p_n)^{-1} e(p_n) \tag{7.6}$$

This is a multivariate version of *(7.3.1)*

(Here $J_e(p_n)$ is the Jacobian of e evaluated at p_n.)

The iteration starts from some initial guess of the price vector p_0.

Here, instead of coding Jacobian by hand, we use the `jacobian()` function in the `jax` library to auto-differentiate and calculate the Jacobian.

With only slight modification, we can generalize *our previous attempt* to multidimensional problems

```
def newton(f, x_0, tol=1e-5, max_iter=10):
    x = x_0
    f_jac = jax.jacobian(f)

    @jax.jit
    def q(x):
        return x - jnp.linalg.solve(f_jac(x), f(x))

    error = tol + 1
    n = 0
    while error > tol:
        n += 1
        if n > max_iter:
            raise Exception("Max iteration reached without convergence")
        y = q(x)
        if any(jnp.isnan(y)):
            raise Exception("Solution not found with NaN generated")
        error = jnp.linalg.norm(x - y)
        x = y
        print(f"iteration {n}, error = {error:.5f}")
    print("\n" + f"Result = {x} \n")
    return x
```

We find the algorithm terminates in 4 steps

```
%%time
p = newton(lambda p: e(p, A, b, c), init_p)
```

```
iteration 1, error = 0.62515
iteration 2, error = 0.11152
iteration 3, error = 0.00258
iteration 4, error = 0.00000

Result = [1.57080182 1.46928838]

CPU times: user 331 ms, sys: 22.8 ms, total: 354 ms
Wall time: 449 ms
```

```
e_p = jnp.max(jnp.abs(e(p, A, b, c)))
e_p.item()
```

```
1.4632739464559563e-13
```

The result is very accurate.

With the larger overhead, the speed is not better than the optimized `scipy` function.

7.4.2 A high-dimensional problem

Our next step is to investigate a large market with 3,000 goods.

The excess demand function is essentially the same, but now the matrix A is 3000×3000 and the parameter vectors b and c are 3000×1.

```
dim = 3000

# Create JAX random key
key = jax.random.PRNGKey(0)

# Create a random matrix A and normalize the columns to sum to one
A = jax.random.uniform(key, (dim, dim))
s = jnp.sum(A, axis=0)
A = A / s

# Set up b and c
b = jnp.ones(dim)
c = jnp.ones(dim)
```

Here's our initial condition

```
init_p = jnp.ones(dim)
```

```
%%time
p = newton(lambda p: e(p, A, b, c), init_p)
```

```
iteration 1, error = 23.22262
iteration 2, error = 3.94537
iteration 3, error = 0.08500
iteration 4, error = 0.00004
iteration 5, error = 0.00000

Result = [1.50723773 1.51041603 1.50134795 ... 1.49941629 1.49033692 1.49666807]

CPU times: user 7.65 s, sys: 1.89 s, total: 9.54 s
Wall time: 9.18 s
```

```
e_p = jnp.max(jnp.abs(e(p, A, b, c)))
e_p.item()
```

```
4.440892098500626e-16
```

With the same tolerance, we compare the runtime and accuracy of Newton's method to SciPy's `root` function

```
%%time
solution = root(
    lambda p: e(p, A, b, c),
    init_p,
    jac = lambda p: jax.jacobian(e)(p, A, b, c),
    method="hybr",
    tol=1e-5,
)
```

```
CPU times: user 36.3 s, sys: 99.6 ms, total: 36.4 s
Wall time: 36.8 s
```

```
p = solution.x
e_p = jnp.max(jnp.abs(e(p, A, b, c)))
e_p.item()
```

```
9.209231102147442e-07
```

7.5 Exercises

Exercise 7.5.1

Consider a three-dimensional extension of the Solow fixed point problem with

$$A = \begin{bmatrix} 2 & 3 & 3 \\ 2 & 4 & 2 \\ 1 & 5 & 1 \end{bmatrix}, \quad s = 0.2, \quad \alpha = 0.5, \quad \delta = 0.8$$

As before the law of motion is

$$k_{t+1} = g(k_t) \quad \text{where} \quad g(k) := sAk^\alpha + (1 - \delta)k$$

However, k_t is now a 3×1 vector.

Solve for the fixed point using Newton's method with the following initial values:

$$\begin{aligned} k1_0 &= (1, 1, 1) \\ k2_0 &= (3, 5, 5) \\ k3_0 &= (50, 50, 50) \end{aligned}$$

Hint

- The computation of the fixed point is equivalent to computing k^* such that $g(k^*) - k^* = 0$.
- If you are unsure about your solution, you can start with the solved example:

$$A = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{bmatrix}$$

with $s = 0.3$, $\alpha = 0.3$, and $\delta = 0.4$ and starting value:

$$k_0 = (1, 1, 1)$$

The result should converge to the *analytical solution*.

Solution

Let's first define the parameters for this problem

```
A = jnp.array([[2.0, 3.0, 3.0], [2.0, 4.0, 2.0], [1.0, 5.0, 1.0]])

s = 0.2
α = 0.5
δ = 0.8

initLs = [jnp.ones(3), jnp.array([3.0, 5.0, 5.0]), jnp.repeat(50.0, 3)]
```

Then define the multivariate version of the formula for the *(7.2.1)*

```
@jax.jit
def multivariate_solow(k, A=A, s=s, α=α, δ=δ):
    return s * jnp.dot(A, k**α) + (1 - δ) * k
```

Let's run through each starting value and see the output

```
attempt = 1
for init in initLs:
    print(f'Attempt {attempt}: Starting value is {init} \n')
    %time k = newton(lambda k: multivariate_solow(k) - k, \
                    init)
    print('-'*64)
    attempt += 1
```

```
Attempt 1: Starting value is [1. 1. 1.]
iteration 1, error = 50.49630
iteration 2, error = 41.10937
iteration 3, error = 4.29413
iteration 4, error = 0.38543
iteration 5, error = 0.00544
iteration 6, error = 0.00000

Result = [3.84058108 3.87071771 3.41091933]

CPU times: user 311 ms, sys: 17.2 ms, total: 328 ms
Wall time: 404 ms
----------------------------------------------------------------
Attempt 2: Starting value is [3. 5. 5.]

iteration 1, error = 2.07011
iteration 2, error = 0.12642
iteration 3, error = 0.00060
iteration 4, error = 0.00000

Result = [3.84058108 3.87071771 3.41091933]

CPU times: user 119 ms, sys: 5.74 ms, total: 125 ms
Wall time: 137 ms
----------------------------------------------------------------
Attempt 3: Starting value is [50. 50. 50.]
iteration 1, error = 73.00943
iteration 2, error = 6.49379
iteration 3, error = 0.68070
iteration 4, error = 0.01620
iteration 5, error = 0.00001
iteration 6, error = 0.00000

Result = [3.84058108 3.87071771 3.41091933]

CPU times: user 277 ms, sys: 11.4 ms, total: 288 ms
Wall time: 327 ms
```

```
----------------------------------------------------------------
```

We find that the results are invariant to the starting values given the well-defined property of this question.

But the number of iterations it takes to converge is dependent on the starting values.

Let's substitute the output back into the formula to check our last result

```
multivariate_solow(k) - k
```

```
Array([0., 0., 0.], dtype=float64)
```

Note the error is very small.

We can also test our results on the known solution

```
A = jnp.array([[2.0, 0.0, 0.0], [0.0, 2.0, 0.0], [0.0, 0.0, 2.0]])

s = 0.3
α = 0.3
δ = 0.4

init = jnp.repeat(1.0, 3)
```

```
%%time

k = newton(lambda k: multivariate_solow(k, A=A, s=s, α=α, δ=δ) - k, init)
```

```
iteration 1, error = 1.57459
iteration 2, error = 0.21345
iteration 3, error = 0.00205
iteration 4, error = 0.00000

Result = [1.78467418 1.78467418 1.78467418]

CPU times: user 272 ms, sys: 9.63 ms, total: 282 ms
Wall time: 322 ms
```

The result is very close to the ground truth but still slightly different.

```
%%time

k = newton(
    lambda k: multivariate_solow(k, A=A, s=s, α=α, δ=δ) - k, init, tol=1e-7
)
```

```
iteration 1, error = 1.57459
iteration 2, error = 0.21345
iteration 3, error = 0.00205
iteration 4, error = 0.00000
iteration 5, error = 0.00000

Result = [1.78467418 1.78467418 1.78467418]

CPU times: user 240 ms, sys: 12 ms, total: 252 ms
Wall time: 279 ms
```

We can see it steps towards a more accurate solution.

Exercise 7.5.2

In this exercise, let's try different initial values and check how Newton's method responds to different starting points.

Let's define a three-good problem with the following default values:

$$A = \begin{bmatrix} 0.2 & 0.1 & 0.7 \\ 0.3 & 0.2 & 0.5 \\ 0.1 & 0.8 & 0.1 \end{bmatrix}, \qquad b = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \qquad \text{and} \qquad c = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

For this exercise, use the following extreme price vectors as initial values:

$$\begin{aligned} p1_0 &= (5, 5, 5) \\ p2_0 &= (1, 1, 1) \\ p3_0 &= (4.5, 0.1, 4) \end{aligned}$$

Set the tolerance to $1e-15$ for more accurate output.

Solution

Define parameters and initial values

```
A = jnp.array([[0.2, 0.1, 0.7], [0.3, 0.2, 0.5], [0.1, 0.8, 0.1]])
b = jnp.array([1.0, 1.0, 1.0])
c = jnp.array([1.0, 1.0, 1.0])

initLs = [jnp.repeat(5.0, 3), jnp.ones(3), jnp.array([4.5, 0.1, 4.0])]
```

Let's run through each initial guess and check the output

```
for attempt, init in enumerate(initLs, start=1):
    print(f"Attempt {attempt}: Starting value is {init} \n")
    %time p = newton(lambda p: e(p, A, b, c), init, tol=1e-15, max_iter=15)
    print("-" * 64)
```

```
Attempt 1: Starting value is [5. 5. 5.]
iteration 1, error = 9.24381
CPU times: user 299 ms, sys: 13.2 ms, total: 312 ms
Wall time: 360 ms
```

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
Cell In[51], line 3
      1 for attempt, init in enumerate(initLs, start=1):
      2     print(f"Attempt {attempt}: Starting value is {init} \n")
----> 3     get_ipython().run_line_magic('time', 'p = newton(lambda p: e(p, A,␣
↪b, c), init, tol=1e-15, max_iter=15)')
      4     print("-" * 64)

File ~/miniconda3/envs/quantecon/lib/python3.13/site-packages/IPython/core/
↪interactiveshell.py:2504, in InteractiveShell.run_line_magic(self, magic_name,
↪ line, _stack_depth)
   2502     kwargs['local_ns'] = self.get_local_scope(stack_depth)
   2503 with self.builtin_trap:
-> 2504     result = fn(*args, **kwargs)
   2506 # The code below prevents the output from being displayed
   2507 # when using magics with decorator @output_can_be_silenced
   2508 # when the last Python token in the expression is a ';'.
   2509 if getattr(fn, magic.MAGIC_OUTPUT_CAN_BE_SILENCED, False):

File ~/miniconda3/envs/quantecon/lib/python3.13/site-packages/IPython/core/
```

```
↪magics/execution.py:1452, in ExecutionMagics.time(self, line, cell, local_ns)
   1450 if interrupt_occured:
   1451     if exit_on_interrupt and captured_exception:
-> 1452         raise captured_exception
   1453     return
   1454 return out

File ~/miniconda3/envs/quantecon/lib/python3.13/site-packages/IPython/core/
↪magics/execution.py:1416, in ExecutionMagics.time(self, line, cell, local_ns)
   1414 st = clock2()
   1415 try:
-> 1416     exec(code, glob, local_ns)
   1417     out = None
   1418     # multi-line %%time case

File <timed exec>:1

Cell In[34], line 17, in newton(f, x_0, tol, max_iter)
     15 y = q(x)
     16 if any(jnp.isnan(y)):
---> 17     raise Exception("Solution not found with NaN generated")
     18 error = jnp.linalg.norm(x - y)
     19 x = y

Exception: Solution not found with NaN generated
```

We can see that Newton's method may fail for some starting values.

Sometimes it may take a few initial guesses to achieve convergence.

Substitute the result back to the formula to check our result using the second initial guess which converges

```
p_solution = newton(lambda p: e(p, A, b, c), initLs[1], tol=1e-15, max_iter=15)
e(p_solution, A, b, c)
```

```
iteration 1, error = 0.73419
iteration 2, error = 0.12472
iteration 3, error = 0.00269
iteration 4, error = 0.00000
iteration 5, error = 0.00000
iteration 6, error = 0.00000

Result = [1.49744442 1.49744442 1.49744442]
```

```
Array([0.00000000e+00, 0.00000000e+00, 2.22044605e-16], dtype=float64)
```

We can see the result is very accurate.

Part II

Elementary Statistics

CHAPTER

EIGHT

ELEMENTARY PROBABILITY WITH MATRICES

This lecture uses matrix algebra to illustrate some basic ideas about probability theory.

After introducing underlying objects, we'll use matrices and vectors to describe probability distributions.

Among concepts that we'll be studying include

- a joint probability distribution
- marginal distributions associated with a given joint distribution
- conditional probability distributions
- statistical independence of two random variables
- joint distributions associated with a prescribed set of marginal distributions
  - couplings
  - copulas
- the probability distribution of a sum of two independent random variables
  - convolution of marginal distributions
- parameters that define a probability distribution
- sufficient statistics as data summaries

We'll use a matrix to represent a bivariate or multivariate probability distribution and a vector to represent a univariate probability distribution

This *companion lecture* describes some popular probability distributions and describes how to use Python to sample from them.

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install prettytable
```

As usual, we'll start with some imports

```
import numpy as np
import matplotlib.pyplot as plt
import prettytable as pt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib_inline.backend_inline import set_matplotlib_formats
set_matplotlib_formats('retina')
```

8.1 Sketch of Basic Concepts

We'll briefly define what we mean by a **probability space**, a **probability measure**, and a **random variable**.

For most of this lecture, we sweep these objects into the background

> **Note**
>
> Nevertheless, they'll be lurking beneath **induced distributions** of random variables that we'll focus on here. These deeper objects are essential for defining and analysing the concepts of stationarity and ergodicity that underly laws of large numbers. For a relatively nontechnical presentation of some of these results see this chapter from Lars Peter Hansen and Thomas J. Sargent's online monograph titled "Risk, Uncertainty, and Values":https://lphansen.github.io/QuantMFR/book/1_stochastic_processes.html.

Let Ω be a set of possible underlying outcomes and let $\omega \in \Omega$ be a particular underlying outcomes.

Let $\mathcal{G} \subset \Omega$ be a subset of Ω.

Let $\mathcal{F}$ be a collection of such subsets $\mathcal{G} \subset \Omega$.

The pair $\Omega, \mathcal{F}$ forms our **probability space** on which we want to put a probability measure.

A **probability measure** μ maps a set of possible underlying outcomes $\mathcal{G} \in \mathcal{F}$ into a scalar number between 0 and 1

- this is the "probability" that X belongs to A, denoted by $\text{Prob}\{X \in A\}$.

A **random variable** $X(\omega)$ is a function of the underlying outcome $\omega \in \Omega$.

The random variable $X(\omega)$ has a **probability distribution** that is induced by the underlying probability measure μ and the function $X(\omega)$:

$$\text{Prob}(X \in A) = \int_{\mathcal{G}} \mu(\omega) d\omega \tag{8.1}$$

where $\mathcal{G}$ is the subset of Ω for which $X(\omega) \in A$.

We call this the induced probability distribution of random variable X.

Instead of working explicitly with an underlying probability space $\Omega, \mathcal{F}$ and probability measure μ, applied statisticians often proceed simply by specifying a form for an induced distribution for a random variable X.

That is how we'll proceed in this lecture and in many subsequent lectures.

8.2 What Does Probability Mean?

Before diving in, we'll say a few words about what probability theory means and how it connects to statistics.

We also touch on these topics in the quantecon lectures https://python.quantecon.org/prob_meaning.html and https://python.quantecon.org/navy_captain.html.

For much of this lecture we'll be discussing fixed "population" probabilities.

These are purely mathematical objects.

To appreciate how statisticians connect probabilities to data, the key is to understand the following concepts:

- A single draw from a probability distribution
- Repeated independently and identically distributed (i.i.d.) draws of "samples" or "realizations" from the same probability distribution

- A **statistic** defined as a function of a sequence of samples
- An **empirical distribution** or **histogram** (a binned empirical distribution) that records observed **relative frequencies**
- The idea that a population probability distribution is what we anticipate **relative frequencies** will be in a long sequence of i.i.d. draws. Here the following mathematical machinery makes precise what is meant by **anticipated relative frequencies**
  - **Law of Large Numbers (LLN)**
  - **Central Limit Theorem (CLT)**

Scalar example

Let X be a scalar random variable that takes on the I possible values $0, 1, 2, \ldots, I-1$ with probabilities

$$\text{Prob}(X = i) = f_i,$$

where

$$f_i \geqslant 0, \quad \sum_i f_i = 1.$$

We sometimes write

$$X \sim \{f_i\}_{i=0}^{I-1}$$

as a short-hand way of saying that the random variable X is described by the probability distribution $\{f_i\}_{i=0}^{I-1}$.

Consider drawing a sample $x_0, x_1, \ldots, x_{N-1}$ of N independent and identically distributoed draws of X.

What do the "identical" and "independent" mean in IID or iid ("identically and independently distributed")?

- "identical" means that each draw is from the same distribution.
- "independent" means that joint distribution equal products of marginal distributions, i.e.,

$$\begin{aligned} \text{Prob}\{x_0 = i_0, x_1 = i_1, \ldots, x_{N-1} = i_{N-1}\} &= \text{Prob}\{x_0 = i_0\} \cdot \dots \cdot \text{Prob}\{x_{I-1} = i_{I-1}\} \\ &= f_{i_0} f_{i_1} \cdot \dots \cdot f_{i_{N-1}} \end{aligned}$$

We define an **empirical distribution** as follows.

For each $i = 0, \ldots, I-1$, let

$$\begin{aligned} N_i &= \text{number of times } X = i, \\ N &= \sum_{i=0}^{I-1} N_i \quad \text{total number of draws}, \\ \tilde{f}_i &= \frac{N_i}{N} \sim \text{ frequency of draws for which } X = i \end{aligned}$$

Key concepts that connect probability theory with statistics are laws of large numbers and central limit theorems

LLN:

- A Law of Large Numbers (LLN) states that $\tilde{f}_i \to f_i$ as $N \to \infty$

CLT:

- A Central Limit Theorem (CLT) describes a **rate** at which $\tilde{f}_i \to f_i$

Remarks

- For "frequentist" statisticians, **anticipated relative frequency** is **all** that a probability distribution means.
- But for a Bayesian it means something else – something partly subjective and purely personal.
  - we say "partly" because a Bayesian also pays attention to relative frequencies

8.3 Representing Probability Distributions

A probability distribution $\text{Prob}(X \in A)$ can be described by its **cumulative distribution function (CDF)**

$$F_X(x) = \text{Prob}\{X \leq x\}.$$

Sometimes, but not always, a random variable can also be described by **density function** $f(x)$ that is related to its CDF by

$$\text{Prob}\{X \in B\} = \int_{t \in B} f(t)dt$$

$$F(x) = \int_{-\infty}^{x} f(t)dt$$

Here B is a set of possible X's whose probability of occurring we want to compute.

When a probability density exists, a probability distribution can be characterized either by its CDF or by its density.

For a **discrete-valued** random variable

- the number of possible values of X is finite or countably infinite
- we replace a **density** with a **probability mass function**, a non-negative sequence that sums to one
- we replace integration with summation in the formula like (8.1) that relates a CDF to a probability mass function

In this lecture, we mostly discuss discrete random variables.

Doing this enables us to confine our tool set basically to linear algebra.

Later we'll briefly discuss how to approximate a continuous random variable with a discrete random variable.

8.4 Univariate Probability Distributions

We'll devote most of this lecture to discrete-valued random variables, but we'll say a few things about continuous-valued random variables.

8.4.1 Discrete random variable

Let X be a discrete random variable that takes possible values: $i = 0, 1, \ldots, I - 1 = \bar{X}$.

Here, we choose the maximum index $I - 1$ because of how this aligns nicely with Python's index convention.

Define $f_i \equiv \text{Prob}\{X = i\}$ and assemble the non-negative vector

$$f = \begin{bmatrix} f_0 \\ f_1 \\ \vdots \\ f_{I-1} \end{bmatrix} \tag{8.2}$$

for which $f_i \in [0, 1]$ for each i and $\sum_{i=0}^{I-1} f_i = 1$.

This vector defines a **probability mass function**.

The distribution (8.2) has **parameters** $\{f_i\}_{i=0,1,\cdots,I-2}$ since $f_{I-1} = 1 - \sum_{i=0}^{I-2} f_i$.

These parameters pin down the shape of the distribution.

(Sometimes $I = \infty$.)

Such a "non-parametric" distribution has as many "parameters" as there are possible values of the random variable.

We often work with special distributions that are characterized by a small number parameters.

In these special parametric distributions,

$$f_i = g(i; \theta)$$

where θ is a vector of parameters that is of much smaller dimension than I.

Remarks:

- A **statistical model** is a joint probability distribution characterized by a list of **parameters**
- The concept of **parameter** is intimately related to the notion of **sufficient statistic**.
- A **statistic** is a nonlinear function of a data set.
- **Sufficient statistics** summarize all **information** that a data set contains about parameters of statistical model.
  - Note that a sufficient statistic corresponds to a particular statistical model.
  - Sufficient statistics are key tools that AI uses to summarize or compress a **big data** set.
- R. A. Fisher provided a rigorous definition of **information** – see https://en.wikipedia.org/wiki/Fisher_information

An example of a parametric probability distribution is a **geometric distribution**.

It is described by

$$f_i = \text{Prob}\{X = i\} = (1 - \lambda)\lambda^i, \quad \lambda \in [0, 1], \quad i = 0, 1, 2, \ldots$$

Evidently, $\sum_{i=0}^{\infty} f_i = 1$.

Let θ be a vector of parameters of the distribution described by f, then

$$f_i(\theta) \geq 0, \sum_{i=0}^{\infty} f_i(\theta) = 1$$

8.4.2 Continuous random variable

Let X be a continous random variable that takes values $X \in \tilde{X} \equiv [X_U, X_L]$ whose distributions have parameters θ.

$$\text{Prob}\{X \in A\} = \int_{x \in A} f(x; \theta)\, dx; \quad f(x; \theta) \geq 0$$

where A is a subset of $\tilde{X}$ and

$$\text{Prob}\{X \in \tilde{X}\} = 1$$

8.5 Bivariate Probability Distributions

We'll now discuss a bivariate **joint distribution**.

To begin, we restrict ourselves to two discrete random variables.

Let X, Y be two discrete random variables that take values:

$$X \in \{0, \dots, I-1\}$$

$$Y \in \{0, \dots, J-1\}$$

Then their **joint distribution** is described by a matrix

$$F_{I \times J} = [f_{ij}]_{i \in \{0,\dots,I-1\}, j \in \{0,\dots,J-1\}}$$

whose elements are

$$f_{ij} = \text{Prob}\{X = i, Y = j\} \geq 0$$

where

$$\sum_i \sum_j f_{ij} = 1$$

8.6 Marginal Probability Distributions

The joint distribution induce marginal distributions

$$\text{Prob}\{X = i\} = \sum_{j=0}^{J-1} f_{ij} = \mu_i, \quad i = 0, \dots, I-1$$

$$\text{Prob}\{Y = j\} = \sum_{i=0}^{I-1} f_{ij} = \nu_j, \quad j = 0, \dots, J-1$$

For example, let a joint distribution over (X, Y) be

$$F = \begin{bmatrix} .25 & .1 \\ .15 & .5 \end{bmatrix} \tag{8.3}$$

The implied marginal distributions are:

$$\begin{aligned} \text{Prob}\{X = 0\} &= .25 + .1 = .35 \\ \text{Prob}\{X = 1\} &= .15 + .5 = .65 \\ \text{Prob}\{Y = 0\} &= .25 + .15 = .4 \\ \text{Prob}\{Y = 1\} &= .1 + .5 = .6 \end{aligned}$$

Digression: If two random variables X, Y are continuous and have joint density $f(x, y)$, then marginal distributions can be computed by

$$\begin{aligned} f(x) &= \int_{\mathbb{R}} f(x, y) dy \\ f(y) &= \int_{\mathbb{R}} f(x, y) dx \end{aligned}$$

8.7 Conditional Probability Distributions

Conditional probabilities are defined according to

$$\text{Prob}\{A \mid B\} = \frac{\text{Prob}\{A \cap B\}}{\text{Prob}\{B\}}$$

where A, B are two events.

For a pair of discrete random variables, we have the **conditional distribution**

$$\text{Prob}\{X = i|Y = j\} = \frac{f_{ij}}{\sum_i f_{ij}} = \frac{\text{Prob}\{X = i, Y = j\}}{\text{Prob}\{Y = j\}}$$

where $i = 0, \dots, I - 1, \quad j = 0, \dots, J - 1$.

Note that

$$\sum_i \text{Prob}\{X_i = i|Y_j = j\} = \frac{\sum_i f_{ij}}{\sum_i f_{ij}} = 1$$

Remark: The mathematics of conditional probability implies:

$$\text{Prob}\{X = i|Y = j\} = \frac{\text{Prob}\{X = i, Y = j\}}{\text{Prob}\{Y = j\}} = \frac{\text{Prob}\{Y = j|X = i\}\text{Prob}\{X = i\}}{\text{Prob}\{Y = j\}} \tag{8.4}$$

> **Note**
>
> Formula (8.4) is also what a Bayesian calls **Bayes' Law**. A Bayesian statistician regards marginal probability distribution $\text{Prob}(X = i), i = 1, \dots, J$ as a **prior** distribution that describes his personal subjective beliefs about X. He then interprets formula (8.4) as a procedure for constructing a **posterior** distribution that describes how he would revise his subjective beliefs after observing that Y equals j.

For the joint distribution (8.3)

$$\text{Prob}\{X = 0|Y = 1\} = \frac{.1}{.1 + .5} = \frac{.1}{.6}$$

8.8 Transition Probability Matrix

Consider the following joint probability distribution of two random variables.

Let X, Y be discrete random variables with joint distribution

$$\text{Prob}\{X = i, Y = j\} = \rho_{ij}$$

where $i = 0, \dots, I - 1; j = 0, \dots, J - 1$ and

$$\sum_i \sum_j \rho_{ij} = 1, \quad \rho_{ij} \geqslant 0.$$

An associated conditional distribution is

$$\text{Prob}\{Y = i|X = j\} = \frac{\rho_{ij}}{\sum_j \rho_{ij}} = \frac{\text{Prob}\{Y = j, X = i\}}{\text{Prob}\{X = i\}}$$

We can define a transition probability matrix P with i, j component

$$p_{ij} = \text{Prob}\{Y = j | X = i\} = \frac{\rho_{ij}}{\sum_j \rho_{ij}}$$

where

$$\begin{bmatrix} p_{11} & p_{12} \\ p_{21} & p_{22} \end{bmatrix}$$

The first row is the probability that $Y = j, j = 0, 1$ conditional on $X = 0$.

The second row is the probability that $Y = j, j = 0, 1$ conditional on $X = 1$.

Note that

- $\sum_j \rho_{ij} = \frac{\sum_j \rho_{ij}}{\sum_j \rho_{ij}} = 1$, so each row of the transition matrix P is a probability distribution (not so for each column).

8.9 Application: Forecasting a Time Series

Suppose that there are two time periods.

- $t = 0$ "today"
- $t = 1$ "tomorrow"

Let $X(0)$ be a random variable to be realized at $t = 0$, $X(1)$ be a random variable to be realized at $t = 1$.

Suppose that

$$\text{Prob}\{X(0) = i, X(1) = j\} = f_{ij} \geq 0 \; i = 0, \cdots, I - 1$$
$$\sum_i \sum_j f_{ij} = 1$$

f_{ij} is a joint distribution over $[X(0), X(1)]$.

A conditional distribution is

$$\text{Prob}\{X(1) = j | X(0) = i\} = \frac{f_{ij}}{\sum_j f_{ij}}$$

Remark:

- This formula is a workhorse for applied economic forecasters.

8.10 Statistical Independence

Random variables X and Y are statistically **independent** if

$$\text{Prob}\{X = i, Y = j\} = f_i g_j$$

where

$$\text{Prob}\{X = i\} = f_i \geq 0 \; \sum f_i = 1$$
$$\text{Prob}\{Y = j\} = g_j \geq 0 \; \sum g_j = 1$$

Conditional distributions are

$$\begin{aligned}
\textrm{Prob}\{X=i|Y=j\} &= \frac{f_ig_j}{\sum_i f_ig_j} = \frac{f_ig_j}{g_j} = f_i \\
\textrm{Prob}\{Y=j|X=i\} &= \frac{f_ig_j}{\sum_j f_ig_j} = \frac{f_ig_j}{f_i} = g_j
\end{aligned}$$

8.11 Means and Variances

The mean and variance of a discrete random variable X are

$$\begin{aligned}
\mu_X &\equiv \mathbb{E}\left[X\right] = \sum_k k \textrm{Prob}\{X=k\} \\
\sigma_X^2 &\equiv \mathbb{D}\left[X\right] = \sum_k \left(k - \mathbb{E}\left[X\right]\right)^2 \textrm{Prob}\{X=k\}
\end{aligned}$$

A continuous random variable having density $f_X(x)$) has mean and variance

$$\begin{aligned}
\mu_X &\equiv \mathbb{E}\left[X\right] = \int_{-\infty}^{\infty} x f_X(x) dx \\
\sigma_X^2 \equiv \mathbb{D}\left[X\right] = \mathrm{E}\left[\left(X - \mu_X\right)^2\right] &= \int_{-\infty}^{\infty} \left(x - \mu_X\right)^2 f_X(x) dx
\end{aligned}$$

8.12 Matrix Representations of Some Bivariate Distributions

Let's use matrices to represent a joint distribution, conditional distribution, marginal distribution, and the mean and variance of a bivariate random variable.

The table below illustrates a probability distribution for a bivariate random variable.

$$F = [f_{ij}] = \left[\begin{array}{cc} 0.3 & 0.2 \\ 0.1 & 0.4 \end{array}\right]$$

Marginal distributions are

$$\textrm{Prob}(X=i) = \sum_j f_{ij} = u_i$$

$$\textrm{Prob}(Y=j) = \sum_i f_{ij} = v_j$$

Sampling:

Let's write some Python code that let's us draw some long samples and compute relative frequencies.

The code will let us check whether the "sampling" distribution agrees with the "population" distribution - confirming that the population distribution correctly tells us the relative frequencies that we should expect in a large sample.

```
# specify parameters
xs = np.array([0, 1])
ys = np.array([10, 20])
f = np.array([[0.3, 0.2], [0.1, 0.4]])
f_cum = np.cumsum(f)
```

(continues on next page)

(continued from previous page)

```
# draw random numbers
p = np.random.rand(1_000_000)
x = np.vstack([xs[1]*np.ones(p.shape), ys[1]*np.ones(p.shape)])
# map to the bivariate distribution

x[0, p < f_cum[2]] = xs[1]
x[1, p < f_cum[2]] = ys[0]

x[0, p < f_cum[1]] = xs[0]
x[1, p < f_cum[1]] = ys[1]

x[0, p < f_cum[0]] = xs[0]
x[1, p < f_cum[0]] = ys[0]
print(x)
```

```
[[ 0.   1.   1. ...  1.   0.   1.]
 [10. 10. 10. ... 20. 10. 20.]]
```

Note

To generate random draws from the joint distribution F, we use the inverse CDF technique described in *this companion lecture*.

```
# marginal distribution
xp = np.sum(x[0, :] == xs[0])/1_000_000
yp = np.sum(x[1, :] == ys[0])/1_000_000

# print output
print("marginal distribution for x")
xmtb = pt.PrettyTable()
xmtb.field_names = ['x_value', 'x_prob']
xmtb.add_row([xs[0], xp])
xmtb.add_row([xs[1], 1-xp])
print(xmtb)

print("\nmarginal distribution for y")
ymtb = pt.PrettyTable()
ymtb.field_names = ['y_value', 'y_prob']
ymtb.add_row([ys[0], yp])
ymtb.add_row([ys[1], 1-yp])
print(ymtb)
```

```
marginal distribution for x
+---------+--------------------+
| x_value |       x_prob       |
+---------+--------------------+
|    0    |      0.499894      |
|    1    | 0.5001059999999999 |
+---------+--------------------+

marginal distribution for y
+---------+--------------------+
| y_value |       y_prob       |
+---------+--------------------+
```

(continues on next page)

(continued from previous page)

```
|    10    |       0.399909       |
|    20    | 0.6000909999999999 |
+----------+--------------------+
```

```
# conditional distributions
xc1 = x[0, x[1, :] == ys[0]]
xc2 = x[0, x[1, :] == ys[1]]
yc1 = x[1, x[0, :] == xs[0]]
yc2 = x[1, x[0, :] == xs[1]]

xc1p = np.sum(xc1 == xs[0])/len(xc1)
xc2p = np.sum(xc2 == xs[0])/len(xc2)
yc1p = np.sum(yc1 == ys[0])/len(yc1)
yc2p = np.sum(yc2 == ys[0])/len(yc2)

# print output
print("conditional distribution for x")
xctb = pt.PrettyTable()
xctb.field_names = ['y_value', 'prob(x=0)', 'prob(x=1)']
xctb.add_row([ys[0], xc1p, 1-xc1p])
xctb.add_row([ys[1], xc2p, 1-xc2p])
print(xctb)

print("\nconditional distribution for y")
yctb = pt.PrettyTable()
yctb.field_names = ['x_value',  'prob(y=10)', 'prob(y=20)']
yctb.add_row([xs[0], yc1p, 1-yc1p])
yctb.add_row([xs[1], yc2p, 1-yc2p])
print(yctb)
```

```
conditional distribution for x
+---------+--------------------+---------------------+
| y_value |     prob(x=0)      |      prob(x=1)      |
+---------+--------------------+---------------------+
|    10   | 0.748947885643984  | 0.25105211435601604 |
|    20   | 0.3339210219783333 |  0.6660789780216667 |
+---------+--------------------+---------------------+

conditional distribution for y
+---------+--------------------+--------------------+
| x_value |     prob(y=10)     |     prob(y=20)     |
+---------+--------------------+--------------------+
|    0    | 0.5991490195921535 | 0.4008509804078465 |
|    1    | 0.2007534402706626 | 0.7992465597293374 |
+---------+--------------------+--------------------+
```

Let's calculate population marginal and conditional probabilities using matrix algebra.

$$
\left[\begin{array}{ccccccc}
 & \vdots & y_1 & y_2 & \vdots & x \\
\cdots & \vdots & \cdots & \cdots & \vdots & \cdots \\
x_1 & \vdots & 0.3 & 0.2 & \vdots & 0.5 \\
x_2 & \vdots & 0.1 & 0.4 & \vdots & 0.5 \\
\cdots & \vdots & \cdots & \cdots & \vdots & \cdots \\
y & \vdots & 0.4 & 0.6 & \vdots & 1
\end{array}\right]
$$

$\Longrightarrow$

(1) Marginal distribution:

$$
\left[\begin{matrix}var & \vdots & var_1 & var_2 \\ \cdots & \vdots & \cdots & \cdots \\ x & \vdots & 0.5 & 0.5 \\ \cdots & \vdots & \cdots & \cdots \\ y & \vdots & 0.4 & 0.6 \\ \end{matrix}\right]
$$

(2) Conditional distribution:

$$
\left[\begin{matrix}\quad x & \vdots & \quad x_1 & \quad x_2 \\ \cdots\cdots\cdots & \vdots & \cdots\cdots\cdots & \cdots\cdots\cdots \\ y=y_1 & \vdots & \frac{0.3}{0.4}=0.75 & \frac{0.1}{0.4}=0.25 \\ \cdots\cdots\cdots & \vdots & \cdots\cdots\cdots & \cdots\cdots\cdots \\ y=y_2 & \vdots & \frac{0.2}{0.6}\approx 0.33 & \frac{0.4}{0.6}\approx0.67 \\ \end{matrix}\right]
$$

$$
\left[\begin{matrix}\quad y & \vdots & \quad y_1 & \quad y_2 \\ \cdots\cdots\cdots & \vdots & \cdots\cdots\cdots & \cdots\cdots\cdots \\ x=x_1 & \vdots & \frac{0.3}{0.5}=0.6 & \frac{0.2}{0.5}=0.4 \\ \cdots\cdots\cdots & \vdots & \cdots\cdots\cdots & \cdots\cdots\cdots \\ x=x_2 & \vdots & \frac{0.1}{0.5}=0.2 & \frac{0.4}{0.5}=0.8 \\ \end{matrix}\right]
$$

These population objects closely resemble the sample counterparts computed above.

Let's wrap some of the functions we have used in a Python class that will let us generate and sample from a discrete bivariate joint distribution.

```python
class discrete_bijoint:

    def __init__(self, f, xs, ys):
        '''initialization
        -----------------
        parameters:
        f: the bivariate joint probability matrix
        xs: values of x vector
        ys: values of y vector
        '''
        self.f, self.xs, self.ys = f, xs, ys

    def joint_tb(self):
        '''print the joint distribution table'''
        xs = self.xs
        ys = self.ys
        f = self.f
        jtb = pt.PrettyTable()
        jtb.field_names = ['x_value/y_value', *ys, 'marginal sum for x']
        for i in range(len(xs)):
            jtb.add_row([xs[i], *f[i, :], np.sum(f[i, :])])
        jtb.add_row(['marginal_sum for y', *np.sum(f, 0), np.sum(f)])
        print("\nThe joint probability distribution for x and y\n", jtb)
        self.jtb = jtb

    def draw(self, n):
        '''draw random numbers
        ----------------------
        parameters:
        n: number of random numbers to draw
        '''
```

(continues on next page)

(continued from previous page)

```
        xs = self.xs
        ys = self.ys
        f_cum = np.cumsum(self.f)
        p = np.random.rand(n)
        x = np.empty([2, p.shape[0]])
        lf = len(f_cum)
        lx = len(xs)-1
        ly = len(ys)-1
        for i in range(lf):
            x[0, p < f_cum[lf-1-i]] = xs[lx]
            x[1, p < f_cum[lf-1-i]] = ys[ly]
            if ly == 0:
                lx -= 1
                ly = len(ys)-1
            else:
                ly -= 1
        self.x = x
        self.n = n

    def marg_dist(self):
        '''marginal distribution'''
        x = self.x
        xs = self.xs
        ys = self.ys
        n = self.n
        xmp = [np.sum(x[0, :] == xs[i])/n for i in range(len(xs))]
        ymp = [np.sum(x[1, :] == ys[i])/n for i in range(len(ys))]

        # print output
        xmtb = pt.PrettyTable()
        ymtb = pt.PrettyTable()
        xmtb.field_names = ['x_value', 'x_prob']
        ymtb.field_names = ['y_value', 'y_prob']
        for i in range(max(len(xs), len(ys))):
            if i < len(xs):
                xmtb.add_row([xs[i], xmp[i]])
            if i < len(ys):
                ymtb.add_row([ys[i], ymp[i]])
        xmtb.add_row(['sum', np.sum(xmp)])
        ymtb.add_row(['sum', np.sum(ymp)])
        print("\nmarginal distribution for x\n", xmtb)
        print("\nmarginal distribution for y\n", ymtb)

        self.xmp = xmp
        self.ymp = ymp

    def cond_dist(self):
        '''conditional distribution'''
        x = self.x
        xs = self.xs
        ys = self.ys
        n = self.n
        xcp = np.empty([len(ys), len(xs)])
        ycp = np.empty([len(xs), len(ys)])
        for i in range(max(len(ys), len(xs))):
            if i < len(ys):
                xi = x[0, x[1, :] == ys[i]]
```

(continues on next page)

(continued from previous page)

```
                idx = xi.reshape(len(xi), 1) == xs.reshape(1, len(xs))
                xcp[i, :] = np.sum(idx, 0)/len(xi)
            if i < len(xs):
                yi = x[1, x[0, :] == xs[i]]
                idy = yi.reshape(len(yi), 1) == ys.reshape(1, len(ys))
                ycp[i, :] = np.sum(idy, 0)/len(yi)

        # print output
        xctb = pt.PrettyTable()
        yctb = pt.PrettyTable()
        xctb.field_names = ['x_value', *xs, 'sum']
        yctb.field_names = ['y_value', *ys, 'sum']
        for i in range(max(len(xs), len(ys))):
            if i < len(ys):
                xctb.add_row([ys[i], *xcp[i], np.sum(xcp[i])])
            if i < len(xs):
                yctb.add_row([xs[i], *ycp[i], np.sum(ycp[i])])
        print("\nconditional distribution for x\n", xctb)
        print("\nconditional distribution for y\n", yctb)

        self.xcp = xcp
        self.xyp = ycp
```

Let's apply our code to some examples.

Example 1

```
# joint
d = discrete_bijoint(f, xs, ys)
d.joint_tb()
```

```
The joint probability distribution for x and y
 +--------------------+-----+--------------------+--------------------+
|  x_value/y_value   |  10 |         20         | marginal sum for x |
+--------------------+-----+--------------------+--------------------+
|         0          | 0.3 |        0.2         |        0.5         |
|         1          | 0.1 |        0.4         |        0.5         |
| marginal_sum for y | 0.4 | 0.6000000000000001 |        1.0         |
+--------------------+-----+--------------------+--------------------+
```

```
# sample marginal
d.draw(1_000_000)
d.marg_dist()
```

```
marginal distribution for x
 +---------+----------+
| x_value |  x_prob  |
+---------+----------+
|    0    | 0.500357 |
|    1    | 0.499643 |
|   sum   |   1.0    |
+---------+----------+

marginal distribution for y
 +---------+----------+
| y_value |  y_prob  |
```

(continues on next page)

(continued from previous page)

```
+----------+----------+
|    10    | 0.400806 |
|    20    | 0.599194 |
|   sum    |   1.0    |
+----------+----------+
```

```
# sample conditional
d.cond_dist()
```

```
conditional distribution for x
 +---------+---------------------+--------------------+-----+
| x_value |          0          |         1          | sum |
+---------+---------------------+--------------------+-----+
|    10   |  0.7493625344929966 | 0.2506374655070034 | 1.0 |
|    20   | 0.33379506470358516 | 0.6662049352964149 | 1.0 |
+---------+---------------------+--------------------+-----+

conditional distribution for y
 +---------+---------------------+---------------------+-----+
| y_value |          10         |          20         | sum |
+---------+---------------------+---------------------+-----+
|    0    |  0.6002694076429429 | 0.39973059235705705 | 1.0 |
|    1    | 0.20105755509433737 |  0.7989424449056627 | 1.0 |
+---------+---------------------+---------------------+-----+
```

Example 2

```
xs_new = np.array([10, 20, 30])
ys_new = np.array([1, 2])
f_new = np.array([[0.2, 0.1], [0.1, 0.3], [0.15, 0.15]])
d_new = discrete_bijoint(f_new, xs_new, ys_new)
d_new.joint_tb()
```

```
The joint probability distribution for x and y
 +--------------------+---------------------+------+---------------------+
|  x_value/y_value   |          1          |  2   |  marginal sum for x |
+--------------------+---------------------+------+---------------------+
|         10         |         0.2         | 0.1  | 0.30000000000000004 |
|         20         |         0.1         | 0.3  |         0.4         |
|         30         |         0.15        | 0.15 |         0.3         |
| marginal_sum for y | 0.45000000000000007 | 0.55 |         1.0         |
+--------------------+---------------------+------+---------------------+
```

```
d_new.draw(1_000_000)
d_new.marg_dist()
```

```
marginal distribution for x
 +---------+--------------------+
| x_value |       x_prob       |
+---------+--------------------+
|    10   |      0.298998      |
|    20   |      0.400873      |
|    30   |      0.300129      |
|   sum   | 0.9999999999999999 |
+---------+--------------------+
```

(continues on next page)

(continued from previous page)

```
marginal distribution for y
 +---------+----------+
| y_value |  y_prob  |
+---------+----------+
|    1    | 0.449673 |
|    2    | 0.550327 |
|   sum   |   1.0    |
+---------+----------+
```

```
d_new.cond_dist()
```

```
conditional distribution for x
 +---------+--------------------+--------------------+--------------------+-----+
| x_value |         10         |         20         |         30         | sum |
+---------+--------------------+--------------------+--------------------+-----+
|    1    | 0.4434911591311909 | 0.22315994066799652 | 0.3333489002008126 | 1.0 |
|    2    | 0.1809324274476811 |  0.546082601798567  | 0.2729849707537519 | 1.0 |
+---------+--------------------+--------------------+--------------------+-----+

conditional distribution for y
 +---------+---------------------+--------------------+-----+
| y_value |          1          |         2          | sum |
+---------+---------------------+--------------------+-----+
|    10   |  0.6669810500404685 | 0.3330189499595315 | 1.0 |
|    20   | 0.25032616314892747 | 0.7496738368510726 | 1.0 |
|    30   |  0.4994452385474246 | 0.5005547614525754 | 1.0 |
+---------+---------------------+--------------------+-----+
```

8.13 A Continuous Bivariate Random Vector

A two-dimensional Gaussian distribution has joint density

$$f(x,y) = (2\pi\sigma_1\sigma_2\sqrt{1-\rho^2})^{-1}\exp\left[-\frac{1}{2(1-\rho^2)}\left(\frac{(x-\mu_1)^2}{\sigma_1^2} - \frac{2\rho(x-\mu_1)(y-\mu_2)}{\sigma_1\sigma_2} + \frac{(y-\mu_2)^2}{\sigma_2^2}\right)\right]$$

$$\frac{1}{2\pi\sigma_1\sigma_2\sqrt{1-\rho^2}}\exp\left[-\frac{1}{2(1-\rho^2)}\left(\frac{(x-\mu_1)^2}{\sigma_1^2} - \frac{2\rho(x-\mu_1)(y-\mu_2)}{\sigma_1\sigma_2} + \frac{(y-\mu_2)^2}{\sigma_2^2}\right)\right]$$

We start with a bivariate normal distribution pinned down by

$$\mu = \begin{bmatrix} 0 \\ 5 \end{bmatrix}, \quad \Sigma = \begin{bmatrix} 5 & .2 \\ .2 & 1 \end{bmatrix}$$

```python
# define the joint probability density function
def func(x, y, μ1=0, μ2=5, σ1=np.sqrt(5), σ2=np.sqrt(1), ρ=.2/np.sqrt(5*1)):
    A = (2 * np.pi * σ1 * σ2 * np.sqrt(1 - ρ**2))**(-1)
    B = -1 / 2 / (1 - ρ**2)
    C1 = (x - μ1)**2 / σ1**2
    C2 = 2 * ρ * (x - μ1) * (y - μ2) / σ1 / σ2
    C3 = (y - μ2)**2 / σ2**2
    return A * np.exp(B * (C1 - C2 + C3))
```

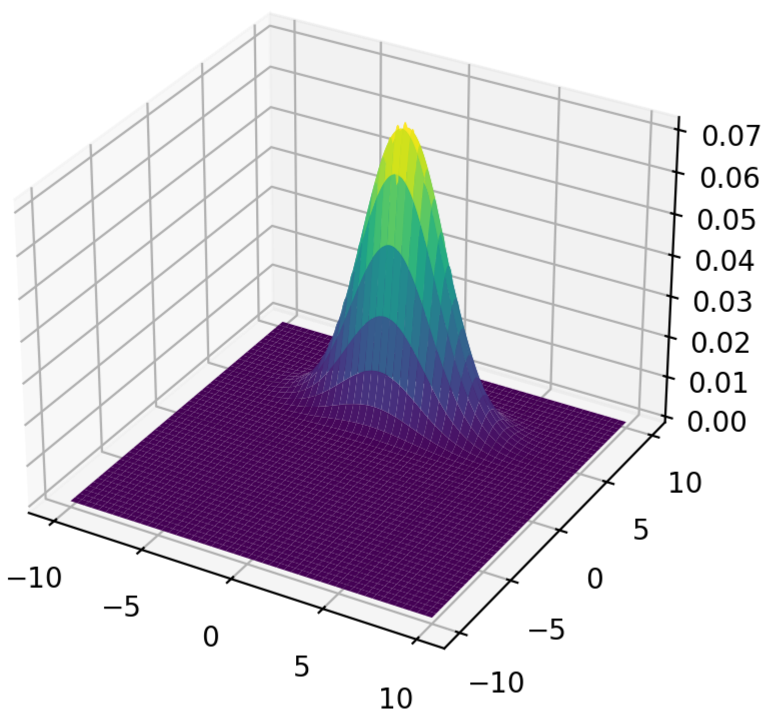
```
μ1 = 0
μ2 = 5
σ1 = np.sqrt(5)
σ2 = np.sqrt(1)
ρ = .2 / np.sqrt(5 * 1)
```

```
x = np.linspace(-10, 10, 1_000)
y = np.linspace(-10, 10, 1_000)
x_mesh, y_mesh = np.meshgrid(x, y, indexing="ij")
```

Joint Distribution

Let's plot the **population** joint density.

```
# %matplotlib notebook

fig = plt.figure()
ax = plt.axes(projection='3d')

surf = ax.plot_surface(x_mesh, y_mesh, func(x_mesh, y_mesh), cmap='viridis')
plt.show()
```

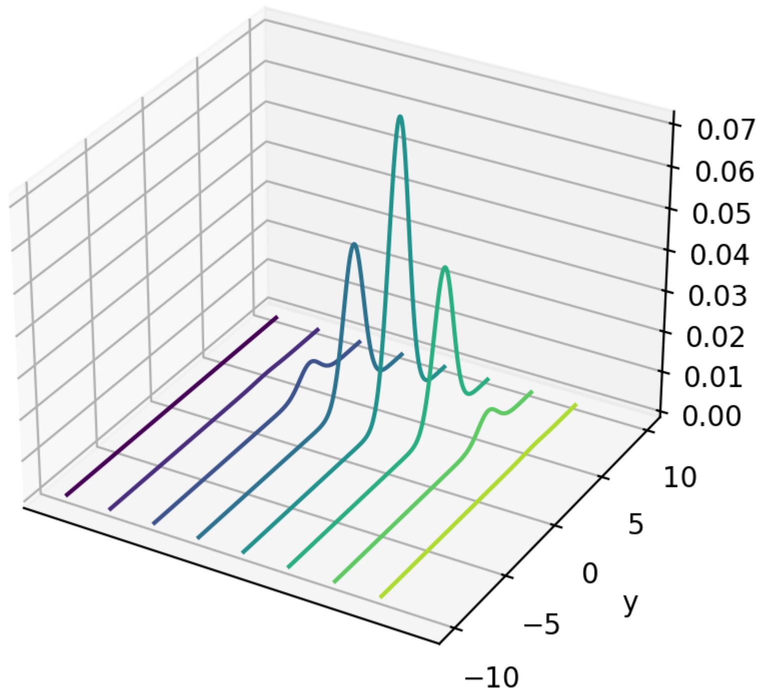



```
# %matplotlib notebook

fig = plt.figure()
ax = plt.axes(projection='3d')

curve = ax.contour(x_mesh, y_mesh, func(x_mesh, y_mesh), zdir='x')
plt.ylabel('y')
```

(continues on next page)

(continued from previous page)

```
ax.set_zlabel('f')
ax.set_xticks([])
plt.show()
```



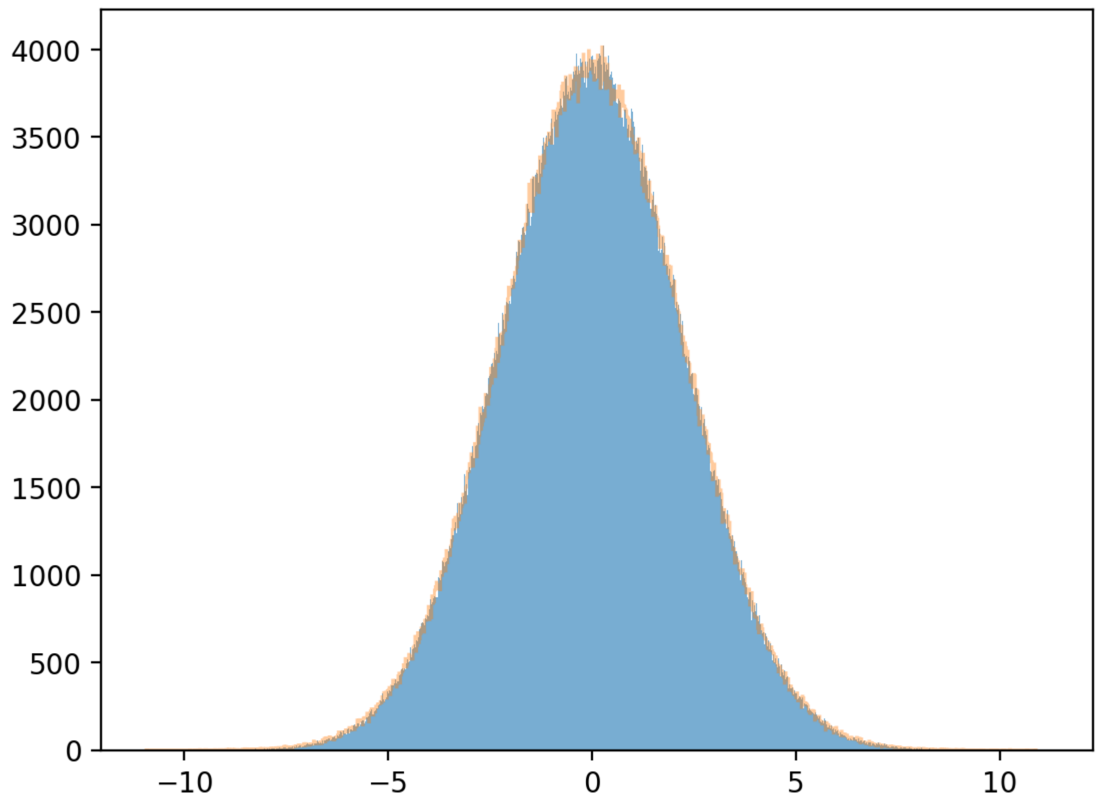
Next we can use a built-in `numpy` function to draw random samples, then calculate a **sample** marginal distribution from the sample mean and variance.

```
μ= np.array([0, 5])
σ= np.array([[5, .2], [.2, 1]])
n = 1_000_000
data = np.random.multivariate_normal(μ, σ, n)
x = data[:, 0]
y = data[:, 1]
```

Marginal distribution

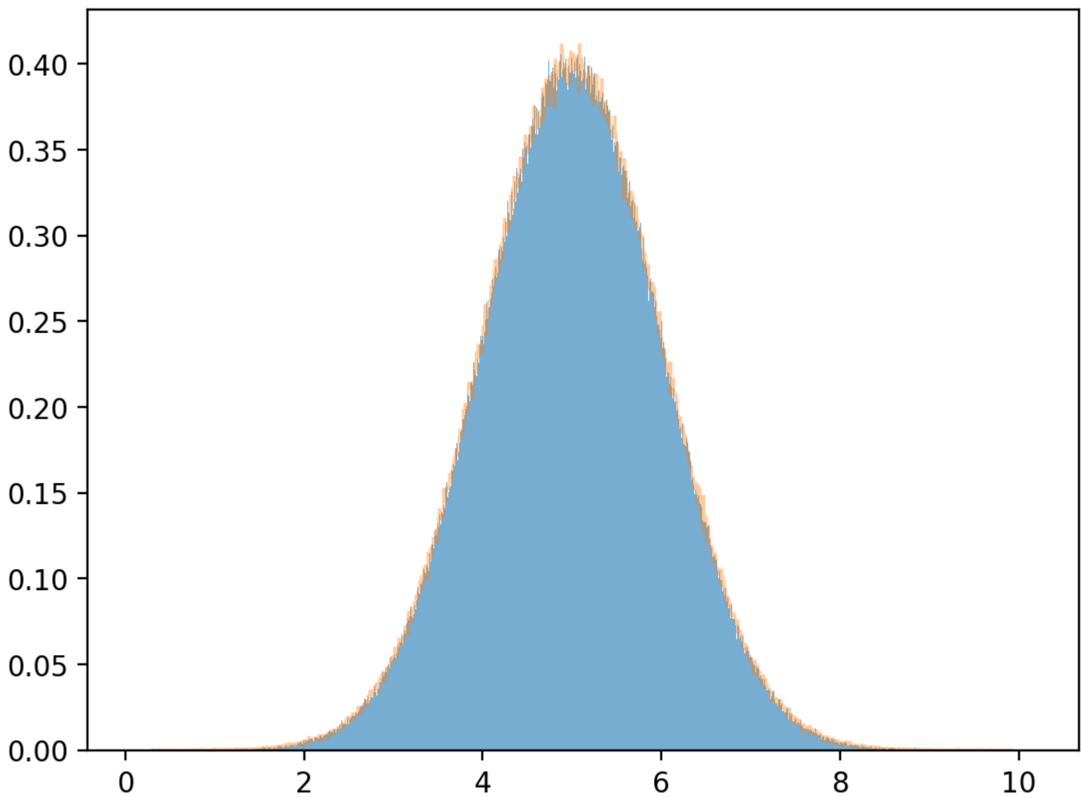
```
plt.hist(x, bins=1_000, alpha=0.6)
μx_hat, σx_hat = np.mean(x), np.std(x)
print(μx_hat, σx_hat)
x_sim = np.random.normal(μx_hat, σx_hat, 1_000_000)
plt.hist(x_sim, bins=1_000, alpha=0.4, histtype="step")
plt.show()
```

```
0.0006882165867244266 2.2374114685106847
```




```
plt.hist(y, bins=1_000, density=True, alpha=0.6)
μy_hat, σy_hat = np.mean(y), np.std(y)
print(μy_hat, σy_hat)
y_sim = np.random.normal(μy_hat, σy_hat, 1_000_000)
plt.hist(y_sim, bins=1_000, density=True, alpha=0.4, histtype="step")
plt.show()
```

```
4.99918993079252 0.9988322369716577
```



Conditional distribution

For a bivariate normal population distribution, the conditional distributions are also normal:

$$[X|Y=y] \sim \mathbb{N}\left[\mu_X + \rho\sigma_X \frac{y-\mu_Y}{\sigma_Y}, \sigma_X^2(1-\rho^2)\right]$$
$$[Y|X=x] \sim \mathbb{N}\left[\mu_Y + \rho\sigma_Y \frac{x-\mu_X}{\sigma_X}, \sigma_Y^2(1-\rho^2)\right]$$

> **Note**
>
> Please see this *quantecon lecture* for more details.

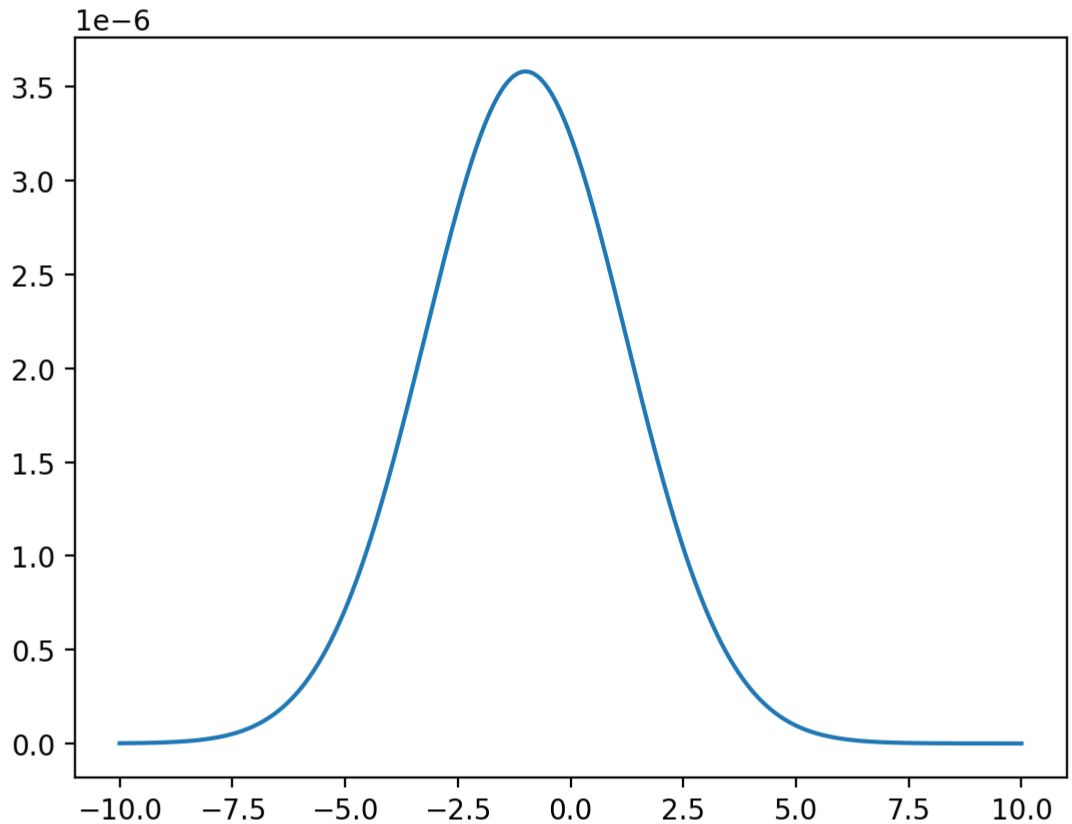
Let's approximate the joint density by discretizing and mapping the approximating joint density into a matrix.

We can compute the discretized marginal density by just using matrix algebra and noting that

$$\text{Prob}\{X=i|Y=j\} = \frac{f_{ij}}{\sum_i f_{ij}}$$

Fix $y=0$.

```
# discretized marginal density
x = np.linspace(-10, 10, 1_000_000)
z = func(x, y=0) / np.sum(func(x, y=0))
plt.plot(x, z)
plt.show()
```

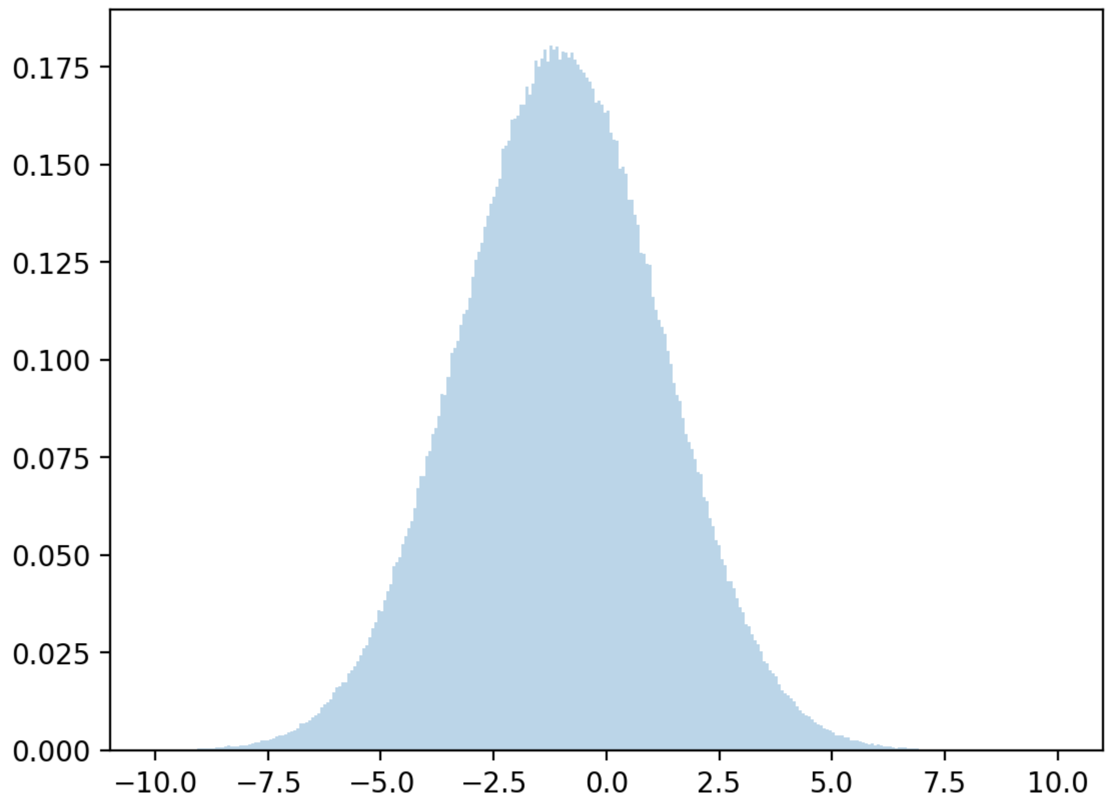


The mean and variance are computed by

$$\mathbb{E}\left[X|Y=j\right]=\sum_{i}iProb\{X=i|Y=j\}=\sum_{i}i\frac{f_{ij}}{\sum_{i}f_{ij}}$$
$$\mathbb{D}\left[X|Y=j\right]=\sum_{i}\left(i-\mu_{X|Y=j}\right)^{2}\frac{f_{ij}}{\sum_{i}f_{ij}}$$

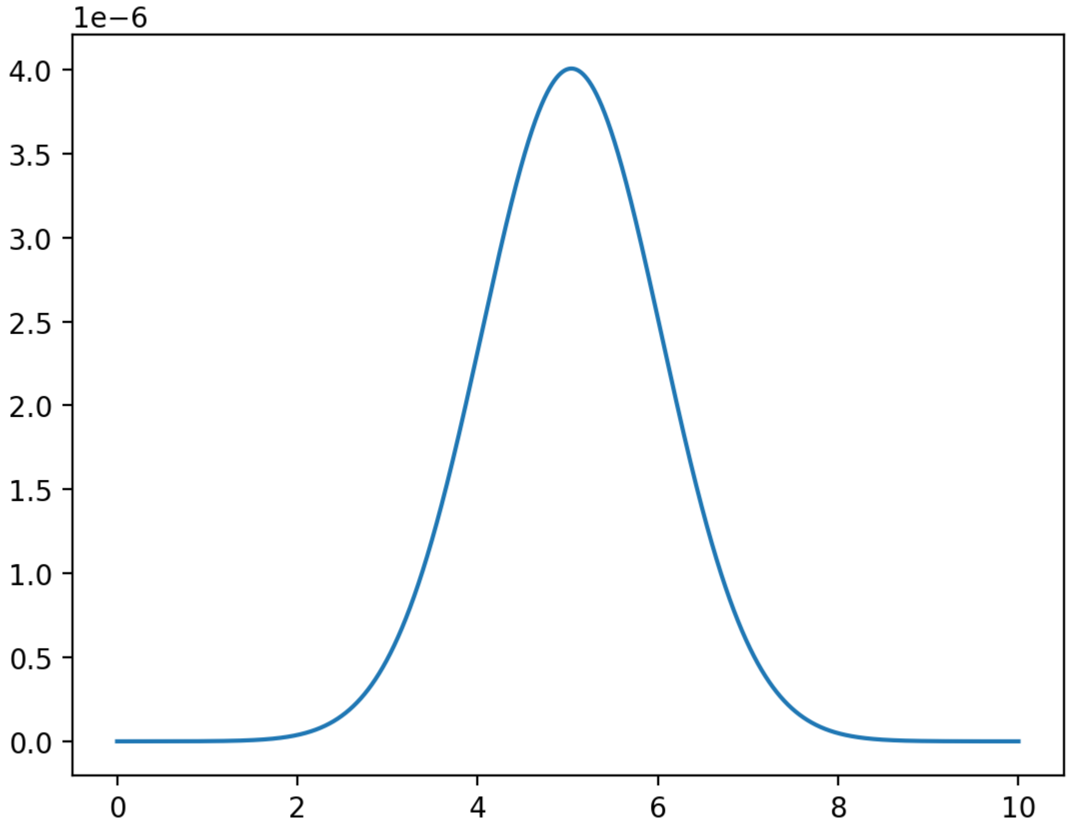
Let's draw from a normal distribution with above mean and variance and check how accurate our approximation is.

```
# discretized mean
μx = np.dot(x, z)

# discretized standard deviation
σx = np.sqrt(np.dot((x - μx)**2, z))

# sample
zz = np.random.normal(μx, σx, 1_000_000)
plt.hist(zz, bins=300, density=True, alpha=0.3, range=[-10, 10])
plt.show()
```

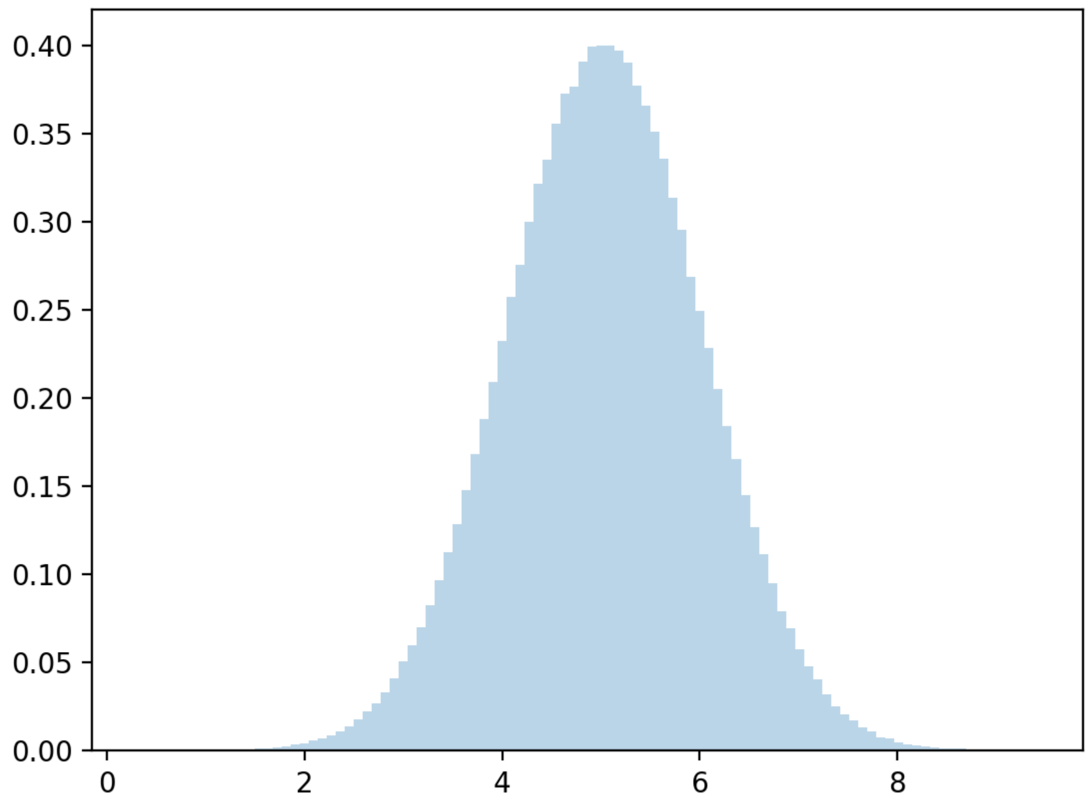


Fix $x = 1$.

```
y = np.linspace(0, 10, 1_000_000)
z = func(x=1, y=y) / np.sum(func(x=1, y=y))
plt.plot(y,z)
plt.show()
```




```
# discretized mean and standard deviation
μy = np.dot(y,z)
σy = np.sqrt(np.dot((y - μy)**2, z))

# sample
zz = np.random.normal(μy,σy,1_000_000)
plt.hist(zz, bins=100, density=True, alpha=0.3)
plt.show()
```



We compare with the analytically computed parameters and note that they are close.

```
print(μx, σx)
print(μ1 + ρ * σ1 * (0 - μ2) / σ2, np.sqrt(σ1**2 * (1 - ρ**2)))

print(μy, σy)
print(μ2 + ρ * σ2 * (1 - μ1) / σ1, np.sqrt(σ2**2 * (1 - ρ**2)))
```

```
-0.9997518414498444 2.2265841331697698
-1.0 2.227105745132009
5.039999456960769 0.9959851265795593
5.04 0.9959919678390986
```

8.14 Sum of Two Independently Distributed Random Variables

Let X, Y be two independent discrete random variables that take values in $\bar{X}, \bar{Y}$, respectively.

Define a new random variable $Z = X + Y$.

Evidently, Z takes values from $\bar{Z}$ defined as follows:

$$
\begin{aligned}
\bar{X} &= \{0, 1, \dots, I-1\}; \qquad f_i = \text{Prob}\{X = i\} \\
\bar{Y} &= \{0, 1, \dots, J-1\}; \qquad g_j = \text{Prob}\{Y = j\} \\
\bar{Z} &= \{0, 1, \dots, I+J-2\}; \qquad h_k = \text{Prob}\{X + Y = k\}
\end{aligned}
$$

Independence of X and Y implies that

$$\begin{aligned} h_k &= \text{Prob}\{X=0,Y=k\} + \text{Prob}\{X=1,Y=k-1\} + \ldots + \text{Prob}\{X=k,Y=0\} \\ h_k &= f_0 g_k + f_1 g_{k-1} + \ldots + f_{k-1} g_1 + f_k g_0 \qquad \text{for} \quad k=0,1,\ldots,I+J-2 \end{aligned}$$

Thus, we have:

$$h_k = \sum_{i=0}^{k} f_i g_{k-i} \equiv f * g$$

where $f * g$ denotes the **convolution** of the f and g sequences.

Similarly, for two random variables X, Y with densities f_X, g_Y, the density of $Z = X + Y$ is

$$f_Z(z) = \int_{-\infty}^{\infty} f_X(x) f_Y(z-x) dx \equiv f_X * g_Y$$

where $f_X * g_Y$ denotes the **convolution** of the f_X and g_Y functions.

8.15 Coupling

Start with a joint distribution

$$\begin{aligned} f_{ij} &= \text{Prob}\{X=i,Y=j\} \\ i &= 0, \cdots \ I-1 \\ j &= 0, \cdots \ J-1 \\ & \text{stacked to an } I \times J \text{ matrix} \\ & e.g. \quad I=1, J=1 \end{aligned}$$

where

$$\begin{bmatrix} f_{11} & f_{12} \\ f_{21} & f_{22} \end{bmatrix}$$

From the joint distribution, we have shown above that we obtain **unique** marginal distributions.

Now we'll try to go in a reverse direction.

We'll find that from two marginal distributions, can we usually construct more than one joint distribution that verifies these marginals.

Each of these joint distributions is called a **coupling** of the two marginal distributions.

Let's start with marginal distributions

$$\begin{aligned} \text{Prob}\{X=i\} &= \sum_j f_{ij} = \mu_i, i = 0, \cdots, I-1 \\ \text{Prob}\{Y=j\} &= \sum_i f_{ij} = \nu_j, j = 0, \cdots, J-1 \end{aligned}$$

Given two marginal distribution, μ for X and ν for Y, a joint distribution f_{ij} is said to be a **coupling** of μ and ν.

Example:

Consider the following bivariate example.

$$
\begin{aligned}
\text{Prob}\{X=0\} =& 1-q=\mu_0 \\
\text{Prob}\{X=1\} =& q=\mu_1 \\
\text{Prob}\{Y=0\} =& 1-r=\nu_0 \\
\text{Prob}\{Y=1\} =& r=\nu_1 \\
\text{where } 0\leq q<r\leq 1
\end{aligned}
$$

We construct two couplings.

The first coupling if our two marginal distributions is the joint distribution

$$
f_{ij} = \begin{bmatrix} (1-q)(1-r) & (1-q)r \\ q(1-r) & qr \end{bmatrix}
$$

To verify that it is a coupling, we check that

$$
\begin{aligned}
(1-q)(1-r)+(1-q)r+q(1-r)+qr &= 1 \\
\mu_0 = (1-q)(1-r)+(1-q)r &= 1-q \\
\mu_1 = q(1-r)+qr &= q \\
\nu_0 = (1-q)(1-r)+(1-r)q &= 1-r \\
\mu_1 = r(1-q)+qr &= r
\end{aligned}
$$

A second coupling of our two marginal distributions is the joint distribution

$$
f_{ij} = \begin{bmatrix} (1-r) & r-q \\ 0 & q \end{bmatrix}
$$

The verify that this is a coupling, note that

$$
\begin{aligned}
1-r+r-q+q &= 1 \\
\mu_0 &= 1-q \\
\mu_1 &= q \\
\nu_0 &= 1-r \\
\nu_1 &= r
\end{aligned}
$$

Thus, our two proposed joint distributions have the same marginal distributions.

But the joint distributions differ.

Thus, multiple joint distributions $[f_{ij}]$ can have the same marginals.

Remark:

- Couplings are important in optimal transport problems and in Markov processes. Please see this *lecture about optimal transport*

8.16 Copula Functions

Suppose that $X_1, X_2, \dots, X_n$ are N random variables and that

- their marginal distributions are $F_1(x_1), F_2(x_2), \dots, F_N(x_N)$, and
- their joint distribution is $H(x_1, x_2, \dots, x_N)$

Then there exists a **copula function** $C(\cdot)$ that verifies

$$H(x_1, x_2, \dots, x_N) = C(F_1(x_1), F_2(x_2), \dots, F_N(x_N)).$$

We can obtain

$$C(u_1, u_2, \dots, u_n) = H[F_1^{-1}(u_1), F_2^{-1}(u_2), \dots, F_N^{-1}(u_N)]$$

In a reverse direction of logic, given univariate **marginal distributions** $F_1(x_1), F_2(x_2), \dots, F_N(x_N)$ and a copula function $C(\cdot)$, the function $H(x_1, x_2, \dots, x_N) = C(F_1(x_1), F_2(x_2), \dots, F_N(x_N))$ is a **coupling** of $F_1(x_1), F_2(x_2), \dots, F_N(x_N)$.

Thus, for given marginal distributions, we can use a copula function to determine a joint distribution when the associated univariate random variables are not independent.

Copula functions are often used to characterize **dependence** of random variables.

Discrete marginal distribution

As mentioned above, for two given marginal distributions there can be more than one coupling.

For example, consider two random variables X, Y with distributions

$$\begin{aligned}
\text{Prob}(X = 0) &= 0.6, \\
\text{Prob}(X = 1) &= 0.4, \\
\text{Prob}(Y = 0) &= 0.3, \\
\text{Prob}(Y = 1) &= 0.7,
\end{aligned}$$

For these two random variables there can be more than one coupling.

Let's first generate X and Y.

```
# define parameters
mu = np.array([0.6, 0.4])
nu = np.array([0.3, 0.7])

# number of draws
draws = 1_000_000

# generate draws from uniform distribution
p = np.random.rand(draws)

# generate draws of X and Y via uniform distribution
x = np.ones(draws)
y = np.ones(draws)
x[p <= mu[0]] = 0
x[p > mu[0]] = 1
y[p <= nu[0]] = 0
y[p > nu[0]] = 1
```

```
# calculate parameters from draws
q_hat = sum(x[x == 1])/draws
r_hat = sum(y[y == 1])/draws

# print output
print("distribution for x")
xmtb = pt.PrettyTable()
xmtb.field_names = ['x_value', 'x_prob']
```

(continues on next page)

(continued from previous page)

```
xmtb.add_row([0, 1-q_hat])
xmtb.add_row([1, q_hat])
print(xmtb)

print("distribution for y")
ymtb = pt.PrettyTable()
ymtb.field_names = ['y_value', 'y_prob']
ymtb.add_row([0, 1-r_hat])
ymtb.add_row([1, r_hat])
print(ymtb)
```

```
distribution for x
+---------+----------+
| x_value |  x_prob  |
+---------+----------+
|    0    | 0.600358 |
|    1    | 0.399642 |
+---------+----------+
distribution for y
+---------+---------------------+
| y_value |        y_prob       |
+---------+---------------------+
|    0    | 0.30045900000000003 |
|    1    |       0.699541      |
+---------+---------------------+
```

Let's now take our two marginal distributions, one for X, the other for Y, and construct two distinct couplings.

For the first joint distribution:

$$\text{Prob}(X = i, Y = j) = f_{ij}$$

where

$$[f_{ij}] = \begin{bmatrix} 0.18 & 0.42 \\ 0.12 & 0.28 \end{bmatrix}$$

Let's use Python to construct this joint distribution and then verify that its marginal distributions are what we want.

```
# define parameters
f1 = np.array([[0.18, 0.42], [0.12, 0.28]])
f1_cum = np.cumsum(f1)

# number of draws
draws1 = 1_000_000

# generate draws from uniform distribution
p = np.random.rand(draws1)

# generate draws of first copuling via uniform distribution
c1 = np.vstack([np.ones(draws1), np.ones(draws1)])
# X=0, Y=0
c1[0, p <= f1_cum[0]] = 0
c1[1, p <= f1_cum[0]] = 0
# X=0, Y=1
c1[0, (p > f1_cum[0])*(p <= f1_cum[1])] = 0
c1[1, (p > f1_cum[0])*(p <= f1_cum[1])] = 1
```

(continues on next page)

(continued from previous page)

```
# X=1, Y=0
c1[0, (p > f1_cum[1])*(p <= f1_cum[2])] = 1
c1[1, (p > f1_cum[1])*(p <= f1_cum[2])] = 0
# X=1, Y=1
c1[0, (p > f1_cum[2])*(p <= f1_cum[3])] = 1
c1[1, (p > f1_cum[2])*(p <= f1_cum[3])] = 1
```

```
# calculate parameters from draws
f1_00 = sum((c1[0, :] == 0)*(c1[1, :] == 0))/draws1
f1_01 = sum((c1[0, :] == 0)*(c1[1, :] == 1))/draws1
f1_10 = sum((c1[0, :] == 1)*(c1[1, :] == 0))/draws1
f1_11 = sum((c1[0, :] == 1)*(c1[1, :] == 1))/draws1

# print output of first joint distribution
print("first joint distribution for c1")
c1_mtb = pt.PrettyTable()
c1_mtb.field_names = ['c1_x_value', 'c1_y_value', 'c1_prob']
c1_mtb.add_row([0, 0, f1_00])
c1_mtb.add_row([0, 1, f1_01])
c1_mtb.add_row([1, 0, f1_10])
c1_mtb.add_row([1, 1, f1_11])
print(c1_mtb)
```

```
first joint distribution for c1
+------------+------------+----------+
| c1_x_value | c1_y_value | c1_prob  |
+------------+------------+----------+
|     0      |     0      | 0.180516 |
|     0      |     1      | 0.419972 |
|     1      |     0      | 0.119884 |
|     1      |     1      | 0.279628 |
+------------+------------+----------+
```

```
# calculate parameters from draws
c1_q_hat = sum(c1[0, :] == 1)/draws1
c1_r_hat = sum(c1[1, :] == 1)/draws1

# print output
print("marginal distribution for x")
c1_x_mtb = pt.PrettyTable()
c1_x_mtb.field_names = ['c1_x_value', 'c1_x_prob']
c1_x_mtb.add_row([0, 1-c1_q_hat])
c1_x_mtb.add_row([1, c1_q_hat])
print(c1_x_mtb)

print("marginal distribution for y")
c1_ymtb = pt.PrettyTable()
c1_ymtb.field_names = ['c1_y_value', 'c1_y_prob']
c1_ymtb.add_row([0, 1-c1_r_hat])
c1_ymtb.add_row([1, c1_r_hat])
print(c1_ymtb)
```

```
marginal distribution for x
+------------+-----------+
| c1_x_value | c1_x_prob |
```

(continues on next page)

(continued from previous page)

```
+------------+-----------+
|     0      |  0.600488 |
|     1      |  0.399512 |
+------------+-----------+
marginal distribution for y
+------------+-----------+
| c1_y_value | c1_y_prob |
+------------+-----------+
|     0      |   0.3004  |
|     1      |   0.6996  |
+------------+-----------+
```

Now, let's construct another joint distribution that is also a coupling of X and Y

$$[f_{ij}] = \left[\begin{array}{cc} 0.3 & 0.3 \\ 0 & 0.4 \end{array} \right]$$

```python
# define parameters
f2 = np.array([[0.3, 0.3], [0, 0.4]])
f2_cum = np.cumsum(f2)

# number of draws
draws2 = 1_000_000

# generate draws from uniform distribution
p = np.random.rand(draws2)

# generate draws of first coupling via uniform distribution
c2 = np.vstack([np.ones(draws2), np.ones(draws2)])
# X=0, Y=0
c2[0, p <= f2_cum[0]] = 0
c2[1, p <= f2_cum[0]] = 0
# X=0, Y=1
c2[0, (p > f2_cum[0])*(p <= f2_cum[1])] = 0
c2[1, (p > f2_cum[0])*(p <= f2_cum[1])] = 1
# X=1, Y=0
c2[0, (p > f2_cum[1])*(p <= f2_cum[2])] = 1
c2[1, (p > f2_cum[1])*(p <= f2_cum[2])] = 0
# X=1, Y=1
c2[0, (p > f2_cum[2])*(p <= f2_cum[3])] = 1
c2[1, (p > f2_cum[2])*(p <= f2_cum[3])] = 1
```

```python
# calculate parameters from draws
f2_00 = sum((c2[0, :] == 0)*(c2[1, :] == 0))/draws2
f2_01 = sum((c2[0, :] == 0)*(c2[1, :] == 1))/draws2
f2_10 = sum((c2[0, :] == 1)*(c2[1, :] == 0))/draws2
f2_11 = sum((c2[0, :] == 1)*(c2[1, :] == 1))/draws2

# print output of second joint distribution
print("first joint distribution for c2")
c2_mtb = pt.PrettyTable()
c2_mtb.field_names = ['c2_x_value', 'c2_y_value', 'c2_prob']
c2_mtb.add_row([0, 0, f2_00])
c2_mtb.add_row([0, 1, f2_01])
c2_mtb.add_row([1, 0, f2_10])
c2_mtb.add_row([1, 1, f2_11])
print(c2_mtb)
```

```
first joint distribution for c2
+------------+------------+----------+
| c2_x_value | c2_y_value | c2_prob  |
+------------+------------+----------+
|     0      |     0      | 0.299057 |
|     0      |     1      | 0.300228 |
|     1      |     0      |   0.0    |
|     1      |     1      | 0.400715 |
+------------+------------+----------+
```

```
# calculate parameters from draws
c2_q_hat = sum(c2[0, :] == 1)/draws2
c2_r_hat = sum(c2[1, :] == 1)/draws2

# print output
print("marginal distribution for x")
c2_x_mtb = pt.PrettyTable()
c2_x_mtb.field_names = ['c2_x_value', 'c2_x_prob']
c2_x_mtb.add_row([0, 1-c2_q_hat])
c2_x_mtb.add_row([1, c2_q_hat])
print(c2_x_mtb)

print("marginal distribution for y")
c2_ymtb = pt.PrettyTable()
c2_ymtb.field_names = ['c2_y_value', 'c2_y_prob']
c2_ymtb.add_row([0, 1-c2_r_hat])
c2_ymtb.add_row([1, c2_r_hat])
print(c2_ymtb)
```

```
marginal distribution for x
+------------+--------------------+
| c2_x_value |     c2_x_prob      |
+------------+--------------------+
|     0      | 0.5992850000000001 |
|     1      |      0.400715      |
+------------+--------------------+
marginal distribution for y
+------------+-----------+
| c2_y_value | c2_y_prob |
+------------+-----------+
|     0      |  0.299057 |
|     1      |  0.700943 |
+------------+-----------+
```

We have verified that both joint distributions, c_1 and c_2, have identical marginal distributions of X and Y, respectively. So they are both couplings of X and Y.

CHAPTER

NINE

SOME PROBABILITY DISTRIBUTIONS

This lecture is a supplement to *this lecture on statistics with matrices*.

It describes some popular distributions and uses Python to sample from them.

It also describes a way to sample from an arbitrary probability distribution that you make up by transforming a sample from a uniform probability distribution.

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install prettytable
```

As usual, we'll start with some imports

```
import numpy as np
import matplotlib.pyplot as plt
import prettytable as pt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib_inline.backend_inline import set_matplotlib_formats
set_matplotlib_formats('retina')
```

9.1 Some Discrete Probability Distributions

Let's write some Python code to compute means and variances of some univariate random variables.

We'll use our code to

- compute population means and variances from the probability distribution
- generate a sample of N independently and identically distributed draws and compute sample means and variances
- compare population and sample means and variances

9.2 Geometric distribution

A discrete geometric distribution has probability mass function

$$\textrm{Prob}(X=k)=(1-p)^{k-1}p, k=1,2,\ldots,\quad p\in(0,1)$$

where $k=1,2,\ldots$ is the number of trials before the first success.

The mean and variance of this one-parameter probability distribution are

$$\mathbb{E}(X) = \frac{1}{p}$$
$$\mathbb{Var}(X) = \frac{1-p}{p^2}$$

Let's use Python draw observations from the distribution and compare the sample mean and variance with the theoretical results.

```
# specify parameters
p, n = 0.3, 1_000_000

# draw observations from the distribution
x = np.random.geometric(p, n)

# compute sample mean and variance
μ_hat = np.mean(x)
σ2_hat = np.var(x)

print("The sample mean is: ", μ_hat, "\nThe sample variance is: ", σ2_hat)

# compare with theoretical results
print("\nThe population mean is: ", 1/p)
print("The population variance is: ", (1-p)/(p**2))
```

```
The sample mean is:  3.329139
The sample variance is:  7.768654518679001

The population mean is:  3.3333333333333335
The population variance is:  7.777777777777778
```

9.3 Pascal (negative binomial) distribution

Consider a sequence of independent Bernoulli trials.

Let p be the probability of success.

Let X be a random variable that represents the number of failures before we get r successes.

Its distribution is

$$X \sim NB(r, p)$$
$$\text{Prob}(X = k; r, p) = \begin{bmatrix} k+r-1 \\ r-1 \end{bmatrix} p^r (1-p)^k$$

Here, we choose from among $k + r - 1$ possible outcomes because the last draw is by definition a success.

We compute the mean and variance to be

$$\mathbb{E}(X) = \frac{k(1-p)}{p}$$
$$\mathbb{V}(X) = \frac{k(1-p)}{p^2}$$

```
# specify parameters
r, p, n = 10, 0.3, 1_000_000

# draw observations from the distribution
x = np.random.negative_binomial(r, p, n)

# compute sample mean and variance
μ_hat = np.mean(x)
σ2_hat = np.var(x)

print("The sample mean is: ", μ_hat, "\nThe sample variance is: ", σ2_hat)
print("\nThe population mean is: ", r*(1-p)/p)
print("The population variance is: ", r*(1-p)/p**2)
```

```
The sample mean is:  23.334072
The sample variance is:  78.015829898816

The population mean is:  23.333333333333336
The population variance is:  77.77777777777779
```

9.4 Newcomb–Benford distribution

The **Newcomb–Benford law** fits many data sets, e.g., reports of incomes to tax authorities, in which the leading digit is more likely to be small than large.

See https://en.wikipedia.org/wiki/Benford's_law

A Benford probability distribution is

$$\text{Prob}\{X = d\} = \log_{10}(d+1) - \log_{10}(d) = \log_{10}\left(1 + \frac{1}{d}\right)$$

where $d \in \{1, 2, \cdots, 9\}$ can be thought of as a **first digit** in a sequence of digits.

This is a well defined discrete distribution since we can verify that probabilities are nonnegative and sum to 1.

$$\log_{10}\left(1 + \frac{1}{d}\right) \geq 0, \quad \sum_{d=1}^{9} \log_{10}\left(1 + \frac{1}{d}\right) = 1$$

The mean and variance of a Benford distribution are

$$\begin{aligned}
\mathbb{E}[X] &= \sum_{d=1}^{9} d \log_{10}\left(1 + \frac{1}{d}\right) \simeq 3.4402 \\
\mathbb{V}[X] &= \sum_{d=1}^{9} \left(d - \mathbb{E}[X]\right)^2 \log_{10}\left(1 + \frac{1}{d}\right) \simeq 6.0565
\end{aligned}$$

We verify the above and compute the mean and variance using `numpy`.

```
Benford_pmf = np.array([np.log10(1+1/d) for d in range(1,10)])
k = np.arange(1, 10)

# mean
mean = k @ Benford_pmf
```

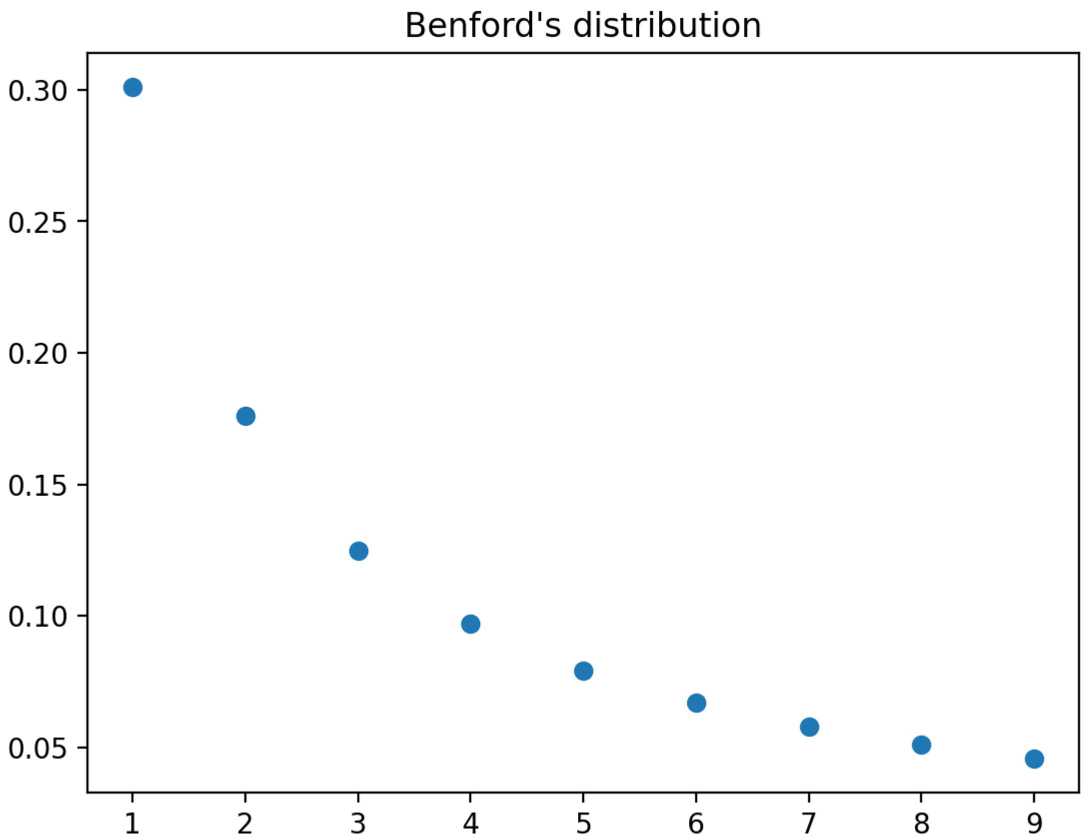
(continues on next page)

(continued from previous page)

```
# variance
var = ((k - mean) ** 2) @ Benford_pmf

# verify sum to 1
print(np.sum(Benford_pmf))
print(mean)
print(var)
```

```
0.9999999999999999
3.4402369671232065
6.056512631375667
```

```
# plot distribution
plt.plot(range(1,10), Benford_pmf, 'o')
plt.title('Benford\'s distribution')
plt.show()
```



Now let's turn to some continuous random variables.

9.5 Univariate Gaussian distribution

We write

$$X \sim N(\mu, \sigma^2)$$

to indicate the probability distribution

$$f(x|u, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{[-\frac{1}{2\sigma^2}(x-u)^2]}$$

In the below example, we set $\mu = 0, \sigma = 0.1$.

```
# specify parameters
μ, σ = 0, 0.1

# specify number of draws
n = 1_000_000

# draw observations from the distribution
x = np.random.normal(μ, σ, n)

# compute sample mean and variance
μ_hat = np.mean(x)
σ_hat = np.std(x)

print("The sample mean is: ", μ_hat)
print("The sample standard deviation is: ", σ_hat)
```

```
The sample mean is:  -0.00011526135254952218
The sample standard deviation is:  0.09986625370568251
```

```
# compare
print(μ-μ_hat < 1e-3)
print(σ-σ_hat < 1e-3)
```

```
True
True
```

9.6 Uniform Distribution

$$X \sim U[a, b]$$

$$f(x) = \begin{cases} \frac{1}{b-a}, & a \leq x \leq b \\ 0, & \text{otherwise} \end{cases}$$

The population mean and variance are

$$\mathbb{E}(X) = \frac{a+b}{2}$$

$$\mathbb{V}(X) = \frac{(b-a)^2}{12}$$

```
# specify parameters
a, b = 10, 20

# specify number of draws
n = 1_000_000

# draw observations from the distribution
x = a + (b-a)*np.random.rand(n)

# compute sample mean and variance
μ_hat = np.mean(x)
σ2_hat = np.var(x)

print("The sample mean is: ", μ_hat, "\nThe sample variance is: ", σ2_hat)
print("\nThe population mean is: ", (a+b)/2)
print("The population variance is: ", (b-a)**2/12)
```

```
The sample mean is:  15.000180944104585
The sample variance is:  8.330881181556308

The population mean is:  15.0
The population variance is:  8.333333333333334
```

9.7 A Mixed Discrete-Continuous Distribution

We'll motivate this example with a little story.

Suppose that to apply for a job you take an interview and either pass or fail it.

You have 5% chance to pass an interview and you know your salary will uniformly distributed in the interval 300~400 a day only if you pass.

We can describe your daily salary as a discrete-continuous variable with the following probabilities:

$$P(X = 0) = 0.95$$

$$P(300 \le X \le 400) = \int_{300}^{400} f(x)\, dx = 0.05$$

$$f(x) = 0.0005$$

Let's start by generating a random sample and computing sample moments.

```
x = np.random.rand(1_000_000)
# x[x > 0.95] = 100*x[x > 0.95]+300
x[x > 0.95] = 100*np.random.rand(len(x[x > 0.95]))+300
x[x <= 0.95] = 0

μ_hat = np.mean(x)
σ2_hat = np.var(x)

print("The sample mean is: ", μ_hat, "\nThe sample variance is: ", σ2_hat)
```

```
The sample mean is:  17.532090366953398
The sample variance is:  5871.996808415499
```

The analytical mean and variance can be computed:

$$
\begin{aligned}
\mu &= \int_{300}^{400} x f(x) dx \\
&= 0.0005 \int_{300}^{400} x dx \\
&= 0.0005 \times \frac{1}{2} x^2 \bigg|_{300}^{400}
\end{aligned}
$$

$$
\begin{aligned}
\sigma^2 &= 0.95 \times (0 - 17.5)^2 + \int_{300}^{400} (x - 17.5)^2 f(x) dx \\
&= 0.95 \times 17.5^2 + 0.0005 \int_{300}^{400} (x - 17.5)^2 dx \\
&= 0.95 \times 17.5^2 + 0.0005 \times \frac{1}{3} (x - 17.5)^3 \bigg|_{300}^{400}
\end{aligned}
$$

```
mean = 0.0005*0.5*(400**2 - 300**2)
var = 0.95*17.5**2+0.0005/3*((400-17.5)**3-(300-17.5)**3)
print("mean: ", mean)
print("variance: ", var)
```

```
mean:  17.5
variance:  5860.416666666666
```

9.8 Drawing a Random Number from a Particular Distribution

Suppose we have at our disposal a pseudo random number that draws a uniform random variable, i.e., one with probability distribution

$$
\text{Prob}\{\tilde{X} = i\} = \frac{1}{I}, \quad i = 0, \ldots, I - 1
$$

How can we transform $\tilde{X}$ to get a random variable X for which $\text{Prob}\{X = i\} = f_i, \quad i = 0, \ldots, I - 1$, where f_i is an arbitary discrete probability distribution on $i = 0, 1, \ldots, I - 1$?

The key tool is the inverse of a cumulative distribution function (CDF).

Observe that the CDF of a distribution is monotone and non-decreasing, taking values between 0 and 1.

We can draw a sample of a random variable X with a known CDF as follows:

- draw a random variable u from a uniform distribution on $[0, 1]$
- pass the sample value of u into the **"inverse"** target CDF for X
- X has the target CDF

Thus, knowing the **"inverse"** CDF of a distribution is enough to simulate from this distribution.

> **Note**
>
> The "inverse" CDF needs to exist for this method to work.

The inverse CDF is

$$F^{-1}(u) \equiv \inf\{x \in \mathbb{R} : F(x) \geq u\} \quad (0 < u < 1)$$

Here we use infimum because a CDF is a non-decreasing and right-continuous function.

Thus, suppose that

- U is a uniform random variable $U \in [0, 1]$
- We want to sample a random variable X whose CDF is F.

It turns out that if we use draw uniform random numbers U and then compute X from

$$X = F^{-1}(U),$$

then X is a random variable with CDF $F_X(x) = F(x) = \text{Prob}\{X \leq x\}$.

We'll verify this in the special case in which F is continuous and bijective so that its inverse function exists and can be denoted by F^{-1}.

Note that

$$\begin{aligned}
F_X(x) &= \text{Prob}\{X \leq x\} \\
&= \text{Prob}\{F^{-1}(U) \leq x\} \\
&= \text{Prob}\{U \leq F(x)\} \\
&= F(x)
\end{aligned}$$

where the last equality occurs because U is distributed uniformly on $[0, 1]$ while $F(x)$ is a constant given x that also lies on $[0, 1]$.

Let's use `numpy` to compute some examples.

Example: A continuous geometric (exponential) distribution

Let X follow a geometric distribution, with parameter $\lambda > 0$.

Its density function is

$$f(x) = \lambda e^{-\lambda x}$$

Its CDF is

$$F(x) = \int_0^\infty \lambda e^{-\lambda x} = 1 - e^{-\lambda x}$$

Let U follow a uniform distribution on $[0, 1]$.

X is a random variable such that $U = F(X)$.

The distribution X can be deduced from

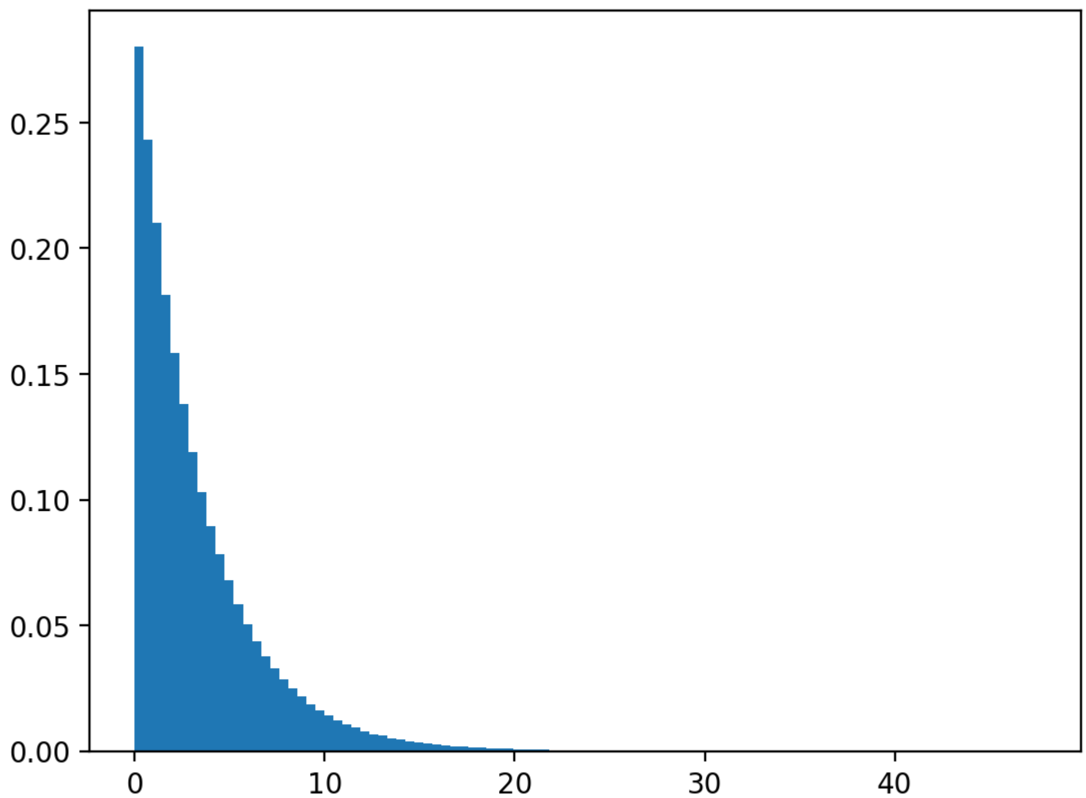
$$\begin{aligned}
& U = F(X) = 1 - e^{-\lambda X} \\
\implies \quad & -U = e^{-\lambda X} \\
\implies \quad & \log(1 - U) = -\lambda X \\
\implies \quad & X = \frac{(1 - U)}{-\lambda}
\end{aligned}$$

Let's draw u from $U[0, 1]$ and calculate $x = \frac{log(1-U)}{-\lambda}$.

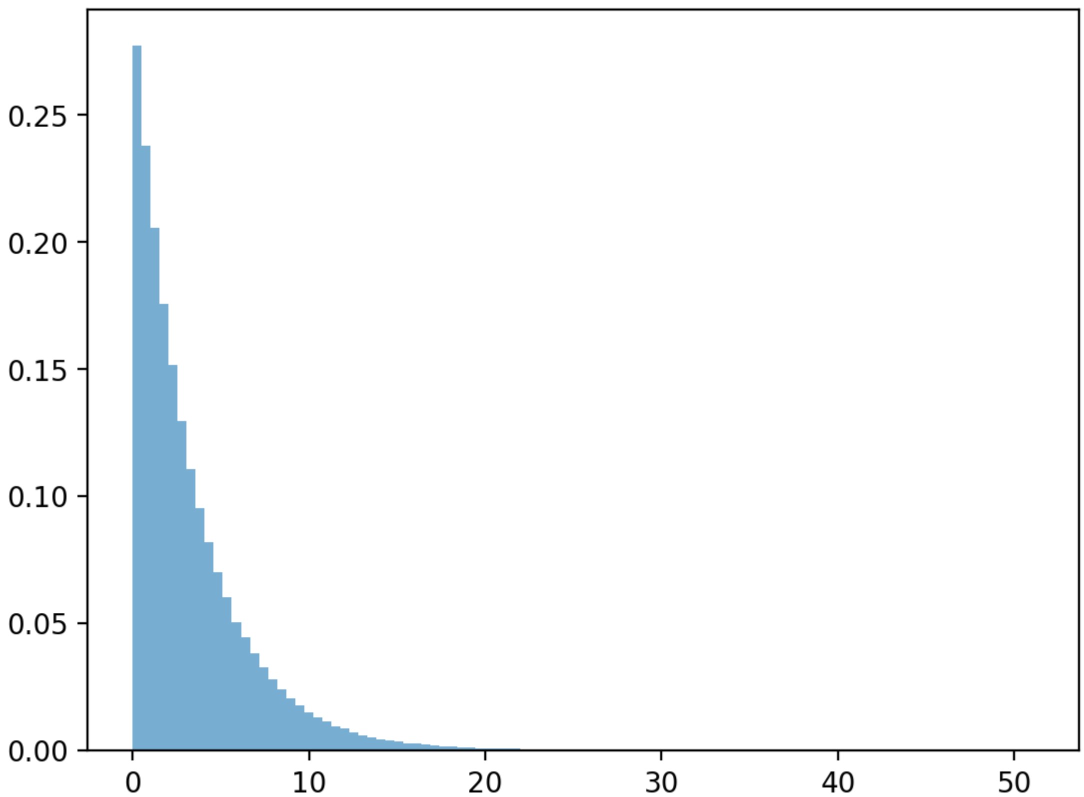
We'll check whether X seems to follow a **continuous geometric** (exponential) distribution.

Let's check with `numpy`.

```
n, λ = 1_000_000, 0.3

# draw uniform numbers
u = np.random.rand(n)

# transform
x = -np.log(1-u)/λ

# draw geometric distributions
x_g = np.random.exponential(1 / λ, n)

# plot and compare
plt.hist(x, bins=100, density=True)
plt.show()
```




```
plt.hist(x_g, bins=100, density=True, alpha=0.6)
plt.show()
```



Geometric distribution

Let X distributed geometrically, that is

$$\text{Prob}(X = i) = (1-\lambda)\lambda^i, \quad \lambda \in (0,1), \quad i = 0, 1, \dots$$

$$\sum_{i=0}^{\infty} \text{Prob}(X = i) = 1 \longleftrightarrow (1-\lambda)\sum_{i=0}^{\infty} \lambda^i = \frac{1-\lambda}{1-\lambda} = 1$$

Its CDF is given by

$$\begin{aligned}
\text{Prob}(X \le i) &= (1-\lambda)\sum_{j=0}^{i} \lambda^i \\
&= (1-\lambda)[\frac{1-\lambda^{i+1}}{1-\lambda}] \\
&= 1-\lambda^{i+1} \\
&= F(X) = F_i
\end{aligned}$$

Again, let $\tilde{U}$ follow a uniform distribution and we want to find X such that $F(X) = \tilde{U}$.

Let's deduce the distribution of X from

$$\begin{aligned}
\tilde{U} &= F(X) = 1-\lambda^{x+1} \\
1-\tilde{U} &= \lambda^{x+1} \\
\log(1-\tilde{U}) &= (x+1)\log\lambda \\
\frac{\log(1-\tilde{U})}{\log\lambda} &= x+1 \\
\frac{\log(1-\tilde{U})}{\log\lambda} - 1 &= x
\end{aligned}$$

However, $\tilde{U} = F^{-1}(X)$ may not be an integer for any $x \geq 0$.

So let

$$x = \lceil \frac{\log(1-\tilde{U})}{\log \lambda} - 1 \rceil$$

where $\lceil . \rceil$ is the ceiling function.

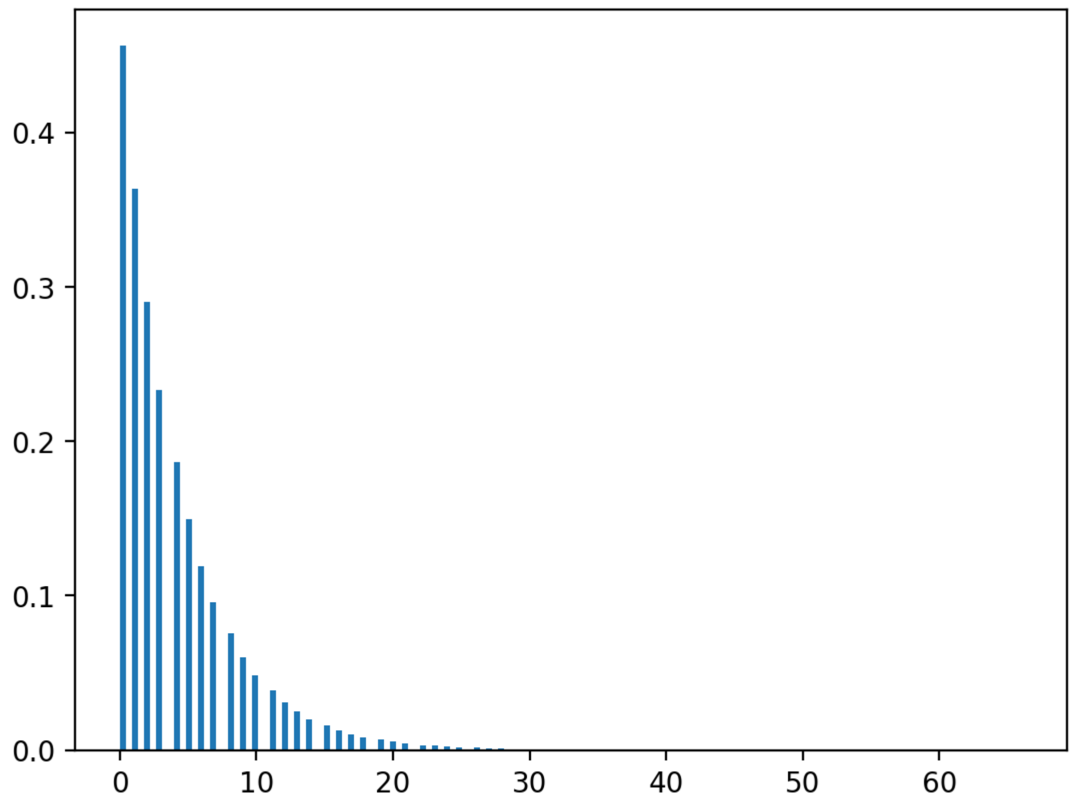
Thus x is the smallest integer such that the discrete geometric CDF is greater than or equal to $\tilde{U}$.

We can verify that x is indeed geometrically distributed by the following `numpy` program.

Note

The exponential distribution is the continuous analog of geometric distribution.

```
n, λ = 1_000_000, 0.8

# draw uniform numbers
u = np.random.rand(n)

# transform
x = np.ceil(np.log(1-u)/np.log(λ) - 1)

# draw geometric distributions
x_g = np.random.geometric(1-λ, n)

# plot and compare
plt.hist(x, bins=150, density=True)
plt.show()
```



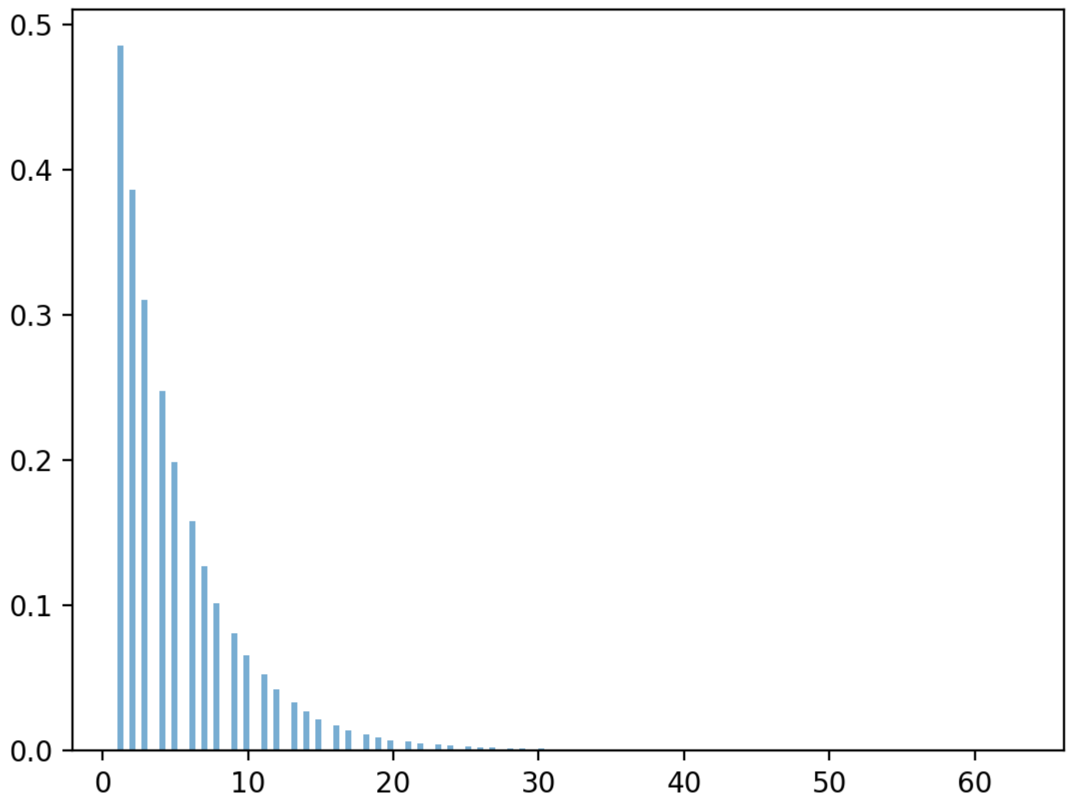

```
np.random.geometric(1-λ, n).max()
```

```
np.int64(64)
```

```
np.log(0.4)/np.log(0.3)
```

```
np.float64(0.7610560044063083)
```

```
plt.hist(x_g, bins=150, density=True, alpha=0.6)
plt.show()
```


0.5
0.4
0.3
0.2
0.1
0.0
0
10
20
30
40
50
60

CHAPTER

TEN

LLN AND CLT

Contents

10.1 Overview

This lecture illustrates two of the most important theorems of probability and statistics: The law of large numbers (LLN) and the central limit theorem (CLT).

These beautiful theorems lie behind many of the most fundamental results in econometrics and quantitative economic modeling.

The lecture is based around simulations that show the LLN and CLT in action.

We also demonstrate how the LLN and CLT break down when the assumptions they are based on do not hold.

In addition, we examine several useful extensions of the classical theorems, such as

- The delta method, for smooth functions of random variables, and
- the multivariate case.

Some of these extensions are presented as exercises.

We'll need the following imports:

```
import matplotlib.pyplot as plt
import random
import numpy as np
from scipy.stats import t, beta, lognorm, expon, gamma, uniform
from scipy.stats import gaussian_kde, poisson, binom, norm, chi2
from mpl_toolkits.mplot3d import Axes3D
```

(continues on next page)

(continued from previous page)

```
from matplotlib.collections import PolyCollection
from scipy.linalg import inv, sqrtm
```

10.2 Relationships

The CLT refines the LLN.

The LLN gives conditions under which sample moments converge to population moments as sample size increases.

The CLT provides information about the rate at which sample moments converge to population moments as sample size increases.

10.3 LLN

We begin with the law of large numbers, which tells us when sample averages will converge to their population means.

10.3.1 The Classical LLN

The classical law of large numbers concerns independent and identically distributed (IID) random variables.

Here is the strongest version of the classical LLN, known as **Kolmogorov's strong law**.

Let $X_1, \ldots, X_n$ be independent and identically distributed scalar random variables, with common distribution F.

When it exists, let μ denote the common mean of this sample:

$$\mu := \mathbb{E}X = \int xF(dx)$$

In addition, let

$$\bar{X}_n := \frac{1}{n} \sum_{i=1}^{n} X_i$$

Kolmogorov's strong law states that, if $\mathbb{E}|X|$ is finite, then

$$\mathbb{P}\left\{\bar{X}_n \to \mu \text{ as } n \to \infty\right\} = 1 \tag{10.1}$$

What does this last expression mean?

Let's think about it from a simulation perspective, imagining for a moment that our computer can generate perfect random samples (which of course it can't).

Let's also imagine that we can generate infinite sequences so that the statement $\bar{X}_n \to \mu$ can be evaluated.

In this setting, (10.1) should be interpreted as meaning that the probability of the computer producing a sequence where $\bar{X}_n \to \mu$ fails to occur is zero.

10.3.2 Proof

The proof of Kolmogorov's strong law is nontrivial – see, for example, theorem 8.3.5 of [Dudley, 2002].

On the other hand, we can prove a weaker version of the LLN very easily and still get most of the intuition.

The version we prove is as follows: If $X_1, \ldots, X_n$ is IID with $\mathbb{E}X_i^2 < \infty$, then, for any $\epsilon > 0$, we have

$$\mathbb{P}\left\{|\bar{X}_n - \mu| \geq \epsilon\right\} \to 0 \quad \text{as} \quad n \to \infty \tag{10.2}$$

(This version is weaker because we claim only convergence in probability rather than almost sure convergence, and assume a finite second moment)

To see that this is so, fix $\epsilon > 0$, and let σ^2 be the variance of each X_i.

Recall the Chebyshev inequality, which tells us that

$$\mathbb{P}\left\{|\bar{X}_n - \mu| \geq \epsilon\right\} \leq \frac{\mathbb{E}[(\bar{X}_n - \mu)^2]}{\epsilon^2} \tag{10.3}$$

Now observe that

$$\begin{aligned}
\mathbb{E}[(\bar{X}_n - \mu)^2] &= \mathbb{E}\left\{\left[\frac{1}{n}\sum_{i=1}^n (X_i - \mu)\right]^2\right\} \\
&= \frac{1}{n^2}\sum_{i=1}^n \sum_{j=1}^n \mathbb{E}(X_i - \mu)(X_j - \mu) \\
&= \frac{1}{n^2}\sum_{i=1}^n \mathbb{E}(X_i - \mu)^2 \\
&= \frac{\sigma^2}{n}
\end{aligned}$$

Here the crucial step is at the third equality, which follows from independence.

Independence means that if $i \neq j$, then the covariance term $\mathbb{E}(X_i - \mu)(X_j - \mu)$ drops out.

As a result, $n^2 - n$ terms vanish, leading us to a final expression that goes to zero in n.

Combining our last result with (10.3), we come to the estimate

$$\mathbb{P}\left\{|\bar{X}_n - \mu| \geq \epsilon\right\} \leq \frac{\sigma^2}{n\epsilon^2} \tag{10.4}$$

The claim in (10.2) is now clear.

Of course, if the sequence $X_1, \ldots, X_n$ is correlated, then the cross-product terms $\mathbb{E}(X_i - \mu)(X_j - \mu)$ are not necessarily zero.

While this doesn't mean that the same line of argument is impossible, it does mean that if we want a similar result then the covariances should be "almost zero" for "most" of these terms.

In a long sequence, this would be true if, for example, $\mathbb{E}(X_i - \mu)(X_j - \mu)$ approached zero when the difference between i and j became large.

In other words, the LLN can still work if the sequence $X_1, \ldots, X_n$ has a kind of "asymptotic independence", in the sense that correlation falls to zero as variables become further apart in the sequence.

This idea is very important in time series analysis, and we'll come across it again soon enough.

10.3.3 Illustration

Let's now illustrate the classical IID law of large numbers using simulation.

In particular, we aim to generate some sequences of IID random variables and plot the evolution of $\bar{X}_n$ as n increases.

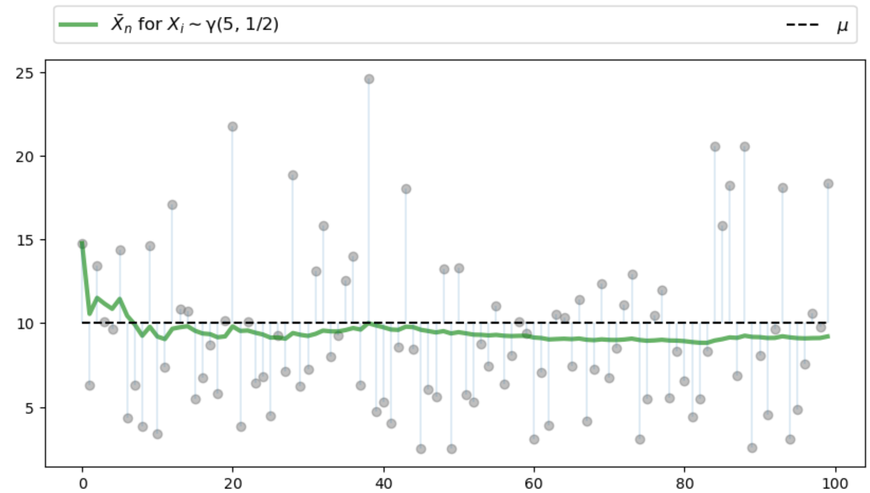
Below is a figure that does just this (as usual, you can click on it to expand it).

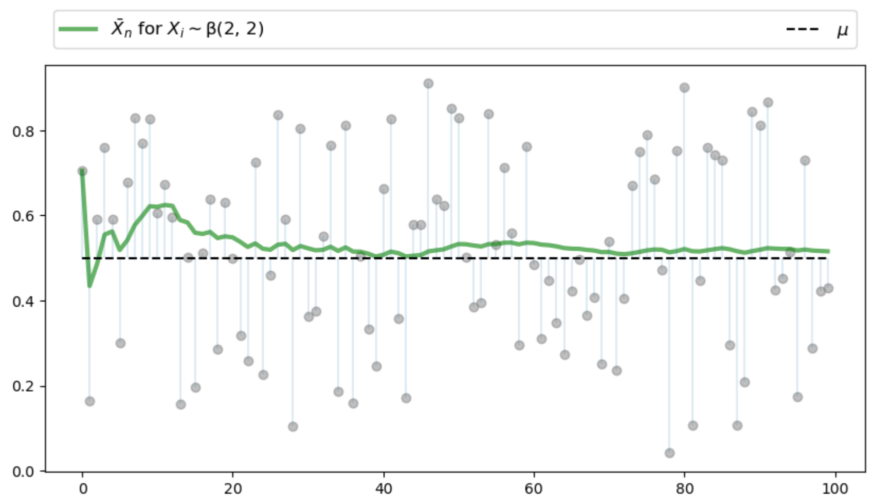
It shows IID observations from three different distributions and plots $\bar{X}_n$ against n in each case.

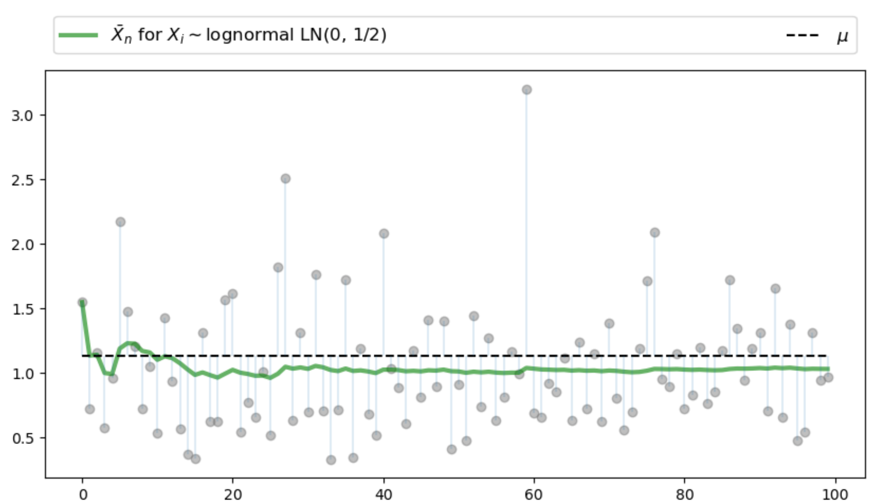
The dots represent the underlying observations X_i for $i = 1, \ldots, 100$.

In each of the three cases, convergence of $\bar{X}_n$ to μ occurs as predicted

```
n = 100

# Arbitrary collection of distributions
distributions = {"student's t with 10 degrees of freedom": t(10),
                 "β(2, 2)": beta(2, 2),
                 "lognormal LN(0, 1/2)": lognorm(0.5),
                 "γ(5, 1/2)": gamma(5, scale=2),
                 "poisson(4)": poisson(4),
                 "exponential with λ = 1": expon(1)}

# Create a figure and some axes
num_plots = 3
fig, axes = plt.subplots(num_plots, 1, figsize=(10, 20))

# Set some plotting parameters to improve layout
bbox = (0., 1.02, 1., .102)
legend_args = {'ncol': 2,
               'bbox_to_anchor': bbox,
               'loc': 3,
               'mode': 'expand'}
plt.subplots_adjust(hspace=0.5)

for ax in axes:
    # Choose a randomly selected distribution
    name = random.choice(list(distributions.keys()))
    distribution = distributions.pop(name)

    # Generate n draws from the distribution
    data = distribution.rvs(n)

    # Compute sample mean at each n
    sample_mean = np.empty(n)
    for i in range(n):
        sample_mean[i] = np.mean(data[:i+1])

    # Plot
    ax.plot(list(range(n)), data, 'o', color='grey', alpha=0.5)
    axlabel = r'$\bar{X}_n$ for $X_i \sim$' + name
    ax.plot(list(range(n)), sample_mean, 'g-', lw=3, alpha=0.6, label=axlabel)
    m = distribution.mean()
    ax.plot(list(range(n)), [m] * n, 'k--', lw=1.5, label=r'$\mu$')
    ax.vlines(list(range(n)), m, data, lw=0.2)
    ax.legend(**legend_args, fontsize=12)

plt.show()
```


$\bar{X}_n$ for $X_i \sim \gamma(5, 1/2)$
μ


$\bar{X}_n$ for $X_i \sim \beta(2, 2)$
μ


$\bar{X}_n$ for $X_i \sim$ lognormal LN(0, 1/2)
μ

The three distributions are chosen at random from a selection stored in the dictionary `distributions`.

10.4 CLT

Next, we turn to the central limit theorem, which tells us about the distribution of the deviation between sample averages and population means.

10.4.1 Statement of the Theorem

The central limit theorem is one of the most remarkable results in all of mathematics.

In the classical IID setting, it tells us the following:

If the sequence $X_1, \ldots, X_n$ is IID, with common mean μ and common variance $\sigma^2 \in (0, \infty)$, then

$$\sqrt{n}(\bar{X}_n - \mu) \stackrel{d}{\to} N(0, \sigma^2) \quad \text{as} \quad n \to \infty \tag{10.5}$$

Here $\stackrel{d}{\to} N(0, \sigma^2)$ indicates convergence in distribution to a centered (i.e, zero mean) normal with standard deviation σ.

10.4.2 Intuition

The striking implication of the CLT is that for **any** distribution with finite second moment, the simple operation of adding independent copies **always** leads to a Gaussian curve.

A relatively simple proof of the central limit theorem can be obtained by working with characteristic functions (see, e.g., theorem 9.5.6 of [Dudley, 2002]).

The proof is elegant but almost anticlimactic, and it provides surprisingly little intuition.

In fact, all of the proofs of the CLT that we know are similar in this respect.

Why does adding independent copies produce a bell-shaped distribution?

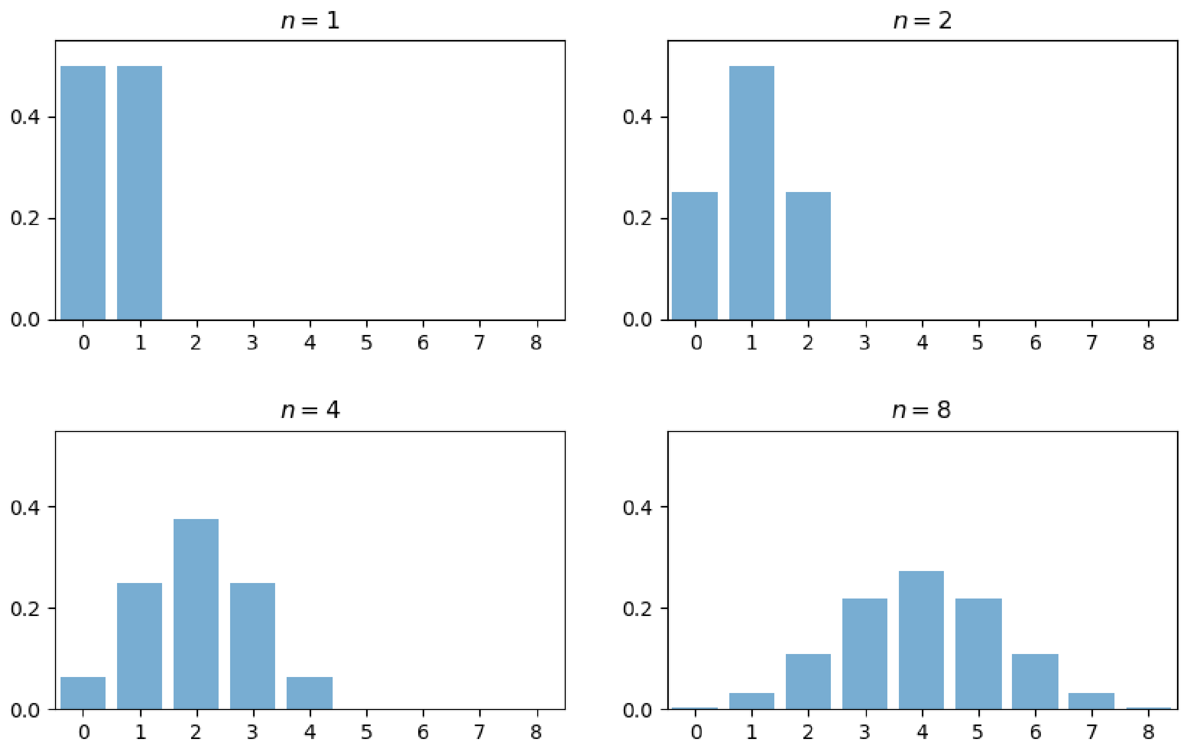
Part of the answer can be obtained by investigating the addition of independent Bernoulli random variables.

In particular, let X_i be binary, with $\mathbb{P}\{X_i = 0\} = \mathbb{P}\{X_i = 1\} = 0.5$, and let $X_1, \ldots, X_n$ be independent.

Think of $X_i = 1$ as a "success", so that $Y_n = \sum_{i=1}^n X_i$ is the number of successes in n trials.

The next figure plots the probability mass function of Y_n for $n = 1, 2, 4, 8$

```
fig, axes = plt.subplots(2, 2, figsize=(10, 6))
plt.subplots_adjust(hspace=0.4)
axes = axes.flatten()
ns = [1, 2, 4, 8]
dom = list(range(9))

for ax, n in zip(axes, ns):
    b = binom(n, 0.5)
    ax.bar(dom, b.pmf(dom), alpha=0.6, align='center')
    ax.set(xlim=(-0.5, 8.5), ylim=(0, 0.55),
           xticks=list(range(9)), yticks=(0, 0.2, 0.4),
           title=f'$n = {n}$')

plt.show()
```



When $n = 1$, the distribution is flat — one success or no successes have the same probability.

When $n = 2$ we can either have 0, 1 or 2 successes.

Notice the peak in probability mass at the mid-point $k = 1$.

The reason is that there are more ways to get 1 success (“fail then succeed” or “succeed then fail”) than to get zero or two successes.

Moreover, the two trials are independent, so the outcomes “fail then succeed” and “succeed then fail” are just as likely as the outcomes “fail then fail” and “succeed then succeed”.

(If there was positive correlation, say, then “succeed then fail” would be less likely than “succeed then succeed”)

Here, already we have the essence of the CLT: addition under independence leads probability mass to pile up in the middle and thin out at the tails.

For $n = 4$ and $n = 8$ we again get a peak at the “middle” value (halfway between the minimum and the maximum possible value).

The intuition is the same — there are simply more ways to get these middle outcomes.

If we continue, the bell-shaped curve becomes even more pronounced.

We are witnessing the binomial approximation of the normal distribution.

10.4.3 Simulation 1

Since the CLT seems almost magical, running simulations that verify its implications is one good way to build intuition.

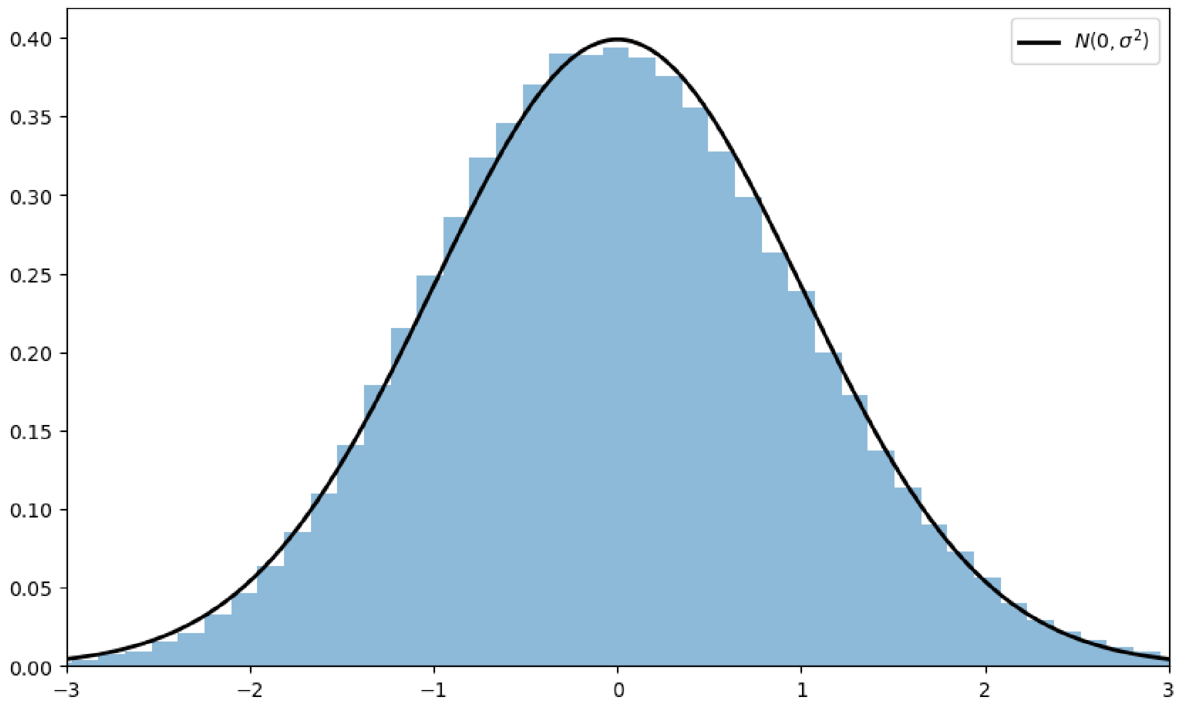
To this end, we now perform the following simulation

1. Choose an arbitrary distribution F for the underlying observations X_i.
2. Generate independent draws of $Y_n := \sqrt{n}(\bar{X}_n - \mu)$.
3. Use these draws to compute some measure of their distribution — such as a histogram.
4. Compare the latter to $N(0, \sigma^2)$.

Here's some code that does exactly this for the exponential distribution $F(x) = 1 - e^{-\lambda x}$.

(Please experiment with other choices of F, but remember that, to conform with the conditions of the CLT, the distribution must have a finite second moment.)

```
# Set parameters
n = 250                  # Choice of n
k = 100000               # Number of draws of Y_n
distribution = expon(2)  # Exponential distribution, λ = 1/2
μ, s = distribution.mean(), distribution.std()

# Draw underlying RVs. Each row contains a draw of X_1,..,X_n
data = distribution.rvs((k, n))
# Compute mean of each row, producing k draws of \bar X_n
sample_means = data.mean(axis=1)
# Generate observations of Y_n
Y = np.sqrt(n) * (sample_means - μ)

# Plot
fig, ax = plt.subplots(figsize=(10, 6))
xmin, xmax = -3 * s, 3 * s
ax.set_xlim(xmin, xmax)
ax.hist(Y, bins=60, alpha=0.5, density=True)
xgrid = np.linspace(xmin, xmax, 200)
ax.plot(xgrid, norm.pdf(xgrid, scale=s), 'k-', lw=2, label=r'$N(0, \sigma^2)$')
ax.legend()

plt.show()
```



Notice the absence of for loops — every operation is vectorized, meaning that the major calculations are all shifted to highly optimized C code.

The fit to the normal density is already tight and can be further improved by increasing `n`.

You can also experiment with other specifications of F.

10.4.4 Simulation 2

Our next simulation is somewhat like the first, except that we aim to track the distribution of $Y_n := \sqrt{n}(\bar{X}_n - \mu)$ as n increases.

In the simulation, we'll be working with random variables having $\mu = 0$.

Thus, when $n = 1$, we have $Y_1 = X_1$, so the first distribution is just the distribution of the underlying random variable.

For $n = 2$, the distribution of Y_2 is that of $(X_1 + X_2)/\sqrt{2}$, and so on.

What we expect is that, regardless of the distribution of the underlying random variable, the distribution of Y_n will smooth out into a bell-shaped curve.

The next figure shows this process for $X_i \sim f$, where f was specified as the convex combination of three different beta densities.

(Taking a convex combination is an easy way to produce an irregular shape for f.)

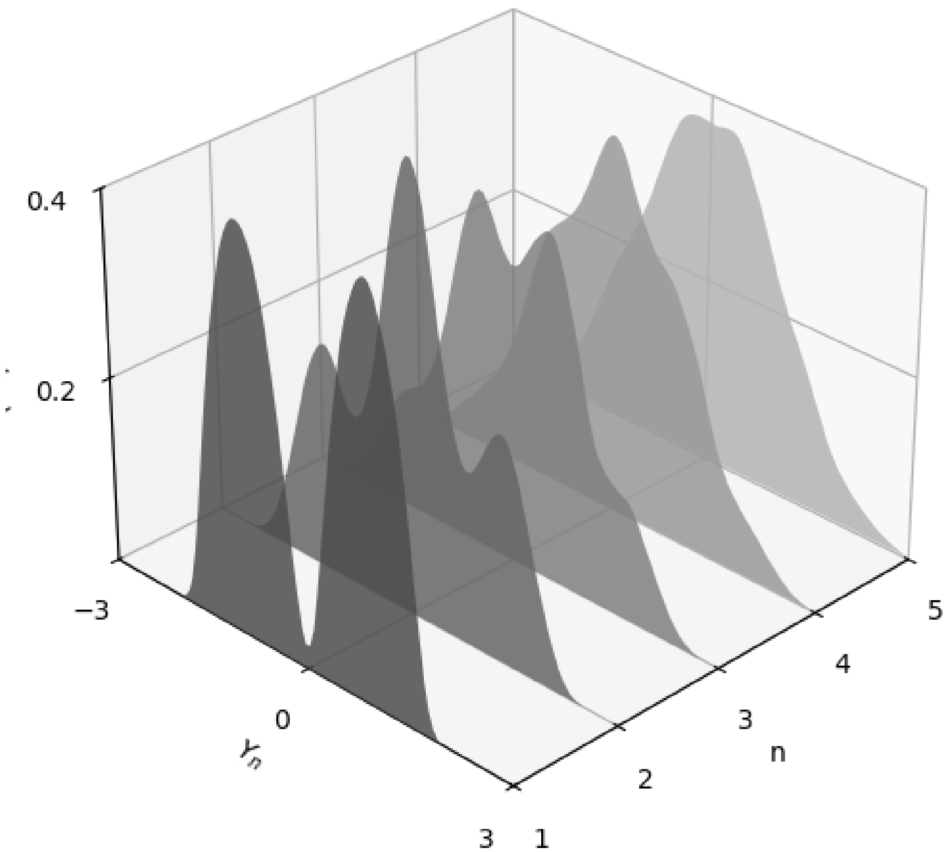
In the figure, the closest density is that of Y_1, while the furthest is that of Y_5

```
beta_dist = beta(2, 2)

def gen_x_draws(k):
    """
    Returns a flat array containing k independent draws from the
    distribution of X, the underlying random variable.  This distribution
```

(continues on next page)

(continued from previous page)

```
    is itself a convex combination of three beta distributions.
    """
    bdraws = beta_dist.rvs((3, k))
    # Transform rows, so each represents a different distribution
    bdraws[0, :] -= 0.5
    bdraws[1, :] += 0.6
    bdraws[2, :] -= 1.1
    # Set X[i] = bdraws[j, i], where j is a random draw from {0, 1, 2}
    js = np.random.randint(0, 2, size=k)
    X = bdraws[js, np.arange(k)]
    # Rescale, so that the random variable is zero mean
    m, sigma = X.mean(), X.std()
    return (X - m) / sigma

nmax = 5
reps = 100000
ns = list(range(1, nmax + 1))

# Form a matrix Z such that each column is reps independent draws of X
Z = np.empty((reps, nmax))
for i in range(nmax):
    Z[:, i] = gen_x_draws(reps)
# Take cumulative sum across columns
S = Z.cumsum(axis=1)
# Multiply j-th column by sqrt j
Y = (1 / np.sqrt(ns)) * S

# Plot
ax = plt.figure(figsize = (10, 6)).add_subplot(projection='3d')

a, b = -3, 3
gs = 100
xs = np.linspace(a, b, gs)

# Build verts
greys = np.linspace(0.3, 0.7, nmax)
verts = []
for n in ns:
    density = gaussian_kde(Y[:, n-1])
    ys = density(xs)
    verts.append(list(zip(xs, ys)))

poly = PolyCollection(verts, facecolors=[str(g) for g in greys])
poly.set_alpha(0.85)
ax.add_collection3d(poly, zs=ns, zdir='x')

ax.set(xlim3d=(1, nmax), xticks=(ns), ylabel='$Y_n$', zlabel='$p(y_n)$',
       xlabel=("n"), yticks=((-3, 0, 3)), ylim3d=(a, b),
       zlim3d=(0, 0.4), zticks=((0.2, 0.4)))
ax.invert_xaxis()
# Rotates the plot 30 deg on z axis and 45 deg on x axis
ax.view_init(30, 45)
plt.show()
```



As expected, the distribution smooths out into a bell curve as n increases.

We leave you to investigate its contents if you wish to know more.

If you run the file from the ordinary IPython shell, the figure should pop up in a window that you can rotate with your mouse, giving different views on the density sequence.

10.4.5 The Multivariate Case

The law of large numbers and central limit theorem work just as nicely in multidimensional settings.

To state the results, let's recall some elementary facts about random vectors.

A random vector $\mathbf{X}$ is just a sequence of k random variables $(X_1, \ldots, X_k)$.

Each realization of $\mathbf{X}$ is an element of $\mathbb{R}^k$.

A collection of random vectors $\mathbf{X}_1, \ldots, \mathbf{X}_n$ is called independent if, given any n vectors $\mathbf{x}_1, \ldots, \mathbf{x}_n$ in $\mathbb{R}^k$, we have

$$\mathbb{P}\{\mathbf{X}_1 \leq \mathbf{x}_1, \ldots, \mathbf{X}_n \leq \mathbf{x}_n\} = \mathbb{P}\{\mathbf{X}_1 \leq \mathbf{x}_1\} \times \cdots \times \mathbb{P}\{\mathbf{X}_n \leq \mathbf{x}_n\}$$

(The vector inequality $\mathbf{X} \leq \mathbf{x}$ means that $X_j \leq x_j$ for $j = 1, \ldots, k$)

Let $\mu_j := \mathbb{E}[X_j]$ for all $j = 1, \ldots, k$.

The expectation $\mathbb{E}[\mathbf{X}]$ of $\mathbf{X}$ is defined to be the vector of expectations:

$$\mathbb{E}[\mathbf{X}] := \begin{pmatrix} \mathbb{E}[X_1] \\ \mathbb{E}[X_2] \\ \vdots \\ \mathbb{E}[X_k] \end{pmatrix} = \begin{pmatrix} \mu_1 \\ \mu_2 \\ \vdots \\ \mu_k \end{pmatrix} =: \mu$$

The **variance-covariance matrix** of random vector $\mathbf{X}$ is defined as

$$\mathrm{Var}[\mathbf{X}] := \mathbb{E}[(\mathbf{X} - \mu)(\mathbf{X} - \mu)']$$

Expanding this out, we get

$$\mathrm{Var}[\mathbf{X}] = \begin{pmatrix} \mathbb{E}[(X_1 - \mu_1)(X_1 - \mu_1)] & \cdots & \mathbb{E}[(X_1 - \mu_1)(X_k - \mu_k)] \\ \mathbb{E}[(X_2 - \mu_2)(X_1 - \mu_1)] & \cdots & \mathbb{E}[(X_2 - \mu_2)(X_k - \mu_k)] \\ \vdots & \vdots & \vdots \\ \mathbb{E}[(X_k - \mu_k)(X_1 - \mu_1)] & \cdots & \mathbb{E}[(X_k - \mu_k)(X_k - \mu_k)] \end{pmatrix}$$

The j, k-th term is the scalar covariance between X_j and X_k.

With this notation, we can proceed to the multivariate LLN and CLT.

Let $\mathbf{X}_1, \ldots, \mathbf{X}_n$ be a sequence of independent and identically distributed random vectors, each one taking values in $\mathbb{R}^k$.

Let μ be the vector $\mathbb{E}[\mathbf{X}_i]$, and let Σ be the variance-covariance matrix of $\mathbf{X}_i$.

Interpreting vector addition and scalar multiplication in the usual way (i.e., pointwise), let

$$\bar{\mathbf{X}}_n := \frac{1}{n} \sum_{i=1}^{n} \mathbf{X}_i$$

In this setting, the LLN tells us that

$$\mathbb{P}\left\{\bar{\mathbf{X}}_n \to \mu \text{ as } n \to \infty\right\} = 1 \tag{10.6}$$

Here $\bar{\mathbf{X}}_n \to \mu$ means that $\|\bar{\mathbf{X}}_n - \mu\| \to 0$, where $\|\cdot\|$ is the standard Euclidean norm.

The CLT tells us that, provided Σ is finite,

$$\sqrt{n}(\bar{\mathbf{X}}_n - \mu) \stackrel{d}{\to} N(\mathbf{0}, \Sigma) \quad \text{as} \quad n \to \infty \tag{10.7}$$

10.5 Exercises

Exercise 10.5.1

One very useful consequence of the central limit theorem is as follows.

Assume the conditions of the CLT as *stated above*.

If $g \colon \mathbb{R} \to \mathbb{R}$ is differentiable at μ and $g'(\mu) \neq 0$, then

$$\sqrt{n}\{g(\bar{X}_n) - g(\mu)\} \stackrel{d}{\to} N(0, g'(\mu)^2\sigma^2) \quad \text{as} \quad n \to \infty \tag{10.8}$$

This theorem is used frequently in statistics to obtain the asymptotic distribution of estimators — many of which can be expressed as functions of sample means.

(These kinds of results are often said to use the "delta method".)

The proof is based on a Taylor expansion of g around the point μ.

Taking the result as given, let the distribution F of each X_i be uniform on $[0, \pi/2]$ and let $g(x) = \sin(x)$.

Derive the asymptotic distribution of $\sqrt{n}\{g(\bar{X}_n) - g(\mu)\}$ and illustrate convergence in the same spirit as the program discussed *above*.

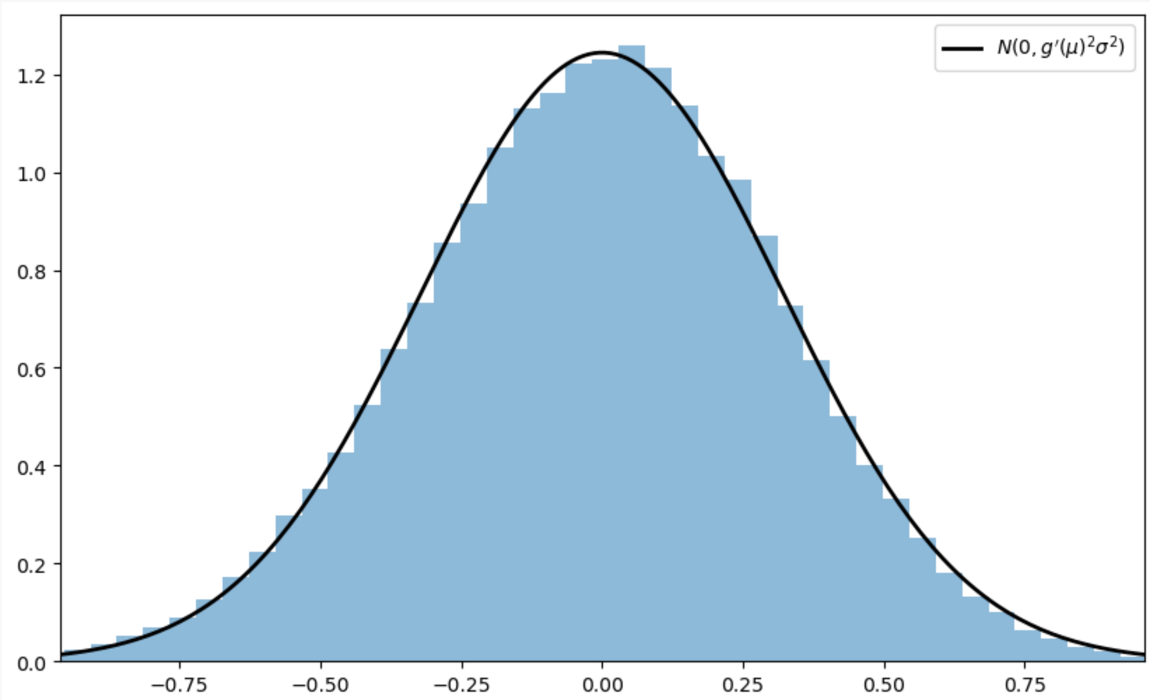
What happens when you replace $[0, \pi/2]$ with $[0, \pi]$?

What is the source of the problem?

Solution

Here is one solution

```
"""
Illustrates the delta method, a consequence of the central limit theorem.
"""

# Set parameters
n = 250
replications = 100000
distribution = uniform(loc=0, scale=(np.pi / 2))
μ, s = distribution.mean(), distribution.std()

g = np.sin
g_prime = np.cos

# Generate obs of sqrt{n} (g(X_n) - g(μ))
data = distribution.rvs((replications, n))
sample_means = data.mean(axis=1)  # Compute mean of each row
error_obs = np.sqrt(n) * (g(sample_means) - g(μ))

# Plot
asymptotic_sd = g_prime(μ) * s
fig, ax = plt.subplots(figsize=(10, 6))
xmin = -3 * g_prime(μ) * s
xmax = -xmin
ax.set_xlim(xmin, xmax)
ax.hist(error_obs, bins=60, alpha=0.5, density=True)
xgrid = np.linspace(xmin, xmax, 200)
lb = r"$N(0, g'(\mu)^2  \sigma^2)$"
ax.plot(xgrid, norm.pdf(xgrid, scale=asymptotic_sd), 'k-', lw=2, label=lb)
ax.legend()
plt.show()
```



What happens when you replace $[0, \pi/2]$ with $[0, \pi]$?

In this case, the mean μ of this distribution is $\pi/2$, and since $g' = \cos$, we have $g'(\mu) = 0$.

Hence the conditions of the delta theorem are not satisfied.

Exercise 10.5.2

Here's a result that's often used in developing statistical tests, and is connected to the multivariate central limit theorem.

If you study econometric theory, you will see this result used again and again.

Assume the setting of the multivariate CLT *discussed above*, so that

1. $\mathbf{X}_1, \ldots, \mathbf{X}_n$ is a sequence of IID random vectors, each taking values in $\mathbb{R}^k$.
2. $\mu := \mathbb{E}[\mathbf{X}_i]$, and Σ is the variance-covariance matrix of $\mathbf{X}_i$.
3. The convergence

$$\sqrt{n}(\bar{\mathbf{X}}_n - \mu) \xrightarrow{d} N(\mathbf{0}, \Sigma) \tag{10.9}$$

is valid.

In a statistical setting, one often wants the right-hand side to be **standard** normal so that confidence intervals are easily computed.

This normalization can be achieved on the basis of three observations.

First, if $\mathbf{X}$ is a random vector in $\mathbb{R}^k$ and $\mathbf{A}$ is constant and $k \times k$, then

$$\mathrm{Var}[\mathbf{A}\mathbf{X}] = \mathbf{A}\,\mathrm{Var}[\mathbf{X}]\mathbf{A}'$$

Second, by the continuous mapping theorem, if $\mathbf{Z}_n \stackrel{d}{\to} \mathbf{Z}$ in $\mathbb{R}^k$ and $\mathbf{A}$ is constant and $k \times k$, then

$$\mathbf{A}\mathbf{Z}_n \stackrel{d}{\to} \mathbf{A}\mathbf{Z}$$

Third, if $\mathbf{S}$ is a $k \times k$ symmetric positive definite matrix, then there exists a symmetric positive definite matrix $\mathbf{Q}$, called the inverse square root of $\mathbf{S}$, such that

$$\mathbf{Q}\mathbf{S}\mathbf{Q}' = \mathbf{I}$$

Here $\mathbf{I}$ is the $k \times k$ identity matrix.

Putting these things together, your first exercise is to show that if $\mathbf{Q}$ is the inverse square root of ⍰, then

$$\mathbf{Z}_n := \sqrt{n}\mathbf{Q}(\bar{\mathbf{X}}_n - \mu) \stackrel{d}{\to} \mathbf{Z} \sim N(\mathbf{0}, \mathbf{I})$$

Applying the continuous mapping theorem one more time tells us that

$$\|\mathbf{Z}_n\|^2 \stackrel{d}{\to} \|\mathbf{Z}\|^2$$

Given the distribution of $\mathbf{Z}$, we conclude that

$$n\|\mathbf{Q}(\bar{\mathbf{X}}_n - \mu)\|^2 \stackrel{d}{\to} \chi^2(k) \tag{10.10}$$

where $\chi^2(k)$ is the chi-squared distribution with k degrees of freedom.

(Recall that k is the dimension of $\mathbf{X}_i$, the underlying random vectors.)

Your second exercise is to illustrate the convergence in (10.10) with a simulation.

In doing so, let

$$\mathbf{X}_i := \begin{pmatrix} W_i \\ U_i + W_i \end{pmatrix}$$

where

- each W_i is an IID draw from the uniform distribution on $[-1, 1]$.
- each U_i is an IID draw from the uniform distribution on $[-2, 2]$.
- U_i and W_i are independent of each other.

Hint

1. `scipy.linalg.sqrtm(A)` computes the square root of `A`. You still need to invert it.
2. You should be able to work out Σ from the preceding information.

Solution

First we want to verify the claim that

$$\sqrt{n}\mathbf{Q}(\bar{\mathbf{X}}_n - \mu) \stackrel{d}{\to} N(\mathbf{0}, \mathbf{I})$$

This is straightforward given the facts presented in the exercise.

Let

$$\mathbf{Y}_n := \sqrt{n}(\bar{\mathbf{X}}_n - \mu) \quad \text{and} \quad \mathbf{Y} \sim N(\mathbf{0}, \Sigma)$$

By the multivariate CLT and the continuous mapping theorem, we have

$$\mathbf{Q}\mathbf{Y}_n \stackrel{d}{\to} \mathbf{Q}\mathbf{Y}$$

Since linear combinations of normal random variables are normal, the vector $\mathbf{Q}\mathbf{Y}$ is also normal.

Its mean is clearly $\mathbf{0}$, and its variance-covariance matrix is

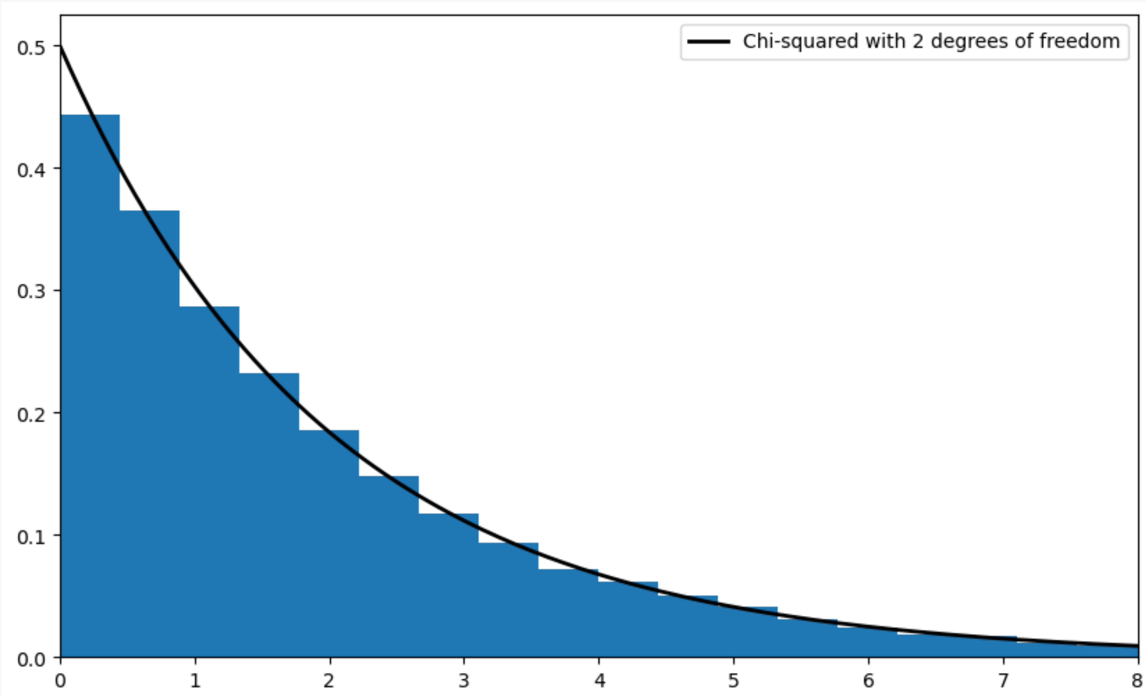
$$\mathrm{Var}[\mathbf{Q}\mathbf{Y}] = \mathbf{Q}\mathrm{Var}[\mathbf{Y}]\mathbf{Q}' = \mathbf{Q}\Sigma\mathbf{Q}' = \mathbf{I}$$

In conclusion, $\mathbf{Q}\mathbf{Y}_n \stackrel{d}{\to} \mathbf{Q}\mathbf{Y} \sim N(\mathbf{0}, \mathbf{I})$, which is what we aimed to show.

Now we turn to the simulation exercise.

Our solution is as follows

```
# Set parameters
n = 250
replications = 50000
dw = uniform(loc=-1, scale=2)  # Uniform(-1, 1)
du = uniform(loc=-2, scale=4)  # Uniform(-2, 2)
sw, su = dw.std(), du.std()
vw, vu = sw**2, su**2
Σ = ((vw, vw), (vw, vw + vu))
Σ = np.array(Σ)

# Compute Σ^{-1/2}
Q = inv(sqrtm(Σ))

# Generate observations of the normalized sample mean
error_obs = np.empty((2, replications))
for i in range(replications):
    # Generate one sequence of bivariate shocks
    X = np.empty((2, n))
    W = dw.rvs(n)
    U = du.rvs(n)
    # Construct the n observations of the random vector
    X[0, :] = W
    X[1, :] = W + U
    # Construct the i-th observation of Y_n
    error_obs[:, i] = np.sqrt(n) * X.mean(axis=1)

# Premultiply by Q and then take the squared norm
temp = Q @ error_obs
chisq_obs = np.sum(temp**2, axis=0)

# Plot
fig, ax = plt.subplots(figsize=(10, 6))
xmax = 8
ax.set_xlim(0, xmax)
xgrid = np.linspace(0, xmax, 200)
lb = "Chi-squared with 2 degrees of freedom"
ax.plot(xgrid, chi2.pdf(xgrid, 2), 'k-', lw=2, label=lb)
ax.legend()
ax.hist(chisq_obs, bins=50, density=True)
plt.show()
```


Chi-squared with 2 degrees of freedom
0.5
0.4
0.3
0.2
0.1
0.0
0
1
2
3
4
5
6
7
8

CHAPTER

ELEVEN

TWO MEANINGS OF PROBABILITY

11.1 Overview

This lecture illustrates two distinct interpretations of a **probability distribution**

- A frequentist interpretation as **relative frequencies** anticipated to occur in a large i.i.d. sample
- A Bayesian interpretation as a **personal opinion** (about a parameter or list of parameters) after seeing a collection of observations

We recommend watching this video about **hypothesis testing** within the frequentist approach

https://youtu.be/8JIe_cz6qGA

After you watch that video, please watch the following video on the Bayesian approach to constructing **coverage intervals**

https://youtu.be/Pahyv9i_X2k

After you are familiar with the material in these videos, this lecture uses the Socratic method to to help consolidate your understanding of the different questions that are answered by

- a frequentist confidence interval
- a Bayesian coverage interval

We do this by inviting you to write some Python code.

It would be especially useful if you tried doing this after each question that we pose for you, before proceeding to read the rest of the lecture.

We provide our own answers as the lecture unfolds, but you'll learn more if you try writing your own code before reading and running ours.

Code for answering questions:

In addition to what's in Anaconda, this lecture will deploy the following library:

```
pip install prettytable
```

To answer our coding questions, we'll start with some imports

```
import numpy as np
import pandas as pd
import prettytable as pt
import matplotlib.pyplot as plt
from scipy.stats import binom
import scipy.stats as st
```

Empowered with these Python tools, we'll now explore the two meanings described above.

11.2 Frequentist Interpretation

Consider the following classic example.

The random variable X takes on possible values $k = 0, 1, 2, \dots, n$ with probabilties

$$\text{Prob}(X = k|\theta) = \left(\frac{n!}{k!(n-k)!}\right) \theta^k (1-\theta)^{n-k}$$

where the fixed parameter $\theta \in (0, 1)$.

This is called the **binomial distribution**.

Here

- θ is the probability that one toss of a coin will be a head, an outcome that we encode as $Y = 1$.
- $1 - \theta$ is the probability that one toss of the coin will be a tail, an outcome that we denote $Y = 0$.
- X is the total number of heads that came up after flipping the coin n times.

Consider the following experiment:

Take I **independent** sequences of n **independent** flips of the coin

Notice the repeated use of the adjective **independent**:

- we use it once to describe that we are drawing n independent times from a **Bernoulli** distribution with parameter θ to arrive at one draw from a **Binomial** distribution with parameters θ, n.
- we use it again to describe that we are then drawing I sequences of n coin draws.

Let $y_h^i \in \{0, 1\}$ be the realized value of Y on the hth flip during the ith sequence of flips.

Let $\sum_{h=1}^n y_h^i$ denote the total number of times heads come up during the ith sequence of n independent coin flips.

Let f_k record the fraction of samples of length n for which $\sum_{h=1}^n y_h^i = k$:

$$f_k^I = \frac{\text{number of samples of length n for which } \sum_{h=1}^n y_h^i = k}{I}$$

The probability $\text{Prob}(X = k|\theta)$ answers the following question:

- As I becomes large, in what fraction of I independent draws of n coin flips should we anticipate k heads to occur?

As usual, a law of large numbers justifies this answer.

Exercise 11.2.1

1. Please write a Python class to compute f_k^I
2. Please use your code to compute $f_k^I, k = 0, \dots, n$ and compare them to $\text{Prob}(X = k|\theta)$ for various values of θ, n and I
3. With the Law of Large numbers in mind, use your code to say something

Solution

Here is one solution:

```python
class frequentist:

    def __init__(self, θ, n, I):

        '''
        initialization
        -----------------
        parameters:
        θ : probability that one toss of a coin will be a head with Y = 1
        n : number of independent flips in each independent sequence of draws
        I : number of independent sequence of draws

        '''

        self.θ, self.n, self.I = θ, n, I

    def binomial(self, k):

        '''compute the theoretical probability for specific input k'''

        θ, n = self.θ, self.n
        self.k = k
        self.P = binom.pmf(k, n, θ)

    def draw(self):

        '''draw n independent flips for I independent sequences'''

        θ, n, I = self.θ, self.n, self.I
        sample = np.random.rand(I, n)
        Y = (sample <= θ) * 1
        self.Y = Y

    def compute_fk(self, kk):

        '''compute f_{k}^I for specific input k'''

        Y, I = self.Y, self.I
        K = np.sum(Y, 1)
        f_kI = np.sum(K == kk) / I
        self.f_kI = f_kI
        self.kk = kk

    def compare(self):

        '''compute and print the comparison'''

        n = self.n
        comp = pt.PrettyTable()
        comp.field_names = ['k', 'Theoretical', 'Frequentist']
        self.draw()
        for i in range(n):
            self.binomial(i+1)
            self.compute_fk(i+1)
            comp.add_row([i+1, self.P, self.f_kI])
```

```
        print(comp)
θ, n, k, I = 0.7, 20, 10, 1_000_000

freq = frequentist(θ, n, I)

freq.compare()
+----+------------------------+-------------+
| k  |      Theoretical       | Frequentist |
+----+------------------------+-------------+
| 1  | 1.6271660538000033e-09 |     0.0     |
| 2  | 3.606884752589999e-08  |     0.0     |
| 3  |  5.04963865362601e-07  |    1e-06    |
| 4  | 5.007558331512455e-06  |    9e-06    |
| 5  | 3.7389768875293014e-05 |    4e-05    |
| 6  | 0.00021810698510587546 |   0.000218  |
| 7  |  0.001017832597160754  |   0.001059  |
| 8  |  0.003859281930901185  |   0.003763  |
| 9  |  0.012006654896137007  |   0.012021  |
| 10 |  0.030817080900085007  |   0.031048  |
| 11 |  0.06536956554563476   |   0.06504   |
| 12 |  0.11439673970486108   |   0.114415  |
| 13 |   0.1642619852172365   |   0.164112  |
| 14 |  0.19163898275344252   |   0.19146   |
| 15 |  0.17886305056987967   |   0.179672  |
| 16 |   0.1304209743738704   |   0.130439  |
| 17 |  0.07160367220526209   |   0.071222  |
| 18 |  0.027845872524268643  |   0.027799  |
| 19 |  0.006839337111223895  |   0.006906  |
| 20 | 0.0007979226629761189  |   0.000776  |
+----+------------------------+-------------+
```

From the table above, can you see the law of large numbers at work?

Let's do some more calculations.

Comparison with different θ

Now we fix

$$n = 20, k = 10, I = 1,000,000$$

We'll vary θ from 0.01 to 0.99 and plot outcomes against θ.

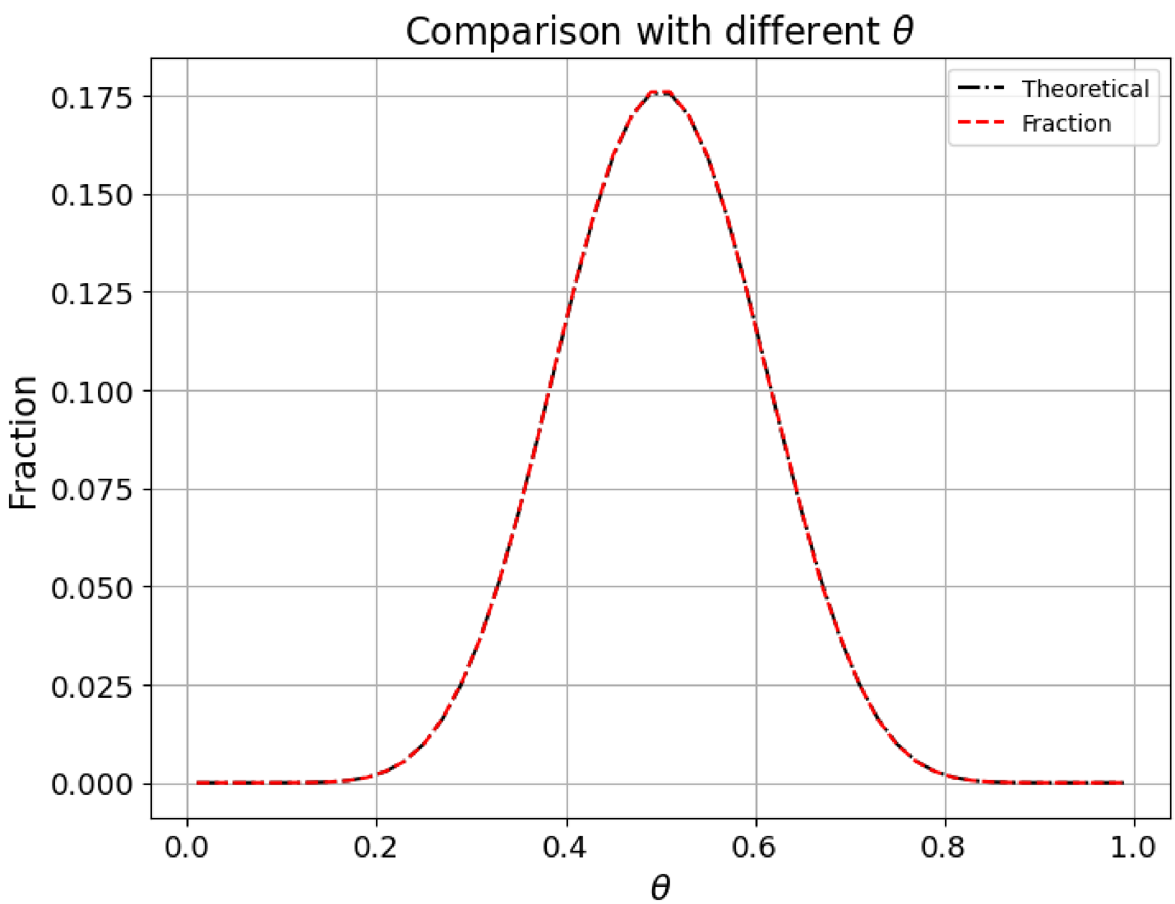
```
θ_low, θ_high, npt = 0.01, 0.99, 50
thetas = np.linspace(θ_low, θ_high, npt)
P = []
f_kI = []
for i in range(npt):
    freq = frequentist(thetas[i], n, I)
    freq.binomial(k)
    freq.draw()
    freq.compute_fk(k)
    P.append(freq.P)
    f_kI.append(freq.f_kI)
```

```
fig, ax = plt.subplots(figsize=(8, 6))
ax.grid()
```

(continues on next page)

(continued from previous page)

```
ax.plot(thetas, P, 'k-.', label='Theoretical')
ax.plot(thetas, f_kI, 'r--', label='Fraction')
plt.title(r'Comparison with different $\theta$', fontsize=16)
plt.xlabel(r'$\theta$', fontsize=15)
plt.ylabel('Fraction', fontsize=15)
plt.tick_params(labelsize=13)
plt.legend()
plt.show()
```



Comparison with different n

Now we fix $\theta = 0.7, k = 10, I = 1,000,000$ and vary n from 1 to 100.

Then we'll plot outcomes.

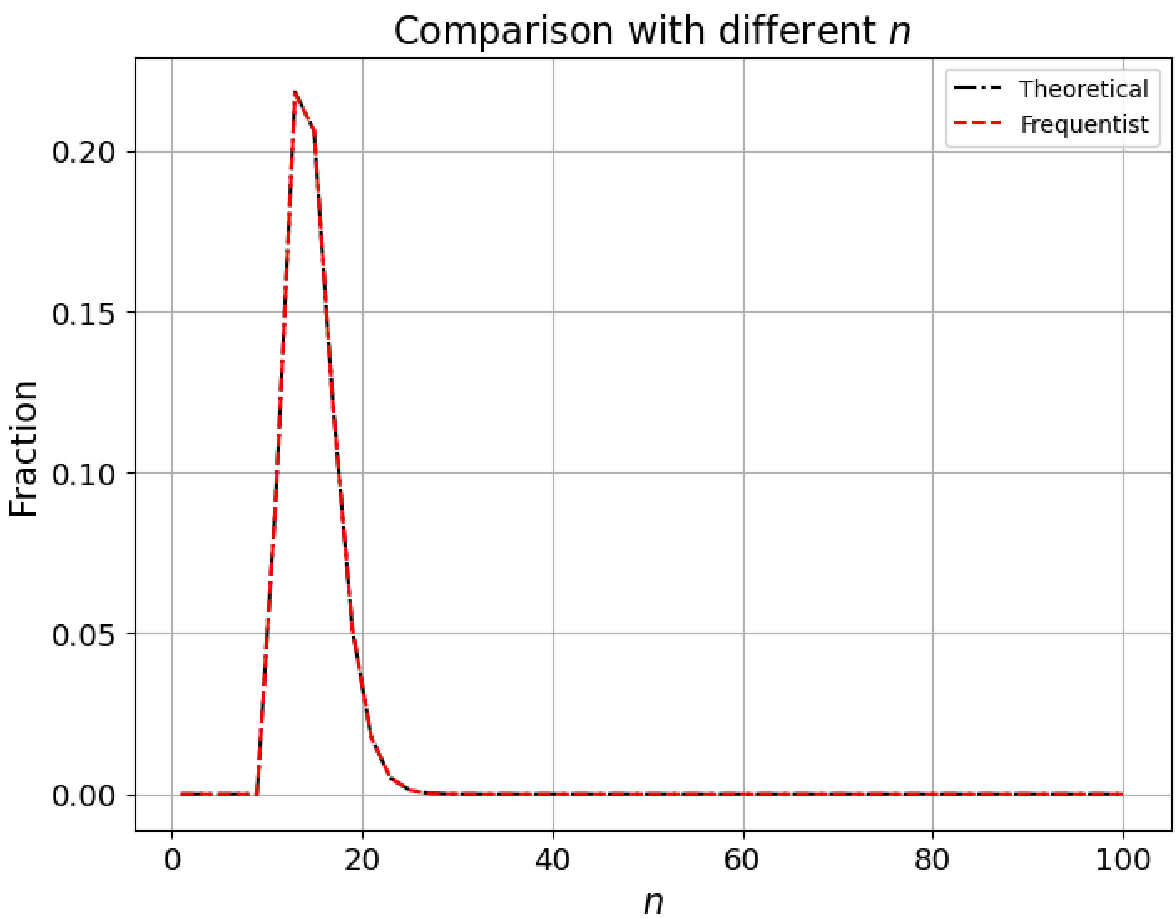
```
n_low, n_high, nn = 1, 100, 50
ns = np.linspace(n_low, n_high, nn, dtype='int')
P = []
f_kI = []
for i in range(nn):
    freq = frequentist(θ, ns[i], I)
    freq.binomial(k)
    freq.draw()
    freq.compute_fk(k)
    P.append(freq.P)
```

(continues on next page)

(continued from previous page)

```
    f_kI.append(freq.f_kI)
```

```
fig, ax = plt.subplots(figsize=(8, 6))
ax.grid()
ax.plot(ns, P, 'k-.', label='Theoretical')
ax.plot(ns, f_kI, 'r--', label='Frequentist')
plt.title(r'Comparison with different $n$', fontsize=16)
plt.xlabel(r'$n$', fontsize=15)
plt.ylabel('Fraction', fontsize=15)
plt.tick_params(labelsize=13)
plt.legend()
plt.show()
```



Comparison with different I

Now we fix $\theta = 0.7, n = 20, k = 10$ and vary $\log(I)$ from 2 to 7.

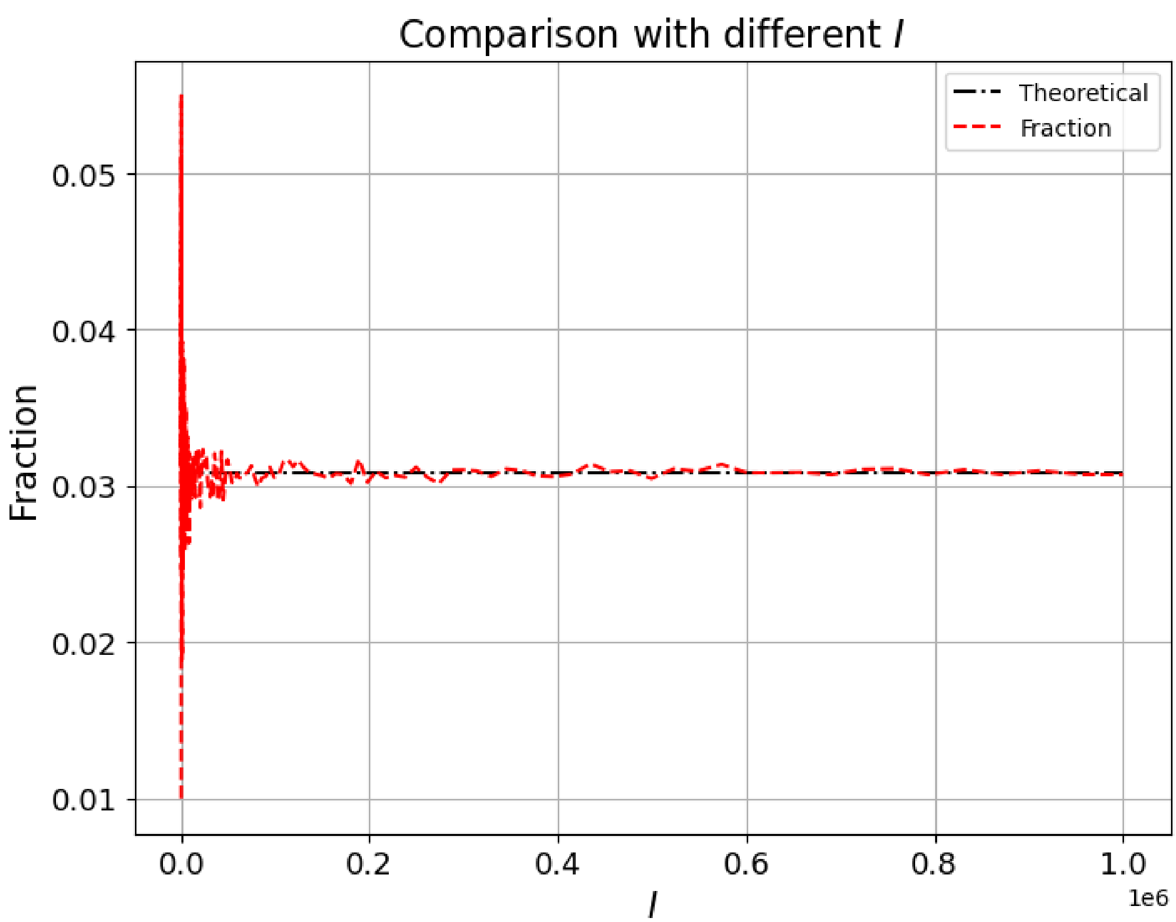
```
I_log_low, I_log_high, nI = 2, 6, 200
log_Is = np.linspace(I_log_low, I_log_high, nI)
Is = np.power(10, log_Is).astype(int)
P = []
f_kI = []
for i in range(nI):
    freq = frequentist(θ, n, Is[i])
```

(continues on next page)

(continued from previous page)

```
    freq.binomial(k)
    freq.draw()
    freq.compute_fk(k)
    P.append(freq.P)
    f_kI.append(freq.f_kI)
```

```
fig, ax = plt.subplots(figsize=(8, 6))
ax.grid()
ax.plot(Is, P, 'k-.', label='Theoretical')
ax.plot(Is, f_kI, 'r--', label='Fraction')
plt.title(r'Comparison with different $I$', fontsize=16)
plt.xlabel(r'$I$', fontsize=15)
plt.ylabel('Fraction', fontsize=15)
plt.tick_params(labelsize=13)
plt.legend()
plt.show()
```



From the above graphs, we can see that I, **the number of independent sequences,** plays an important role.

When I becomes larger, the difference between theoretical probability and frequentist estimate becomes smaller.

Also, as long as I is large enough, changing θ or n does not substantially change the accuracy of the observed fraction as an approximation of θ.

The Law of Large Numbers is at work here.

For each draw of an independent sequence, $\text{Prob}(X_i = k|\theta)$ is the same, so aggregating all draws forms an i.i.d sequence of a binary random variable $\rho_{k,i}, i = 1, 2, ...I$, with a mean of $\text{Prob}(X = k|\theta)$ and a variance of

$$n \cdot \text{Prob}(X = k|\theta) \cdot (1 - \text{Prob}(X = k|\theta)).$$

So, by the LLN, the average of $P_{k,i}$ converges to:

$$E[\rho_{k,i}] = \text{Prob}(X = k|\theta) = \left(\frac{n!}{k!(n-k)!}\right) \theta^k (1-\theta)^{n-k}$$

as I goes to infinity.

11.3 Bayesian Interpretation

We again use a binomial distribution.

But now we don't regard θ as being a fixed number.

Instead, we think of it as a **random variable**.

θ is described by a probability distribution.

But now this probability distribution means something different than a relative frequency that we can anticipate to occur in a large i.i.d. sample.

Instead, the probability distribution of θ is now a summary of our views about likely values of θ either

- **before** we have seen **any** data at all, or
- **before** we have seen **more** data, after we have seen **some** data

Thus, suppose that, before seeing any data, you have a personal prior probability distribution saying that

$$P(\theta) = \frac{\theta^{\alpha-1}(1-\theta)^{\beta-1}}{B(\alpha, \beta)}$$

where $B(\alpha, \beta)$ is a **beta function** , so that $P(\theta)$ is a **beta distribution** with parameters α, β.

Exercise 11.3.1

a) Please write down the **likelihood function** for a sample of length n from a binomial distribution with parameter θ.

b) Please write down the **posterior** distribution for θ after observing one flip of the coin.

c) Now pretend that the true value of $\theta = .4$ and that someone who doesn't know this has a beta prior distribution with parameters with $\beta = \alpha = .5$. Please write a Python class to simulate this person's personal posterior distribution for θ for a *single* sequence of n draws.

d) Please plot the posterior distribution for θ as a function of θ as n grows as $1, 2, ...$

e) For various n's, please describe and compute a Bayesian coverage interval for the interval $[.45, .55]$.

f) Please tell what question a Bayesian coverage interval answers.

g) Please compute the Posterior probabililty that $\theta \in [.45, .55]$ for various values of sample size n.

h) Please use your Python class to study what happens to the posterior distribution as $n \to +\infty$, again assuming that the true value of $\theta = .4$, though it is unknown to the person doing the updating via Bayes' Law.

Solution

a) Please write down the **likelihood function** and the **posterior** distribution for θ after observing one flip of our coin.

Suppose the outcome is **Y**.

The likelihood function is:

$$L(Y|\theta) = \text{Prob}(X = Y|\theta) = \theta^Y (1-\theta)^{1-Y}$$

b) Please write the **posterior** distribution for θ after observing one flip of our coin.

The prior distribution is

$$\text{Prob}(\theta) = \frac{\theta^{\alpha-1}(1-\theta)^{\beta-1}}{B(\alpha, \beta)}$$

We can derive the posterior distribution for θ via

$$\begin{aligned}
\text{Prob}(\theta|Y) &= \frac{\text{Prob}(Y|\theta)\text{Prob}(\theta)}{\text{Prob}(Y)} \\
&= \frac{\text{Prob}(Y|\theta)\text{Prob}(\theta)}{\int_0^1 \text{Prob}(Y|\theta)\text{Prob}(\theta)d\theta} \\
&= \frac{\theta^Y (1-\theta)^{1-Y} \frac{\theta^{\alpha-1}(1-\theta)^{\beta-1}}{B(\alpha,\beta)}}{\int_0^1 \theta^Y (1-\theta)^{1-Y} \frac{\theta^{\alpha-1}(1-\theta)^{\beta-1}}{B(\alpha,\beta)} d\theta} \\
&= \frac{\theta^{Y+\alpha-1}(1-\theta)^{1-Y+\beta-1}}{\int_0^1 \theta^{Y+\alpha-1}(1-\theta)^{1-Y+\beta-1} d\theta}
\end{aligned}$$

which means that

$$\text{Prob}(\theta|Y) \sim \text{Beta}(\alpha + Y, \beta + (1-Y))$$

Now please pretend that the true value of $\theta = .4$ and that someone who doesn't know this has a beta prior with $\beta = \alpha = .5$.

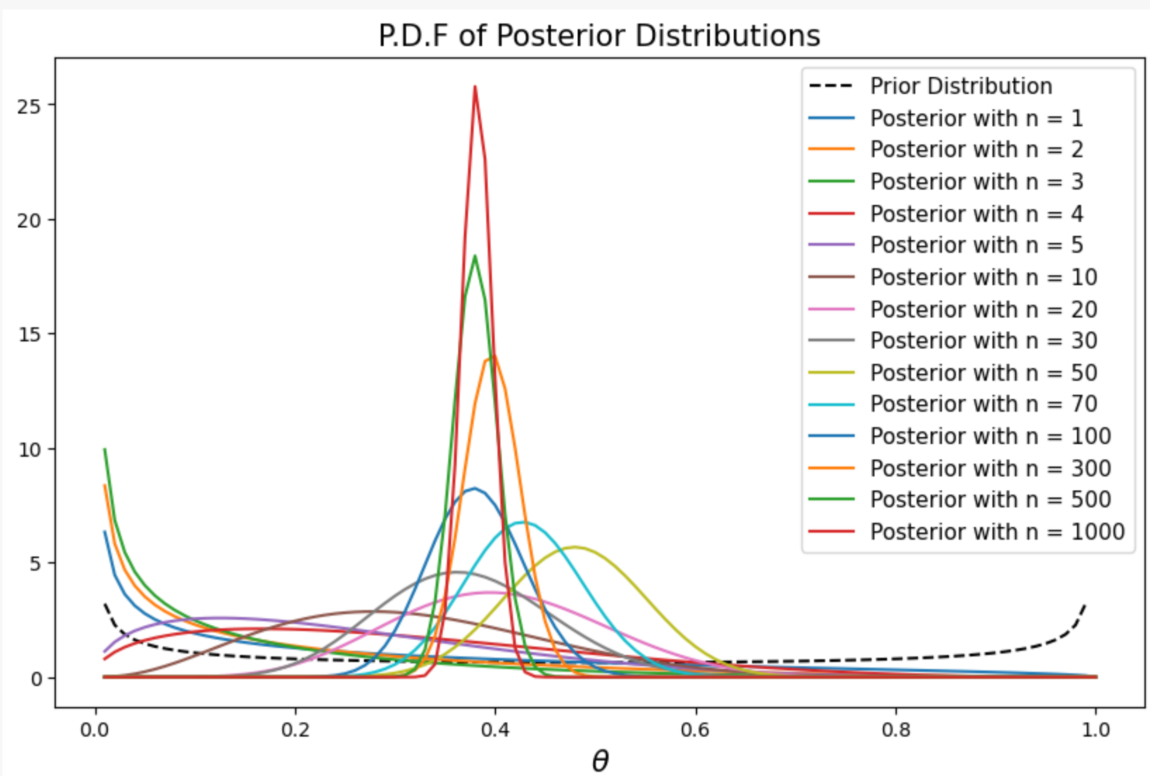
c) Now pretend that the true value of $\theta = .4$ and that someone who doesn't know this has a beta prior distribution with parameters with $\beta = \alpha = .5$. Please write a Python class to simulate this person's personal posterior distribution for θ for a *single* sequence of n draws.

```python
class Bayesian:

    def __init__(self, θ=0.4, n=1_000_000, α=0.5, β=0.5):
        """
        Parameters:
        ----------
        θ : float, ranging from [0,1].
           probability that one toss of a coin will be a head with Y = 1

        n : int.
           number of independent flips in an independent sequence of draws

        α&β : int or float.
             parameters of the prior distribution on θ

        """
        self.θ, self.n, self.α, self.β = θ, n, α, β
```

```
        self.prior = st.beta(α, β)

    def draw(self):
        """
        simulate a single sequence of draws of length n, given probability θ

        """
        array = np.random.rand(self.n)
        self.draws = (array < self.θ).astype(int)

    def form_single_posterior(self, step_num):
        """
        form a posterior distribution after observing the first step_num elements␣
↪of the draws

        Parameters
        ----------
        step_num: int.
               number of steps observed to form a posterior distribution

        Returns
        ------
        the posterior distribution for sake of plotting in the subsequent steps

        """
        heads_num = self.draws[:step_num].sum()
        tails_num = step_num - heads_num

        return st.beta(self.α+heads_num, self.β+tails_num)

    def form_posterior_series(self,num_obs_list):
        """
        form a series of posterior distributions that form after observing␣
↪different number of draws.

        Parameters
        ----------
        num_obs_list: a list of int.
               a list of the number of observations used to form a series of␣
↪posterior distributions.

        """
        self.posterior_list = []
        for num in num_obs_list:
            self.posterior_list.append(self.form_single_posterior(num))
```

d) Please plot the posterior distribution for θ as a function of θ as n grows from $1, 2, \ldots$.

```
Bay_stat = Bayesian()
Bay_stat.draw()

num_list = [1, 2, 3, 4, 5, 10, 20, 30, 50, 70, 100, 300, 500, 1000, # this line␣
↪for finite n
            5000, 10_000, 50_000, 100_000, 200_000, 300_000]  # this line for␣
↪approximately infinite n

Bay_stat.form_posterior_series(num_list)

θ_values = np.linspace(0.01, 1, 100)

fig, ax = plt.subplots(figsize=(10, 6))

ax.plot(θ_values, Bay_stat.prior.pdf(θ_values), label='Prior Distribution', color=
↪'k', linestyle='--')

for ii, num in enumerate(num_list[:14]):
    ax.plot(θ_values, Bay_stat.posterior_list[ii].pdf(θ_values), label='Posterior␣
↪with n = %d' % num)

ax.set_title('P.D.F of Posterior Distributions', fontsize=15)
ax.set_xlabel(r"$\theta$", fontsize=15)

ax.legend(fontsize=11)
plt.show()
```



e) For various n's, please describe and compute .05 and .95 quantiles for posterior probabilities.

```python
upper_bound = [ii.ppf(0.05) for ii in Bay_stat.posterior_list[:14]]
lower_bound = [ii.ppf(0.95) for ii in Bay_stat.posterior_list[:14]]

interval_df = pd.DataFrame()
interval_df['upper'] = upper_bound
interval_df['lower'] = lower_bound
interval_df.index = num_list[:14]
interval_df = interval_df.T
interval_df
```

```
             1         2         3         4         5         10        20    \
upper  0.001543  0.000868  0.000603  0.046007  0.036447  0.117329  0.237639
lower  0.771480  0.569259  0.444067  0.650707  0.562845  0.558127  0.582446

             30        50        70        100       300       500       1000
upper  0.235206  0.366730  0.334679  0.303401  0.351081  0.344870  0.357002
lower  0.516350  0.594938  0.526749  0.461664  0.443718  0.416153  0.407502
```

As n increases, we can see that Bayesian coverage intervals narrow and move toward 0.4.

f) Please tell what question a Bayesian coverage interval answers.

The Bayesian coverage interval tells the range of θ that corresponds to the $[p_1, p_2]$ quantiles of the cumulative probability distribution (CDF) of the posterior distribution.

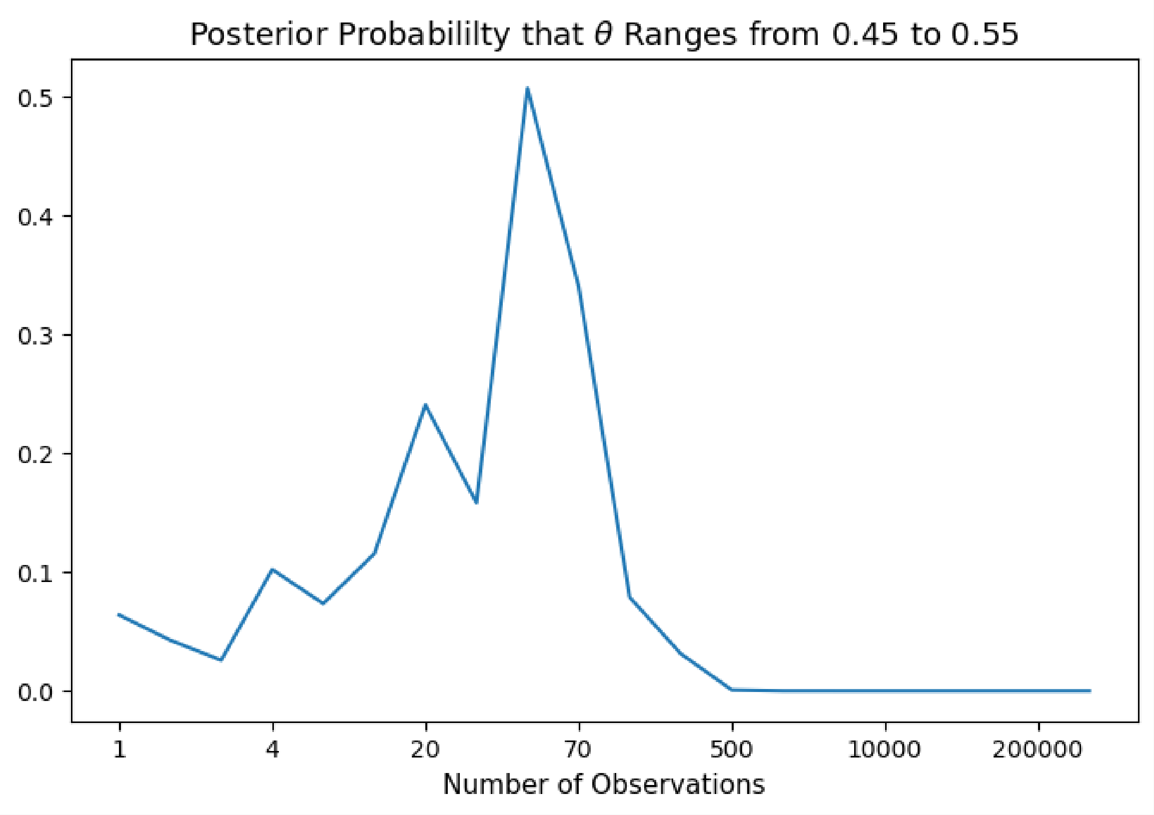
To construct the coverage interval we first compute a posterior distribution of the unknown parameter θ.

If the CDF is $F(\theta)$, then the Bayesian coverage interval $[a, b]$ for the interval $[p_1, p_2]$ is described by

$$F(a) = p_1, F(b) = p_2$$

g) Please compute the Posterior probabililty that $\theta \in [.45, .55]$ for various values of sample size n.

```python
left_value, right_value = 0.45, 0.55

posterior_prob_list=[ii.cdf(right_value)-ii.cdf(left_value) for ii in Bay_stat.
↪posterior_list]

fig, ax = plt.subplots(figsize=(8, 5))
ax.plot(posterior_prob_list)
ax.set_title('Posterior Probabililty that '+ r"$\theta$" +' Ranges from %.2f to %.
↪2f'%(left_value, right_value),
             fontsize=13)
ax.set_xticks(np.arange(0, len(posterior_prob_list), 3))
ax.set_xticklabels(num_list[::3])
ax.set_xlabel('Number of Observations', fontsize=11)

plt.show()
```



Notice that in the graph above the posterior probabililty that $\theta \in [.45, .55]$ typically exhibits a hump shape as n increases.

Two opposing forces are at work.

The first force is that the individual adjusts his belief as he observes new outcomes, so his posterior probability distribution becomes more and more realistic, which explains the rise of the posterior probabililty.

However, $[.45, .55]$ actually excludes the true $\theta = .4$ that generates the data.

As a result, the posterior probabililty drops as larger and larger samples refine his posterior probability distribution of θ.

The descent seems precipitous only because of the scale of the graph that has the number of observations increasing disproportionately.

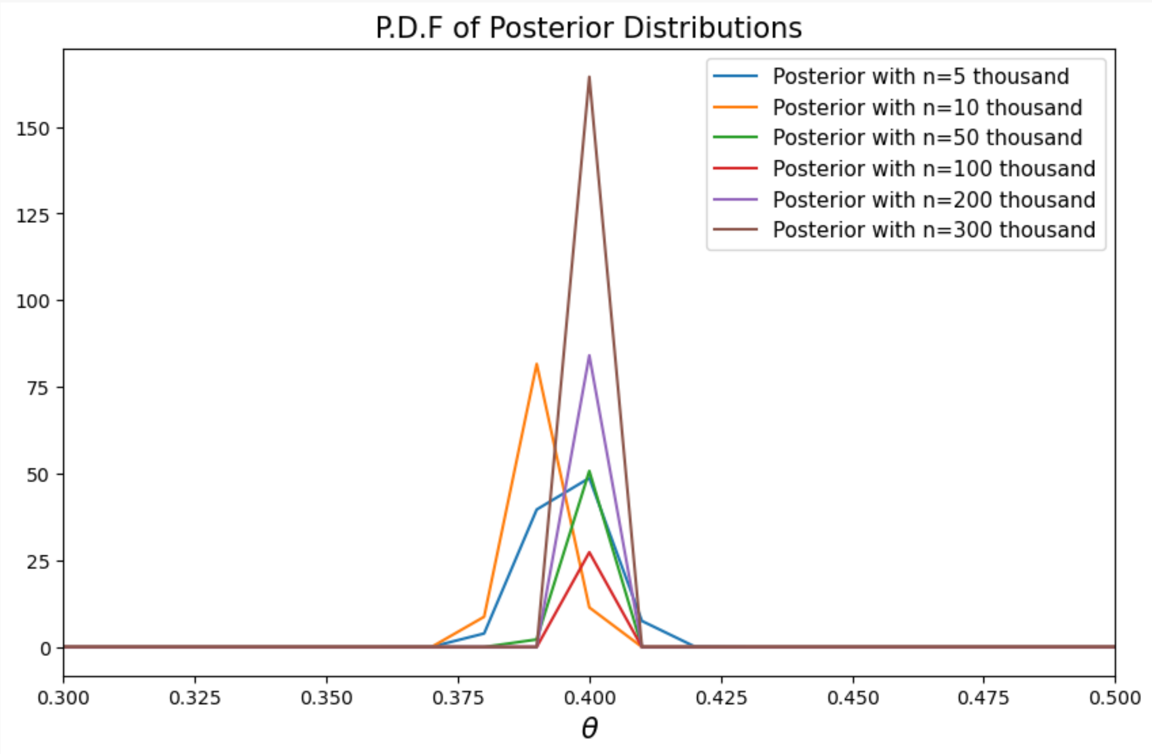
When the number of observations becomes large enough, our Bayesian becomes so confident about θ that he considers $\theta \in [.45, .55]$ very unlikely.

That is why we see a nearly horizontal line when the number of observations exceeds 500.

h) Please use your Python class to study what happens to the posterior distribution as $n \to +\infty$, again assuming that the true value of $\theta = .4$, though it is unknown to the person doing the updating via Bayes' Law.

Using the Python class we made above, we can see the evolution of posterior distributions as n approaches infinity.

```
fig, ax = plt.subplots(figsize=(10, 6))

for ii, num in enumerate(num_list[14:]):
    ii += 14
    ax.plot(θ_values, Bay_stat.posterior_list[ii].pdf(θ_values),
            label='Posterior with n=%d thousand' % (num/1000))

ax.set_title('P.D.F of Posterior Distributions', fontsize=15)
ax.set_xlabel(r"$\theta$", fontsize=15)
ax.set_xlim(0.3, 0.5)

ax.legend(fontsize=11)
plt.show()
```

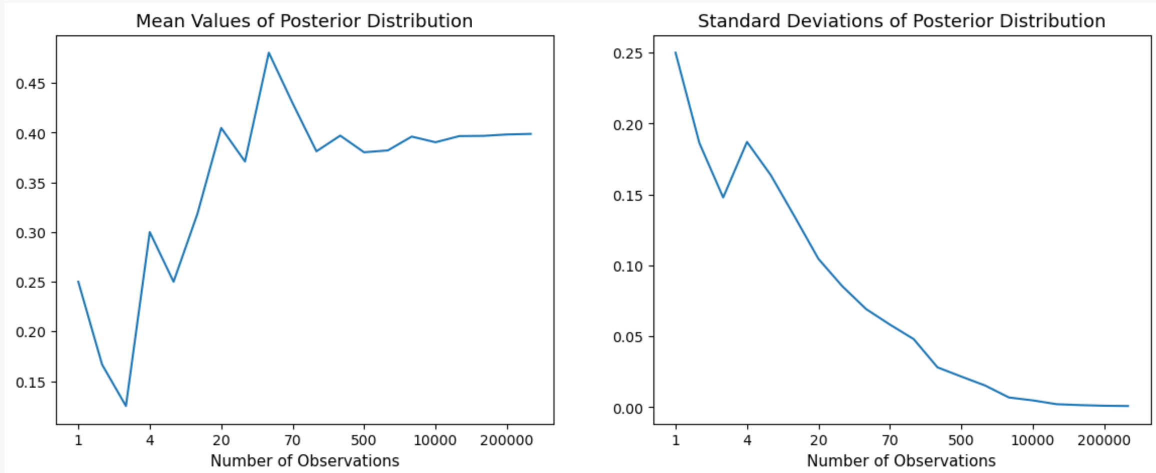


As n increases, we can see that the probability density functions *concentrate* on 0.4, the true value of θ.

Here the posterior means converges to 0.4 while the posterior standard deviations converges to 0 from above.

To show this, we compute the means and variances statistics of the posterior distributions.

```
mean_list = [ii.mean() for ii in Bay_stat.posterior_list]
std_list = [ii.std() for ii in Bay_stat.posterior_list]

fig, ax = plt.subplots(1, 2, figsize=(14, 5))

ax[0].plot(mean_list)
ax[0].set_title('Mean Values of Posterior Distribution', fontsize=13)
ax[0].set_xticks(np.arange(0, len(mean_list), 3))
ax[0].set_xticklabels(num_list[::3])
ax[0].set_xlabel('Number of Observations', fontsize=11)

ax[1].plot(std_list)
ax[1].set_title('Standard Deviations of Posterior Distribution', fontsize=13)
ax[1].set_xticks(np.arange(0, len(std_list), 3))
ax[1].set_xticklabels(num_list[::3])
ax[1].set_xlabel('Number of Observations', fontsize=11)

plt.show()
```



How shall we interpret the patterns above?

The answer is encoded in the Bayesian updating formulas.

It is natural to extend the one-step Bayesian update to an n-step Bayesian update.

$$\begin{aligned}
\textrm{Prob}(\theta|k) &= \frac{\textrm{Prob}(\theta,k)}{\textrm{Prob}(k)} = \frac{\textrm{Prob}(k|\theta)*\textrm{Prob}(\theta)}{\textrm{Prob}(k)} = \frac{\textrm{Prob}(k|\theta)*\textrm{Prob}(\theta)}{\int_0^1 \textrm{Prob}(k|\theta)*\textrm{Prob}(\theta)d\theta} \\
&= \frac{\binom{N}{k}(1-\theta)^{N-k}\theta^k*\frac{\theta^{\alpha-1}(1-\theta)^{\beta-1}}{B(\alpha,\beta)}}{\int_0^1 \binom{N}{k}(1-\theta)^{N-k}\theta^k*\frac{\theta^{\alpha-1}(1-\theta)^{\beta-1}}{B(\alpha,\beta)}d\theta} \\
&= \frac{(1-\theta)^{\beta+N-k-1}*\theta^{\alpha+k-1}}{\int_0^1 (1-\theta)^{\beta+N-k-1}*\theta^{\alpha+k-1}d\theta} \\
&= Beta(\alpha+k,\beta+N-k)
\end{aligned}$$

A beta distribution with α and β has the following mean and variance.

The mean is $\frac{\alpha}{\alpha+\beta}$

The variance is $\frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)}$

- α can be viewed as the number of successes
- β can be viewed as the number of failures

The random variables k and $N-k$ are governed by Binomial Distribution with $\theta = 0.4$.

Call this the true data generating process.

According to the Law of Large Numbers, for a large number of observations, observed frequencies of k and $N-k$ will be described by the true data generating process, i.e., the population probability distribution that we assumed when generating the observations on the computer. (See Exercise 11.2.1).

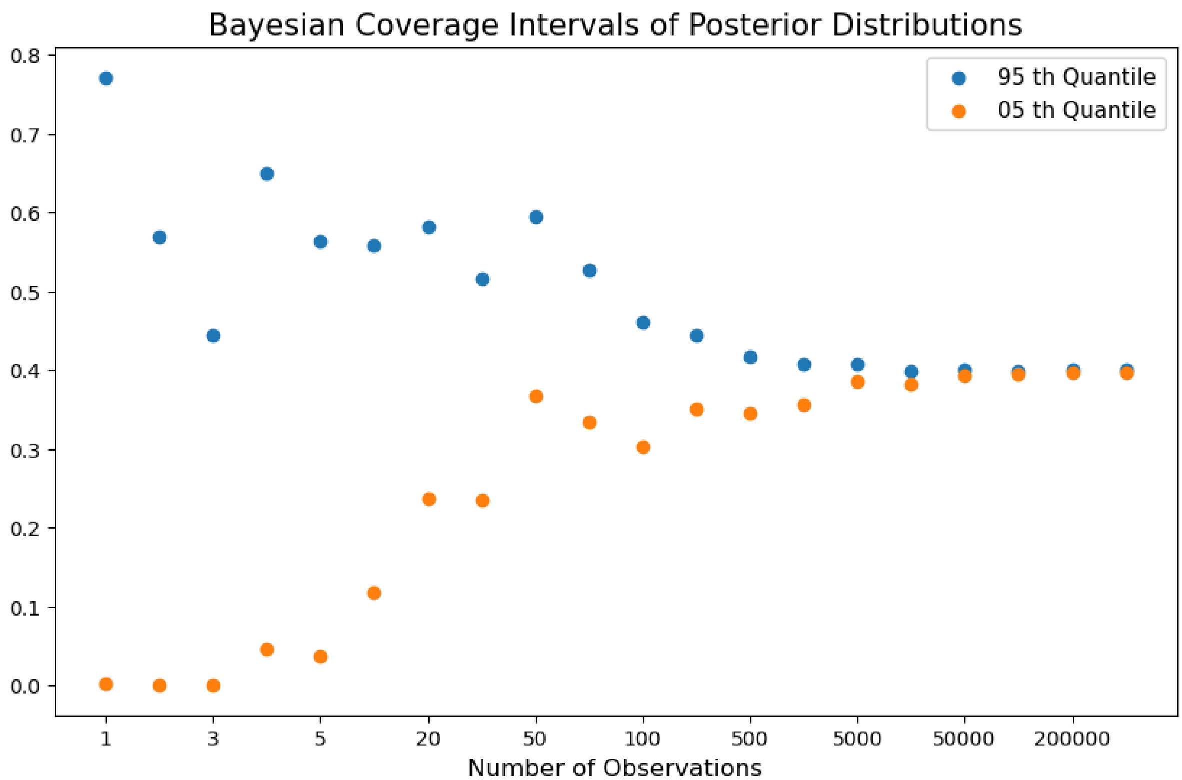
Consequently, the mean of the posterior distribution converges to 0.4 and the variance withers to zero.

```
upper_bound = [ii.ppf(0.95) for ii in Bay_stat.posterior_list]
lower_bound = [ii.ppf(0.05) for ii in Bay_stat.posterior_list]

fig, ax = plt.subplots(figsize=(10, 6))
```

(continues on next page)

(continued from previous page)

```
ax.scatter(np.arange(len(upper_bound)), upper_bound, label='95 th Quantile')
ax.scatter(np.arange(len(lower_bound)), lower_bound, label='05 th Quantile')

ax.set_xticks(np.arange(0, len(upper_bound), 2))
ax.set_xticklabels(num_list[::2])
ax.set_xlabel('Number of Observations', fontsize=12)
ax.set_title('Bayesian Coverage Intervals of Posterior Distributions', fontsize=15)

ax.legend(fontsize=11)
plt.show()
```



After observing a large number of outcomes, the posterior distribution collapses around 0.4.

Thus, the Bayesian statististian comes to believe that θ is near $.4$.

As shown in the figure above, as the number of observations grows, the Bayesian coverage intervals (BCIs) become narrower and narrower around 0.4.

However, if you take a closer look, you will find that the centers of the BCIs are not exactly 0.4, due to the persistent influence of the prior distribution and the randomness of the simulation path.

11.4 Role of a Conjugate Prior

We have made assumptions that link functional forms of our likelihood function and our prior in a way that has eased our calculations considerably.

In particular, our assumptions that the likelihood function is **binomial** and that the prior distribution is a **beta distribution** have the consequence that the posterior distribution implied by Bayes' Law is also a **beta distribution**.

So posterior and prior are both beta distributions, albeit ones with different parameters.

When a likelihood function and prior fit together like hand and glove in this way, we can say that the prior and posterior are **conjugate distributions**.

In this situation, we also sometimes say that we have **conjugate prior** for the likelihood function $\text{Prob}(X|\theta)$.

Typically, the functional form of the likelihood function determines the functional form of a **conjugate prior**.

A natural question to ask is why should a person's personal prior about a parameter θ be restricted to be described by a conjugate prior?

Why not some other functional form that more sincerely describes the person's beliefs?

To be argumentative, one could ask, why should the form of the likelihood function have *anything* to say about my personal beliefs about θ?

A dignified response to that question is, well, it shouldn't, but if you want to compute a posterior easily you'll just be happier if your prior is conjugate to your likelihood.

Otherwise, your posterior won't have a convenient analytical form and you'll be in the situation of wanting to apply the Markov chain Monte Carlo techniques deployed in *this quantecon lecture*.

We also apply these powerful methods to approximating Bayesian posteriors for non-conjugate priors in *this quantecon lecture* and *this quantecon lecture*

CHAPTER

TWELVE

MULTIVARIATE HYPERGEOMETRIC DISTRIBUTION

Contents

12.1 Overview

This lecture describes how an administrator deployed a **multivariate hypergeometric distribution** in order to access the fairness of a procedure for awarding research grants.

In the lecture we'll learn about

- properties of the multivariate hypergeometric distribution
- first and second moments of a multivariate hypergeometric distribution
- using a Monte Carlo simulation of a multivariate normal distribution to evaluate the quality of a normal approximation
- the administrator's problem and why the multivariate hypergeometric distribution is the right tool

12.2 The Administrator's Problem

An administrator in charge of allocating research grants is in the following situation.

To help us forget details that are none of our business here and to protect the anonymity of the administrator and the subjects, we call research proposals **balls** and continents of residence of authors of a proposal a **color**.

There are K_i balls (proposals) of color i.

There are c distinct colors (continents of residence).

Thus, $i = 1, 2, \ldots, c$

So there is a total of $N = \sum_{i=1}^{c} K_i$ balls.

All N of these balls are placed in an urn.

Then n balls are drawn randomly.

The selection procedure is supposed to be **color blind** meaning that **ball quality**, a random variable that is supposed to be independent of **ball color**, governs whether a ball is drawn.

Thus, the selection procedure is supposed randomly to draw n balls from the urn.

The n balls drawn represent successful proposals and are awarded research funds.

The remaining $N - n$ balls receive no research funds.

12.2.1 Details of the Awards Procedure Under Study

Let k_i be the number of balls of color i that are drawn.

Things have to add up so $\sum_{i=1}^{c} k_i = n$.

Under the hypothesis that the selection process judges proposals on their quality and that quality is independent of continent of the author's continent of residence, the administrator views the outcome of the selection procedure as a random vector

$$X = \begin{bmatrix} k_1 \\ k_2 \\ \vdots \\ k_c \end{bmatrix}.$$

To evaluate whether the selection procedure is **color blind** the administrator wants to study whether the particular realization of X drawn can plausibly be said to be a random draw from the probability distribution that is implied by the **color blind** hypothesis.

The appropriate probability distribution is the one described here.

Let's now instantiate the administrator's problem, while continuing to use the colored balls metaphor.

The administrator has an urn with $N = 238$ balls.

157 balls are blue, 11 balls are green, 46 balls are yellow, and 24 balls are black.

So $(K_1, K_2, K_3, K_4) = (157, 11, 46, 24)$ and $c = 4$.

15 balls are drawn without replacement.

So $n = 15$.

The administrator wants to know the probability distribution of outcomes

$$X = \begin{bmatrix} k_1 \\ k_2 \\ \vdots \\ k_4 \end{bmatrix}.$$

In particular, he wants to know whether a particular outcome - in the form of a 4×1 vector of integers recording the numbers of blue, green, yellow, and black balls, respectively, - contains evidence against the hypothesis that the selection process is *fair*, which here means *color blind* and truly are random draws without replacement from the population of N balls.

The right tool for the administrator's job is the **multivariate hypergeometric distribution**.

12.2.2 Multivariate Hypergeometric Distribution

Let's start with some imports.

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.special import comb
from scipy.stats import normaltest
from numba import jit, prange
```

To recapitulate, we assume there are in total c types of objects in an urn.

If there are K_i type i object in the urn and we take n draws at random without replacement, then the numbers of type i objects in the sample $(k_1, k_2, \dots, k_c)$ has the multivariate hypergeometric distribution.

Note again that $N = \sum_{i=1}^{c} K_i$ is the total number of objects in the urn and $n = \sum_{i=1}^{c} k_i$.

Notation

We use the following notation for **binomial coefficients**: $\binom{m}{q} = \frac{m!}{(m-q)!}$.

The multivariate hypergeometric distribution has the following properties:

Probability mass function:

$$\Pr\{X_i = k_i \ \forall i\} = \frac{\prod_{i=1}^{c} \binom{K_i}{k_i}}{\binom{N}{n}}$$

Mean:

$$\mathrm{E}(X_i) = n \frac{K_i}{N}$$

Variances and covariances:

$$\mathrm{Var}(X_i) = n \frac{N-n}{N-1} \frac{K_i}{N} \left(1 - \frac{K_i}{N}\right)$$

$$\mathrm{Cov}(X_i, X_j) = -n \frac{N-n}{N-1} \frac{K_i}{N} \frac{K_j}{N}$$

To do our work for us, we'll write an `Urn` class.

```
class Urn:

    def __init__(self, K_arr):
        """
        Initialization given the number of each type i object in the urn.

        Parameters
        ----------
        K_arr: ndarray(int)
            number of each type i object.
        """

        self.K_arr = np.array(K_arr)
        self.N = np.sum(K_arr)
        self.c = len(K_arr)

    def pmf(self, k_arr):
        """
```

(continues on next page)

(continued from previous page)

```
        Probability mass function.

        Parameters
        ----------
        k_arr: ndarray(int)
            number of observed successes of each object.
        """

        K_arr, N = self.K_arr, self.N

        k_arr = np.atleast_2d(k_arr)
        n = np.sum(k_arr, 1)

        num = np.prod(comb(K_arr, k_arr), 1)
        denom = comb(N, n)

        pr = num / denom

        return pr

    def moments(self, n):
        """
        Compute the mean and variance-covariance matrix for
        multivariate hypergeometric distribution.

        Parameters
        ----------
        n: int
            number of draws.
        """

        K_arr, N, c = self.K_arr, self.N, self.c

        # mean
        μ = n * K_arr / N

        # variance-covariance matrix
        Σ = np.full((c, c), n * (N - n) / (N - 1) / N ** 2)
        for i in range(c-1):
            Σ[i, i] *= K_arr[i] * (N - K_arr[i])
            for j in range(i+1, c):
                Σ[i, j] *= - K_arr[i] * K_arr[j]
                Σ[j, i] = Σ[i, j]

        Σ[-1, -1] *= K_arr[-1] * (N - K_arr[-1])

        return μ, Σ

    def simulate(self, n, size=1, seed=None):
        """
        Simulate a sample from multivariate hypergeometric
        distribution where at each draw we take n objects
        from the urn without replacement.

        Parameters
        ----------
        n: int
```

(continues on next page)

(continued from previous page)

```
            number of objects for each draw.
        size: int(optional)
            sample size.
        seed: int(optional)
            random seed.
        """

        K_arr = self.K_arr

        gen = np.random.Generator(np.random.PCG64(seed))
        sample = gen.multivariate_hypergeometric(K_arr, n, size=size)

        return sample
```

12.3 Usage

12.3.1 First example

Apply this to an example from wiki:

Suppose there are 5 black, 10 white, and 15 red marbles in an urn. If six marbles are chosen without replacement, the probability that exactly two of each color are chosen is

$$P(2 \text{ black}, 2 \text{ white}, 2 \text{ red}) = \frac{\binom{5}{2}\binom{10}{2}\binom{15}{2}}{\binom{30}{6}} = 0.079575596816976$$

```
# construct the urn
K_arr = [5, 10, 15]
urn = Urn(K_arr)
```

Now use the Urn Class method `pmf` to compute the probability of the outcome $X = \begin{bmatrix} 2 & 2 & 2 \end{bmatrix}$

```
k_arr = [2, 2, 2] # array of number of observed successes
urn.pmf(k_arr)
```

```
array([0.0795756])
```

We can use the code to compute probabilities of a list of possible outcomes by constructing a 2-dimensional array `k_arr` and `pmf` will return an array of probabilities for observing each case.

```
k_arr = [[2, 2, 2], [1, 3, 2]]
urn.pmf(k_arr)
```

```
array([0.0795756, 0.1061008])
```

Now let's compute the mean vector and variance-covariance matrix.

```
n = 6
μ, Σ = urn.moments(n)
```

```
μ
```

```
array([1., 2., 3.])
```

```
Σ
```

```
array([[ 0.68965517, -0.27586207, -0.4137931 ],
       [-0.27586207,  1.10344828, -0.82758621],
       [-0.4137931 , -0.82758621,  1.24137931]])
```

12.3.2 Back to The Administrator's Problem

Now let's turn to the grant administrator's problem.

Here the array of numbers of i objects in the urn is $(157, 11, 46, 24)$.

```
K_arr = [157, 11, 46, 24]
urn = Urn(K_arr)
```

Let's compute the probability of the outcome $(10, 1, 4, 0)$.

```
k_arr = [10, 1, 4, 0]
urn.pmf(k_arr)
```

```
array([0.01547738])
```

We can compute probabilities of three possible outcomes by constructing a 3-dimensional arrays `k_arr` and utilizing the method `pmf` of the `Urn` class.

```
k_arr = [[5, 5, 4 ,1], [10, 1, 2, 2], [13, 0, 2, 0]]
urn.pmf(k_arr)
```

```
array([6.21412534e-06, 2.70935969e-02, 1.61839976e-02])
```

Now let's compute the mean and variance-covariance matrix of X when $n = 6$.

```
n = 6 # number of draws
μ, Σ = urn.moments(n)
```

```
# mean
μ
```

```
array([3.95798319, 0.27731092, 1.15966387, 0.60504202])
```

```
# variance-covariance matrix
Σ
```

```
array([[ 1.31862604, -0.17907267, -0.74884935, -0.39070401],
       [-0.17907267,  0.25891399, -0.05246715, -0.02737417],
       [-0.74884935, -0.05246715,  0.91579029, -0.11447379],
       [-0.39070401, -0.02737417, -0.11447379,  0.53255196]])
```

We can simulate a large sample and verify that sample means and covariances closely approximate the population means and covariances.

```
size = 10_000_000
sample = urn.simulate(n, size=size)
```

```
# mean
np.mean(sample, 0)
```

```
array([3.9579126, 0.2773305, 1.1595624, 0.6051945])
```

```
# variance covariance matrix
np.cov(sample.T)
```

```
array([[ 1.31916158, -0.1793529 , -0.74882221, -0.39098648],
       [-0.1793529 ,  0.25906652, -0.05227543, -0.0274382 ],
       [-0.74882221, -0.05227543,  0.91585813, -0.1147605 ],
       [-0.39098648, -0.0274382 , -0.1147605 ,  0.53318517]])
```

Evidently, the sample means and covariances approximate their population counterparts well.

12.3.3 Quality of Normal Approximation

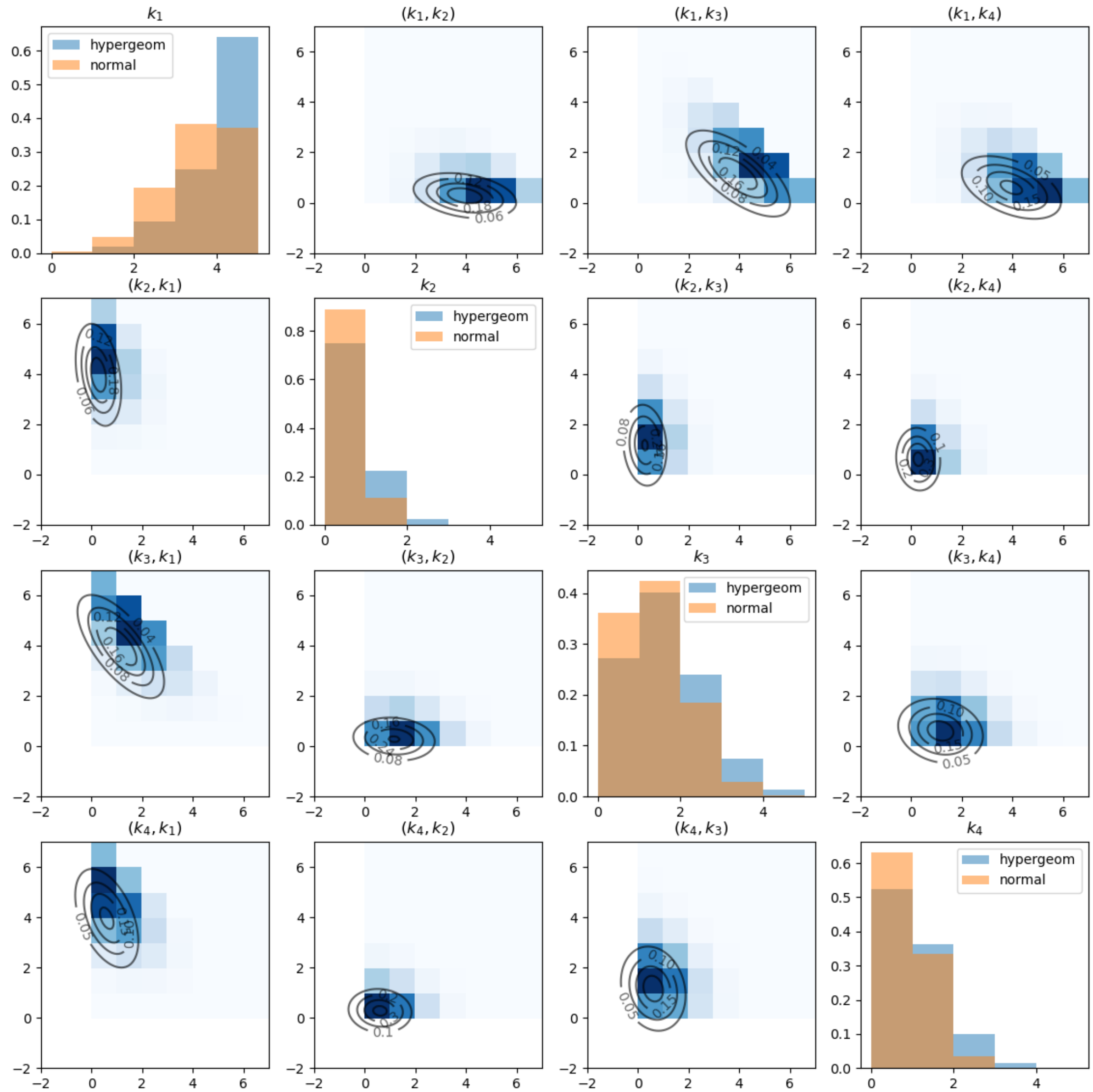
To judge the quality of a multivariate normal approximation to the multivariate hypergeometric distribution, we draw a large sample from a multivariate normal distribution with the mean vector and covariance matrix for the corresponding multivariate hypergeometric distribution and compare the simulated distribution with the population multivariate hypergeometric distribution.

```
sample_normal = np.random.multivariate_normal(μ, Σ, size=size)
```

```
def bivariate_normal(x, y, μ, Σ, i, j):

    μ_x, μ_y = μ[i], μ[j]
    σ_x, σ_y = np.sqrt(Σ[i, i]), np.sqrt(Σ[j, j])
    σ_xy = Σ[i, j]

    x_μ = x - μ_x
    y_μ = y - μ_y

    ρ = σ_xy / (σ_x * σ_y)
    z = x_μ**2 / σ_x**2 + y_μ**2 / σ_y**2 - 2 * ρ * x_μ * y_μ / (σ_x * σ_y)
    denom = 2 * np.pi * σ_x * σ_y * np.sqrt(1 - ρ**2)

    return np.exp(-z / (2 * (1 - ρ**2))) / denom
```

```
@jit
def count(vec1, vec2, n):
    size = sample.shape[0]

    count_mat = np.zeros((n+1, n+1))
    for i in prange(size):
        count_mat[vec1[i], vec2[i]] += 1

    return count_mat
```

```python
c = urn.c
fig, axs = plt.subplots(c, c, figsize=(14, 14))

# grids for ploting the bivariate Gaussian
x_grid = np.linspace(-2, n+1, 100)
y_grid = np.linspace(-2, n+1, 100)
X, Y = np.meshgrid(x_grid, y_grid)

for i in range(c):
    axs[i, i].hist(sample[:, i], bins=np.arange(0, n, 1), alpha=0.5, density=True,␣
↪label='hypergeom')
    axs[i, i].hist(sample_normal[:, i], bins=np.arange(0, n, 1), alpha=0.5,␣
↪density=True, label='normal')
    axs[i, i].legend()
    axs[i, i].set_title('$k_{' +str(i+1) +'}$')
    for j in range(c):
        if i == j:
            continue

        # bivariate Gaussian density function
        Z = bivariate_normal(X, Y, μ, Σ, i, j)
        cs = axs[i, j].contour(X, Y, Z, 4, colors="black", alpha=0.6)
        axs[i, j].clabel(cs, inline=1, fontsize=10)

        # empirical multivariate hypergeometric distrbution
        count_mat = count(sample[:, i], sample[:, j], n)
        axs[i, j].pcolor(count_mat.T/size, cmap='Blues')
        axs[i, j].set_title('$(k_{' +str(i+1) +'}, k_{' + str(j+1) + '})$')

plt.show()
```



The diagonal graphs plot the marginal distributions of k_i for each i using histograms.

Note the substantial differences between hypergeometric distribution and the approximating normal distribution.

The off-diagonal graphs plot the empirical joint distribution of k_i and k_j for each pair (i, j).

The darker the blue, the more data points are contained in the corresponding cell. (Note that k_i is on the x-axis and k_j is on the y-axis).

The contour maps plot the bivariate Gaussian density function of (k_i, k_j) with the population mean and covariance given by slices of μ and Σ that we computed above.

Let's also test the normality for each k_i using `scipy.stats.normaltest` that implements D'Agostino and Pearson's test that combines skew and kurtosis to form an omnibus test of normality.

The null hypothesis is that the sample follows normal distribution.

> `normaltest` returns an array of p-values associated with tests for each k_i sample.

```
test_multihyper = normaltest(sample)
test_multihyper.pvalue
```

```
array([0., 0., 0., 0.])
```

As we can see, all the p-values are almost 0 and the null hypothesis is soundly rejected.

By contrast, the sample from normal distribution does not reject the null hypothesis.

```
test_normal = normaltest(sample_normal)
test_normal.pvalue
```

```
array([0.52261218, 0.57634582, 0.05691857, 0.68951118])
```

The lesson to take away from this is that the normal approximation is imperfect.

CHAPTER

THIRTEEN

MULTIVARIATE NORMAL DISTRIBUTION

Contents

13.1 Overview

This lecture describes a workhorse in probability theory, statistics, and economics, namely, the **multivariate normal distribution**.

In this lecture, you will learn formulas for

- the joint distribution of a random vector x of length N
- marginal distributions for all subvectors of x
- conditional distributions for subvectors of x conditional on other subvectors of x

We will use the multivariate normal distribution to formulate some useful models:

- a factor analytic model of an intelligence quotient, i.e., IQ
- a factor analytic model of two independent inherent abilities, say, mathematical and verbal.
- a more general factor analytic model
- Principal Components Analysis (PCA) as an approximation to a factor analytic model
- time series generated by linear stochastic difference equations
- optimal linear filtering theory

13.2 The Multivariate Normal Distribution

This lecture defines a Python class `MultivariateNormal` to be used to generate **marginal** and **conditional** distributions associated with a multivariate normal distribution.

For a multivariate normal distribution it is very convenient that

- conditional expectations equal linear least squares projections
- conditional distributions are characterized by multivariate linear regressions

We apply our Python class to some examples.

We use the following imports:

```
import matplotlib.pyplot as plt
import numpy as np
from numba import jit
import statsmodels.api as sm
```

Assume that an $N \times 1$ random vector z has a multivariate normal probability density.

This means that the probability density takes the form

$$f(z;\mu,\Sigma) = (2\pi)^{-\left(\frac{N}{2}\right)} \det(\Sigma)^{-\frac{1}{2}} \exp\left(-.5\,(z-\mu)'\,\Sigma^{-1}\,(z-\mu)\right)$$

where $\mu = Ez$ is the mean of the random vector z and $\Sigma = E(z-\mu)(z-\mu)'$ is the covariance matrix of z.

The covariance matrix Σ is symmetric and positive definite.

```
@jit
def f(z, μ, Σ):
    """
    The density function of multivariate normal distribution.

    Parameters
    ---------------
    z: ndarray(float, dim=2)
        random vector, N by 1
    μ: ndarray(float, dim=1 or 2)
        the mean of z, N by 1
    Σ: ndarray(float, dim=2)
        the covarianece matrix of z, N by 1
    """

    z = np.atleast_2d(z)
    μ = np.atleast_2d(μ)
    Σ = np.atleast_2d(Σ)
```

(continues on next page)

(continued from previous page)

```
    N = z.size

    temp1 = np.linalg.det(Σ) ** (-1/2)
    temp2 = np.exp(-.5 * (z - μ).T @ np.linalg.inv(Σ) @ (z - μ))

    return (2 * np.pi) ** (-N/2) * temp1 * temp2
```

For some integer $k \in \{1, \ldots, N-1\}$, partition z as

$$z = \left[\begin{array}{c} z_1 \\ z_2 \end{array} \right],$$

where z_1 is an $(N-k) \times 1$ vector and z_2 is a $k \times 1$ vector.

Let

$$\mu = \left[\begin{array}{c} \mu_1 \\ \mu_2 \end{array} \right], \quad \Sigma = \left[\begin{array}{cc} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{array} \right]$$

be corresponding partitions of μ and Σ.

The **marginal** distribution of z_1 is

- multivariate normal with mean μ_1 and covariance matrix Σ_{11}.

The **marginal** distribution of z_2 is

- multivariate normal with mean μ_2 and covariance matrix Σ_{22}.

The distribution of z_1 **conditional** on z_2 is

- multivariate normal with mean

$$\hat{\mu}_1 = \mu_1 + \beta \left(z_2 - \mu_2 \right)$$

and covariance matrix

$$\hat{\Sigma}_{11} = \Sigma_{11} - \Sigma_{12} \Sigma_{22}^{-1} \Sigma_{21} = \Sigma_{11} - \beta \Sigma_{22} \beta'$$

where

$$\beta = \Sigma_{12} \Sigma_{22}^{-1}$$

is an $(N-k) \times k$ matrix of **population regression coefficients** of the $(N-k) \times 1$ random vector $z_1 - \mu_1$ on the $k \times 1$ random vector $z_2 - \mu_2$.

The following class constructs a multivariate normal distribution instance with two methods.

- a method `partition` computes β, taking k as an input
- a method `cond_dist` computes either the distribution of z_1 conditional on z_2 or the distribution of z_2 conditional on z_1

```
class MultivariateNormal:
    """
    Class of multivariate normal distribution.

    Parameters
    ----------
    μ: ndarray(float, dim=1)
```

(continues on next page)

(continued from previous page)

```
        the mean of z, N by 1
    Σ: ndarray(float, dim=2)
        the covarianece matrix of z, N by 1

    Arguments
    ---------
    μ, Σ:
        see parameters
    μs: list(ndarray(float, dim=1))
        list of mean vectors μ1 and μ2 in order
    Σs: list(list(ndarray(float, dim=2)))
        2 dimensional list of covariance matrices
        Σ11, Σ12, Σ21, Σ22 in order
    βs: list(ndarray(float, dim=1))
        list of regression coefficients β1 and β2 in order
    """

    def __init__(self, μ, Σ):
        "initialization"
        self.μ = np.array(μ)
        self.Σ = np.atleast_2d(Σ)

    def partition(self, k):
        """
        Given k, partition the random vector z into a size k vector z1
        and a size N-k vector z2. Partition the mean vector μ into
        μ1 and μ2, and the covariance matrix Σ into Σ11, Σ12, Σ21, Σ22
        correspondingly. Compute the regression coefficients β1 and β2
        using the partitioned arrays.
        """
        μ = self.μ
        Σ = self.Σ

        self.μs = [μ[:k], μ[k:]]
        self.Σs = [[Σ[:k, :k], Σ[:k, k:]],
                   [Σ[k:, :k], Σ[k:, k:]]]

        self.βs = [self.Σs[0][1] @ np.linalg.inv(self.Σs[1][1]),
                   self.Σs[1][0] @ np.linalg.inv(self.Σs[0][0])]

    def cond_dist(self, ind, z):
        """
        Compute the conditional distribution of z1 given z2, or reversely.
        Argument ind determines whether we compute the conditional
        distribution of z1 (ind=0) or z2 (ind=1).

        Returns
        ---------
        μ_hat: ndarray(float, ndim=1)
            The conditional mean of z1 or z2.
        Σ_hat: ndarray(float, ndim=2)
            The conditional covariance matrix of z1 or z2.
        """
        β = self.βs[ind]
        μs = self.μs
        Σs = self.Σs
```

(continues on next page)

(continued from previous page)

```
        μ_hat = μs[ind] + β @ (z - μs[1-ind])
        Σ_hat = Σs[ind][ind] - β @ Σs[1-ind][1-ind] @ β.T

        return μ_hat, Σ_hat
```

Let's put this code to work on a suite of examples.

We begin with a simple bivariate example; after that we'll turn to a trivariate example.

We'll compute population moments of some conditional distributions using our `MultivariateNormal` class.

For fun we'll also compute sample analogs of the associated population regressions by generating simulations and then computing linear least squares regressions.

We'll compare those linear least squares regressions for the simulated data to their population counterparts.

13.3 Bivariate Example

We start with a bivariate normal distribution pinned down by

$$\mu = \begin{bmatrix} .5 \\ 1.0 \end{bmatrix}, \quad \Sigma = \begin{bmatrix} 1 & .5 \\ .5 & 1 \end{bmatrix}$$

```
μ = np.array([.5, 1.])
Σ = np.array([[1., .5], [.5 ,1.]])

# construction of the multivariate normal instance
multi_normal = MultivariateNormal(μ, Σ)
```

```
k = 1 # choose partition

# partition and compute regression coefficients
multi_normal.partition(k)
multi_normal.βs[0],multi_normal.βs[1]
```

```
(array([[0.5]]), array([[0.5]]))
```

Let's illustrate the fact that you *can regress anything on anything else*.

We have computed everything we need to compute two regression lines, one of z_2 on z_1, the other of z_1 on z_2.

We'll represent these regressions as

$$z_1 = a_1 + b_1 z_2 + \epsilon_1$$

and

$$z_2 = a_2 + b_2 z_1 + \epsilon_2$$

where we have the population least squares orthogonality conditions

$$E\epsilon_1 z_2 = 0$$

and

$$E\epsilon_2 z_1 = 0$$

Let's compute a_1, a_2, b_1, b_2.

```
beta = multi_normal.βs

a1 = μ[0] - beta[0]*μ[1]
b1 = beta[0]

a2 = μ[1] - beta[1]*μ[0]
b2 = beta[1]
```

Let's print out the intercepts and slopes.

For the regression of z_1 on z_2 we have

```
print ("a1 = ", a1)
print ("b1 = ", b1)
```

```
a1 =  [[0.]]
b1 =  [[0.5]]
```

For the regression of z_2 on z_1 we have

```
print ("a2 = ", a2)
print ("b2 = ", b2)
```

```
a2 =  [[0.75]]
b2 =  [[0.5]]
```

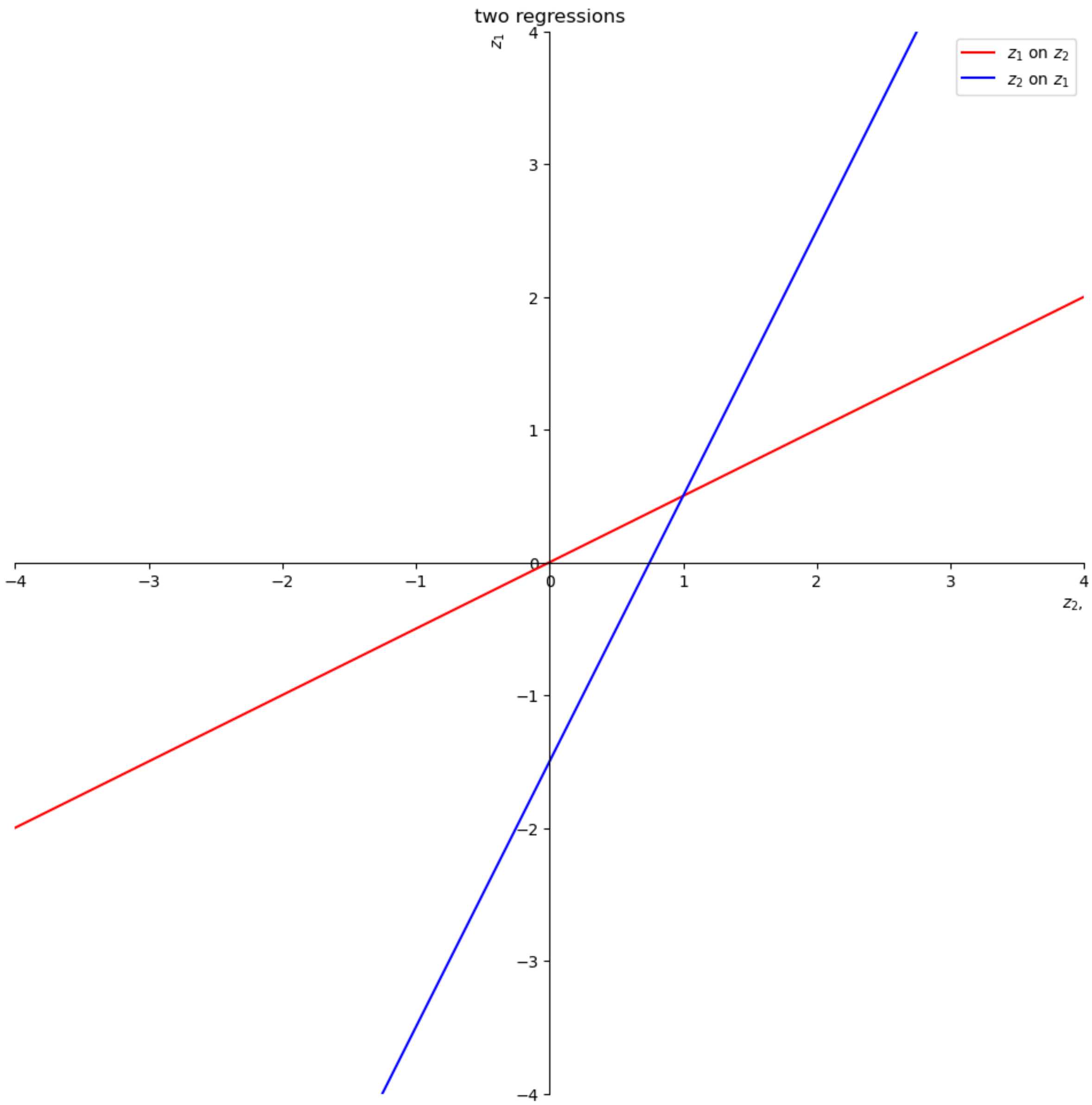
Now let's plot the two regression lines and stare at them.

```
z2 = np.linspace(-4,4,100)


a1 = np.squeeze(a1)
b1 = np.squeeze(b1)

a2 = np.squeeze(a2)
b2 = np.squeeze(b2)

z1  = b1*z2 + a1


z1h = z2/b2 - a2/b2


fig = plt.figure(figsize=(12,12))
ax = fig.add_subplot(1, 1, 1)
ax.set(xlim=(-4, 4), ylim=(-4, 4))
ax.spines['left'].set_position('center')
ax.spines['bottom'].set_position('zero')
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')
plt.ylabel('$z_1$', loc = 'top')
plt.xlabel('$z_2$,', loc = 'right')
plt.title('two regressions')
plt.plot(z2,z1, 'r', label = "$z_1$ on $z_2$")
plt.plot(z2,z1h, 'b', label = "$z_2$ on $z_1$")
```

(continues on next page)

(continued from previous page)

```
plt.legend()
plt.show()
```



The red line is the expectation of z_1 conditional on z_2.

The intercept and slope of the red line are

```
print("a1 = ", a1)
print("b1 = ", b1)
```

```
a1 =  0.0
b1 =  0.5
```

The blue line is the expectation of z_2 conditional on z_1.

The intercept and slope of the blue line are

```
print("-a2/b2 = ", - a2/b2)
print("1/b2 = ", 1/b2)
```

```
-a2/b2 =  -1.5
1/b2 =  2.0
```

We can use these regression lines or our code to compute conditional expectations.

Let's compute the mean and variance of the distribution of z_2 conditional on $z_1 = 5$.

After that we'll reverse what are on the left and right sides of the regression.

```
# compute the cond. dist. of z1
ind = 1
z1 = np.array([5.]) # given z1

μ2_hat, Σ2_hat = multi_normal.cond_dist(ind, z1)
print('μ2_hat, Σ2_hat = ', μ2_hat, Σ2_hat)
```

```
μ2_hat, Σ2_hat =  [3.25] [[0.75]]
```

Now let's compute the mean and variance of the distribution of z_1 conditional on $z_2 = 5$.

```
# compute the cond. dist. of z1
ind = 0
z2 = np.array([5.]) # given z2

μ1_hat, Σ1_hat = multi_normal.cond_dist(ind, z2)
print('μ1_hat, Σ1_hat = ', μ1_hat, Σ1_hat)
```

```
μ1_hat, Σ1_hat =  [2.5] [[0.75]]
```

Let's compare the preceding population mean and variance with outcomes from drawing a large sample and then regressing $z_1 - \mu_1$ on $z_2 - \mu_2$.

We know that

$$Ez_1|z_2 = (\mu_1 - \beta\mu_2) + \beta z_2$$

which can be arranged to

$$z_1 - \mu_1 = \beta\left(z_2 - \mu_2\right) + \epsilon,$$

We anticipate that for larger and larger sample sizes, estimated OLS coefficients will converge to β and the estimated variance of ϵ will converge to $\hat{\Sigma}_1$.

```
n = 1_000_000 # sample size

# simulate multivariate normal random vectors
data = np.random.multivariate_normal(μ, Σ, size=n)
z1_data = data[:, 0]
z2_data = data[:, 1]

# OLS regression
μ1, μ2 = multi_normal.μs
results = sm.OLS(z1_data - μ1, z2_data - μ2).fit()
```

Let's compare the preceding population β with the OLS sample estimate on $z_2 - \mu_2$

```
multi_normal.βs[0], results.params
```

```
(array([[0.5]]), array([0.50029482]))
```

Let's compare our population $\hat{\Sigma}_1$ with the degrees-of-freedom adjusted estimate of the variance of ϵ

```
Σ1_hat, results.resid @ results.resid.T / (n - 1)
```

```
(array([[0.75]]), np.float64(0.749283355563193))
```

Lastly, let's compute the estimate of $\widehat{Ez_1|z_2}$ and compare it with $\hat{\mu}_1$

```
μ1_hat, results.predict(z2 - μ2) + μ1
```

```
(array([2.5]), array([2.50117929]))
```

Thus, in each case, for our very large sample size, the sample analogues closely approximate their population counterparts.

A Law of Large Numbers explains why sample analogues approximate population objects.

13.4 Trivariate Example

Let's apply our code to a trivariate example.

We'll specify the mean vector and the covariance matrix as follows.

```
μ = np.random.random(3)
C = np.random.random((3, 3))
Σ = C @ C.T # positive semi-definite

multi_normal = MultivariateNormal(μ, Σ)
```

```
μ, Σ
```

```
(array([0.64701751, 0.3394191 , 0.50890234]),
 array([[1.08141667, 1.19827171, 1.30763619],
        [1.19827171, 1.63553175, 1.41592525],
        [1.30763619, 1.41592525, 1.96688636]]))
```

```
k = 1
multi_normal.partition(k)
```

Let's compute the distribution of z_1 conditional on $z_2 = \begin{bmatrix} 2 \\ 5 \end{bmatrix}$.

```
ind = 0
z2 = np.array([2., 5.])

μ1_hat, Σ1_hat = multi_normal.cond_dist(ind, z2)
```

```
n = 1_000_000
data = np.random.multivariate_normal(μ, Σ, size=n)
z1_data = data[:, :k]
z2_data = data[:, k:]
```

```
μ1, μ2 = multi_normal.μs
results = sm.OLS(z1_data - μ1, z2_data - μ2).fit()
```

As above, we compare population and sample regression coefficients, the conditional covariance matrix, and the conditional mean vector in that order.

```
multi_normal.βs[0], results.params
```

```
(array([[0.41693386, 0.36468248]]), array([0.41748255, 0.36444232]))
```

```
Σ1_hat, results.resid @ results.resid.T / (n - 1)
```

```
(array([[0.10494461]]), np.float64(0.10495864020461919))
```

```
μ1_hat, results.predict(z2 - μ2) + μ1
```

```
(array([2.97719457]), array([2.97702712]))
```

Once again, sample analogues do a good job of approximating their populations counterparts.

13.5 One Dimensional Intelligence (IQ)

Let's move closer to a real-life example, namely, inferring a one-dimensional measure of intelligence called IQ from a list of test scores.

The ith test score y_i equals the sum of an unknown scalar IQ θ and a random variable w_i.

$$y_i = \theta + \sigma_y w_i, \quad i = 1, \dots, n$$

The distribution of IQ's for a cross-section of people is a normal random variable described by

$$\theta = \mu_\theta + \sigma_\theta w_{n+1}.$$

We assume that the noises $\{w_i\}_{i=1}^N$ in the test scores are IID and not correlated with IQ.

We also assume that $\{w_i\}_{i=1}^{n+1}$ are i.i.d. standard normal:

$$w = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \\ w_{n+1} \end{bmatrix} \sim N\left(0, I_{n+1}\right)$$

The following system describes the $(n+1) \times 1$ random vector X that interests us:

$$X = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \\ \theta \end{bmatrix} = \begin{bmatrix} \mu_\theta \\ \mu_\theta \\ \vdots \\ \mu_\theta \\ \mu_\theta \end{bmatrix} + \begin{bmatrix} \sigma_y & 0 & \cdots & 0 & \sigma_\theta \\ 0 & \sigma_y & \cdots & 0 & \sigma_\theta \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & \sigma_y & \sigma_\theta \\ 0 & 0 & \cdots & 0 & \sigma_\theta \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \\ w_{n+1} \end{bmatrix},$$

or equivalently,

$$X = \mu_\theta 1_{n+1} + Dw$$

where $X = \begin{bmatrix} y \\ \theta \end{bmatrix}$, 1_{n+1} is a vector of 1s of size $n+1$, and D is an $n+1$ by $n+1$ matrix.

Let's define a Python function that constructs the mean μ and covariance matrix Σ of the random vector X that we know is governed by a multivariate normal distribution.

As arguments, the function takes the number of tests n, the mean μ_θ and the standard deviation σ_θ of the IQ distribution, and the standard deviation of the randomness in test scores σ_y.

```
def construct_moments_IQ(n, μθ, σθ, σy):

    μ_IQ = np.full(n+1, μθ)

    D_IQ = np.zeros((n+1, n+1))
    D_IQ[range(n), range(n)] = σy
    D_IQ[:, n] = σθ

    Σ_IQ = D_IQ @ D_IQ.T

    return μ_IQ, Σ_IQ, D_IQ
```

Now let's consider a specific instance of this model.

Assume we have recorded 50 test scores and we know that $\mu_\theta = 100$, $\sigma_\theta = 10$, and $\sigma_y = 10$.

We can compute the mean vector and covariance matrix of X easily with our `construct_moments_IQ` function as follows.

```
n = 50
μθ, σθ, σy = 100., 10., 10.

μ_IQ, Σ_IQ, D_IQ = construct_moments_IQ(n, μθ, σθ, σy)
μ_IQ, Σ_IQ, D_IQ
```

```
(array([100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
        100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
        100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
        100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
        100., 100., 100., 100., 100., 100., 100.]),
 array([[200., 100., 100., ..., 100., 100., 100.],
        [100., 200., 100., ..., 100., 100., 100.],
        [100., 100., 200., ..., 100., 100., 100.],
        ...,
        [100., 100., 100., ..., 200., 100., 100.],
        [100., 100., 100., ..., 100., 200., 100.],
        [100., 100., 100., ..., 100., 100., 100.]], shape=(51, 51)),
 array([[10.,  0.,  0., ...,  0.,  0., 10.],
        [ 0., 10.,  0., ...,  0.,  0., 10.],
        [ 0.,  0., 10., ...,  0.,  0., 10.],
        ...,
        [ 0.,  0.,  0., ..., 10.,  0., 10.],
        [ 0.,  0.,  0., ...,  0., 10., 10.],
        [ 0.,  0.,  0., ...,  0.,  0., 10.]], shape=(51, 51)))
```

We can now use our `MultivariateNormal` class to construct an instance, then partition the mean vector and covariance matrix as we wish.

We want to regress IQ, the random variable θ (*what we don't know*), on the vector y of test scores (*what we do know*).

We choose `k=n` so that $z_1 = y$ and $z_2 = \theta$.

```
multi_normal_IQ = MultivariateNormal(μ_IQ, Σ_IQ)

k = n
multi_normal_IQ.partition(k)
```

Using the generator `multivariate_normal`, we can make one draw of the random vector from our distribution and then compute the distribution of θ conditional on our test scores.

Let's do that and then print out some pertinent quantities.

```
x = np.random.multivariate_normal(μ_IQ, Σ_IQ)
y = x[:-1] # test scores
θ = x[-1]  # IQ
```

```
# the true value
θ
```

```
np.float64(99.70637905687724)
```

The method `cond_dist` takes test scores y as input and returns the conditional normal distribution of the IQ θ.

In the following code, `ind` sets the variables on the right side of the regression.

Given the way we have defined the vector X, we want to set `ind=1` in order to make θ the left side variable in the population regression.

```
ind = 1
multi_normal_IQ.cond_dist(ind, y)
```

```
(array([101.10833632]), array([[1.96078431]]))
```

The first number is the conditional mean $\hat{\mu}_\theta$ and the second is the conditional variance $\hat{\Sigma}_\theta$.

How do additional test scores affect our inferences?

To shed light on this, we compute a sequence of conditional distributions of θ by varying the number of test scores in the conditioning set from 1 to n.

We'll make a pretty graph showing how our judgment of the person's IQ change as more test results come in.

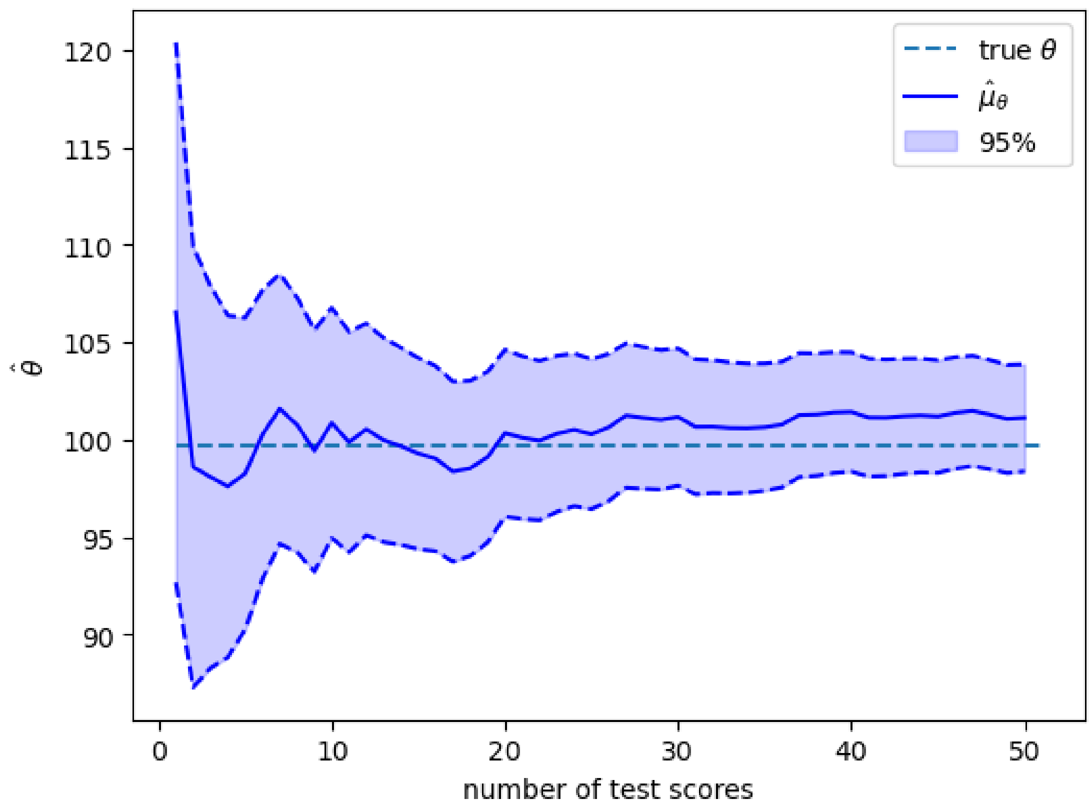
```
# array for containing moments
μθ_hat_arr = np.empty(n)
Σθ_hat_arr = np.empty(n)

# loop over number of test scores
for i in range(1, n+1):
    # construction of multivariate normal distribution instance
    μ_IQ_i, Σ_IQ_i, D_IQ_i = construct_moments_IQ(i, μθ, σθ, σy)
    multi_normal_IQ_i = MultivariateNormal(μ_IQ_i, Σ_IQ_i)

    # partition and compute conditional distribution
    multi_normal_IQ_i.partition(i)
    scores_i = y[:i]
    μθ_hat_i, Σθ_hat_i = multi_normal_IQ_i.cond_dist(1, scores_i)

    # store the results
    μθ_hat_arr[i-1] = μθ_hat_i[0]
    Σθ_hat_arr[i-1] = Σθ_hat_i[0, 0]
```

(continues on next page)

(continued from previous page)

```
# transform variance to standard deviation
σθ_hat_arr = np.sqrt(Σθ_hat_arr)
```

```
μθ_hat_lower = μθ_hat_arr - 1.96 * σθ_hat_arr
μθ_hat_higher = μθ_hat_arr + 1.96 * σθ_hat_arr

plt.hlines(θ, 1, n+1, ls='--', label='true $θ$')
plt.plot(range(1, n+1), μθ_hat_arr, color='b', label=r'$\hat{μ}_{θ}$')
plt.plot(range(1, n+1), μθ_hat_lower, color='b', ls='--')
plt.plot(range(1, n+1), μθ_hat_higher, color='b', ls='--')
plt.fill_between(range(1, n+1), μθ_hat_lower, μθ_hat_higher,
                 color='b', alpha=0.2, label='95%')

plt.xlabel('number of test scores')
plt.ylabel(r'$\hat{θ}$')
plt.legend()

plt.show()
```



The solid blue line in the plot above shows $\hat{\mu}_\theta$ as a function of the number of test scores that we have recorded and conditioned on.

The blue area shows the span that comes from adding or subtracting $1.96\hat{\sigma}_\theta$ from $\hat{\mu}_\theta$.

Therefore, 95% of the probability mass of the conditional distribution falls in this range.

The value of the random θ that we drew is shown by the black dotted line.

As more and more test scores come in, our estimate of the person's θ become more and more reliable.

By staring at the changes in the conditional distributions, we see that adding more test scores makes $\hat{\theta}$ settle down and approach θ.

Thus, each y_i adds information about θ.

If we were to drive the number of tests $n \rightarrow +\infty$, the conditional standard deviation $\hat{\sigma}_\theta$ would converge to 0 at rate $\frac{1}{n^{.5}}$.

13.6 Information as Surprise

By using a different representation, let's look at things from a different perspective.

We can represent the random vector X defined above as

$$X = \mu_\theta 1_{n+1} + C\epsilon, \quad \epsilon \sim N\left(0, I\right)$$

where C is a lower triangular **Cholesky factor** of Σ so that

$$\Sigma \equiv DD' = CC'$$

and

$$E\epsilon\epsilon' = I.$$

It follows that

$$\epsilon \sim N(0, I).$$

Let $G = C^{-1}$

G is also lower triangular.

We can compute ϵ from the formula

$$\epsilon = G\left(X - \mu_\theta 1_{n+1}\right)$$

This formula confirms that the orthonormal vector ϵ contains the same information as the non-orthogonal vector $\left(X - \mu_\theta 1_{n+1}\right)$.

We can say that ϵ is an orthogonal basis for $\left(X - \mu_\theta 1_{n+1}\right)$.

Let c_i be the ith element in the last row of C.

Then we can write

$$\theta = \mu_\theta + c_1\epsilon_1 + c_2\epsilon_2 + \dots + c_n\epsilon_n + c_{n+1}\epsilon_{n+1} \tag{13.1}$$

The mutual orthogonality of the ϵ_i's provides us with an informative way to interpret them in light of equation (13.1).

Thus, relative to what is known from tests $i = 1, \dots, n-1$, $c_i\epsilon_i$ is the amount of **new information** about θ brought by the test number i.

Here **new information** means **surprise** or what could not be predicted from earlier information.

Formula (13.1) also provides us with an enlightening way to express conditional means and conditional variances that we computed earlier.

In particular,

$$E\left[\theta \mid y_1, \dots, y_k\right] = \mu_\theta + c_1\epsilon_1 + \dots + c_k\epsilon_k$$

and

$$Var(\theta \mid y_1, \dots, y_k) = c_{k+1}^2 + c_{k+2}^2 + \dots + c_{n+1}^2.$$

```
C = np.linalg.cholesky(Σ_IQ)
G = np.linalg.inv(C)

ε = G @ (x - μθ)
```

```
cε = C[n, :] * ε

# compute the sequence of μθ and Σθ conditional on y1, y2, ..., yk
μθ_hat_arr_C = np.array([np.sum(cε[:k+1]) for k in range(n)]) + μθ
Σθ_hat_arr_C = np.array([C[n, i+1:n+1] @ C[n, i+1:n+1] for i in range(n)])
```

To confirm that these formulas give the same answers that we computed earlier, we can compare the means and variances of θ conditional on $\{y_i\}_{i=1}^k$ with what we obtained above using the formulas implemented in the class `MultivariateNormal` built on our original representation of conditional distributions for multivariate normal distributions.

```
# conditional mean
np.max(np.abs(μθ_hat_arr - μθ_hat_arr_C)) < 1e-10
```

```
np.True_
```

```
# conditional variance
np.max(np.abs(Σθ_hat_arr - Σθ_hat_arr_C)) < 1e-10
```

```
np.True_
```

13.7 Cholesky Factor Magic

Evidently, the Cholesky factorizations automatically computes the population **regression coefficients** and associated statistics that are produced by our `MultivariateNormal` class.

The Cholesky factorization computes these things **recursively**.

Indeed, in formula (13.1),

- the random variable $c_i \epsilon_i$ is information about θ that is not contained by the information in $\epsilon_1, \epsilon_2, \dots, \epsilon_{i-1}$
- the coefficient c_i is the simple population regression coefficient of $\theta - \mu_\theta$ on ϵ_i

13.8 Math and Verbal Intelligence

We can alter the preceding example to be more realistic.

There is ample evidence that IQ is not a scalar.

Some people are good in math skills but poor in language skills.

Other people are good in language skills but poor in math skills.

So now we shall assume that there are two dimensions of IQ, θ and η.

These determine average performances in math and language tests, respectively.

We observe math scores $\{y_i\}_{i=1}^{n}$ and language scores $\{y_i\}_{i=n+1}^{2n}$.

When $n = 2$, we assume that outcomes are draws from a multivariate normal distribution with representation

$$X = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ \theta \\ \eta \end{bmatrix} = \begin{bmatrix} \mu_\theta \\ \mu_\theta \\ \mu_\eta \\ \mu_\eta \\ \mu_\theta \\ \mu_\eta \end{bmatrix} + \begin{bmatrix} \sigma_y & 0 & 0 & 0 & \sigma_\theta & 0 \\ 0 & \sigma_y & 0 & 0 & \sigma_\theta & 0 \\ 0 & 0 & \sigma_y & 0 & 0 & \sigma_\eta \\ 0 & 0 & 0 & \sigma_y & 0 & \sigma_\eta \\ 0 & 0 & 0 & 0 & \sigma_\theta & 0 \\ 0 & 0 & 0 & 0 & 0 & \sigma_\eta \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \\ w_5 \\ w_6 \end{bmatrix}$$

where $w \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_6 \end{bmatrix}$ is a standard normal random vector.

We construct a Python function `construct_moments_IQ2d` to construct the mean vector and covariance matrix of the joint normal distribution.

```
def construct_moments_IQ2d(n, μθ, σθ, μη, ση, σy):

    μ_IQ2d = np.empty(2*(n+1))
    μ_IQ2d[:n] = μθ
    μ_IQ2d[2*n] = μθ
    μ_IQ2d[n:2*n] = μη
    μ_IQ2d[2*n+1] = μη


    D_IQ2d = np.zeros((2*(n+1), 2*(n+1)))
    D_IQ2d[range(2*n), range(2*n)] = σy
    D_IQ2d[:n, 2*n] = σθ
    D_IQ2d[2*n, 2*n] = σθ
    D_IQ2d[n:2*n, 2*n+1] = ση
    D_IQ2d[2*n+1, 2*n+1] = ση

    Σ_IQ2d = D_IQ2d @ D_IQ2d.T

    return μ_IQ2d, Σ_IQ2d, D_IQ2d
```

Let's put the function to work.

```
n = 2
# mean and variance of θ, η, and y
μθ, σθ, μη, ση, σy = 100., 10., 100., 10, 10

μ_IQ2d, Σ_IQ2d, D_IQ2d = construct_moments_IQ2d(n, μθ, σθ, μη, ση, σy)
μ_IQ2d, Σ_IQ2d, D_IQ2d
```

```
(array([100., 100., 100., 100., 100., 100.]),
 array([[200., 100.,   0.,   0., 100.,   0.],
        [100., 200.,   0.,   0., 100.,   0.],
        [  0.,   0., 200., 100.,   0., 100.],
        [  0.,   0., 100., 200.,   0., 100.],
        [100., 100.,   0.,   0., 100.,   0.],
        [  0.,   0., 100., 100.,   0., 100.]]),
 array([[10.,  0.,  0.,  0., 10.,  0.],
        [ 0., 10.,  0.,  0., 10.,  0.],
        [ 0.,  0., 10.,  0.,  0., 10.],
```

(continues on next page)

(continued from previous page)

```
       [ 0.,   0.,   0., 10.,   0., 10.],
       [ 0.,   0.,   0.,   0., 10.,   0.],
       [ 0.,   0.,   0.,   0.,   0., 10.]]))
```

```
# take one draw
x = np.random.multivariate_normal(μ_IQ2d, Σ_IQ2d)
y1 = x[:n]
y2 = x[n:2*n]
θ = x[2*n]
η = x[2*n+1]

# the true values
θ, η
```

```
(np.float64(85.39576376650763), np.float64(89.89199777745883))
```

We first compute the joint normal distribution of (θ, η).

```
multi_normal_IQ2d = MultivariateNormal(μ_IQ2d, Σ_IQ2d)

k = 2*n # the length of data vector
multi_normal_IQ2d.partition(k)

multi_normal_IQ2d.cond_dist(1, [*y1, *y2])
```

```
(array([91.82885281, 93.38617802]),
 array([[33.33333333,  0.        ],
        [ 0.        , 33.33333333]]))
```

Now let's compute distributions of θ and μ separately conditional on various subsets of test scores.

It will be fun to compare outcomes with the help of an auxiliary function `cond_dist_IQ2d` that we now construct.

```
def cond_dist_IQ2d(μ, Σ, data):

    n = len(μ)

    multi_normal = MultivariateNormal(μ, Σ)
    multi_normal.partition(n-1)
    μ_hat, Σ_hat = multi_normal.cond_dist(1, data)

    return μ_hat, Σ_hat
```

Let's see how things work for an example.

```
for indices, IQ, conditions in [([*range(2*n), 2*n], 'θ', 'y1, y2, y3, y4'),
                                ([*range(n), 2*n], 'θ', 'y1, y2'),
                                ([*range(n, 2*n), 2*n], 'θ', 'y3, y4'),
                                ([*range(2*n), 2*n+1], 'η', 'y1, y2, y3, y4'),
                                ([*range(n), 2*n+1], 'η', 'y1, y2'),
                                ([*range(n, 2*n), 2*n+1], 'η', 'y3, y4')]:

    μ_hat, Σ_hat = cond_dist_IQ2d(μ_IQ2d[indices], Σ_IQ2d[indices][:, indices],
↪x[indices[:-1]])
    print(f'The mean and variance of {IQ} conditional on {conditions: <15} are ' +
          f'{μ_hat[0]:1.2f} and {Σ_hat[0, 0]:1.2f} respectively')
```

```
The mean and variance of θ conditional on y1, y2, y3, y4  are 91.83 and 33.33␣
↪respectively
The mean and variance of θ conditional on y1, y2          are 91.83 and 33.33␣
↪respectively
The mean and variance of θ conditional on y3, y4          are 100.00 and 100.00␣
↪respectively
The mean and variance of η conditional on y1, y2, y3, y4  are 93.39 and 33.33␣
↪respectively
The mean and variance of η conditional on y1, y2          are 100.00 and 100.00␣
↪respectively
The mean and variance of η conditional on y3, y4          are 93.39 and 33.33␣
↪respectively
```

Evidently, math tests provide no information about μ and language tests provide no information about η.

13.9 Univariate Time Series Analysis

We can use the multivariate normal distribution and a little matrix algebra to present foundations of univariate linear time series analysis.

Let x_t, y_t, v_t, w_{t+1} each be scalars for $t \geq 0$.

Consider the following model:

$$
\begin{aligned}
x_0 &\sim N\left(0, \sigma_0^2\right) \\
x_{t+1} &= a x_t + b w_{t+1}, \quad w_{t+1} \sim N\left(0, 1\right), t \geq 0 \\
y_t &= c x_t + d v_t, \quad v_t \sim N\left(0, 1\right), t \geq 0
\end{aligned}
$$

We can compute the moments of x_t

1. $E x_{t+1}^2 = a^2 E x_t^2 + b^2, t \geq 0$, where $E x_0^2 = \sigma_0^2$
2. $E x_{t+j} x_t = a^j E x_t^2, \forall t \ \forall j$

Given some T, we can formulate the sequence $\{x_t\}_{t=0}^T$ as a random vector

$$
X = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_T \end{bmatrix}
$$

and the covariance matrix Σ_x can be constructed using the moments we have computed above.

Similarly, we can define

$$
Y = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_T \end{bmatrix}, \quad v = \begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ v_T \end{bmatrix}
$$

and therefore

$$
Y = CX + DV
$$

where C and D are both diagonal matrices with constant c and d as diagonal respectively.

Consequently, the covariance matrix of Y is

$$
\Sigma_y = E Y Y' = C \Sigma_x C' + D D'
$$

By stacking X and Y, we can write

$$Z = \begin{bmatrix} X \\ Y \end{bmatrix}$$

and

$$\Sigma_z = EZZ' = \begin{bmatrix} \Sigma_x & \Sigma_x C' \\ C\Sigma_x & \Sigma_y \end{bmatrix}$$

Thus, the stacked sequences $\{x_t\}_{t=0}^T$ and $\{y_t\}_{t=0}^T$ jointly follow the multivariate normal distribution $N(0, \Sigma_z)$.

```
# as an example, consider the case where T = 3
T = 3
```

```
# variance of the initial distribution x_0
σ0 = 1.

# parameters of the equation system
a = .9
b = 1.
c = 1.0
d = .05
```

```
# construct the covariance matrix of X
Σx = np.empty((T+1, T+1))

Σx[0, 0] = σ0 ** 2
for i in range(T):
    Σx[i, i+1:] = Σx[i, i] * a ** np.arange(1, T+1-i)
    Σx[i+1:, i] = Σx[i, i+1:]

    Σx[i+1, i+1] = a ** 2 * Σx[i, i] + b ** 2
```

```
Σx
```

```
array([[1.      , 0.9     , 0.81    , 0.729   ],
       [0.9     , 1.81    , 1.629   , 1.4661  ],
       [0.81    , 1.629   , 2.4661  , 2.21949 ],
       [0.729   , 1.4661  , 2.21949 , 2.997541]])
```

```
# construct the covariance matrix of Y
C = np.eye(T+1) * c
D = np.eye(T+1) * d

Σy = C @ Σx @ C.T + D @ D.T
```

```
# construct the covariance matrix of Z
Σz = np.empty((2*(T+1), 2*(T+1)))

Σz[:T+1, :T+1] = Σx
Σz[:T+1, T+1:] = Σx @ C.T
Σz[T+1:, :T+1] = C @ Σx
Σz[T+1:, T+1:] = Σy
```

```
Σz
```

```
array([[1.      , 0.9     , 0.81    , 0.729   , 1.      , 0.9     ,
        0.81    , 0.729   ],
       [0.9     , 1.81    , 1.629   , 1.4661  , 0.9     , 1.81    ,
        1.629   , 1.4661  ],
       [0.81    , 1.629   , 2.4661  , 2.21949 , 0.81    , 1.629   ,
        2.4661  , 2.21949 ],
       [0.729   , 1.4661  , 2.21949 , 2.997541, 0.729   , 1.4661  ,
        2.21949 , 2.997541],
       [1.      , 0.9     , 0.81    , 0.729   , 1.0025  , 0.9     ,
        0.81    , 0.729   ],
       [0.9     , 1.81    , 1.629   , 1.4661  , 0.9     , 1.8125  ,
        1.629   , 1.4661  ],
       [0.81    , 1.629   , 2.4661  , 2.21949 , 0.81    , 1.629   ,
        2.4686  , 2.21949 ],
       [0.729   , 1.4661  , 2.21949 , 2.997541, 0.729   , 1.4661  ,
        2.21949 , 3.000041]])
```

```
# construct the mean vector of Z
μz = np.zeros(2*(T+1))
```

The following Python code lets us sample random vectors X and Y.

This is going to be very useful for doing the conditioning to be used in the fun exercises below.

```
z = np.random.multivariate_normal(μz, Σz)

x = z[:T+1]
y = z[T+1:]
```

13.9.1 Smoothing Example

This is an instance of a classic `smoothing` calculation whose purpose is to compute $EX \mid Y$.

An interpretation of this example is

- X is a random sequence of hidden Markov state variables x_t
- Y is a sequence of observed signals y_t bearing information about the hidden state

```
# construct a MultivariateNormal instance
multi_normal_ex1 = MultivariateNormal(μz, Σz)
x = z[:T+1]
y = z[T+1:]
```

```
# partition Z into X and Y
multi_normal_ex1.partition(T+1)
```

```
# compute the conditional mean and covariance matrix of X given Y=y

print("X = ", x)
print("Y = ", y)
print(" E [ X | Y] = ", )

multi_normal_ex1.cond_dist(0, y)
```

```
X =  [-1.86903552 -0.98966078  0.06213788 -1.87778974]
Y =  [-1.86046702 -0.94807184 -0.00479698 -1.89975518]
 E [ X | Y] =
```

```
(array([-1.85420968, -0.94797928, -0.01114336, -1.89504265]),
 array([[2.48875094e-03, 5.57449314e-06, 1.24861727e-08, 2.80242496e-11],
        [5.57449314e-06, 2.48876343e-03, 5.57452116e-06, 1.25113948e-08],
        [1.24861727e-08, 5.57452116e-06, 2.48876346e-03, 5.58575339e-06],
        [2.80242496e-11, 1.25113948e-08, 5.58575339e-06, 2.49377812e-03]]))
```

13.9.2 Filtering Exercise

Compute $E[x_t \mid y_{t-1}, y_{t-2}, \dots, y_0]$.

To do so, we need to first construct the mean vector and the covariance matrix of the subvector $[x_t, y_0, \dots, y_{t-2}, y_{t-1}]$.

For example, let's say that we want the conditional distribution of x_3.

```
t = 3
```

```
# mean of the subvector
sub_μz = np.zeros(t+1)

# covariance matrix of the subvector
sub_Σz = np.empty((t+1, t+1))

sub_Σz[0, 0] = Σz[t, t] # x_t
sub_Σz[0, 1:] = Σz[t, T+1:T+t+1]
sub_Σz[1:, 0] = Σz[T+1:T+t+1, t]
sub_Σz[1:, 1:] = Σz[T+1:T+t+1, T+1:T+t+1]
```

```
sub_Σz
```

```
array([[2.997541, 0.729   , 1.4661  , 2.21949 ],
       [0.729   , 1.0025  , 0.9     , 0.81    ],
       [1.4661  , 0.9     , 1.8125  , 1.629   ],
       [2.21949 , 0.81    , 1.629   , 2.4686  ]])
```

```
multi_normal_ex2 = MultivariateNormal(sub_μz, sub_Σz)
multi_normal_ex2.partition(1)
```

```
sub_y = y[:t]

multi_normal_ex2.cond_dist(0, sub_y)
```

```
(array([-0.00622137]), array([[1.00201996]]))
```

13.9.3 Prediction Exercise

Compute $E\left[y_{t}\mid y_{t-j},\dots,y_{0}\right]$.

As what we did in exercise 2, we will construct the mean vector and covariance matrix of the subvector $\left[y_{t},y_{0},\dots,y_{t-j-1},y_{t-j}\right]$.

For example, we take a case in which $t=3$ and $j=2$.

```
t = 3
j = 2
```

```
sub_μz = np.zeros(t-j+2)
sub_Σz = np.empty((t-j+2, t-j+2))

sub_Σz[0, 0] = Σz[T+t+1, T+t+1]
sub_Σz[0, 1:] = Σz[T+t+1, T+1:T+t-j+2]
sub_Σz[1:, 0] = Σz[T+1:T+t-j+2, T+t+1]
sub_Σz[1:, 1:] = Σz[T+1:T+t-j+2, T+1:T+t-j+2]
```

```
sub_Σz
```

```
array([[3.000041, 0.729   , 1.4661  ],
       [0.729   , 1.0025  , 0.9     ],
       [1.4661  , 0.9     , 1.8125  ]])
```

```
multi_normal_ex3 = MultivariateNormal(sub_μz, sub_Σz)
multi_normal_ex3.partition(1)
```

```
sub_y = y[:t-j+1]

multi_normal_ex3.cond_dist(0, sub_y)
```

```
(array([-0.76939401]), array([[1.81413617]]))
```

13.9.4 Constructing a Wold Representation

Now we'll apply Cholesky decomposition to decompose $\Sigma_{y}=HH^{\prime}$ and form

$$\epsilon=H^{-1}Y.$$

Then we can represent y_{t} as

$$y_{t}=h_{t,t}\epsilon_{t}+h_{t,t-1}\epsilon_{t-1}+\dots+h_{t,0}\epsilon_{0}.$$

```
H = np.linalg.cholesky(Σy)

H
```

```
array([[1.00124922, 0.        , 0.        , 0.        ],
       [0.8988771 , 1.00225743, 0.        , 0.        ],
       [0.80898939, 0.89978675, 1.00225743, 0.        ],
       [0.72809046, 0.80980808, 0.89978676, 1.00225743]])
```

```
ε = np.linalg.inv(H) @ y

ε
```

```
array([-1.85814579,  0.72054628,  0.84817057, -1.88926892])
```

```
y
```

```
array([-1.86046702, -0.94807184, -0.00479698, -1.89975518])
```

This example is an instance of what is known as a **Wold representation** in time series analysis.

13.10 Stochastic Difference Equation

Consider the stochastic second-order linear difference equation

$$y_t = \alpha_0 + \alpha_1 y_{y-1} + \alpha_2 y_{t-2} + u_t$$

where $u_t \sim N\left(0, \sigma_u^2\right)$ and

$$\begin{bmatrix} y_{-1} \\ y_0 \end{bmatrix} \sim N\left(\mu_{\tilde{y}}, \Sigma_{\tilde{y}}\right)$$

It can be written as a stacked system

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ -\alpha_1 & 1 & 0 & 0 & \cdots & 0 & 0 & 0 \\ -\alpha_2 & -\alpha_1 & 1 & 0 & \cdots & 0 & 0 & 0 \\ 0 & -\alpha_2 & -\alpha_1 & 1 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & -\alpha_2 & -\alpha_1 & 1 \end{bmatrix}}_{\equiv A} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ \vdots \\ y_T \end{bmatrix} = \underbrace{\begin{bmatrix} \alpha_0 + \alpha_1 y_0 + \alpha_2 y_{-1} \\ \alpha_0 + \alpha_2 y_0 \\ \alpha_0 \\ \alpha_0 \\ \vdots \\ \alpha_0 \end{bmatrix}}_{\equiv b} + \underbrace{\begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ \vdots \\ u_T \end{bmatrix}}_{\equiv u}$$

We can compute y by solving the system

$$y = A^{-1}\left(b + u\right)$$

We have

$$\begin{aligned} \mu_y &= A^{-1}\mu_b \\ \Sigma_y &= A^{-1}E\left[\left(b - \mu_b + u\right)\left(b - \mu_b + u\right)'\right]\left(A^{-1}\right)' \\ &= A^{-1}\left(\Sigma_b + \Sigma_u\right)\left(A^{-1}\right)' \end{aligned}$$

where

$$\mu_b = \begin{bmatrix} \alpha_0 + \alpha_1 \mu_{y_0} + \alpha_2 \mu_{y_{-1}} \\ \alpha_0 + \alpha_2 \mu_{y_0} \\ \alpha_0 \\ \vdots \\ \alpha_0 \end{bmatrix}$$

$$\Sigma_b = \begin{bmatrix} C\Sigma_{\tilde{y}}C' & 0_{N-2\times N-2} \\ 0_{N-2\times 2} & 0_{N-2\times N-2} \end{bmatrix}, \quad C = \begin{bmatrix} \alpha_2 & \alpha_1 \\ 0 & \alpha_2 \end{bmatrix}$$

$$\Sigma_u = \begin{bmatrix} \sigma_u^2 & 0 & \cdots & 0 \\ 0 & \sigma_u^2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & \sigma_u^2 \end{bmatrix}$$

```
# set parameters
T = 80
T = 160
# coefficients of the second order difference equation
𝛼0 = 10
𝛼1 = 1.53
𝛼2 = -.9

# variance of u
σu = 1.
σu = 10.

# distribution of y_{-1} and y_{0}
μy_tilde = np.array([1., 0.5])
Σy_tilde = np.array([[2., 1.], [1., 0.5]])
```

```
# construct A and A^{\prime}
A = np.zeros((T, T))

for i in range(T):
    A[i, i] = 1

    if i-1 >= 0:
        A[i, i-1] = -𝛼1

    if i-2 >= 0:
        A[i, i-2] = -𝛼2

A_inv = np.linalg.inv(A)
```

```
# compute the mean vectors of b and y
μb = np.full(T, 𝛼0)
μb[0] += 𝛼1 * μy_tilde[1] + 𝛼2 * μy_tilde[0]
μb[1] += 𝛼2 * μy_tilde[1]

μy = A_inv @ μb
```

```
# compute the covariance matrices of b and y
Σu = np.eye(T) * σu ** 2

Σb = np.zeros((T, T))

C = np.array([[𝛼2, 𝛼1], [0, 𝛼2]])
Σb[:2, :2] = C @ Σy_tilde @ C.T

Σy = A_inv @ (Σb + Σu) @ A_inv.T
```

13.11 Application to Stock Price Model

Let

$$p_t = \sum_{j=0}^{T-t} \beta^j y_{t+j}$$

Form

$$\underbrace{\left[\begin{array}{c} p_1 \\ p_2 \\ p_3 \\ \vdots \\ p_T \end{array}\right]}_{\equiv p} = \underbrace{\left[\begin{array}{ccccc} 1 & \beta & \beta^2 & \cdots & \beta^{T-1} \\ 0 & 1 & \beta & \cdots & \beta^{T-2} \\ 0 & 0 & 1 & \cdots & \beta^{T-3} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{array}\right]}_{\equiv B} \left[\begin{array}{c} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_T \end{array}\right]$$

we have

$$\mu_p = B\mu_y$$
$$\Sigma_p = B\Sigma_y B'$$

```
β = .96
```

```
# construct B
B = np.zeros((T, T))

for i in range(T):
    B[i, i:] = β ** np.arange(0, T-i)
```

Denote

$$z = \left[\begin{array}{c} y \\ p \end{array}\right] = \underbrace{\left[\begin{array}{c} I \\ B \end{array}\right]}_{\equiv D} y$$

Thus, $\{y_t\}_{t=1}^T$ and $\{p_t\}_{t=1}^T$ jointly follow the multivariate normal distribution $N(\mu_z, \Sigma_z)$, where

$$\mu_z = D\mu_y$$

$$\Sigma_z = D\Sigma_y D'$$

```
D = np.vstack([np.eye(T), B])
```

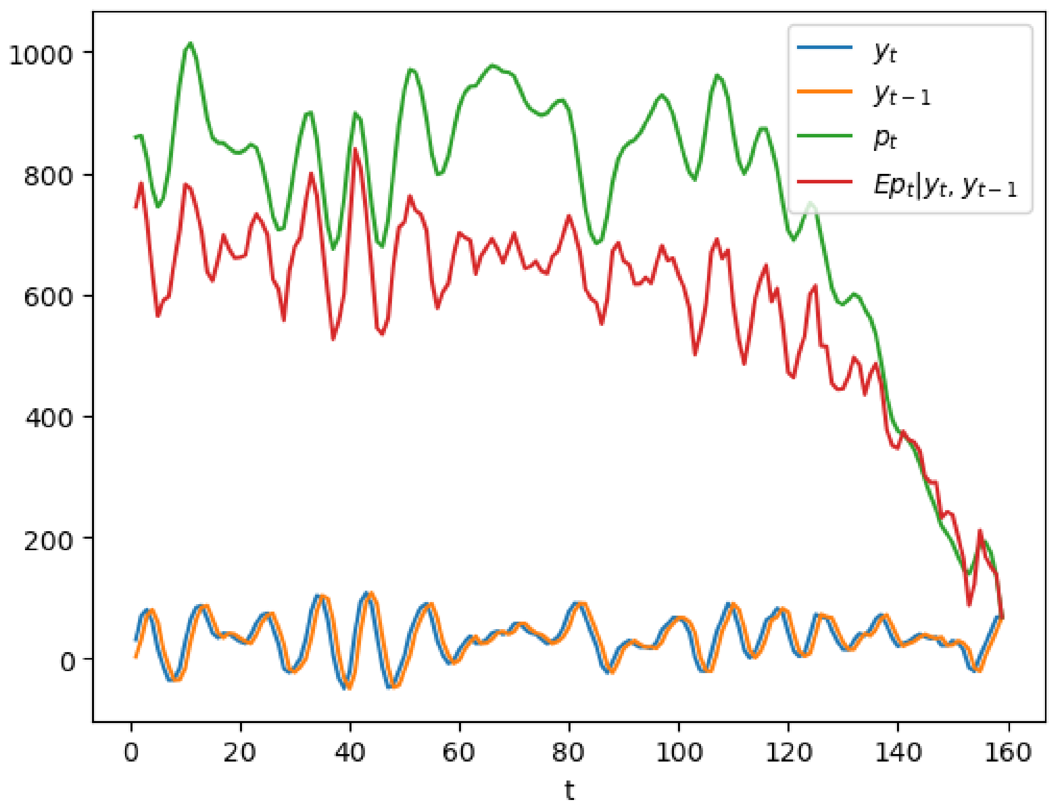
```
μz = D @ μy
Σz = D @ Σy @ D.T
```

We can simulate paths of y_t and p_t and compute the conditional mean $E[p_t \mid y_{t-1}, y_t]$ using the `MultivariateNormal` class.

```
z = np.random.multivariate_normal(μz, Σz)
y, p = z[:T], z[T:]
```

```
cond_Ep = np.empty(T-1)

sub_μ = np.empty(3)
sub_Σ = np.empty((3, 3))
for t in range(2, T+1):
    sub_μ[:] = μz[[t-2, t-1, T-1+t]]
    sub_Σ[:, :] = Σz[[t-2, t-1, T-1+t], :][:, [t-2, t-1, T-1+t]]

    multi_normal = MultivariateNormal(sub_μ, sub_Σ)
    multi_normal.partition(2)

    cond_Ep[t-2] = multi_normal.cond_dist(1, y[t-2:t])[0][0]
```

```
plt.plot(range(1, T), y[1:], label='$y_{t}$')
plt.plot(range(1, T), y[:-1], label='$y_{t-1}$')
plt.plot(range(1, T), p[1:], label='$p_{t}$')
plt.plot(range(1, T), cond_Ep, label='$Ep_{t}|y_{t}, y_{t-1}$')

plt.xlabel('t')
plt.legend(loc=1)
plt.show()
```



In the above graph, the green line is what the price of the stock would be if people had perfect foresight about the path of dividends while the green line is the conditional expectation $Ep_t|y_t, y_{t-1}$, which is what the price would be if people did not have perfect foresight but were optimally predicting future dividends on the basis of the information y_t, y_{t-1} at time t.

13.12 Filtering Foundations

Assume that x_0 is an $n \times 1$ random vector and that y_0 is a $p \times 1$ random vector determined by the *observation equation*

$$y_0 = Gx_0 + v_0, \quad x_0 \sim \mathcal{N}(\hat{x}_0, \Sigma_0), \quad v_0 \sim \mathcal{N}(0, R)$$

where v_0 is orthogonal to x_0, G is a $p \times n$ matrix, and R is a $p \times p$ positive definite matrix.

We consider the problem of someone who

- *observes* y_0
- does not observe x_0,
- knows $\hat{x}_0, \Sigma_0, G, R$ and therefore the joint probability distribution of the vector $\begin{bmatrix} x_0 \\ y_0 \end{bmatrix}$
- wants to infer x_0 from y_0 in light of what he knows about that joint probability distribution.

Therefore, the person wants to construct the probability distribution of x_0 conditional on the random vector y_0.

The joint distribution of $\begin{bmatrix} x_0 \\ y_0 \end{bmatrix}$ is multivariate normal $\mathcal{N}(\mu, \Sigma)$ with

$$\mu = \begin{bmatrix} \hat{x}_0 \\ G\hat{x}_0 \end{bmatrix}, \quad \Sigma = \begin{bmatrix} \Sigma_0 & \Sigma_0 G' \\ G\Sigma_0 & G\Sigma_0 G' + R \end{bmatrix}$$

By applying an appropriate instance of the above formulas for the mean vector $\hat{\mu}_1$ and covariance matrix $\hat{\Sigma}_{11}$ of z_1 conditional on z_2, we find that the probability distribution of x_0 conditional on y_0 is $\mathcal{N}(\tilde{x}_0, \tilde{\Sigma}_0)$ where

$$\begin{aligned} \beta_0 &= \Sigma_0 G'(G\Sigma_0 G' + R)^{-1} \\ \tilde{x}_0 &= \hat{x}_0 + \beta_0(y_0 - G\hat{x}_0) \\ \tilde{\Sigma}_0 &= \Sigma_0 - \Sigma_0 G'(G\Sigma_0 G' + R)^{-1} G\Sigma_0 \end{aligned}$$

We can express our finding that the probability distribution of x_0 conditional on y_0 is $\mathcal{N}(\tilde{x}_0, \tilde{\Sigma}_0)$ by representing x_0 as

$$x_0 = \tilde{x}_0 + \zeta_0 \tag{13.2}$$

where ζ_0 is a Gaussian random vector that is orthogonal to $\tilde{x}_0$ and y_0 and that has mean vector 0 and conditional covariance matrix $E[\zeta_0 \zeta_0' | y_0] = \tilde{\Sigma}_0$.

13.12.1 Step toward dynamics

Now suppose that we are in a time series setting and that we have the one-step state transition equation

$$x_1 = Ax_0 + Cw_1, \quad w_1 \sim \mathcal{N}(0, I)$$

where A is an $n \times n$ matrix and C is an $n \times m$ matrix.

Using equation (13.2), we can also represent x_1 as

$$x_1 = A(\tilde{x}_0 + \zeta_0) + Cw_1$$

It follows that

$$Ex_1 | y_0 = A\tilde{x}_0$$

and that the corresponding conditional covariance matrix $E(x_1 - Ex_1|y_0)(x_1 - Ex_1|y_0)' \equiv \Sigma_1$ is

$$\Sigma_1 = A\tilde{\Sigma}_0 A' + CC'$$

or

$$\Sigma_1 = A\Sigma_0 A' - A\Sigma_0 G'(G\Sigma_0 G' + R)^{-1} G\Sigma_0 A'$$

We can write the mean of x_1 conditional on y_0 as

$$\hat{x}_1 = A\hat{x}_0 + A\Sigma_0 G'(G\Sigma_0 G' + R)^{-1}(y_0 - G\hat{x}_0)$$

or

$$\hat{x}_1 = A\hat{x}_0 + K_0(y_0 - G\hat{x}_0)$$

where

$$K_0 = A\Sigma_0 G'(G\Sigma_0 G' + R)^{-1}$$

13.12.2 Dynamic version

Suppose now that for $t \geq 0$, $\{x_{t+1}, y_t\}_{t=0}^{\infty}$ are governed by the equations

$$\begin{aligned} x_{t+1} &= Ax_t + Cw_{t+1} \\ y_t &= Gx_t + v_t \end{aligned}$$

where as before $x_0 \sim \mathcal{N}(\hat{x}_0, \Sigma_0)$, w_{t+1} is the $t+1$th component of an i.i.d. stochastic process distributed as $w_{t+1} \sim \mathcal{N}(0, I)$, and v_t is the tth component of an i.i.d. process distributed as $v_t \sim \mathcal{N}(0, R)$ and the $\{w_{t+1}\}_{t=0}^{\infty}$ and $\{v_t\}_{t=0}^{\infty}$ processes are orthogonal at all pairs of dates.

The logic and formulas that we applied above imply that the probability distribution of x_t conditional on $y_0, y_1, \ldots, y_{t-1} = y^{t-1}$ is

$$x_t | y^{t-1} \sim \mathcal{N}(A\tilde{x}_t, A\tilde{\Sigma}_t A' + CC')$$

where $\{\tilde{x}_t, \tilde{\Sigma}_t\}_{t=1}^{\infty}$ can be computed by iterating on the following equations starting from $t = 1$ and initial conditions for $\tilde{x}_0, \tilde{\Sigma}_0$ computed as we have above:

$$\begin{aligned} \Sigma_t &= A\tilde{\Sigma}_{t-1} A' + CC' \\ \hat{x}_t &= A\tilde{x}_{t-1} \\ \beta_t &= \Sigma_t G'(G\Sigma_t G' + R)^{-1} \\ \tilde{x}_t &= \hat{x}_t + \beta_t(y_t - G\hat{x}_t) \\ \tilde{\Sigma}_t &= \Sigma_t - \Sigma_t G'(G\Sigma_t G' + R)^{-1} G\Sigma_t \end{aligned}$$

If we shift the first equation forward one period and then substitute the expression for $\tilde{\Sigma}_t$ on the right side of the fifth equation into it we obtain

$$\Sigma_{t+1} = CC' + A\Sigma_t A' - A\Sigma_t G'(G\Sigma_t G' + R)^{-1} G\Sigma_t A'.$$

This is a matrix Riccati difference equation that is closely related to another matrix Riccati difference equation that appears in a quantecon lecture on the basics of linear quadratic control theory.

That equation has the form

$$P_{t-1} = R + A' P_t A - A' P_t B (B' P_t B + Q)^{-1} B' P_t A.$$

Stare at the two preceding equations for a moment or two, the first being a matrix difference equation for a conditional covariance matrix, the second being a matrix difference equation in the matrix appearing in a quadratic form for an intertemporal cost of value function.

Although the two equations are not identical, they display striking family resemblences.

- the first equation tells dynamics that work **forward** in time
- the second equation tells dynamics that work **backward** in time
- while many of the terms are similar, one equation seems to apply matrix transformations to some matrices that play similar roles in the other equation

The family resemblences of these two equations reflects a transcendent **duality** that prevails between control theory and filtering theory.

13.12.3 An example

We can use the Python class *MultivariateNormal* to construct examples.

Here is an example for a single period problem at time 0

```
G = np.array([[1., 3.]])
R = np.array([[1.]])

x0_hat = np.array([0., 1.])
Σ0 = np.array([[1., .5], [.3, 2.]])

μ = np.hstack([x0_hat, G @ x0_hat])
Σ = np.block([[Σ0, Σ0 @ G.T], [G @ Σ0, G @ Σ0 @ G.T + R]])
```

```
# construction of the multivariate normal instance
multi_normal = MultivariateNormal(μ, Σ)
```

```
multi_normal.partition(2)
```

```
# the observation of y
y0 = 2.3

# conditional distribution of x0
μ1_hat, Σ11 = multi_normal.cond_dist(0, y0)
μ1_hat, Σ11
```

```
(array([-0.078125,  0.803125]),
 array([[ 0.72098214, -0.203125  ],
        [-0.403125  ,  0.228125  ]]))
```

```
A = np.array([[0.5, 0.2], [-0.1, 0.3]])
C = np.array([[2.], [1.]])

# conditional distribution of x1
x1_cond = A @ μ1_hat
```

(continues on next page)

(continued from previous page)

```
Σ1_cond = C @ C.T + A @ Σ11 @ A.T
x1_cond, Σ1_cond
```

```
(array([0.1215625, 0.24875  ]),
 array([[4.12874554, 1.95523214],
        [1.92123214, 1.04592857]]))
```

13.12.4 Code for Iterating

Here is code for solving a dynamic filtering problem by iterating on our equations, followed by an example.

```
def iterate(x0_hat, Σ0, A, C, G, R, y_seq):

    p, n = G.shape

    T = len(y_seq)
    x_hat_seq = np.empty((T+1, n))
    Σ_hat_seq = np.empty((T+1, n, n))

    x_hat_seq[0] = x0_hat
    Σ_hat_seq[0] = Σ0

    for t in range(T):
        xt_hat = x_hat_seq[t]
        Σt = Σ_hat_seq[t]
        μ = np.hstack([xt_hat, G @ xt_hat])
        Σ = np.block([[Σt, Σt @ G.T], [G @ Σt, G @ Σt @ G.T + R]])

        # filtering
        multi_normal = MultivariateNormal(μ, Σ)
        multi_normal.partition(n)
        x_tilde, Σ_tilde = multi_normal.cond_dist(0, y_seq[t])

        # forecasting
        x_hat_seq[t+1] = A @ x_tilde
        Σ_hat_seq[t+1] = C @ C.T + A @ Σ_tilde @ A.T

    return x_hat_seq, Σ_hat_seq
```

```
iterate(x0_hat, Σ0, A, C, G, R, [2.3, 1.2, 3.2])
```

```
(array([[0.        , 1.        ],
        [0.1215625 , 0.24875   ],
        [0.18680212, 0.06904689],
        [0.75576875, 0.05558463]]),
 array([[[1.        , 0.5       ],
         [0.3       , 2.        ]],

        [[4.12874554, 1.95523214],
         [1.92123214, 1.04592857]],

        [[4.08198663, 1.99218488],
         [1.98640488, 1.00886423]],
```

(continues on next page)

(continued from previous page)

```
       [[4.06457628, 2.00041999],
        [1.99943739, 1.00275526]]]))
```

The iterative algorithm just described is a version of the celebrated **Kalman filter**.

We describe the Kalman filter and some applications of it in *A First Look at the Kalman Filter*

13.13 Classic Factor Analysis Model

The factor analysis model widely used in psychology and other fields can be represented as

$$Y = \Lambda f + U$$

where

1. Y is $n \times 1$ random vector, $EUU' = D$ is a diagonal matrix,
2. Λ is $n \times k$ coefficient matrix,
3. f is $k \times 1$ random vector, $Eff' = I$,
4. U is $n \times 1$ random vector, and $U \perp f$ (i.e., $EUf' = 0$)
5. It is presumed that k is small relative to n; often k is only 1 or 2, as in our IQ examples.

This implies that

$$\begin{aligned} \Sigma_y = EYY' &= \Lambda\Lambda' + D \\ EYf' &= \Lambda \\ EfY' &= \Lambda' \end{aligned}$$

Thus, the covariance matrix Σ_Y is the sum of a diagonal matrix D and a positive semi-definite matrix $\Lambda\Lambda'$ of rank k.

This means that all covariances among the n components of the Y vector are intermediated by their common dependencies on the $k <$ factors.

Form

$$Z = \begin{pmatrix} f \\ Y \end{pmatrix}$$

the covariance matrix of the expanded random vector Z can be computed as

$$\Sigma_z = EZZ' = \begin{pmatrix} I & \Lambda' \\ \Lambda & \Lambda\Lambda' + D \end{pmatrix}$$

In the following, we first construct the mean vector and the covariance matrix for the case where $N = 10$ and $k = 2$.

```
N = 10
k = 2
```

We set the coefficient matrix Λ and the covariance matrix of U to be

$$\Lambda = \begin{pmatrix} 1 & 0 \\ \vdots & \vdots \\ 1 & 0 \\ 0 & 1 \\ \vdots & \vdots \\ 0 & 1 \end{pmatrix}, \quad D = \begin{pmatrix} \sigma_u^2 & 0 & \cdots & 0 \\ 0 & \sigma_u^2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & \sigma_u^2 \end{pmatrix}$$

where the first half of the first column of Λ is filled with 1s and 0s for the rest half, and symmetrically for the second column.

D is a diagonal matrix with parameter σ_u^2 on the diagonal.

```
Λ = np.zeros((N, k))
Λ[:N//2, 0] = 1
Λ[N//2:, 1] = 1

σu = .5
D = np.eye(N) * σu ** 2
```

```
# compute Σy
Σy = Λ @ Λ.T + D
```

We can now construct the mean vector and the covariance matrix for Z.

```
μz = np.zeros(k+N)

Σz = np.empty((k+N, k+N))

Σz[:k, :k] = np.eye(k)
Σz[:k, k:] = Λ.T
Σz[k:, :k] = Λ
Σz[k:, k:] = Σy
```

```
z = np.random.multivariate_normal(μz, Σz)

f = z[:k]
y = z[k:]
```

```
multi_normal_factor = MultivariateNormal(μz, Σz)
multi_normal_factor.partition(k)
```

Let's compute the conditional distribution of the hidden factor f on the observations Y, namely, $f \mid Y = y$.

```
multi_normal_factor.cond_dist(0, y)
```

```
(array([0.14142014, 0.67340051]),
 array([[0.04761905, 0.        ],
        [0.        , 0.04761905]]))
```

We can verify that the conditional mean $E[f \mid Y = y] = BY$ where $B = \Lambda' \Sigma_y^{-1}$.

```
B = Λ.T @ np.linalg.inv(Σy)

B @ y
```

```
array([0.14142014, 0.67340051])
```

Similarly, we can compute the conditional distribution $Y \mid f$.

```
multi_normal_factor.cond_dist(1, f)
```

```
(array([0.17252565, 0.17252565, 0.17252565, 0.17252565, 0.17252565,
        0.77007632, 0.77007632, 0.77007632, 0.77007632, 0.77007632]),
```

(continues on next page)

(continued from previous page)

```
array([[0.25, 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 0.25, 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 0.  , 0.25, 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 0.  , 0.  , 0.25, 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 0.  , 0.  , 0.  , 0.25, 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 0.  , 0.  , 0.  , 0.  , 0.25, 0.  , 0.  , 0.  , 0.  ],
       [0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.25, 0.  , 0.  , 0.  ],
       [0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.25, 0.  , 0.  ],
       [0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.25, 0.  ],
       [0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.25]]))
```

It can be verified that the mean is $\Lambda I^{-1} f = \Lambda f$.

```
Λ @ f
```

```
array([0.17252565, 0.17252565, 0.17252565, 0.17252565, 0.17252565,
       0.77007632, 0.77007632, 0.77007632, 0.77007632, 0.77007632])
```

13.14 PCA and Factor Analysis

To learn about Principal Components Analysis (PCA), please see this lecture *Singular Value Decompositions*.

For fun, let's apply a PCA decomposition to a covariance matrix Σ_y that in fact is governed by our factor-analytic model.

Technically, this means that the PCA model is misspecified. (Can you explain why?)

Nevertheless, this exercise will let us study how well the first two principal components from a PCA can approximate the conditional expectations $Ef_i|Y$ for our two factors f_i, $i = 1, 2$ for the factor analytic model that we have assumed truly governs the data on Y we have generated.

So we compute the PCA decomposition

$$\Sigma_y = P\tilde{\Lambda}P'$$

where $\tilde{\Lambda}$ is a diagonal matrix.

We have

$$Y = P\epsilon$$

and

$$\epsilon = P'Y$$

Note that we will arrange the eigenvectors in P in the *descending* order of eigenvalues.

```
λ_tilde, P = np.linalg.eigh(Σy)

# arrange the eigenvectors by eigenvalues
ind = sorted(range(N), key=lambda x: λ_tilde[x], reverse=True)

P = P[:, ind]
λ_tilde = λ_tilde[ind]
Λ_tilde = np.diag(λ_tilde)

print('λ_tilde =', λ_tilde)
```

```
Λ_tilde = [5.25 5.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25]
```

```
# verify the orthogonality of eigenvectors
np.abs(P @ P.T - np.eye(N)).max()
```

```
np.float64(4.440892098500626e-16)
```

```
# verify the eigenvalue decomposition is correct
P @ Λ_tilde @ P.T
```

```
array([[1.25, 1.  , 1.  , 1.  , 1.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [1.  , 1.25, 1.  , 1.  , 1.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [1.  , 1.  , 1.25, 1.  , 1.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [1.  , 1.  , 1.  , 1.25, 1.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [1.  , 1.  , 1.  , 1.  , 1.25, 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 0.  , 0.  , 0.  , 0.  , 1.25, 1.  , 1.  , 1.  , 1.  ],
       [0.  , 0.  , 0.  , 0.  , 0.  , 1.  , 1.25, 1.  , 1.  , 1.  ],
       [0.  , 0.  , 0.  , 0.  , 0.  , 1.  , 1.  , 1.25, 1.  , 1.  ],
       [0.  , 0.  , 0.  , 0.  , 0.  , 1.  , 1.  , 1.  , 1.25, 1.  ],
       [0.  , 0.  , 0.  , 0.  , 0.  , 1.  , 1.  , 1.  , 1.  , 1.25]])
```

```
ε = P.T @ y

print("ε = ", ε)
```

```
ε =  [ 0.3320363   1.58105778 -0.06875162  0.11773158 -0.25902923  0.29371387
 -0.12879333  0.20139587  0.54589253  0.98391678]
```

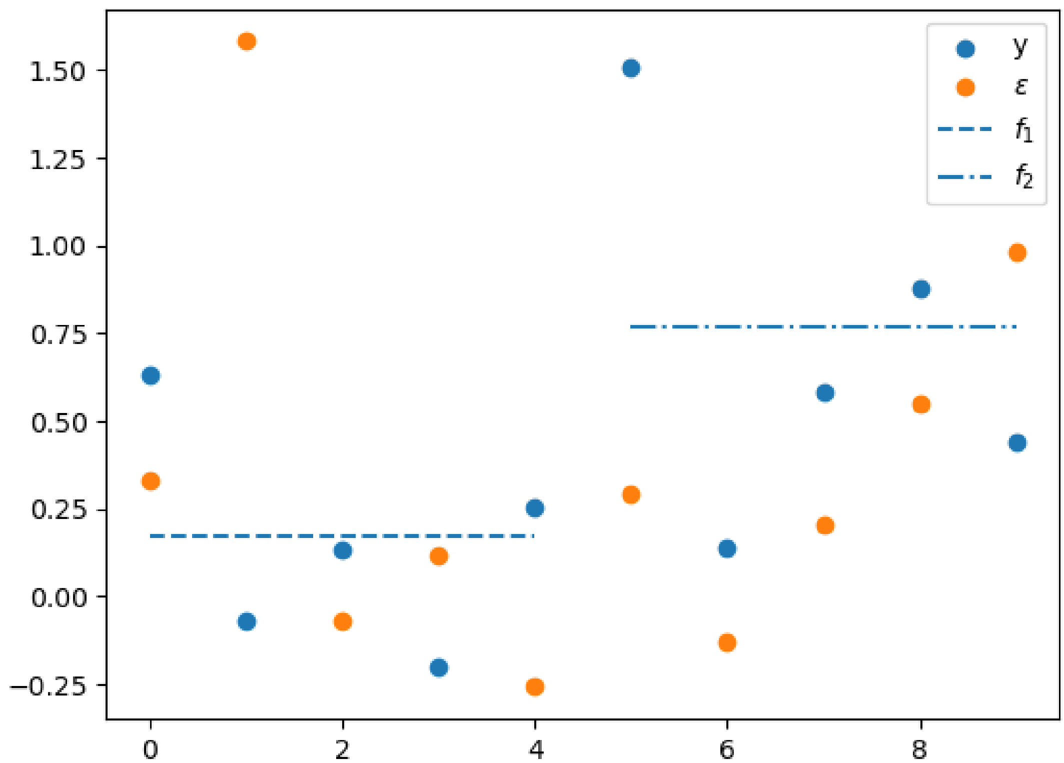
```
# print the values of the two factors

print('f = ', f)
```

```
f =  [0.17252565 0.77007632]
```

Below we'll plot several things

- the N values of y
- the N values of the principal components ϵ
- the value of the first factor f_1 plotted only for the first $N/2$ observations of y for which it receives a non-zero loading in Λ
- the value of the second factor f_2 plotted only for the final $N/2$ observations for which it receives a non-zero loading in Λ

```
plt.scatter(range(N), y, label='y')
plt.scatter(range(N), ε, label=r'$\epsilon$')
plt.hlines(f[0], 0, N//2-1, ls='--', label='$f_{1}$')
plt.hlines(f[1], N//2, N-1, ls='-.', label='$f_{2}$')
plt.legend()

plt.show()
```



Consequently, the first two ϵ_j correspond to the largest two eigenvalues.

Let's look at them, after which we'll look at $Ef|y = By$

```
ε[:2]
```

```
array([0.3320363 , 1.58105778])
```

```
# compare with Ef|y
B @ y
```

```
array([0.14142014, 0.67340051])
```

The fraction of variance in y_t explained by the first two principal components can be computed as below.

```
λ_tilde[:2].sum() / λ_tilde.sum()
```

```
np.float64(0.84)
```

Compute

$$\hat{Y} = P_j \epsilon_j + P_k \epsilon_k$$

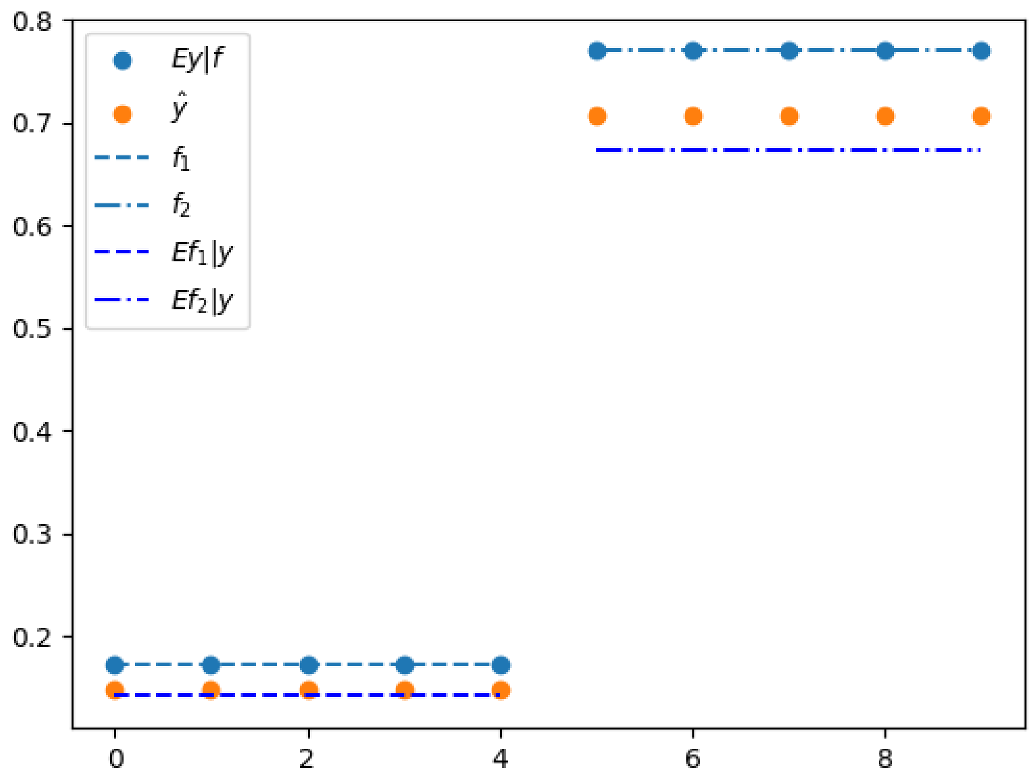
where P_j and P_k correspond to the largest two eigenvalues.

```
y_hat = P[:, :2] @ ε[:2]
```

In this example, it turns out that the projection $\hat{Y}$ of Y on the first two principal components does a good job of approximating $Ef \mid y$.

We confirm this in the following plot of f, $Ey \mid f$, $Ef \mid y$, and $\hat{y}$ on the coordinate axis versus y on the ordinate axis.

```
plt.scatter(range(N), Λ @ f, label='$Ey|f$')
plt.scatter(range(N), y_hat, label=r'$\hat{y}$')
plt.hlines(f[0], 0, N//2-1, ls='--', label='$f_{1}$')
plt.hlines(f[1], N//2, N-1, ls='-.', label='$f_{2}$')

Efy = B @ y
plt.hlines(Efy[0], 0, N//2-1, ls='--', color='b', label='$Ef_{1}|y$')
plt.hlines(Efy[1], N//2, N-1, ls='-.', color='b', label='$Ef_{2}|y$')
plt.legend()

plt.show()
```



The covariance matrix of $\hat{Y}$ can be computed by first constructing the covariance matrix of ϵ and then use the upper left block for ϵ_1 and ϵ_2.

```
Σεjk = (P.T @ Σy @ P)[:2, :2]

Pjk = P[:, :2]

Σy_hat = Pjk @ Σεjk @ Pjk.T
print('Σy_hat = \n', Σy_hat)
```

```
Σy_hat =
 [[1.05 1.05 1.05 1.05 1.05 0.   0.   0.   0.   0.  ]
 [1.05 1.05 1.05 1.05 1.05 0.   0.   0.   0.   0.  ]
 [1.05 1.05 1.05 1.05 1.05 0.   0.   0.   0.   0.  ]
 [1.05 1.05 1.05 1.05 1.05 0.   0.   0.   0.   0.  ]
 [1.05 1.05 1.05 1.05 1.05 0.   0.   0.   0.   0.  ]
 [0.   0.   0.   0.   0.   1.05 1.05 1.05 1.05 1.05]
```

(continues on next page)

(continued from previous page)

```
 [0.   0.   0.   0.   0.   1.05 1.05 1.05 1.05 1.05]
 [0.   0.   0.   0.   0.   1.05 1.05 1.05 1.05 1.05]
 [0.   0.   0.   0.   0.   1.05 1.05 1.05 1.05 1.05]
 [0.   0.   0.   0.   0.   1.05 1.05 1.05 1.05 1.05]]
```

CHAPTER

FOURTEEN

FAULT TREE UNCERTAINTIES

14.1 Overview

This lecture puts elementary tools to work to approximate probability distributions of the annual failure rates of a system consisting of a number of critical parts.

We'll use log normal distributions to approximate probability distributions of critical component parts.

To approximate the probability distribution of the **sum** of n log normal probability distributions that describes the failure rate of the entire system, we'll compute the convolution of those n log normal probability distributions.

We'll use the following concepts and tools:

- log normal distributions
- the convolution theorem that describes the probability distribution of the sum independent random variables
- fault tree analysis for approximating a failure rate of a multi-component system
- a hierarchical probability model for describing uncertain probabilities
- Fourier transforms and inverse Fourier tranforms as efficient ways of computing convolutions of sequences

For more about Fourier transforms see this quantecon lecture Circulant Matrices as well as these lecture Covariance Stationary Processes and Estimation of Spectra.

El-Shanawany, Ardron, and Walker [El-Shanawany *et al.*, 2018] and Greenfield and Sargent [Greenfield and Sargent, 1993] used some of the methods described here to approximate probabilities of failures of safety systems in nuclear facilities.

These methods respond to some of the recommendations made by Apostolakis [Apostolakis, 1990] for constructing procedures for quantifying uncertainty about the reliability of a safety system.

We'll start by bringing in some Python machinery.

```
!pip install tabulate
```

```
Requirement already satisfied: tabulate in /home/runner/miniconda3/envs/quantecon/
↪lib/python3.13/site-packages (0.9.0)
```

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import fftconvolve
from tabulate import tabulate
import time
```

```
np.set_printoptions(precision=3, suppress=True)
```

14.2 Log normal distribution

If a random variable x follows a normal distribution with mean μ and variance σ^2, then the natural logarithm of x, say $y = \log(x)$, follows a **log normal distribution** with parameters μ, σ^2.

Notice that we said **parameters** and not **mean and variance** μ, σ^2.

- μ and σ^2 are the mean and variance of $x = \exp(y)$
- they are **not** the mean and variance of y
- instead, the mean of y is $e^{\mu+\frac{1}{2}\sigma^2}$ and the variance of y is $(e^{\sigma^2} - 1)e^{2\mu+\sigma^2}$

A log normal random variable y is nonnegative.

The density for a log normal random variate y is

$$f(y) = \frac{1}{y\sigma\sqrt{2\pi}} \exp\left(\frac{-(\log y - \mu)^2}{2\sigma^2}\right)$$

for $y \geq 0$.

Important features of a log normal random variable are

$$\begin{aligned}
\text{mean:} & \quad e^{\mu+\frac{1}{2}\sigma^2} \\
\text{variance:} & \quad (e^{\sigma^2} - 1)e^{2\mu+\sigma^2} \\
\text{median:} & \quad e^{\mu} \\
\text{mode:} & \quad e^{\mu-\sigma^2} \\
\text{.95 quantile:} & \quad e^{\mu+1.645\sigma} \\
\text{.95-.05 quantile ratio:} & \quad e^{1.645\sigma}
\end{aligned}$$

Recall the following *stability* property of two independent normally distributed random variables:

If x_1 is normal with mean μ_1 and variance σ_1^2 and x_2 is independent of x_1 and normal with mean μ_2 and variance σ_2^2, then $x_1 + x_2$ is normally distributed with mean $\mu_1 + \mu_2$ and variance $\sigma_1^2 + \sigma_2^2$.

Independent log normal distributions have a different *stability* property.

The **product** of independent log normal random variables is also log normal.

In particular, if y_1 is log normal with parameters (μ_1, σ_1^2) and y_2 is log normal with parameters (μ_2, σ_2^2), then the product $y_1 y_2$ is log normal with parameters $(\mu_1 + \mu_2, \sigma_1^2 + \sigma_2^2)$.

> **Note**
>
> While the product of two log normal distributions is log normal, the **sum** of two log normal distributions is **not** log normal.

This observation sets the stage for challenge that confronts us in this lecture, namely, to approximate probability distributions of **sums** of independent log normal random variables.

To compute the probability distribution of the sum of two log normal distributions, we can use the following convolution property of a probability distribution that is a sum of independent random variables.

14.3 The Convolution Property

Let x be a random variable with probability density $f(x)$, where $x \in \mathbf{R}$.

Let y be a random variable with probability density $g(y)$, where $y \in \mathbf{R}$.

Let x and y be independent random variables and let $z = x + y \in \mathbf{R}$.

Then the probability distribution of z is

$$h(z) = (f * g)(z) \equiv \int_{-\infty}^{\infty} f(z)g(z - \tau)d\tau$$

where $(f * g)$ denotes the **convolution** of the two functions f and g.

If the random variables are both nonnegative, then the above formula specializes to

$$h(z) = (f * g)(z) \equiv \int_{0}^{\infty} f(z)g(z - \tau)d\tau$$

Below, we'll use a discretized version of the preceding formula.

In particular, we'll replace both f and g with discretized counterparts, normalized to sum to 1 so that they are probability distributions.

- by **discretized** we mean an equally spaced sampled version

Then we'll use the following version of the above formula

$$h_n = (f * g)_n = \sum_{m=0}^{\infty} f_m g_{n-m}, n \geq 0$$

to compute a discretized version of the probability distribution of the sum of two random variables, one with probability mass function f, the other with probability mass function g.

Before applying the convolution property to sums of log normal distributions, let's practice on some simple discrete distributions.

To take one example, let's consider the following two probability distributions

$$f_j = \text{Prob}(X = j), j = 0, 1$$

and

$$g_j = \text{Prob}(Y = j), j = 0, 1, 2, 3$$

and

$$h_j = \text{Prob}(Z \equiv X + Y = j), j = 0, 1, 2, 3, 4$$

The convolution property tells us that

$$h = f * g = g * f$$

Let's compute an example using the `numpy.convolve` and `scipy.signal.fftconvolve`.

```
f = [.75, .25]
g = [0., .6,  0., .4]
h = np.convolve(f,g)
hf = fftconvolve(f,g)

print("f = ", f,  ", np.sum(f) = ", np.sum(f))
print("g = ", g, ", np.sum(g) = ", np.sum(g))
print("h = ", h, ", np.sum(h) = ", np.sum(h))
print("hf = ", hf, ",np.sum(hf) = ", np.sum(hf))
```

```
f =  [0.75, 0.25] , np.sum(f) =  1.0
g =  [0.0, 0.6, 0.0, 0.4] , np.sum(g) =  1.0
h =  [0.   0.45 0.15 0.3  0.1 ] , np.sum(h) =  1.0
hf =  [0.   0.45 0.15 0.3  0.1 ] ,np.sum(hf) =  1.0000000000000002
```

A little later we'll explain some advantages that come from using `scipy.signal.ftconvolve` rather than `numpy.convolve`.numpy program convolve.

They provide the same answers but `scipy.signal.ftconvolve` is much faster.

That's why we rely on it later in this lecture.

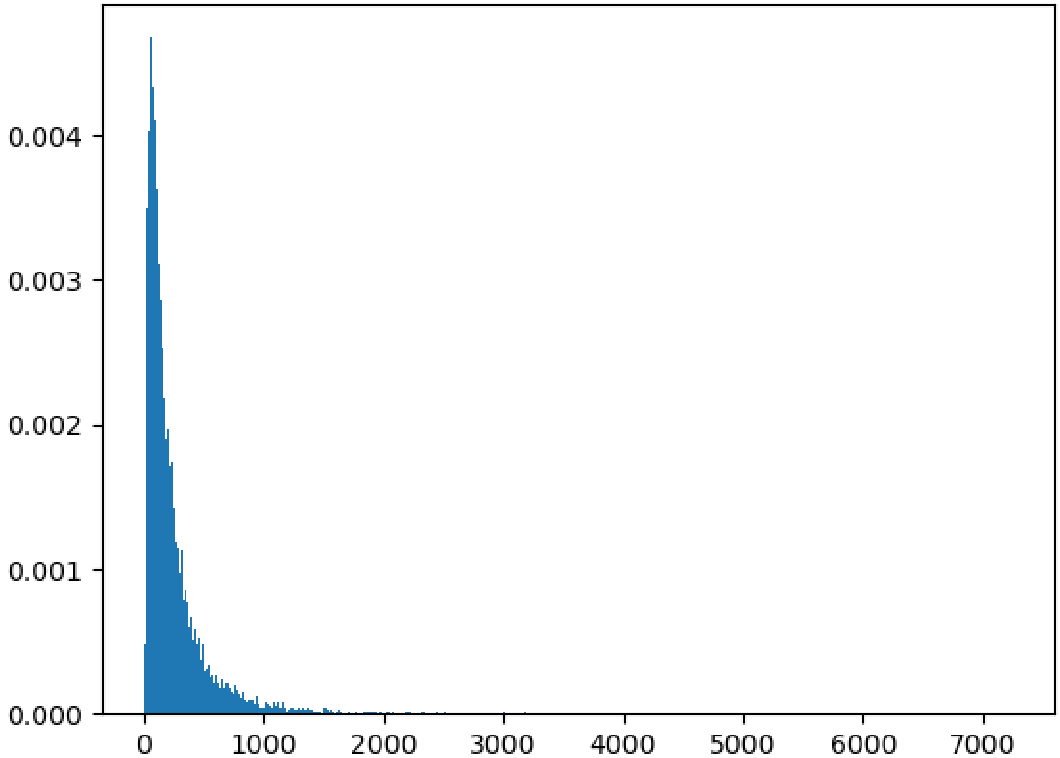
14.4 Approximating Distributions

We'll construct an example to verify that discretized distributions can do a good job of approximating samples drawn from underlying continuous distributions.

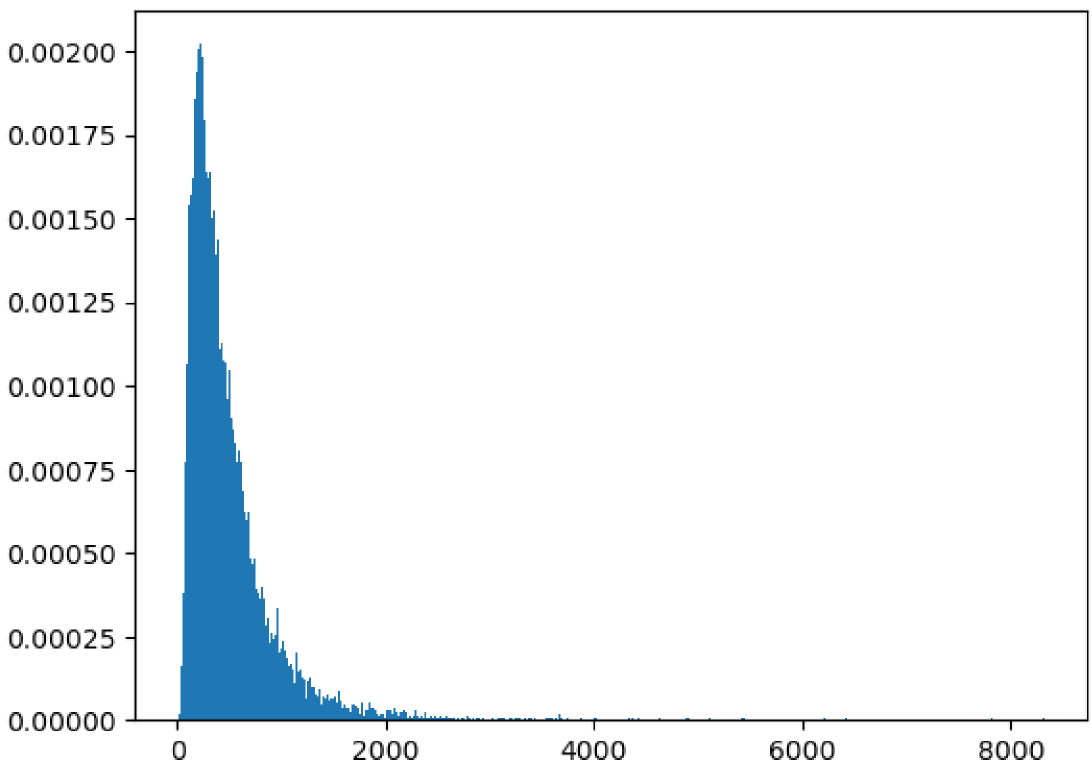
We'll start by generating samples of size 25000 of three independent log normal random variates as well as pairwise and triple-wise sums.

Then we'll plot histograms and compare them with convolutions of appropriate discretized log normal distributions.

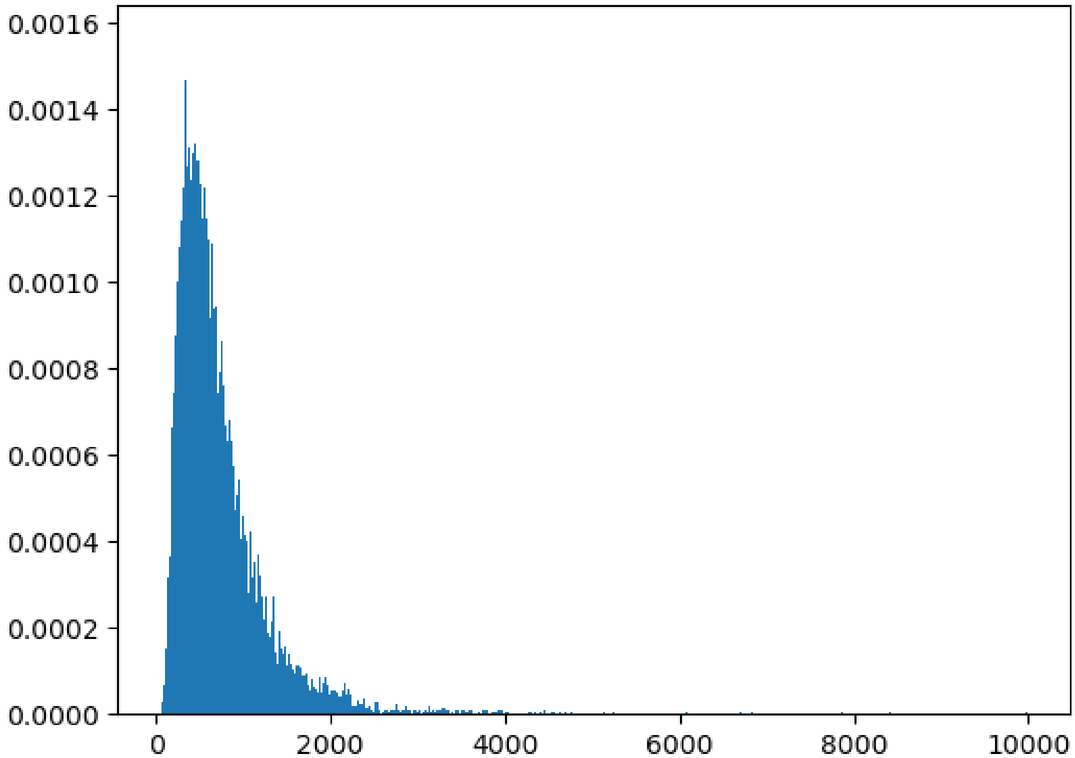
```
## create sums of two and three log normal random variates ssum2 = s1 + s2 and ssum3␣
↪= s1 + s2 + s3


mu1, sigma1 = 5., 1. # mean and standard deviation
s1 = np.random.lognormal(mu1, sigma1, 25000)

mu2, sigma2 = 5., 1. # mean and standard deviation
s2 = np.random.lognormal(mu2, sigma2, 25000)

mu3, sigma3 = 5., 1. # mean and standard deviation
s3 = np.random.lognormal(mu3, sigma3, 25000)

ssum2 = s1 + s2

ssum3 = s1 + s2 + s3

count, bins, ignored = plt.hist(s1, 1000, density=True, align='mid')
```




```
count, bins, ignored = plt.hist(ssum2, 1000, density=True, align='mid')
```



```
count, bins, ignored = plt.hist(ssum3, 1000, density=True, align='mid')
```




```
samp_mean2 = np.mean(s2)
pop_mean2 = np.exp(mu2+ (sigma2**2)/2)

pop_mean2, samp_mean2, mu2, sigma2
```

```
(np.float64(244.69193226422038), np.float64(246.6780743337006), 5.0, 1.0)
```

Here are helper functions that create a discretized version of a log normal probability density function.

```
def p_log_normal(x,μ,σ):
    p = 1 / (σ*x*np.sqrt(2*np.pi)) * np.exp(-1/2*((np.log(x) - μ)/σ)**2)
    return p

def pdf_seq(μ,σ,I,m):
    x = np.arange(1e-7,I,m)
    p_array = p_log_normal(x,μ,σ)
    p_array_norm = p_array/np.sum(p_array)
    return p_array,p_array_norm,x
```

Now we shall set a grid length I and a grid increment size $m = 1$ for our discretizations.

Note

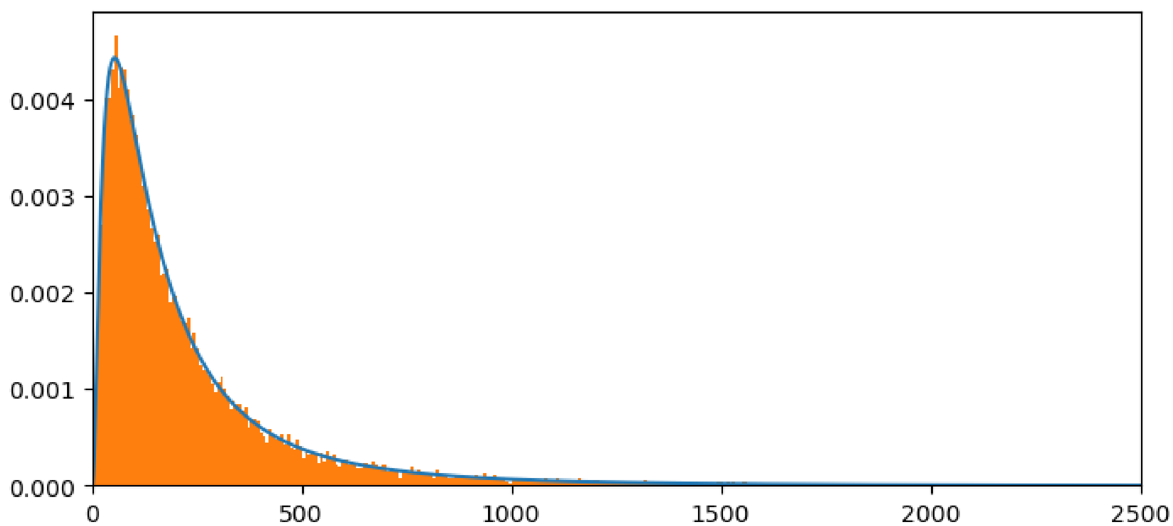
We set I equal to a power of two because we want to be free to use a Fast Fourier Transform to compute a convolution of two sequences (discrete distributions).

We recommend experimenting with different values of the power p of 2.

Setting it to 15 rather than 12, for example, improves how well the discretized probability mass function approximates the original continuous probability density function being studied.

```
p=15
I = 2**p # Truncation value
m = .1 # increment size
```

```
## Cell to check -- note what happens when don't normalize!
## things match up without adjustment. Compare with above

p1,p1_norm,x = pdf_seq(mu1,sigma1,I,m)
## compute number of points to evaluate the probability mass function
NT = x.size

plt.figure(figsize = (8,8))
plt.subplot(2,1,1)
plt.plot(x[:int(NT)],p1[:int(NT)],label = '')
plt.xlim(0,2500)
count, bins, ignored = plt.hist(s1, 1000, density=True, align='mid')

plt.show()
```




```
# Compute mean from discretized pdf and compare with the theoretical value

mean= np.sum(np.multiply(x[:NT],p1_norm[:NT]))
meantheory = np.exp(mu1+.5*sigma1**2)
mean, meantheory
```

```
(np.float64(244.69059898302908), np.float64(244.69193226422038))
```

14.5 Convolving Probability Mass Functions

Now let's use the convolution theorem to compute the probability distribution of a sum of the two log normal random variables we have parameterized above.

We'll also compute the probability of a sum of three log normal distributions constructed above.

Before we do these things, we shall explain our choice of Python algorithm to compute a convolution of two sequences.

Because the sequences that we convolve are long, we use the `scipy.signal.fftconvolve` function rather than the numpy.convove function.

These two functions give virtually equivalent answers but for long sequences `scipy.signal.fftconvolve` is much faster.

The program `scipy.signal.fftconvolve` uses fast Fourier transforms and their inverses to calculate convolutions.

Let's define the Fourier transform and the inverse Fourier transform.

The **Fourier transform** of a sequence $\{x_t\}_{t=0}^{T-1}$ is a sequence of complex numbers $\{x(\omega_j)\}_{j=0}^{T-1}$ given by

$$x(\omega_j) = \sum_{t=0}^{T-1} x_t \exp(-i\omega_j t) \tag{14.1}$$

where $\omega_j = \frac{2\pi j}{T}$ for $j = 0, 1, \dots, T-1$.

The **inverse Fourier transform** of the sequence $\{x(\omega_j)\}_{j=0}^{T-1}$ is

$$x_t = T^{-1} \sum_{j=0}^{T-1} x(\omega_j) \exp(i\omega_j t) \tag{14.2}$$

The sequences $\{x_t\}_{t=0}^{T-1}$ and $\{x(\omega_j)\}_{j=0}^{T-1}$ contain the same information.

The pair of equations (14.1) and (14.2) tell how to recover one series from its Fourier partner.

The program `scipy.signal.fftconvolve` deploys the theorem that a convolution of two sequences $\{f_k\}, \{g_k\}$ can be computed in the following way:

- Compute Fourier transforms $F(\omega), G(\omega)$ of the $\{f_k\}$ and $\{g_k\}$ sequences, respectively
- Form the product $H(\omega) = F(\omega)G(\omega)$
- The convolution of $f * g$ is the inverse Fourier transform of $H(\omega)$

The **fast Fourier transform** and the associated **inverse fast Fourier transform** execute these calculations very quickly.

This is the algorithm that `scipy.signal.fftconvolve` uses.

Let's do a warmup calculation that compares the times taken by `numpy.convove` and `scipy.signal.fftconvolve`.

```
p1,p1_norm,x = pdf_seq(mu1,sigma1,I,m)
p2,p2_norm,x = pdf_seq(mu2,sigma2,I,m)
p3,p3_norm,x = pdf_seq(mu3,sigma3,I,m)

tic = time.perf_counter()

c1 = np.convolve(p1_norm,p2_norm)
c2 = np.convolve(c1,p3_norm)
```

(continues on next page)

(continued from previous page)

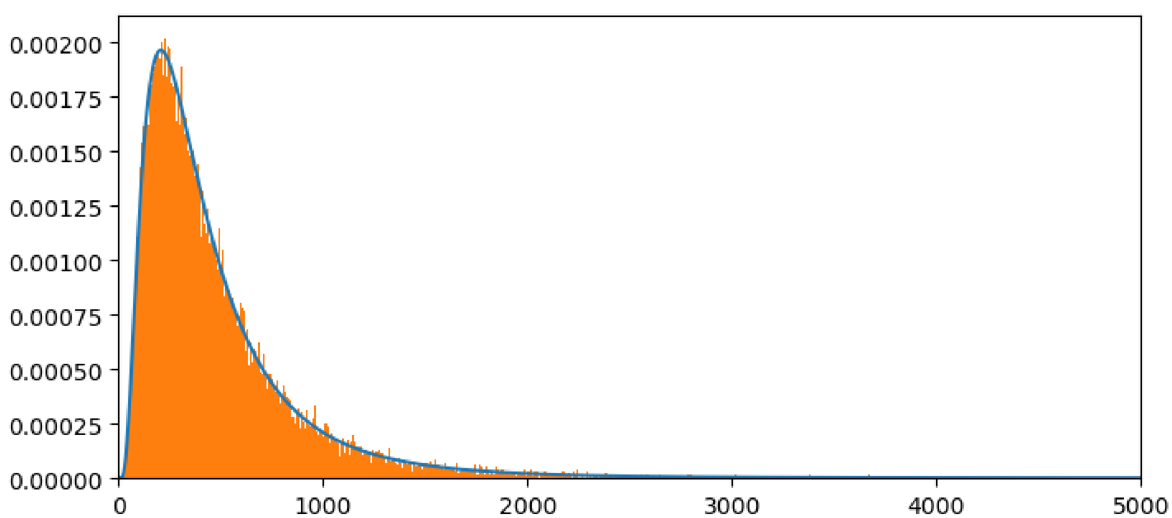
```
toc = time.perf_counter()

tdiff1 = toc - tic

tic = time.perf_counter()

c1f = fftconvolve(p1_norm,p2_norm)
c2f = fftconvolve(c1f,p3_norm)
toc = time.perf_counter()

toc = time.perf_counter()

tdiff2 = toc - tic

print("time with np.convolve = ", tdiff1,  "; time with fftconvolve = ",  tdiff2)
```

```
time with np.convolve =  33.75614000399992 ; time with fftconvolve =  0.
↪1390971159999026
```

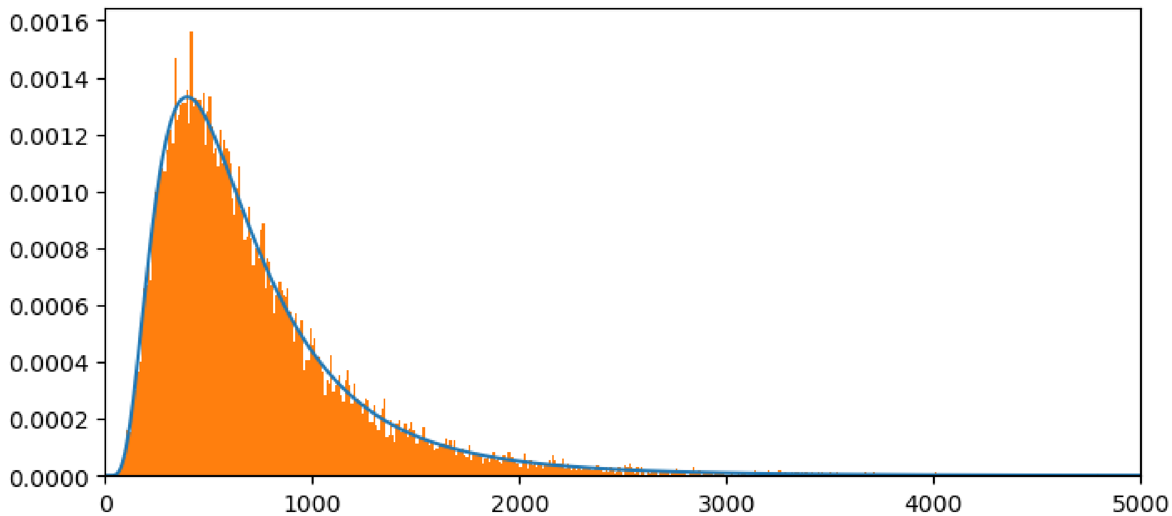
The fast Fourier transform is two orders of magnitude faster than `numpy.convolve`

Now let's plot our computed probability mass function approximation for the sum of two log normal random variables against the histogram of the sample that we formed above.

```
NT= np.size(x)

plt.figure(figsize = (8,8))
plt.subplot(2,1,1)
plt.plot(x[:int(NT)],c1f[:int(NT)]/m,label = '')
plt.xlim(0,5000)

count, bins, ignored = plt.hist(ssum2, 1000, density=True, align='mid')
# plt.plot(P2P3[:10000],label = 'FFT method',linestyle = '--')

plt.show()
```



```
NT= np.size(x)
plt.figure(figsize = (8,8))
plt.subplot(2,1,1)
plt.plot(x[:int(NT)],c2f[:int(NT)]/m,label = '')
plt.xlim(0,5000)

count, bins, ignored = plt.hist(ssum3, 1000, density=True, align='mid')
# plt.plot(P2P3[:10000],label = 'FFT method',linestyle = '--')

plt.show()
```




```
## Let's compute the mean of the discretized pdf
mean= np.sum(np.multiply(x[:NT],c1f[:NT]))
# meantheory = np.exp(mu1+.5*sigma1**2)
mean, 2*meantheory
```

```
(np.float64(489.38109740938535), np.float64(489.38386452844077))
```

```
## Let's compute the mean of the discretized pdf
mean= np.sum(np.multiply(x[:NT],c2f[:NT]))
# meantheory = np.exp(mu1+.5*sigma1**2)
mean, 3*meantheory
```

```
(np.float64(734.0714863312277), np.float64(734.0757967926611))
```

14.6 Failure Tree Analysis

We shall soon apply the convolution theorem to compute the probability of a **top event** in a failure tree analysis.

Before applying the convolution theorem, we first describe the model that connects constituent events to the **top** end whose failure rate we seek to quantify.

The model is an example of the widely used **failure tree analysis** described by El-Shanawany, Ardron, and Walker [El-Shanawany *et al.*, 2018].

To construct the statistical model, we repeatedly use what is called the **rare event approximation**.

We want to compute the probabilty of an event $A \cup B$.

- the union $A \cup B$ is the event that A OR B occurs

A law of probability tells us that A OR B occurs with probability

$$P(A \cup B) = P(A) + P(B) - P(A \cap B)$$

where the intersection $A \cap B$ is the event that A **AND** B both occur and the union $A \cup B$ is the event that A **OR** B occurs.

If A and B are independent, then

$$P(A \cap B) = P(A)P(B)$$

If $P(A)$ and $P(B)$ are both small, then $P(A)P(B)$ is even smaller.

The **rare event approximation** is

$$P(A \cup B) \approx P(A) + P(B)$$

This approximation is widely used in evaluating system failures.

14.7 Application

A system has been designed with the feature a system failure occurs when **any** of n critical components fails.

The failure probability $P(A_i)$ of each event A_i is small.

We assume that failures of the components are statistically independent random variables.

We repeatedly apply a **rare event approximation** to obtain the following formula for the problem of a system failure:

$$P(F) \approx P(A_1) + P(A_2) + \cdots + P(A_n)$$

or

$$P(F) \approx \sum_{i=1}^{n} P(A_i) \tag{14.3}$$

Probabilities for each event are recorded as failure rates per year.

14.8 Failure Rates Unknown

Now we come to the problem that really interests us, following [El-Shanawany *et al.*, 2018] and Greenfield and Sargent [Greenfield and Sargent, 1993] in the spirit of Apostolakis [Apostolakis, 1990].

The constituent probabilities or failure rates $P(A_i)$ are not known a priori and have to be estimated.

We address this problem by specifying **probabilities of probabilities** that capture one notion of not knowing the constituent probabilities that are inputs into a failure tree analysis.

Thus, we assume that a system analyst is uncertain about the failure rates $P(A_i), i = 1, \dots, n$ for components of a system.

The analyst copes with this situation by regarding the systems failure probability $P(F)$ and each of the component probabilities $P(A_i)$ as random variables.

- dispersions of the probability distribution of $P(A_i)$ characterizes the analyst's uncertainty about the failure probability $P(A_i)$
- the dispersion of the implied probability distribution of $P(F)$ characterizes his uncertainty about the probability of a system's failure.

This leads to what is sometimes called a **hierarchical** model in which the analyst has probabilities about the probabilities $P(A_i)$.

The analyst formalizes his uncertainty by assuming that

- the failure probability $P(A_i)$ is itself a log normal random variable with parameters (μ_i, σ_i).
- failure rates $P(A_i)$ and $P(A_j)$ are statistically independent for all pairs with $i \neq j$.

The analyst calibrates the parameters (μ_i, σ_i) for the failure events $i = 1, \ldots, n$ by reading reliability studies in engineering papers that have studied historical failure rates of components that are as similar as possible to the components being used in the system under study.

The analyst assumes that such information about the observed dispersion of annual failure rates, or times to failure, can inform him of what to expect about parts' performances in his system.

The analyst assumes that the random variables $P(A_i)$ are statistically mutually independent.

The analyst wants to approximate a probability mass function and cumulative distribution function of the systems failure probability $P(F)$.

- We say probability mass function because of how we discretize each random variable, as described earlier.

The analyst calculates the probability mass function for the **top event** F, i.e., a **system failure**, by repeatedly applying the convolution theorem to compute the probability distribution of a sum of independent log normal random variables, as described in equation (14.3).

14.9 Waste Hoist Failure Rate

We'll take close to a real world example by assuming that $n = 14$.

The example estimates the annual failure rate of a critical hoist at a nuclear waste facility.

A regulatory agency wants the sytem to be designed in a way that makes the failure rate of the top event small with high probability.

This example is Design Option B-2 (Case I) described in Table 10 on page 27 of [Greenfield and Sargent, 1993].

The table describes parameters μ_i, σ_i for fourteen log normal random variables that consist of **seven pairs** of random variables that are identically and independently distributed.

- Within a pair, parameters μ_i, σ_i are the same
- As described in table 10 of [Greenfield and Sargent, 1993] p. 27, parameters of log normal distributions for the seven unique probabilities $P(A_i)$ have been calibrated to be the values in the following Python code:

```
mu1, sigma1 = 4.28, 1.1947
mu2, sigma2 = 3.39, 1.1947
mu3, sigma3 = 2.795, 1.1947
mu4, sigma4 = 2.717, 1.1947
mu5, sigma5 = 2.717, 1.1947
mu6, sigma6 = 1.444, 1.4632
mu7, sigma7 = -.040, 1.4632
```

> **Note**
>
> Because the failure rates are all very small, log normal distributions with the above parameter values actually describe $P(A_i)$ times 10^{-09}.

So the probabilities that we'll put on the x axis of the probability mass function and associated cumulative distribution function should be multiplied by 10^{-09}

To extract a table that summarizes computed quantiles, we'll use a helper function

```
def find_nearest(array, value):
    array = np.asarray(array)
    idx = (np.abs(array - value)).argmin()
    return idx
```

We compute the required thirteen convolutions in the following code.

(Please feel free to try different values of the power parameter p that we use to set the number of points in our grid for constructing the probability mass functions that discretize the continuous log normal distributions.)

We'll plot a counterpart to the cumulative distribution function (CDF) in figure 5 on page 29 of [Greenfield and Sargent, 1993] and we'll also present a counterpart to their Table 11 on page 28.

```
p=15
I = 2**p # Truncation value
m =  .05 # increment size




p1,p1_norm,x = pdf_seq(mu1,sigma1,I,m)
p2,p2_norm,x = pdf_seq(mu2,sigma2,I,m)
p3,p3_norm,x = pdf_seq(mu3,sigma3,I,m)
p4,p4_norm,x = pdf_seq(mu4,sigma4,I,m)
p5,p5_norm,x = pdf_seq(mu5,sigma5,I,m)
p6,p6_norm,x = pdf_seq(mu6,sigma6,I,m)
p7,p7_norm,x = pdf_seq(mu7,sigma7,I,m)
p8,p8_norm,x = pdf_seq(mu7,sigma7,I,m)
p9,p9_norm,x = pdf_seq(mu7,sigma7,I,m)
p10,p10_norm,x = pdf_seq(mu7,sigma7,I,m)
p11,p11_norm,x = pdf_seq(mu7,sigma7,I,m)
p12,p12_norm,x = pdf_seq(mu7,sigma7,I,m)
p13,p13_norm,x = pdf_seq(mu7,sigma7,I,m)
p14,p14_norm,x = pdf_seq(mu7,sigma7,I,m)

tic = time.perf_counter()

c1 = fftconvolve(p1_norm,p2_norm)
c2 = fftconvolve(c1,p3_norm)
c3 = fftconvolve(c2,p4_norm)
c4 = fftconvolve(c3,p5_norm)
c5 = fftconvolve(c4,p6_norm)
c6 = fftconvolve(c5,p7_norm)
c7 = fftconvolve(c6,p8_norm)
c8 = fftconvolve(c7,p9_norm)
c9 = fftconvolve(c8,p10_norm)
c10 = fftconvolve(c9,p11_norm)
```

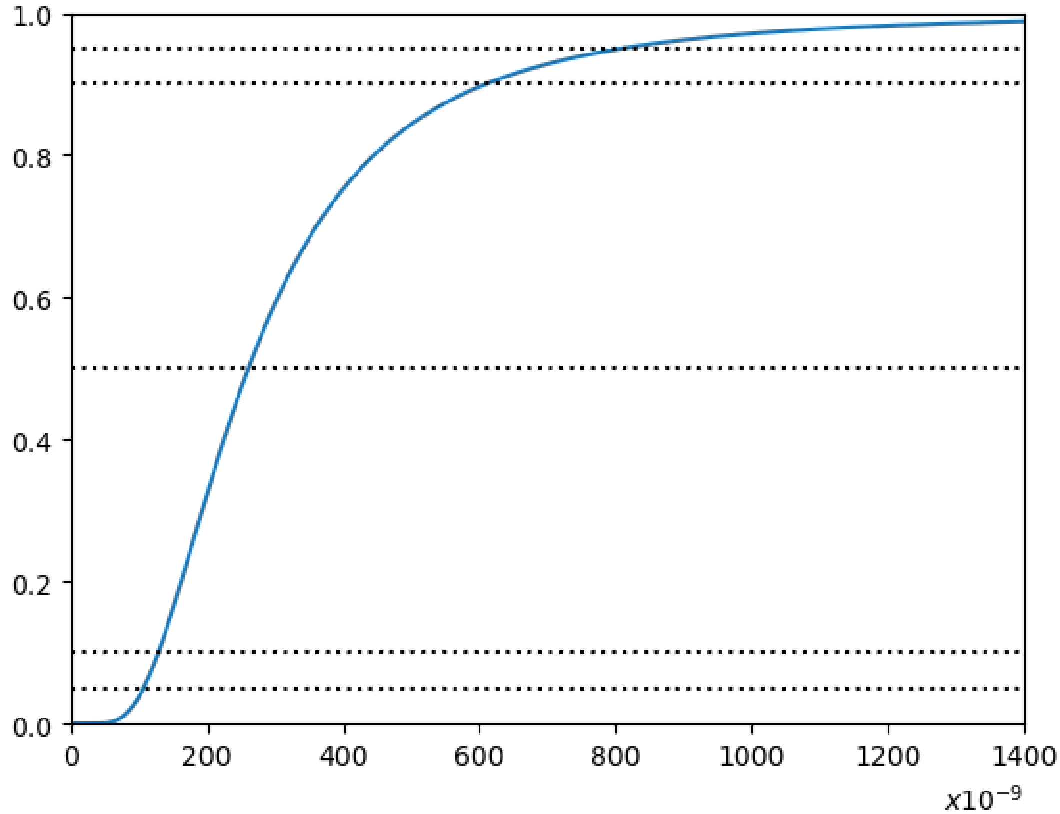
(continues on next page)

(continued from previous page)

```
c11 = fftconvolve(c10,p12_norm)
c12 = fftconvolve(c11,p13_norm)
c13 = fftconvolve(c12,p14_norm)

toc = time.perf_counter()

tdiff13 = toc - tic

print("time for 13 convolutions = ", tdiff13)
```

```
time for 13 convolutions =  7.634427766000044
```

```
d13 = np.cumsum(c13)
Nx=int(1400)
plt.figure()
plt.plot(x[0:int(Nx/m)],d13[0:int(Nx/m)])  # show Yad this -- I multiplied by m --␣
↪step size
plt.hlines(0.5,min(x),Nx,linestyles='dotted',colors = {'black'})
plt.hlines(0.9,min(x),Nx,linestyles='dotted',colors = {'black'})
plt.hlines(0.95,min(x),Nx,linestyles='dotted',colors = {'black'})
plt.hlines(0.1,min(x),Nx,linestyles='dotted',colors = {'black'})
plt.hlines(0.05,min(x),Nx,linestyles='dotted',colors = {'black'})
plt.ylim(0,1)
plt.xlim(0,Nx)
plt.xlabel("$x10^{-9}$",loc = "right")
plt.show()

x_1 = x[find_nearest(d13,0.01)]
x_5 = x[find_nearest(d13,0.05)]
x_10 = x[find_nearest(d13,0.1)]
x_50 = x[find_nearest(d13,0.50)]
x_66 = x[find_nearest(d13,0.665)]
x_85 = x[find_nearest(d13,0.85)]
x_90 = x[find_nearest(d13,0.90)]
x_95 = x[find_nearest(d13,0.95)]
x_99 = x[find_nearest(d13,0.99)]
x_9978 = x[find_nearest(d13,0.9978)]

print(tabulate([
    ['1%',f"{x_1}"],
    ['5%',f"{x_5}"],
    ['10%',f"{x_10}"],
    ['50%',f"{x_50}"],
    ['66.5%',f"{x_66}"],
    ['85%',f"{x_85}"],
    ['90%',f"{x_90}"],
    ['95%',f"{x_95}"],
    ['99%',f"{x_99}"],
    ['99.78%',f"{x_9978}"]],
    headers = ['Percentile', 'x * 1e-9']))
```




```
Percentile      x * 1e-9
------------  ----------
1%                 76.15
5%                106.5
10%               128.2
50%               260.55
66.5%             338.55
85%               509.4
90%               608.8
95%               807.6
99%              1470.2
99.78%           2474.85
```

The above table agrees closely with column 2 of Table 11 on p. 28 of of [Greenfield and Sargent, 1993].

Discrepancies are probably due to slight differences in the number of digits retained in inputting $\mu_i, \sigma_i, i = 1, \ldots, 14$ and in the number of points deployed in the discretizations.

CHAPTER

FIFTEEN

INTRODUCTION TO ARTIFICIAL NEURAL NETWORKS

> **GPU**
>
> This lecture was built using a machine with access to a GPU — although it will also run without one.
>
> Google Colab has a free tier with GPUs that you can access as follows:
>
> 1. Click on the "play" icon top right
> 2. Select Colab
> 3. Set the runtime environment to include a GPU

In addition to what's included in base Anaconda, we need to install the following packages

```
!pip install -U kaleido plotly
!conda install -y -c plotly plotly-orca

# kaleido needs chrome to build images
import kaleido
kaleido.get_chrome_sync()
```

> **Note**
>
> If you are running this on Google Colab the above cell will present an error. This is because Google Colab doesn't use Anaconda to manage the Python packages. However this lecture will still execute as Google Colab has `plotly` installed.

We also need to install JAX to run this lecture

```
!pip install --upgrade jax
```

```
import jax
print(f"JAX backend: {jax.devices()[0].platform}") # to check that gpu is activated␣
↪in environment
```

```
JAX backend: gpu
```

15.1 Overview

Substantial parts of **machine learning** and **artificial intelligence** are about

- approximating an unknown function with a known function
- estimating the known function from a set of data on the left- and right-hand variables

This lecture describes the structure of a plain vanilla **artificial neural network** (ANN) of a type that is widely used to approximate a function f that maps x in a space X into y in a space Y.

To introduce elementary concepts, we study an example in which x and y are scalars.

We'll describe the following concepts that are brick and mortar for neural networks:

- a neuron
- an activation function
- a network of neurons
- A neural network as a composition of functions
- back-propagation and its relationship to the chain rule of differential calculus

15.2 A Deep (but not Wide) Artificial Neural Network

We describe a "deep" neural network of "width" one.

Deep means that the network composes a large number of functions organized into nodes of a graph.

Width refers to the number of right hand side variables on the right hand side of the function being approximated.

Setting "width" to one means that the network composes just univariate functions.

Let $x \in \mathbb{R}$ be a scalar and $y \in \mathbb{R}$ be another scalar.

We assume that y is a nonlinear function of x:

$$y = f(x)$$

We want to approximate $f(x)$ with another function that we define recursively.

For a network of depth $N \geq 1$, each **layer** $i = 1, \ldots N$ consists of

- an input x_i
- an **affine function** $w_i x_i + bI$, where w_i is a scalar **weight** placed on the input x_i and b_i is a scalar **bias**
- an **activation function** h_i that takes $(w_i x_i + b_i)$ as an argument and produces an output x_{i+1}

An example of an activation function h is the **sigmoid** function

$$h(z) = \frac{1}{1 + e^{-z}}$$

Another popular activation function is the **rectified linear unit** (ReLU) function

$$h(z) = \max(0, z)$$

Yet another activation function is the identity function

$$h(z) = z$$

As activation functions below, we'll use the sigmoid function for layers 1 to $N-1$ and the identity function for layer N.

To approximate a function $f(x)$ we construct $\hat{f}(x)$ by proceeding as follows.

Let

$$l_i(x) = w_i x + b_i.$$

We construct $\hat{f}$ by iterating on compositions of functions $h_i \circ l_i$:

$$f(x) \approx \hat{f}(x) = h_N \circ l_N \circ h_{N-1} \circ l_1 \circ \cdots \circ h_1 \circ l_1(x)$$

If $N > 1$, we call the right side a "deep" neural net.

The larger is the integer N, the "deeper" is the neural net.

Evidently, if we know the parameters $\{w_i, b_i\}_{i=1}^N$, then we can compute $\hat{f}(x)$ for a given $x = \tilde{x}$ by iterating on the recursion

$$x_{i+1} = h_i \circ l_i(x_i), \quad , i = 1, \dots N \tag{15.1}$$

starting from $x_1 = \tilde{x}$.

The value of x_{N+1} that emerges from this iterative scheme equals $\hat{f}(\tilde{x})$.

15.3 Calibrating Parameters

We now consider a neural network like the one describe above with width 1, depth N, and activation functions h_i for $1 \leqslant i \leqslant N$ that map $\mathbb{R}$ into itself.

Let $\{(w_i, b_i)\}_{i=1}^N$ denote a sequence of weights and biases.

As mentioned above, for a given input x_1, our approximating function $\hat{f}$ evaluated at x_1 equals the "output" x_{N+1} from our network that can be computed by iterating on $x_{i+1} = h_i(w_i x_i + b_i)$.

For a given **prediction** $\hat{y}(x)$ and **target** $y = f(x)$, consider the loss function

$$\mathcal{L}(\hat{y}, y)(x) = \frac{1}{2}(\hat{y} - y)^2(x).$$

This criterion is a function of the parameters $\{(w_i, b_i)\}_{i=1}^N$ and the point x.

We're interested in solving the following problem:

$$\min_{\{(w_i, b_i)\}_{i=1}^N} \int \mathcal{L}(x_{N+1}, y)(x) d\mu(x)$$

where $\mu(x)$ is some measure of points $x \in \mathbb{R}$ over which we want a good approximation $\hat{f}(x)$ to $f(x)$.

Stack weights and biases into a vector of parameters p:

$$p = \begin{bmatrix} w_1 \\ b_1 \\ w_2 \\ b_2 \\ \vdots \\ w_N \\ b_N \end{bmatrix}$$

Applying a "poor man's version" of a **stochastic gradient descent** algorithm for finding a zero of a function leads to the following update rule for parameters:

$$p_{k+1} = p_k - \alpha \frac{d\mathcal{L}}{dx_{N+1}} \frac{dx_{N+1}}{dp_k} \tag{15.2}$$

where $\frac{d\mathcal{L}}{dx_{N+1}} = -\left(x_{N+1} - y\right)$ and $\alpha > 0$ is a step size.

(See this and this to gather insights about how stochastic gradient descent relates to Newton's method.)

To implement one step of this parameter update rule, we want the vector of derivatives $\frac{dx_{N+1}}{dp_k}$.

In the neural network literature, this step is accomplished by what is known as **back propagation**.

15.4 Back Propagation and the Chain Rule

Thanks to properties of

- the chain and product rules for differentiation from differential calculus, and
- lower triangular matrices

back propagation can actually be accomplished in one step by

- inverting a lower triangular matrix, and
- matrix multiplication

(This idea is from the last 7 minutes of this great youtube video by MIT's Alan Edelman)

https://youtu.be/rZS2LGiurKY

Here goes.

Define the derivative of $h(z)$ with respect to z evaluated at $z = z_i$ as δ_i:

$$\delta_i = \frac{d}{dz} h(z)|_{z=z_i}$$

or

$$\delta_i = h'\left(w_i x_i + b_i\right).$$

Repeated application of the chain rule and product rule to our recursion (15.1) allows us to obtain:

$$dx_{i+1} = \delta_i \left(dw_i x_i + w_i dx_i + b_i\right)$$

After imposing $dx_1 = 0$, we get the following system of equations:

$$\left(\begin{array}{c} dx_2 \\ \vdots \\ dx_{N+1} \end{array}\right) = \underbrace{\left(\begin{array}{ccccc} \delta_1 w_1 & \delta_1 & 0 & 0 & 0 \\ 0 & 0 & \ddots & 0 & 0 \\ 0 & 0 & 0 & \delta_N w_N & \delta_N \end{array}\right)}_{D} \left(\begin{array}{c} dw_1 \\ db_1 \\ \vdots \\ dw_N \\ db_N \end{array}\right) + \underbrace{\left(\begin{array}{cccc} 0 & 0 & 0 & 0 \\ w_2 & 0 & 0 & 0 \\ 0 & \ddots & 0 & 0 \\ 0 & 0 & w_N & 0 \end{array}\right)}_{L} \left(\begin{array}{c} dx_2 \\ \vdots \\ dx_{N+1} \end{array}\right)$$

or

$$dx = Ddp + Ldx$$

which implies that

$$dx = (I - L)^{-1} D dp$$

which in turn implies

$$\begin{pmatrix} dx_{N+1}/dw_1 \\ dx_{N+1}/db_1 \\ \vdots \\ dx_{N+1}/dw_N \\ dx_{N+1}/db_N \end{pmatrix} = e_N (I - L)^{-1} D.$$

We can then solve the above problem by applying our update for p multiple times for a collection of input-output pairs $\{(x_1^i, y^i)\}_{i=1}^M$ that we'll call our "training set".

15.5 Training Set

Choosing a training set amounts to a choice of measure μ in the above formulation of our function approximation problem as a minimization problem.

In this spirit, we shall use a uniform grid of, say, 50 or 200 points.

There are many possible approaches to the minimization problem posed above:

- batch gradient descent in which you use an average gradient over the training set
- stochastic gradient descent in which you sample points randomly and use individual gradients
- something in-between (so-called "mini-batch gradient descent")

The update rule (15.2) described above amounts to a stochastic gradient descent algorithm.

```
from IPython.display import Image
import jax.numpy as jnp
from jax import grad, jit, jacfwd, vmap
from jax import random
import jax
import plotly.graph_objects as go
```

```
# A helper function to randomly initialize weights and biases
# for a dense neural network layer
def random_layer_params(m, n, key, scale=1.):
    w_key, b_key = random.split(key)
    return scale * random.normal(w_key, (n, m)), scale * random.normal(b_key, (n,))

# Initialize all layers for a fully-connected neural network with sizes "sizes"
def init_network_params(sizes, key):
    keys = random.split(key, len(sizes))
    return [random_layer_params(m, n, k) for m, n, k in zip(sizes[:-1], sizes[1:],
↪keys)]
```

```
def compute_xδw_seq(params, x):
    # Initialize arrays
    δ = jnp.zeros(len(params))
    xs = jnp.zeros(len(params) + 1)
    ws = jnp.zeros(len(params))
```

(continues on next page)

(continued from previous page)

```
    bs = jnp.zeros(len(params))

    h = jax.nn.sigmoid

    xs = xs.at[0].set(x)
    for i, (w, b) in enumerate(params[:-1]):
        output = w * xs[i] + b
        activation = h(output[0, 0])

        # Store elements
        δ = δ.at[i].set(grad(h)(output[0, 0]))
        ws = ws.at[i].set(w[0, 0])
        bs = bs.at[i].set(b[0])
        xs = xs.at[i+1].set(activation)

    final_w, final_b = params[-1]
    preds = final_w * xs[-2] + final_b

    # Store elements
    δ = δ.at[-1].set(1.)
    ws = ws.at[-1].set(final_w[0, 0])
    bs = bs.at[-1].set(final_b[0])
    xs = xs.at[-1].set(preds[0, 0])

    return xs, δ, ws, bs


def loss(params, x, y):
    xs, δ, ws, bs = compute_xδw_seq(params, x)
    preds = xs[-1]

    return 1 / 2 * (y - preds) ** 2
```

```
# Parameters
N = 3  # Number of layers
layer_sizes = [1, ] * (N + 1)
param_scale = 0.1
step_size = 0.01
params = init_network_params(layer_sizes, random.PRNGKey(1))
```

```
x = 5
y = 3
xs, δ, ws, bs = compute_xδw_seq(params, x)
```

```
dxs_ad = jacfwd(lambda params, x: compute_xδw_seq(params, x)[0], argnums=0)(params, x)
dxs_ad_mat = jnp.block([dx.reshape((-1, 1)) for dx_tuple in dxs_ad for dx in dx_tuple␣
↪])[1:]
```

```
jnp.block([[δ * xs[:-1]], [δ]])
```

```
Array([[1.0165801 , 0.06087969, 0.09382247],
       [0.20331602, 0.08501981, 1.        ]], dtype=float32)
```

```
L = jnp.diag(δ * ws, k=-1)
```

(continues on next page)

(continued from previous page)

```
L = L[1:, 1:]

D = jax.scipy.linalg.block_diag(*[row.reshape((1, 2)) for row in jnp.block([[δ * xs[:-
↪1]], [δ]]).T])

dxs_la = jax.scipy.linalg.solve_triangular(jnp.eye(N) - L, D, lower=True)
```

```
# Check that the `dx` generated by the linear algebra method
# are the same as the ones generated using automatic differentiation
jnp.max(jnp.abs(dxs_ad_mat - dxs_la))
```

```
Array(0., dtype=float32)
```

```
grad_loss_ad = jnp.block([dx.reshape((-1, 1)) for dx_tuple in grad(loss)(params, x,␣
↪y) for dx in dx_tuple ])
```

```
# Check that the gradient of the loss is the same for both approaches
jnp.max(jnp.abs(-(y - xs[-1]) * dxs_la[-1] - grad_loss_ad))
```

```
Array(5.9604645e-08, dtype=float32)
```

```
@jit
def update_ad(params, x, y):
    grads = grad(loss)(params, x, y)
    return [(w - step_size * dw, b - step_size * db)
          for (w, b), (dw, db) in zip(params, grads)]

@jit
def update_la(params, x, y):
    xs, δ, ws, bs = compute_xδw_seq(params, x)
    N = len(params)
    L = jnp.diag(δ * ws, k=-1)
    L = L[1:, 1:]

    D = jax.scipy.linalg.block_diag(*[row.reshape((1, 2)) for row in jnp.block([[δ *␣
↪xs[:-1]], [δ]]).T])

    dxs_la = jax.scipy.linalg.solve_triangular(jnp.eye(N) - L, D, lower=True)

    grads = -(y - xs[-1]) * dxs_la[-1]

    return [(w - step_size * dw, b - step_size * db)
            for (w, b), (dw, db) in zip(params, grads.reshape((-1, 2)))]
```

```
# Check that both updates are the same
update_la(params, x, y)
```

```
[(Array([[-0.00826643]], dtype=float32), Array([0.94700736], dtype=float32)),
 (Array([[-2.0638916]], dtype=float32), Array([-0.7872697], dtype=float32)),
 (Array([[1.6248171]], dtype=float32), Array([1.5765371], dtype=float32))]
```

```
update_ad(params, x, y)
```

```
[(Array([[-0.00826644]], dtype=float32), Array([0.94700736], dtype=float32)),
 (Array([[-2.0638916]], dtype=float32), Array([-0.7872697], dtype=float32)),
 (Array([[1.6248171]], dtype=float32), Array([1.5765371], dtype=float32))]
```

15.6 Example 1

Consider the function

$$f(x) = -3x + 2$$

on $[0.5, 3]$.

We use a uniform grid of 200 points and update the parameters for each point on the grid 300 times.

h_i is the sigmoid activation function for all layers except the final one for which we use the identity function and $N = 3$.

Weights are initialized randomly.

```
def f(x):
    return -3 * x + 2

M = 200
grid = jnp.linspace(0.5, 3, num=M)
f_val = f(grid)
```

```
indices = jnp.arange(M)
key = random.PRNGKey(0)

def train(params, grid, f_val, key, num_epochs=300):
    for epoch in range(num_epochs):
        key, _ = random.split(key)
        random_permutation = random.permutation(random.PRNGKey(1), indices)
        for x, y in zip(grid[random_permutation], f_val[random_permutation]):
            params = update_la(params, x, y)

    return params
```

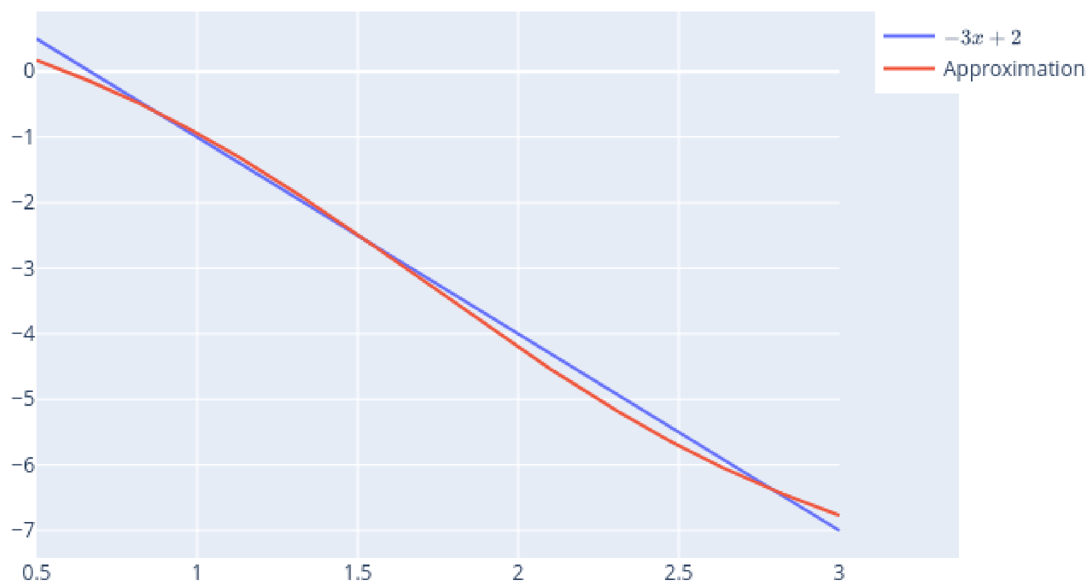
```
# Parameters
N = 3  # Number of layers
layer_sizes = [1, ] * (N + 1)
params_ex1 = init_network_params(layer_sizes, key)
```

```
%%time
params_ex1 = train(params_ex1, grid, f_val, key, num_epochs=500)
```

```
CPU times: user 17.6 s, sys: 4.4 s, total: 22 s
Wall time: 15.1 s
```

```
predictions = vmap(compute_xδw_seq, in_axes=(None, 0))(params_ex1, grid)[0][:, -1]
```

```
fig = go.Figure()
fig.add_trace(go.Scatter(x=grid, y=f_val, name=r'$-3x+2$'))
fig.add_trace(go.Scatter(x=grid, y=predictions, name='Approximation'))

# Export to PNG file
Image(fig.to_image(format="png"))
# fig.show() will provide interactive plot when running
# notebook locally
```



15.7 How Deep?

It is fun to think about how deepening the neural net for the above example affects the quality of approximation

- If the network is too deep, you'll run into the vanishing gradient problem
- Other parameters such as the step size and the number of epochs can be as important or more important than the number of layers in the situation considered in this lecture.
- Indeed, since f is a linear function of x, a one-layer network with the identity map as an activation would probably work best.

15.8 Example 2

We use the same setup as for the previous example with

$$f(x) = \log(x)$$

```
def f(x):
    return jnp.log(x)

grid = jnp.linspace(0.5, 3, num=M)
f_val = f(grid)
```

```
# Parameters
N = 1   # Number of layers
layer_sizes = [1, ] * (N + 1)
params_ex2_1 = init_network_params(layer_sizes, key)
```

```
# Parameters
N = 2   # Number of layers
layer_sizes = [1, ] * (N + 1)
params_ex2_2 = init_network_params(layer_sizes, key)
```

```
# Parameters
N = 3   # Number of layers
layer_sizes = [1, ] * (N + 1)
params_ex2_3 = init_network_params(layer_sizes, key)
```

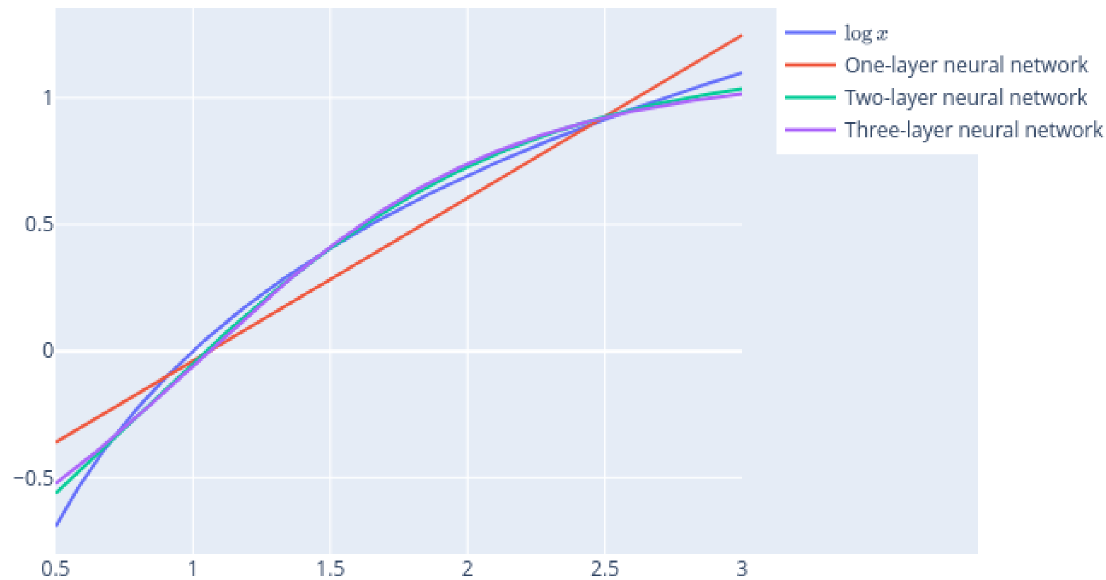
```
params_ex2_1 = train(params_ex2_1, grid, f_val, key, num_epochs=300)
```

```
params_ex2_2 = train(params_ex2_2, grid, f_val, key, num_epochs=300)
```

```
params_ex2_3 = train(params_ex2_3, grid, f_val, key, num_epochs=300)
```

```
predictions_1 = vmap(compute_xδw_seq, in_axes=(None, 0))(params_ex2_1, grid)[0][:, -1]
predictions_2 = vmap(compute_xδw_seq, in_axes=(None, 0))(params_ex2_2, grid)[0][:, -1]
predictions_3 = vmap(compute_xδw_seq, in_axes=(None, 0))(params_ex2_3, grid)[0][:, -1]
```

```
fig = go.Figure()
fig.add_trace(go.Scatter(x=grid, y=f_val, name=r'$\log{x}$'))
fig.add_trace(go.Scatter(x=grid, y=predictions_1, name='One-layer neural network'))
fig.add_trace(go.Scatter(x=grid, y=predictions_2, name='Two-layer neural network'))
fig.add_trace(go.Scatter(x=grid, y=predictions_3, name='Three-layer neural network'))

# Export to PNG file
Image(fig.to_image(format="png"))
# fig.show() will provide interactive plot when running
# notebook locally
```



log x
One-layer neural network
Two-layer neural network
Three-layer neural network
1
0.5
0
−0.5
0.5
1
1.5
2
2.5
3

CHAPTER

SIXTEEN

RANDOMIZED RESPONSE SURVEYS

16.1 Overview

Social stigmas can inhibit people from confessing potentially embarrassing activities or opinions.

When people are reluctant to participate a sample survey about personally sensitive issues, they might decline to participate, and even if they do participate, they might choose to provide incorrect answers to sensitive questions.

These problems induce **selection** biases that present challenges to interpreting and designing surveys.

To illustrate how social scientists have thought about estimating the prevalence of such embarrassing activities and opinions, this lecture describes a classic approach of S. L. Warner [Warner, 1965].

Warner used elementary probability to construct a way to protect the privacy of **individual** respondents to surveys while still estimating the fraction of a **collection** of individuals who have a socially stigmatized characteristic or who engage in a socially stigmatized activity.

Warner's idea was to add **noise** between the respondent's answer and the **signal** about that answer that the survey maker ultimately receives.

Knowing about the structure of the noise assures the respondent that the survey maker does not observe his answer.

Statistical properties of the noise injection procedure provide the respondent **plausible deniability**.

Related ideas underlie modern **differential privacy** systems.

(See https://en.wikipedia.org/wiki/Differential_privacy)

16.2 Warner's Strategy

As usual, let's bring in the Python modules we'll be using.

```
import numpy as np
import pandas as pd
```

Suppose that every person in population either belongs to Group A or Group B.

We want to estimate the proportion π who belong to Group A while protecting individual respondents' privacy.

Warner [Warner, 1965] proposed and analyzed the following procedure.

- A random sample of n people is drawn with replacement from the population and each person is interviewed.
- Draw n random samples from the population with replacement and interview each person.
- Prepare a **random spinner** that with p probability points to the Letter A and with $(1-p)$ probability points to the Letter B.

- Each subject spins a random spinner and sees an outcome (A or B) that the interviewer does **not observe**.
- The subject states whether he belongs to the group to which the spinner points.
- If the spinner points to the group that the spinner belongs, the subject reports "yes"; otherwise he reports "no".
- The subject answers the question truthfully.

Warner constructed a maximum likelihood estimators of the proportion of the population in set A.

Let

- π : True probability of A in the population
- p : Probability that the spinner points to A
- $X_i = \begin{cases} 1, & \text{if the } i\text{th subject says yes} \\ 0, & \text{if the } i\text{th subject says no} \end{cases}$

Index the sample set so that the first n_1 report "yes", while the second $n - n_1$ report "no".

The likelihood function of a sample set is

$$L = [\pi p + (1-\pi)(1-p)]^{n_1} [(1-\pi)p + \pi(1-p)]^{n-n_1} \tag{16.1}$$

The log of the likelihood function is:

$$\log(L) = n_1 \log[\pi p + (1-\pi)(1-p)] + (n-n_1)\log[(1-\pi)p + \pi(1-p)] \tag{16.2}$$

The first-order necessary condition for maximizing the log likelihood function with respect to π is:

$$\frac{(n-n_1)(2p-1)}{(1-\pi)p + \pi(1-p)} = \frac{n_1(2p-1)}{\pi p + (1-\pi)(1-p)}$$

or

$$\pi p + (1-\pi)(1-p) = \frac{n_1}{n} \tag{16.3}$$

If $p \neq \frac{1}{2}$, then the maximum likelihood estimator (MLE) of π is:

$$\hat{\pi} = \frac{p-1}{2p-1} + \frac{n_1}{(2p-1)n} \tag{16.4}$$

We compute the mean and variance of the MLE estimator $\hat{\pi}$ to be:

$$\begin{aligned} \mathbb{E}(\hat{\pi}) &= \frac{1}{2p-1}\left[p - 1 + \frac{1}{n}\sum_{i=1}^{n}\mathbb{E}X_i\right] \\ &= \frac{1}{2p-1}[p-1+\pi p + (1-\pi)(1-p)] \\ &= \pi \end{aligned} \tag{16.5}$$

and

$$\begin{aligned} Var(\hat{\pi}) &= \frac{nVar(X_i)}{(2p-1)^2 n^2} \\ &= \frac{[\pi p + (1-\pi)(1-p)][(1-\pi)p + \pi(1-p)]}{(2p-1)^2 n^2} \\ &= \frac{\frac{1}{4} + (2p^2 - 2p + \frac{1}{2})(-2\pi^2 + 2\pi - \frac{1}{2})}{(2p-1)^2 n^2} \\ &= \frac{1}{n}\left[\frac{1}{16(p-\frac{1}{2})^2} - (\pi - \frac{1}{2})^2\right] \end{aligned} \tag{16.6}$$

Equation (16.5) indicates that $\hat{\pi}$ is an **unbiased estimator** of π while equation (16.6) tell us the variance of the estimator.

To compute a confidence interval, first rewrite (16.6) as:

$$Var(\hat{\pi}) = \frac{\frac{1}{4} - (\pi - \frac{1}{2})^2}{n} + \frac{\frac{1}{16(p-\frac{1}{2})^2} - \frac{1}{4}}{n} \tag{16.7}$$

This equation indicates that the variance of $\hat{\pi}$ can be represented as a sum of the variance due to sampling plus the variance due to the random device.

From the expressions above we can find that:

- When p is $\frac{1}{2}$, expression (16.1) degenerates to a constant.
- When p is 1 or 0, the randomized estimate degenerates to an estimator without randomized sampling.

We shall only discuss situations in which $p \in (\frac{1}{2}, 1)$

(a situation in which $p \in (0, \frac{1}{2})$ is symmetric).

From expressions (16.5) and (16.7) we can deduce that:

- The MSE of $\hat{\pi}$ decreases as p increases.

16.3 Comparing Two Survey Designs

Let's compare the preceding randomized-response method with a stylized non-randomized response method.

In our non-randomized response method, we suppose that:

- Members of Group A tells the truth with probability T_a while the members of Group B tells the truth with probability T_b
- Y_i is 1 or 0 according to whether the sample's ith member's report is in Group A or not.

Then we can estimate π as:

$$\hat{\pi} = \frac{\sum_{i=1}^{n} Y_i}{n} \tag{16.8}$$

We calculate the expectation, bias, and variance of the estimator to be:

$$\mathbb{E}(\hat{\pi}) = \pi T_a + [(1-\pi)(1-T_b)] \tag{16.9}$$

$$\begin{aligned} Bias(\hat{\pi}) &= \mathbb{E}(\hat{\pi} - \pi) \\ &= \pi[T_a + T_b - 2] + [1 - T_b] \end{aligned} \tag{16.10}$$

$$Var(\hat{\pi}) = \frac{[\pi T_a + (1-\pi)(1-T_b)]\,[1 - \pi T_a - (1-\pi)(1-T_b)]}{n} \tag{16.11}$$

It is useful to define a

$$\text{MSE Ratio} = \frac{\text{Mean Square Error Randomized}}{\text{Mean Square Error Regular}}$$

We can compute MSE Ratios for different survey designs associated with different parameter values.

The following Python code computes objects we want to stare at in order to make comparisons under different values of π_A and n:

```python
class Comparison:
    def __init__(self, A, n):
        self.A = A
        self.n = n
        TaTb = np.array([[0.95,  1], [0.9,   1], [0.7,    1],
                         [0.5,   1], [1,  0.95], [1,    0.9],
                         [1,   0.7], [1,   0.5], [0.95, 0.95],
                         [0.9, 0.9], [0.7, 0.7], [0.5,  0.5]])
        self.p_arr = np.array([0.6, 0.7, 0.8, 0.9])
        self.p_map = dict(zip(self.p_arr, [f"MSE Ratio: p = {x}" for x in self.p_
↪arr]))
        self.template = pd.DataFrame(columns=self.p_arr)
        self.template[['T_a','T_b']] = TaTb
        self.template['Bias'] = None

    def theoretical(self):
        A = self.A
        n = self.n
        df = self.template.copy()
        df['Bias'] = A * (df['T_a'] + df['T_b'] - 2) + (1 - df['T_b'])
        for p in self.p_arr:
            df[p] = (1 / (16 * (p - 1/2)**2) - (A - 1/2)**2) / n / \
                    (df['Bias']**2 + ((A * df['T_a'] + (1 - A) * (1 - df['T_b'])) *␣
↪(1 - A * df['T_a'] - (1 - A) * (1 - df['T_b'])) / n))
            df[p] = df[p].round(2)
        df = df.set_index(["T_a", "T_b", "Bias"]).rename(columns=self.p_map)
        return df

    def MCsimulation(self, size=1000, seed=123456):
        A = self.A
        n = self.n
        df = self.template.copy()
        np.random.seed(seed)
        sample = np.random.rand(size, self.n) <= A
        random_device = np.random.rand(size, n)
        mse_rd = {}
        for p in self.p_arr:
            spinner = random_device <= p
            rd_answer = sample * spinner + (1 - sample) * (1 - spinner)
            n1 = rd_answer.sum(axis=1)
            pi_hat = (p - 1) / (2 * p - 1) + n1 / n / (2 * p - 1)
            mse_rd[p] = np.sum((pi_hat - A)**2)
        for inum, irow in df.iterrows():
            truth_a = np.random.rand(size, self.n) <= irow.T_a
            truth_b = np.random.rand(size, self.n) <= irow.T_b
            trad_answer = sample * truth_a + (1 - sample) * (1 - truth_b)
            pi_trad = trad_answer.sum(axis=1) / n
            df.loc[inum, 'Bias'] = pi_trad.mean() - A
            mse_trad = np.sum((pi_trad - A)**2)
            for p in self.p_arr:
                df.loc[inum, p] = (mse_rd[p] / mse_trad).round(2)
        df = df.set_index(["T_a", "T_b", "Bias"]).rename(columns=self.p_map)
        return df
```

Let's put the code to work for parameter values

- $\pi_A = 0.6$
- $n = 1000$

We can generate MSE Ratios theoretically using the above formulas.

We can also perform Monte Carlo simulations of a MSE Ratio.

```
cp1 = Comparison(0.6, 1000)
df1_theoretical = cp1.theoretical()
df1_theoretical
```

```
                  MSE Ratio: p = 0.6  MSE Ratio: p = 0.7  MSE Ratio: p = 0.8  \
T_a  T_b  Bias
0.95 1.00 -0.03                 5.45                1.36                0.60
0.90 1.00 -0.06                 1.62                0.40                0.18
0.70 1.00 -0.18                 0.19                0.05                0.02
0.50 1.00 -0.30                 0.07                0.02                0.01
1.00 0.95  0.02                 9.82                2.44                1.08
     0.90  0.04                 3.41                0.85                0.37
     0.70  0.12                 0.43                0.11                0.05
     0.50  0.20                 0.16                0.04                0.02
0.95 0.95 -0.01                18.25                4.54                2.00
0.90 0.90 -0.02                 9.70                2.41                1.06
0.70 0.70 -0.06                 1.62                0.40                0.18
0.50 0.50 -0.10                 0.61                0.15                0.07

                  MSE Ratio: p = 0.9
T_a  T_b  Bias
0.95 1.00 -0.03                 0.33
0.90 1.00 -0.06                 0.10
0.70 1.00 -0.18                 0.01
0.50 1.00 -0.30                 0.00
1.00 0.95  0.02                 0.60
     0.90  0.04                 0.21
     0.70  0.12                 0.03
     0.50  0.20                 0.01
0.95 0.95 -0.01                 1.11
0.90 0.90 -0.02                 0.59
0.70 0.70 -0.06                 0.10
0.50 0.50 -0.10                 0.04
```

```
df1_mc = cp1.MCsimulation()
df1_mc
```

```
                      MSE Ratio: p = 0.6 MSE Ratio: p = 0.7 MSE Ratio: p = 0.8  \
T_a  T_b  Bias
0.95 1.00 -0.030060                 5.76               1.36               0.63
0.90 1.00 -0.060045                 1.73               0.41               0.19
0.70 1.00 -0.179530                 0.21               0.05               0.02
0.50 1.00 -0.300077                 0.07               0.02               0.01
1.00 0.95  0.019770                10.59                2.5               1.15
     0.90  0.040050                 3.63               0.86               0.39
     0.70  0.120052                 0.46               0.11               0.05
     0.50  0.199746                 0.17               0.04               0.02
0.95 0.95 -0.010137                18.65               4.41               2.02
0.90 0.90 -0.020103                10.48               2.48               1.14
0.70 0.70 -0.060488                 1.71                0.4               0.19
0.50 0.50 -0.099341                 0.66               0.16               0.07

                      MSE Ratio: p = 0.9
```

(continues on next page)

(continued from previous page)

```
T_a  T_b  Bias
0.95 1.00 -0.030060              0.35
0.90 1.00 -0.060045               0.1
0.70 1.00 -0.179530              0.01
0.50 1.00 -0.300077               0.0
1.00 0.95  0.019770              0.64
     0.90  0.040050              0.22
     0.70  0.120052              0.03
     0.50  0.199746              0.01
0.95 0.95 -0.010137              1.12
0.90 0.90 -0.020103              0.63
0.70 0.70 -0.060488               0.1
0.50 0.50 -0.099341              0.04
```

The theoretical calculations do a good job of predicting Monte Carlo results.

We see that in many situations, especially when the bias is not small, the MSE of the randomized-sampling methods is smaller than that of the non-randomized sampling method.

These differences become larger as p increases.

By adjusting parameters π_A and n, we can study outcomes in different situations.

For example, for another situation described in Warner [Warner, 1965]:

- $\pi_A = 0.5$
- $n = 1000$

we can use the code

```
cp2 = Comparison(0.5, 1000)
df2_theoretical = cp2.theoretical()
df2_theoretical
```

```
                  MSE Ratio: p = 0.6  MSE Ratio: p = 0.7  MSE Ratio: p = 0.8  \
T_a  T_b  Bias
0.95 1.00 -0.025                7.15                1.79                0.79
0.90 1.00 -0.050                2.27                0.57                0.25
0.70 1.00 -0.150                0.27                0.07                0.03
0.50 1.00 -0.250                0.10                0.02                0.01
1.00 0.95  0.025                7.15                1.79                0.79
     0.90  0.050                2.27                0.57                0.25
     0.70  0.150                0.27                0.07                0.03
     0.50  0.250                0.10                0.02                0.01
0.95 0.95  0.000               25.00                6.25                2.78
0.90 0.90  0.000               25.00                6.25                2.78
0.70 0.70  0.000               25.00                6.25                2.78
0.50 0.50  0.000               25.00                6.25                2.78

                  MSE Ratio: p = 0.9
T_a  T_b  Bias
0.95 1.00 -0.025                0.45
0.90 1.00 -0.050                0.14
0.70 1.00 -0.150                0.02
0.50 1.00 -0.250                0.01
1.00 0.95  0.025                0.45
     0.90  0.050                0.14
     0.70  0.150                0.02
```

(continues on next page)

(continued from previous page)

```
     0.50  0.250                 0.01
0.95 0.95  0.000                 1.56
0.90 0.90  0.000                 1.56
0.70 0.70  0.000                 1.56
0.50 0.50  0.000                 1.56
```

```
df2_mc = cp2.MCsimulation()
df2_mc
```

```
                    MSE Ratio: p = 0.6 MSE Ratio: p = 0.7 MSE Ratio: p = 0.8  \
T_a  T_b  Bias
0.95 1.00 -0.025230                7.0               1.69               0.75
0.90 1.00 -0.050279               2.23               0.54               0.24
0.70 1.00 -0.149866               0.27               0.07               0.03
0.50 1.00 -0.250211                0.1               0.02               0.01
1.00 0.95  0.024410               7.38               1.78               0.79
     0.90  0.049839               2.26               0.54               0.24
     0.70  0.149769               0.27               0.07               0.03
     0.50  0.249851                0.1               0.02               0.01
0.95 0.95 -0.000260              24.29               5.86               2.59
0.90 0.90 -0.000109              25.73                6.2               2.74
0.70 0.70 -0.000439              25.75               6.21               2.74
0.50 0.50  0.000768              24.91               6.01               2.65

                    MSE Ratio: p = 0.9
T_a  T_b  Bias
0.95 1.00 -0.025230               0.44
0.90 1.00 -0.050279               0.14
0.70 1.00 -0.149866               0.02
0.50 1.00 -0.250211               0.01
1.00 0.95  0.024410               0.46
     0.90  0.049839               0.14
     0.70  0.149769               0.02
     0.50  0.249851               0.01
0.95 0.95 -0.000260               1.52
0.90 0.90 -0.000109               1.61
0.70 0.70 -0.000439               1.61
0.50 0.50  0.000768               1.56
```

We can also revisit a calculation in the concluding section of Warner [Warner, 1965] in which

- $\pi_A = 0.6$
- $n = 2000$

We use the code

```
cp3 = Comparison(0.6, 2000)
df3_theoretical = cp3.theoretical()
df3_theoretical
```

```
                 MSE Ratio: p = 0.6  MSE Ratio: p = 0.7  MSE Ratio: p = 0.8  \
T_a  T_b  Bias
0.95 1.00 -0.03                3.05                0.76                0.33
0.90 1.00 -0.06                0.84                0.21                0.09
0.70 1.00 -0.18                0.10                0.02                0.01
0.50 1.00 -0.30                0.03                0.01                0.00
```

(continues on next page)

(continued from previous page)

```
1.00 0.95  0.02                6.03                1.50                0.66
     0.90  0.04                1.82                0.45                0.20
     0.70  0.12                0.22                0.05                0.02
     0.50  0.20                0.08                0.02                0.01
0.95 0.95 -0.01               14.12                3.51                1.55
0.90 0.90 -0.02                5.98                1.49                0.66
0.70 0.70 -0.06                0.84                0.21                0.09
0.50 0.50 -0.10                0.31                0.08                0.03

                  MSE Ratio: p = 0.9
T_a  T_b  Bias
0.95 1.00 -0.03                0.19
0.90 1.00 -0.06                0.05
0.70 1.00 -0.18                0.01
0.50 1.00 -0.30                0.00
1.00 0.95  0.02                0.37
     0.90  0.04                0.11
     0.70  0.12                0.01
     0.50  0.20                0.00
0.95 0.95 -0.01                0.86
0.90 0.90 -0.02                0.36
0.70 0.70 -0.06                0.05
0.50 0.50 -0.10                0.02
```

```
df3_mc = cp3.MCsimulation()
df3_mc
```

```
                     MSE Ratio: p = 0.6 MSE Ratio: p = 0.7 MSE Ratio: p = 0.8  \
T_a  T_b  Bias
0.95 1.00 -0.030316                3.27                0.8               0.34
0.90 1.00 -0.060352                0.91               0.22               0.09
0.70 1.00 -0.180087                0.11               0.03               0.01
0.50 1.00 -0.299849                0.04               0.01                0.0
1.00 0.95  0.019734                 6.7               1.64               0.69
     0.90  0.039766                2.01               0.49               0.21
     0.70  0.119789                0.24               0.06               0.02
     0.50  0.200138                0.09               0.02               0.01
0.95 0.95 -0.010475               14.78               3.61               1.53
0.90 0.90 -0.020373                6.32               1.54               0.65
0.70 0.70 -0.059945                0.92               0.23                0.1
0.50 0.50 -0.100103                0.34               0.08               0.03

                     MSE Ratio: p = 0.9
T_a  T_b  Bias
0.95 1.00 -0.030316                0.19
0.90 1.00 -0.060352                0.05
0.70 1.00 -0.180087                0.01
0.50 1.00 -0.299849                 0.0
1.00 0.95  0.019734                0.39
     0.90  0.039766                0.12
     0.70  0.119789                0.01
     0.50  0.200138                 0.0
0.95 0.95 -0.010475                0.85
0.90 0.90 -0.020373                0.36
0.70 0.70 -0.059945                0.05
0.50 0.50 -0.100103                0.02
```

Evidently, as n increases, the randomized response method does better performance in more situations.

16.4 Concluding Remarks

This QuantEcon lecture describes some alternative randomized response surveys.

That lecture presents a utilitarian analysis of those alternatives conducted by Lars Ljungqvist [Ljungqvist, 1993].

```
import matplotlib.pyplot as plt
import numpy as np
```

CHAPTER

SEVENTEEN

EXPECTED UTILITIES OF RANDOM RESPONSES

17.1 Overview

This QuantEcon lecture describes randomized response surveys in the tradition of Warner [Warner, 1965] that are designed to protect respondents' privacy.

Lars Ljungqvist [Ljungqvist, 1993] analyzed how a respondent's decision about whether to answer truthfully depends on **expected utility**.

The lecture tells how Ljungqvist used his framework to shed light on alternative randomized response survey techniques proposed, for example, by [Lanke, 1975], [Lanke, 1976], [Leysieffer and Warner, 1976], [Anderson, 1976], [Fligner *et al.*, 1977], [Greenberg *et al.*, 1977], [Greenberg *et al.*, 1969].

17.2 Privacy Measures

We consider randomized response models with only two possible answers, "yes" and "no."

The design determines probabilities

$$\begin{aligned} \text{Pr}(\text{yes}|A) &= 1 - \text{Pr}(\text{no}|A) \\ \text{Pr}(\text{yes}|A^{'}) &= 1 - \text{Pr}(\text{no}|A^{'}) \end{aligned}$$

These design probabilities in turn can be used to compute the conditional probability of belonging to the sensitive group A for a given response, say r:

$$\text{Pr}(A|r) = \frac{\pi_A \text{Pr}(r|A)}{\pi_A \text{Pr}(r|A) + (1-\pi_A)\text{Pr}(r|A^{'})} \tag{17.1}$$

17.3 Zoo of Concepts

At this point we describe some concepts proposed by various researchers

17.3.1 Leysieffer and Warner(1976)

The response r is regarded as jeopardizing with respect to A or $A^{'}$ if

$$
\begin{aligned}
\Pr(A|r) &> \pi_A \\
&\text{or} \\
\Pr(A^{'}|r) &> 1-\pi_A
\end{aligned}
\tag{17.2}
$$

From Bayes's rule:

$$
\frac{\Pr(A|r)}{\Pr(A^{'}|r)} \times \frac{(1-\pi_A)}{\pi_A} = \frac{\Pr(r|A)}{\Pr(r|A^{'})} \tag{17.3}
$$

If this expression is greater (less) than unity, it follows that r is jeopardizing with respect to $A(A^{'})$. Then, the natural measure of jeopardy will be:

$$
\begin{aligned}
g(r|A) &= \frac{\Pr(r|A)}{\Pr(r|A^{'})} \\
&\text{and} \\
g(r|A^{'}) &= \frac{\Pr(r|A^{'})}{\Pr(r|A)}
\end{aligned}
\tag{17.4}
$$

Suppose, without loss of generality, that $\Pr(\text{yes}|A) > \Pr(\text{yes}|A^{'})$, then a yes (no) answer is jeopardizing with respect $A(A^{'})$, that is,

$$
\begin{aligned}
g(\text{yes}|A) &> 1 \\
&\text{and} \\
g(\text{no}|A^{'}) &> 1
\end{aligned}
$$

Leysieffer and Warner proved that the variance of the estimate can only be decreased through an increase in one or both of these two measures of jeopardy.

An efficient randomized response model is, therefore, any model that attains the maximum acceptable levels of jeopardy that are consistent with cooperation of the respondents.

As a special example, Leysieffer and Warner considered "a problem in which there is no jeopardy in a no answer"; that is, $g(\text{no}|A^{'})$ can be of unlimited magnitude.

Evidently, an optimal design must have

$$
\Pr(\text{yes}|A) = 1
$$

which implies that

$$
\Pr(A|\text{no}) = 0
$$

17.3.2 Lanke(1976)

Lanke (1975) [Lanke, 1975] argued that "it is membership in Group A that people may want to hide, not membership in the complementary Group A'."

For that reason, Lanke (1976) [Lanke, 1976] argued that an appropriate measure of protection is to minimize

$$
\max\left\{\Pr(A|\text{yes}), \Pr(A|\text{no})\right\} \tag{17.5}
$$

Holding this measure constant, he explained under what conditions the smallest variance of the estimate was achieved with the unrelated question model or Warner's (1965) original model.

17.3.3 2.3 Fligner, Policello, and Singh

Fligner, Policello, and Singh reached similar conclusion as Lanke (1976). [Fligner *et al.*, 1977]

They measured "private protection" as

$$\frac{1 - \max\{\Pr(A|\text{yes}), \Pr(A|\text{no})\}}{1 - \pi_A} \tag{17.6}$$

17.3.4 2.4 Greenberg, Kuebler, Abernathy, and Horvitz (1977)

[Greenberg *et al.*, 1977]

Greenberg, Kuebler, Abernathy, and Horvitz (1977) stressed the importance of examining the risk to respondents who do not belong to A as well as the risk to those who do belong to the sensitive group.

They defined the hazard for an individual in A as the probability that he or she is perceived as belonging to A:

$$\Pr(\text{yes}|A) \times \Pr(A|\text{yes}) + \Pr(\text{no}|A) \times \Pr(A|\text{no}) \tag{17.7}$$

Similarly, the hazard for an individual who does not belong to A would be

$$\Pr(\text{yes}|A^{'}) \times \Pr(A|\text{yes}) + \Pr(\text{no}|A^{'}) \times \Pr(A|\text{no}) \tag{17.8}$$

Greenberg et al. (1977) also considered an alternative related measure of hazard that "is likely to be closer to the actual concern felt by a respondent."

The "limited hazard" for an individual in A and $A^{'}$ is

$$\Pr(\text{yes}|A) \times \Pr(A|\text{yes}) \tag{17.9}$$

and

$$\Pr(\text{yes}|A^{'}) \times \Pr(A|\text{yes}) \tag{17.10}$$

This measure is just the first term in (17.7), i.e., the probability that an individual answers "yes" and is perceived to belong to A.

17.4 Respondent's Expected Utility

17.4.1 Truth Border

Key assumptions that underlie a randomized response technique for estimating the fraction of a population that belongs to A are:

- **Assumption 1**: Respondents feel discomfort from being thought of as belonging to A.
- **Assumption 2**: Respondents prefer to answer questions truthfully than to lie, so long as the cost of doing so is not too high. The cost is taken to be the discomfort in 1.

Let r_i denote individual i's response to the randomized question.

r_i can only take values "yes" or "no".

For a given design of a randomized response interview and a given belief about the fraction of the population that belongs to A, the respondent's answer is associated with a conditional probability $\Pr(A|r_i)$ that the individual belongs to A.

Given r_i and complete privacy, the individual's utility is higher if r_i represents a truthful answer rather than a lie.

In terms of a respondent's expected utility as a function of $\Pr(A|r_i)$ and r_i

- The higher is $\text{Pr}(A|r_i)$, the lower isindividual i's expected utility.
- expected utility is higher if r_i represents a truthful answer rather than a lie

Define:

- $\phi_i \in \{\text{truth}, \text{lie}\}$, a dichotomous variable that indicates whether or not r_i is a truthful statement.
- $U_i\left(\text{Pr}(A|r_i), \phi_i\right)$, a utility function that is differentiable in its first argument, summarizes individual i's expected utility.

Then there is an r_i such that

$$\frac{\partial U_i\left(\text{Pr}(A|r_i), \phi_i\right)}{\partial \text{Pr}(A|r_i)} < 0, \text{ for } \phi_i \in \{\text{truth}, \text{lie}\} \tag{17.11}$$

and

$$U_i\left(\text{Pr}(A|r_i), \text{truth}\right) > U_i\left(\text{Pr}(A|r_i), \text{lie}\right), \text{ for } \text{Pr}(A|r_i) \in [0,1] \tag{17.12}$$

Suppose now that correct answer for individual i is "yes".

Individual i would choose to answer truthfully if

$$U_i\left(\text{Pr}(A|\text{yes}), \text{truth}\right) \geq U_i\left(\text{Pr}(A|\text{no}), \text{lie}\right) \tag{17.13}$$

If the correct answer is "no", individual i would volunteer the correct answer only if

$$U_i\left(\text{Pr}(A|\text{no}), \text{truth}\right) \geq U_i\left(\text{Pr}(A|\text{yes}), \text{lie}\right) \tag{17.14}$$

Assume that

$$\text{Pr}(A|\text{yes}) > \pi_A > \text{Pr}(A|\text{no})$$

so that a "yes" answer increases the odds that an individual belongs to A.

Constraint (17.14) holds for sure.

Consequently, constraint (17.13) becomes the single necessary condition for individual i always to answer truthfully.

At equality, constraint $(10.\text{a})$ determines conditional probabilities that make the individual indifferent between telling the truth and lying when the correct answer is "yes":

$$U_i\left(\text{Pr}(A|\text{yes}), \text{truth}\right) = U_i\left(\text{Pr}(A|\text{no}), \text{lie}\right) \tag{17.15}$$

Equation (17.15) defines a "truth border".

Differentiating (17.15) with respect to the conditional probabilities shows that the truth border has a positive slope in the space of conditional probabilities:

$$\frac{\partial \text{Pr}(A|\text{no})}{\partial \text{Pr}(A|\text{yes})} = \frac{\frac{\partial U_i\left(\text{Pr}(A|\text{yes}), \text{truth}\right)}{\partial \text{Pr}(A|\text{yes})}}{\frac{\partial U_i\left(\text{Pr}(A|\text{no}), \text{lie}\right)}{\partial \text{Pr}(A|\text{no})}} > 0 \tag{17.16}$$

The source of the positive relationship is:

- The individual is willing to volunteer a truthful "yes" answer so long as the utility from doing so (i.e., the left side of (17.15)) is at least as high as the utility of lying on the right side of (17.15).
- Suppose now that $\text{Pr}(A|\text{yes})$ increases. That reduces the utility of telling the truth. To preserve indifference between a truthful answer and a lie, $\text{Pr}(A|\text{no})$ must increase to reduce the utility of lying.

17.4.2 Drawing a Truth Border

We can deduce two things about the truth border:

- The truth border divides the space of conditional probabilities into two subsets: "truth telling" and "lying". Thus, sufficient privacy elicits a truthful answer, whereas insufficient privacy results in a lie. The truth border depends on a respondent's utility function.
- Assumptions in (17.11) and (17.11) are sufficient only to guarantee a positive slope of the truth border. The truth border can have either a concave or a convex shape.

We can draw some truth borders with the following Python code:

```
x1 = np.arange(0, 1, 0.001)
y1 = x1 - 0.4
x2 = np.arange(0.4**2, 1, 0.001)
y2 = (pow(x2, 0.5) - 0.4)**2
x3 = np.arange(0.4**0.5, 1, 0.001)
y3 = pow(x3**2 - 0.4, 0.5)
plt.figure(figsize=(12, 10))
plt.plot(x1, y1, 'r-', label=r'Truth Border of: $U_i(Pr(A|r_i),\phi_i)=-Pr(A|r_i)+f(\
↪phi_i)$')
plt.fill_between(x1, 0, y1, facecolor='red', alpha=0.05)
plt.plot(x2, y2, 'b-', label=r'Truth Border of: $U_i(Pr(A|r_i),\phi_i)=-Pr(A|r_i)^{2}
↪+f(\phi_i)$')
plt.fill_between(x2, 0, y2, facecolor='blue', alpha=0.05)
plt.plot(x3, y3, 'y-', label=r'Truth Border of: $U_i(Pr(A|r_i),\phi_i)=-\sqrt{Pr(A|r_
↪i)}+f(\phi_i)$')
plt.fill_between(x3, 0, y3, facecolor='green', alpha=0.05)
plt.plot(x1, x1, ':', linewidth=2)
plt.xlim([0, 1])
plt.ylim([0, 1])

plt.xlabel('Pr(A|yes)')
plt.ylabel('Pr(A|no)')
plt.text(0.42, 0.3, "Truth Telling", fontdict={'size':28, 'style':'italic'})
plt.text(0.8, 0.1, "Lying", fontdict={'size':28, 'style':'italic'})

plt.legend(loc=0, fontsize='large')
plt.title('Figure 1.1')
plt.show()
```

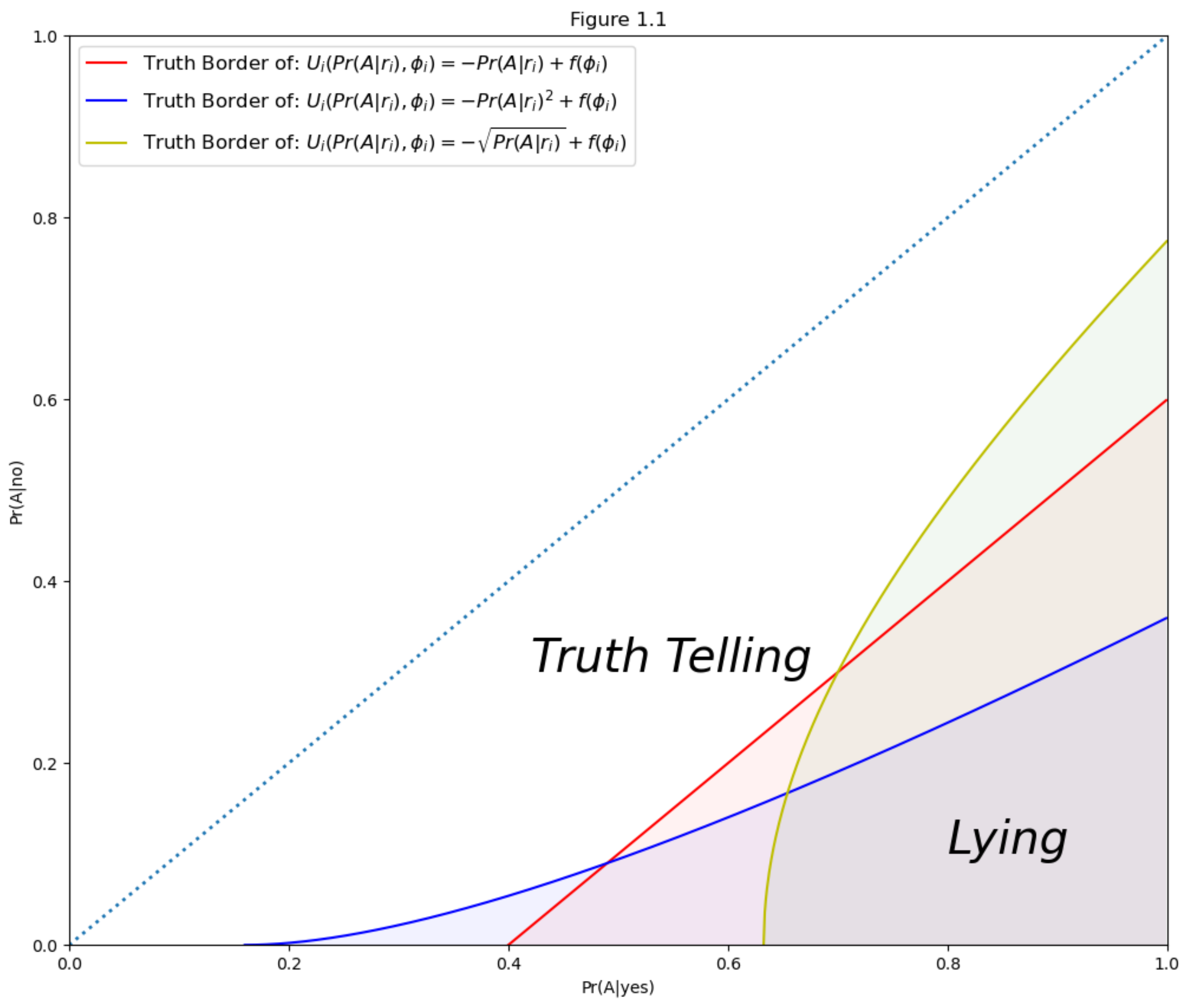


Figure 1.1 three types of truth border.

Without loss of generality, we consider the truth border:

$$U_i(\Pr(A|r_i), \phi_i) = -\Pr(A|r_i) + f(\phi_i)$$

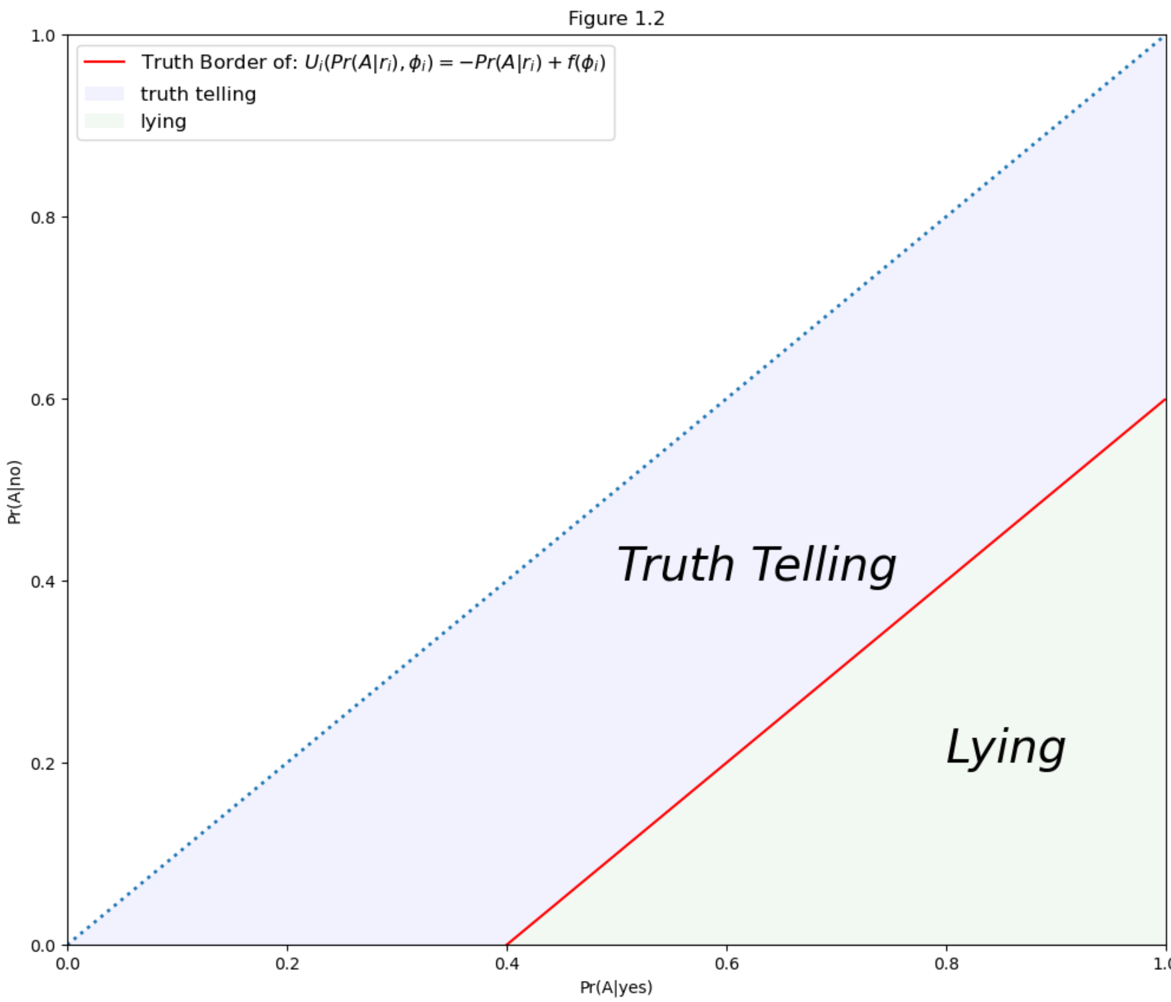
and plot the "truth telling" and "lying area" of individual i in Figure 1.2:

```
x1 = np.arange(0, 1, 0.001)
y1 = x1 - 0.4
z1 = x1
z2 = 0
plt.figure(figsize=(12, 10))
plt.plot(x1, y1,'r-',label=r'Truth Border of: $U_i(Pr(A|r_i),\phi_i)=-Pr(A|r_i)+f(\
↪phi_i)$')
plt.plot(x1, x1, ':', linewidth=2)
plt.fill_between(x1, y1, z1, facecolor='blue', alpha=0.05, label='truth telling')
plt.fill_between(x1, z2, y1, facecolor='green', alpha=0.05, label='lying')
plt.xlim([0, 1])
plt.ylim([0, 1])

plt.xlabel('Pr(A|yes)')
plt.ylabel('Pr(A|no)')
```

(continues on next page)

(continued from previous page)

```
plt.text(0.5, 0.4, "Truth Telling", fontdict={'size':28, 'style':'italic'})
plt.text(0.8, 0.2, "Lying", fontdict={'size':28, 'style':'italic'})

plt.legend(loc=0, fontsize='large')
plt.title('Figure 1.2')
plt.show()
```



17.5 Utilitarian View of Survey Design

17.5.1 Iso-variance Curves

A statistician's objective is

- to find a randomized response survey design that minimizes the bias and the variance of the estimator.

Given a design that ensures truthful answers by all respondents, Anderson(1976, Theorem 1) [Anderson, 1976] showed that the minimum variance estimate in the two-response model has variance

$$V(\Pr(A|\text{yes}), \Pr(A|\text{no})) = \frac{{\pi_A}^2(1-\pi_A)^2}{n} \times \frac{1}{\Pr(A|\text{yes}) - \pi_A} \times \frac{1}{\pi_A - \Pr(A|\text{no})} \tag{17.17}$$

where the random sample with replacement consists of n individuals.

We can use Expression (17.17) to draw iso-variance curves.

The following inequalities restrict the shapes of iso-variance curves:

$$\left.\frac{d\ \Pr(A|\text{no})}{d\ \Pr(A|\text{yes})}\right|_{\text{constant variance}} = \frac{\pi_A - \Pr(A|\text{no})}{\Pr(A|\text{yes}) - \pi_A} > 0 \tag{17.18}$$

$$\left.\frac{d^2\ \Pr(A|\text{no})}{d\ \Pr(A|\text{yes})^2}\right|_{\text{constant variance}} = -\frac{2\left[\pi_A - \Pr(A|\text{no})\right]}{\left[\Pr(A|\text{yes}) - \pi_A\right]^2} < 0 \tag{17.19}$$

From expression (17.17), (17.18) and (17.19) we can see that:

- Variance can be reduced only by increasing the distance of $\Pr(A|\text{yes})$ and/or $\Pr(A|\text{no})$ from r_A.
- Iso-variance curves are always upward-sloping and concave.

17.5.2 Drawing Iso-variance Curves

We use Python code to draw iso-variance curves.

The pairs of conditional probabilities can be attained using Warner's (1965) model.

Note that:

- Any point on the iso-variance curves can be attained with the unrelated question model as long as the statistician can completely control the model design.
- Warner's (1965) original randomized response model is less flexible than the unrelated question model.

```
class Iso_Variance:
    def __init__(self, pi, n):
        self.pi = pi
        self.n = n

    def plotting_iso_variance_curve(self):
        pi = self.pi
        n = self.n

        nv = np.array([0.27, 0.34, 0.49, 0.74, 0.92, 1.1, 1.47, 2.94, 14.7])
        x = np.arange(0, 1, 0.001)
        x0 = np.arange(pi, 1, 0.001)
        x2 = np.arange(0, pi, 0.001)
        y1 = [pi for i in x0]
        y2 = [pi for i in x2]
        y0 = 1 / (1 + (x0 * (1 - pi)**2) / ((1 - x0) * pi**2))

        plt.figure(figsize=(12, 10))
        plt.plot(x0, y0, 'm-', label='Warner')
        plt.plot(x, x, 'c:', linewidth=2)
        plt.plot(x0, y1,'c:', linewidth=2)
        plt.plot(y2, x2, 'c:', linewidth=2)
        for i in range(len(nv)):
            y = pi - (pi**2 * (1 - pi)**2) / (n * (nv[i] / n) * (x0 - pi + 1e-8))
            plt.plot(x0, y, 'k--', alpha=1 - 0.07 * i, label=f'V{i+1}')
        plt.xlim([0, 1])
        plt.ylim([0, 0.5])
        plt.xlabel('Pr(A|yes)')
```

(continues on next page)

(continued from previous page)

```
        plt.ylabel('Pr(A|no)')
        plt.legend(loc=0, fontsize='large')
        plt.text(0.32, 0.28, "High Var", fontdict={'size':15, 'style':'italic'})
        plt.text(0.91, 0.01, "Low Var", fontdict={'size':15, 'style':'italic'})
        plt.title('Figure 2')
        plt.show()
```

Properties of iso-variance curves are:

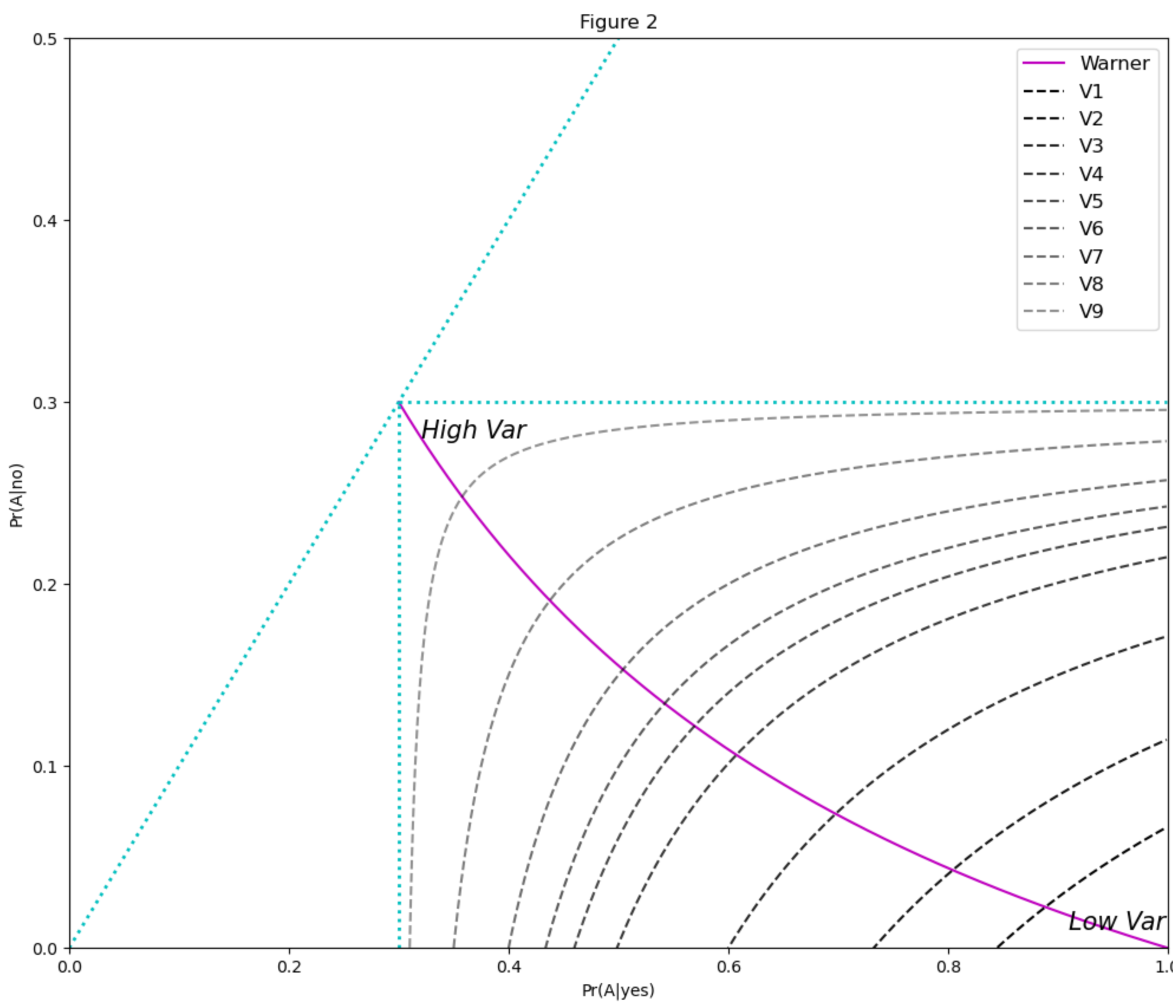
- All points on one iso-variance curve share the same variance
- From V_1 to V_9, the variance of the iso-variance curve increase monotonically, as colors brighten monotonically

Suppose the parameters of the iso-variance model follow those in Ljungqvist [Ljungqvist, 1993], which are:

- $\pi = 0.3$
- $n = 100$

Then we can plot the iso-variance curve in Figure 2:

```
var = Iso_Variance(pi=0.3, n=100)
var.plotting_iso_variance_curve()
```



17.5.3 Optimal Survey

A point on an iso-variance curves can be attained with the unrelated question design.

We now focus on finding an “optimal survey design” that

- Minimizes the variance of the estimator subject to privacy restrictions.

To obtain an optimal design, we first superimpose all individuals’ truth borders on the iso-variance mapping.

To construct an optimal design

- The statistician should find the intersection of areas above all truth borders; that is, the set of conditional probabilities ensuring truthful answers from all respondents.
- The point where this set touches the lowest possible iso-variance curve determines an optimal survey design.

Consquently, a minimum variance unbiased estimator is pinned down by an individual who is the least willing to volunteer a truthful answer.

Here are some comments about the model design:

- An individual’s decision of whether or not to answer truthfully depends on his or her belief about other respondents’ behavior, because this determines the individual’s calculation of $\text{Pr}(A|\text{yes})$ and $\text{Pr}(A|\text{no})$.
- An equilibrium of the optimal design model is a Nash equilibrium of a noncooperative game.
- Assumption (17.12) is sufficient to guarantee existence of an optimal model design. By choosing $\text{Pr}(A|\text{yes})$ and $\text{Pr}(A|\text{no})$ sufficiently close to each other, all respondents will find it optimal to answer truthfully. The closer are these probabilities, the higher the variance of the estimator becomes.
- If respondents experience a large enough increase in expected utility from telling the truth, then there is no need to use a randomized response model. The smallest possible variance of the estimate is then obtained at $\text{Pr}(A|\text{yes}) = 1$ and $\text{Pr}(A|\text{no}) = 0$; that is, when respondents answer truthfully to direct questioning.
- A more general design problem would be to minimize some weighted sum of the estimator’s variance and bias. It would be optimal to accept some lies from the most “reluctant” respondents.

17.6 Criticisms of Proposed Privacy Measures

We can use a utilitarian approach to analyze some privacy measures.

We’ll enlist Python Code to help us.

17.6.1 Analysis of Method of Lanke’s (1976)

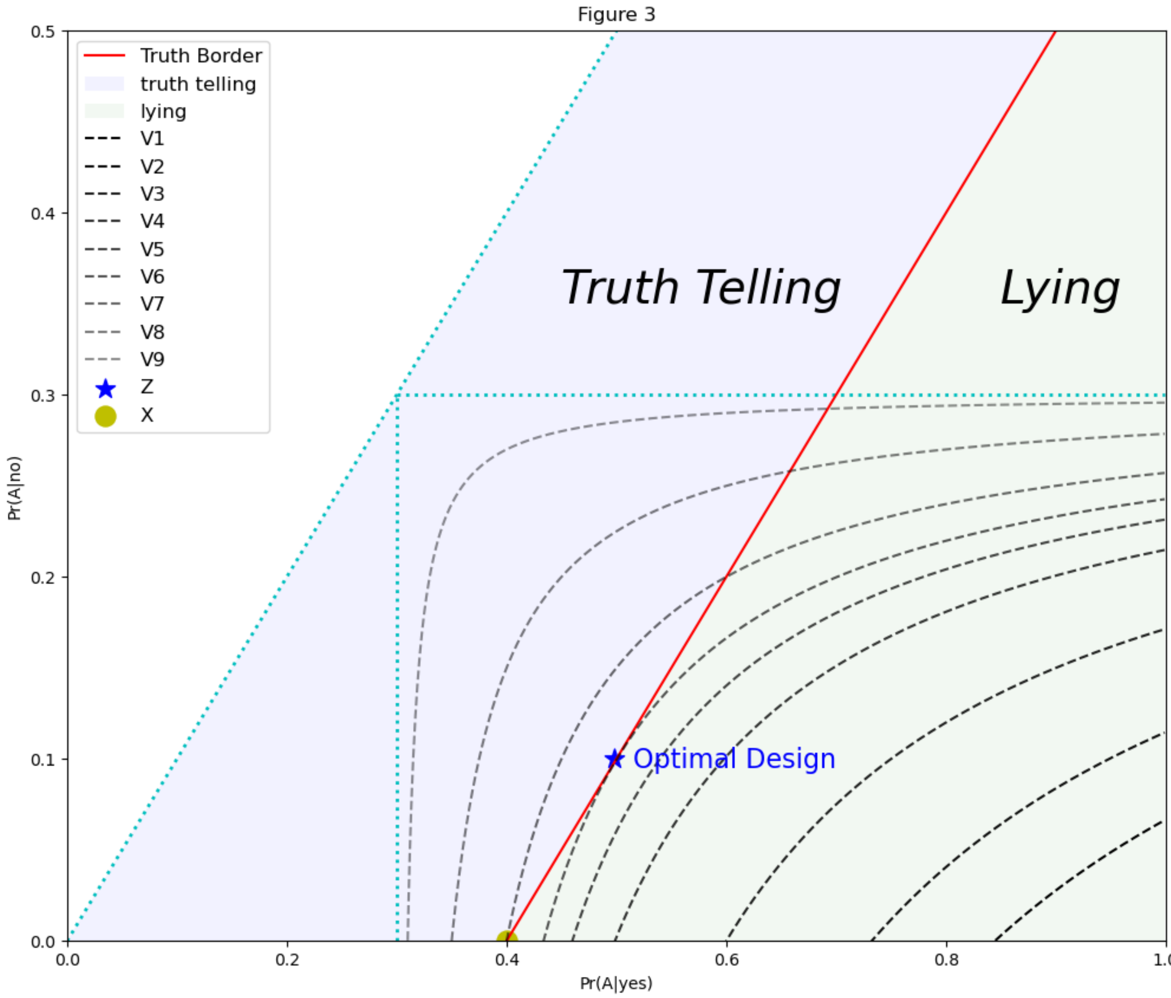
Lanke (1976) recommends a privacy protection criterion that minimizes:

$$\max \left\{ \text{Pr}(A|\text{yes}), \text{Pr}(A|\text{no}) \right\} \tag{17.20}$$

Following Lanke’s suggestion, the statistician should find the highest possible $\text{Pr}(A|\text{yes})$ consistent with truth telling while $\text{Pr}(A|\text{no})$ is fixed at 0. The variance is then minimized at point X in Figure 3.

However, we can see that in Figure 3, point Z offers a smaller variance that still allows cooperation of the respondents, and it is achievable following our discussion of the truth border in Part III:

```python
pi = 0.3
n = 100
nv = [0.27, 0.34, 0.49, 0.74, 0.92, 1.1, 1.47, 2.94, 14.7]
x = np.arange(0, 1, 0.001)
y = x - 0.4
z = x
x0 = np.arange(pi, 1, 0.001)
x2 = np.arange(0, pi, 0.001)
y1 = [pi for i in x0]
y2 = [pi for i in x2]

plt.figure(figsize=(12, 10))
plt.plot(x, x, 'c:', linewidth=2)
plt.plot(x0, y1, 'c:', linewidth=2)
plt.plot(y2, x2, 'c:', linewidth=2)
plt.plot(x, y, 'r-', label='Truth Border')
plt.fill_between(x, y, z, facecolor='blue', alpha=0.05, label='truth telling')
plt.fill_between(x, 0, y, facecolor='green', alpha=0.05, label='lying')
for i in range(len(nv)):
    y = pi - (pi**2 * (1 - pi)**2) / (n * (nv[i] / n) * (x0 - pi + 1e-8))
    plt.plot(x0, y, 'k--', alpha=1 - 0.07 * i, label=f'V{i+1}')


plt.scatter(0.498, 0.1, c='b', marker='*', label='Z', s=150)
plt.scatter(0.4, 0, c='y', label='X', s=150)
plt.xlim([0, 1])
plt.ylim([0, 0.5])
plt.xlabel('Pr(A|yes)')
plt.ylabel('Pr(A|no)')
plt.text(0.45, 0.35, "Truth Telling", fontdict={'size':28, 'style':'italic'})
plt.text(0.85, 0.35, "Lying",fontdict = {'size':28, 'style':'italic'})
plt.text(0.515, 0.095, "Optimal Design", fontdict={'size':16,'color':'b'})
plt.legend(loc=0, fontsize='large')
plt.title('Figure 3')
plt.show()
```



17.6.2 Method of Leysieffer and Warner (1976)

Leysieffer and Warner (1976) recommend a two-dimensional measure of jeopardy that reduces to a single dimension when there is no jeopardy in a 'no' answer", which means that

$$\Pr(\text{yes}|A) = 1$$

and

$$\Pr(A|\text{no}) = 0$$

This is not an optimal choice under a utilitarian approach.

17.6.3 Analysis on the Method of Chaudhuri and Mukerjee's (1988)

[Chadhuri and Mukerjee, 1988]

Chaudhuri and Mukerjee (1988) argued that the individual may find that since "yes" may sometimes relate to the sensitive group A, a clever respondent may falsely but safely always be inclined to respond "no". In this situation, the truth border is such that individuals choose to lie whenever the truthful answer is "yes" and

$$\text{Pr}(A|\text{no}) = 0$$

Here the gain from lying is too high for someone to volunteer a "yes" answer.

This means that

$$U_i\left(\text{Pr}(A|\text{yes}), \text{truth}\right) < U_i\left(\text{Pr}(A|\text{no}), \text{lie}\right)$$

in any situation always.

As a result, there is no attainable model design.

However, under a utilitarian approach there should exist other survey designs that are consistent with truthful answers.

In particular, respondents will choose to answer truthfully if the relative advantage from lying is eliminated.

We can use Python to show that the optimal model design corresponds to point Q in Figure 4:

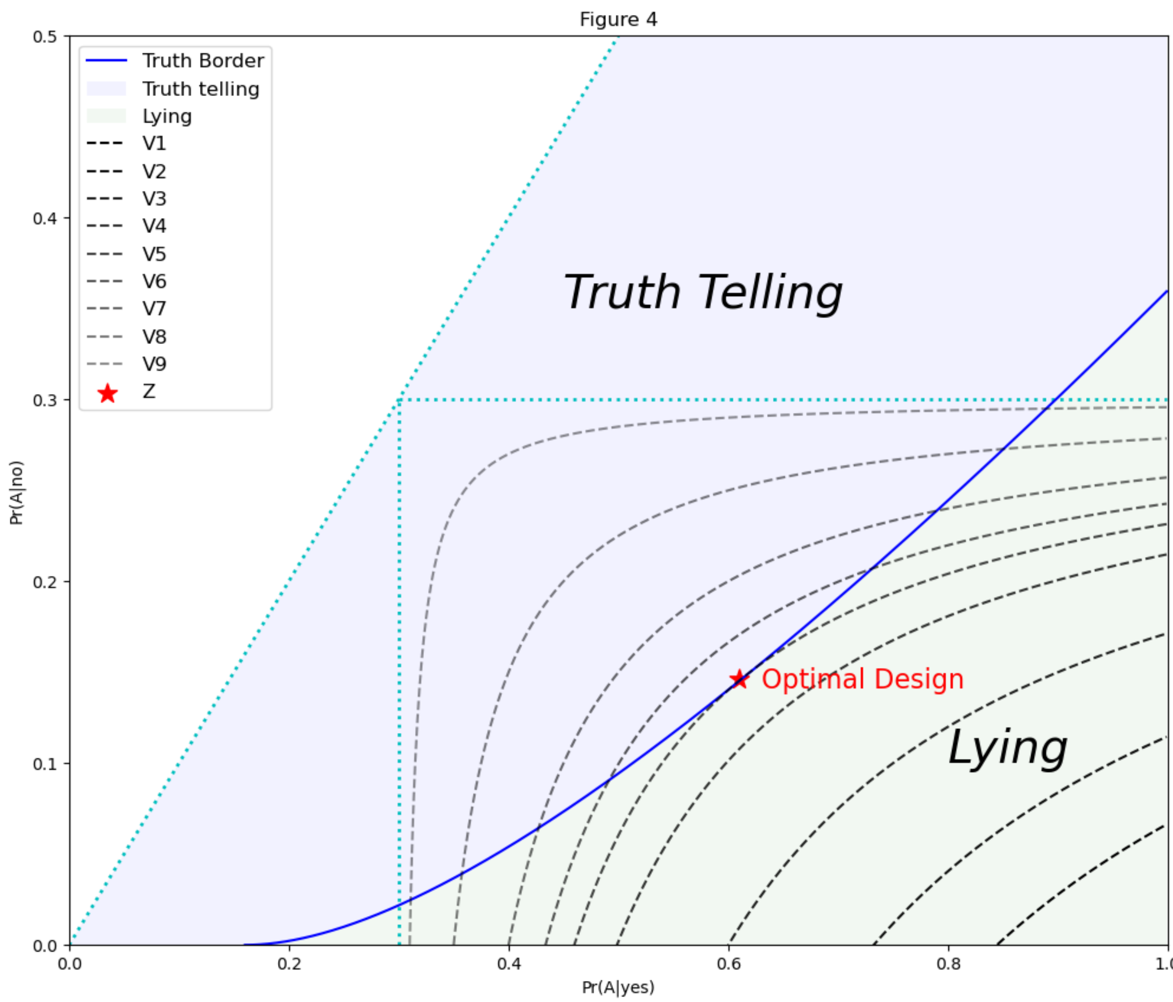
```python
def f(x):
    if x < 0.16:
        return 0
    else:
        return (pow(x, 0.5) - 0.4)**2
```

```python
pi = 0.3
n = 100
nv = [0.27, 0.34, 0.49, 0.74, 0.92, 1.1, 1.47, 2.94, 14.7]
x = np.arange(0, 1, 0.001)
y = [f(i) for i in x]
z = x
x0 = np.arange(pi, 1, 0.001)
x2 = np.arange(0, pi, 0.001)
y1 = [pi for i in x0]
y2 = [pi for i in x2]
x3 = np.arange(0.16, 1, 0.001)
y3 = (pow(x3, 0.5) - 0.4)**2

plt.figure(figsize=(12, 10))
plt.plot(x, x, 'c:', linewidth=2)
plt.plot(x0, y1,'c:', linewidth=2)
plt.plot(y2, x2,'c:', linewidth=2)
plt.plot(x3, y3,'b-', label='Truth Border')
plt.fill_between(x, y, z, facecolor='blue', alpha=0.05, label='Truth telling')
plt.fill_between(x3, 0, y3,facecolor='green', alpha=0.05, label='Lying')
for i in range(len(nv)):
    y = pi - (pi**2 * (1 - pi)**2) / (n * (nv[i] / n) * (x0 - pi + 1e-8))
    plt.plot(x0, y, 'k--', alpha=1 - 0.07 * i, label=f'V{i+1}')
plt.scatter(0.61, 0.146, c='r', marker='*', label='Z', s=150)
plt.xlim([0, 1])
plt.ylim([0, 0.5])
plt.xlabel('Pr(A|yes)')
```

(continues on next page)

(continued from previous page)

```
plt.ylabel('Pr(A|no)')
plt.text(0.45, 0.35, "Truth Telling", fontdict={'size':28, 'style':'italic'})
plt.text(0.8, 0.1, "Lying", fontdict={'size':28, 'style':'italic'})
plt.text(0.63, 0.141, "Optimal Design", fontdict={'size':16,'color':'r'})
plt.legend(loc=0, fontsize='large')
plt.title('Figure 4')
plt.show()
```



17.6.4 Method of Greenberg et al. (1977)

[Greenberg *et al.*, 1977]

Greenberg et al. (1977) defined the hazard for an individual in A as the probability that he or she is perceived as belonging to A:

$$\Pr(\text{yes}|A) \times \Pr(A|\text{yes}) + \Pr(\text{no}|A) \times \Pr(A|\text{no}) \tag{17.21}$$

The hazard for an individual who does not belong to A is

$$\Pr(\text{yes}|A^{'}) \times \Pr(A|\text{yes}) + \Pr(\text{no}|A^{'}) \times \Pr(A|\text{no}) \tag{17.22}$$

They also considered an alternative related measure of hazard that they said "is likely to be closer to the actual concern felt by a respondent."

Their "limited hazard" for an individual in A and A' is

$$\text{Pr}(\text{yes}|A) \times \text{Pr}(A|\text{yes}) \tag{17.23}$$

and

$$\text{Pr}(\text{yes}|A') \times \text{Pr}(A|\text{yes}) \tag{17.24}$$

According to Greenberg et al. (1977), a respondent commits himself or herself to answer truthfully on the basis of a probability in (17.21) or (17.23) **before** randomly selecting the question to be answered.

Suppose that the appropriate privacy measure is captured by the notion of "limited hazard" in (17.23) and (17.24).

Consider an unrelated question model where the unrelated question is replaced by the instruction "Say the word 'no'", which implies that

$$\text{Pr}(A|\text{yes}) = 1$$

and it follows that:

- Hazard for an individual in A' is 0.
- Hazard for an individual in A can also be made arbitrarily small by choosing a sufficiently small $\text{Pr}(\text{yes}|A)$.

Even though this hazard can be set arbitrarily close to 0, an individual in A will completely reveal his or her identity whenever truthfully answering the sensitive question.

However, under utilitarian framework, it is obviously contradictory.

If the individuals are willing to volunteer this information, it seems that the randomized response design was not necessary in the first place.

It ignores the fact that respondents retain the option of lying until they have seen the question to be answered.

17.7 Concluding Remarks

The justifications for a randomized response procedure are that

- Respondents are thought to feel discomfort from being perceived as belonging to the sensitive group.
- Respondents prefer to answer questions truthfully than to lie, unless it is too revealing.

If a privacy measure is not completely consistent with the rational behavior of the respondents, all efforts to derive an optimal model design are futile.

A utilitarian approach provides a systematic way to model respondents' behavior under the assumption that they maximize their expected utilities.

In a utilitarian analysis:

- A truth border divides the space of conditional probabilities of being perceived as belonging to the sensitive group, $\text{Pr}(A|\text{yes})$ and $\text{Pr}(A|\text{no})$, into the truth-telling region and the lying region.
- The optimal model design is obtained at the point where the truth border touches the lowest possible iso-variance curve.

A practical implication of the analysis of [Ljungqvist, 1993] is that uncertainty about respondents' demands for privacy can be acknowledged by **choosing Pr$(A|$yes$)$ and Pr$(A|$no$)$ sufficiently close to each other**.

Part III

Bayes Law

CHAPTER

EIGHTEEN

NON-CONJUGATE PRIORS

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

```
!pip install numpyro jax
```

This lecture is a sequel to the *Two Meanings of Probability*.

That lecture offers a Bayesian interpretation of probability in a setting in which the likelihood function and the prior distribution over parameters just happened to form a **conjugate** pair in which

- application of Bayes' Law produces a posterior distribution that has the same functional form as the prior

Having a likelihood and prior that are conjugate can simplify calculation of a posterior, facilitating analytical or nearly analytical calculations.

But in many situations the likelihood and prior need not form a conjugate pair.

- after all, a person's prior is his or her own business and would take a form conjugate to a likelihood only by remote coincidence

In these situations, computing a posterior can become very challenging.

In this lecture, we illustrate how modern Bayesians confront non-conjugate priors by using Monte Carlo techniques that involve

- first cleverly forming a Markov chain whose invariant distribution is the posterior distribution we want
- simulating the Markov chain until it has converged and then sampling from the invariant distribution to approximate the posterior

We shall illustrate the approach by deploying a powerful Python library, NumPyro that implements this approach.

As usual, we begin by importing some Python code.

```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import scipy.stats as st
```

(continues on next page)

(continued from previous page)

```
from typing import NamedTuple, Sequence
import jax.numpy as jnp
from jax import random

import numpyro
from numpyro import distributions as dist
import numpyro.distributions.constraints as constraints
from numpyro.infer import MCMC, NUTS, SVI, Trace_ELBO
from numpyro.optim import Adam
```

18.1 Unleashing MCMC on a binomial likelihood

This lecture begins with the binomial example in the *Two Meanings of Probability*.

That lecture computed a posterior

- analytically via choosing the conjugate priors,

This lecture instead computes posteriors

- numerically by sampling from the posterior distribution through MCMC methods, and
- using a variational inference (VI) approximation.

We use `numpyro` with assistance from `jax` to approximate a posterior distribution.

We use several alternative prior distributions.

We compare computed posteriors with ones associated with a conjugate prior as described in *Two Meanings of Probability*.

18.1.1 Analytical posterior

Assume that the random variable $X \sim Binom\left(n, \theta\right)$.

This defines a likelihood function

$$L\left(Y|\theta\right) = \text{Prob}(X = k|\theta) = \left(\frac{n!}{k!(n-k)!}\right)\theta^k(1-\theta)^{n-k}$$

where $Y = k$ is an observed data point.

We view θ as a random variable for which we assign a prior distribution having density $f(\theta)$.

We will try alternative priors later, but for now, suppose the prior is distributed as $\theta \sim Beta\left(\alpha, \beta\right)$, i.e.,

$$f(\theta) = \text{Prob}(\theta) = \frac{\theta^{\alpha-1}(1-\theta)^{\beta-1}}{B(\alpha, \beta)}$$

We choose this as our prior for now because we know that a conjugate prior for the binomial likelihood function is a beta distribution.

After observing k successes among N sample observations, the posterior probability distribution of θ is

$$\text{Prob}(\theta|k) = \frac{\text{Prob}(\theta, k)}{\text{Prob}(k)} = \frac{\text{Prob}(k|\theta)\text{Prob}(\theta)}{\text{Prob}(k)} = \frac{\text{Prob}(k|\theta)\text{Prob}(\theta)}{\int_0^1 \text{Prob}(k|\theta)\text{Prob}(\theta)d\theta}$$

$$= \frac{\binom{N}{k}(1-\theta)^{N-k}\theta^k \frac{\theta^{\alpha-1}(1-\theta)^{\beta-1}}{B(\alpha,\beta)}}{\int_0^1 \binom{N}{k}(1-\theta)^{N-k}\theta^k \frac{\theta^{\alpha-1}(1-\theta)^{\beta-1}}{B(\alpha,\beta)} d\theta}$$

$$= \frac{(1-\theta)^{\beta+N-k-1}\theta^{\alpha+k-1}}{\int_0^1 (1-\theta)^{\beta+N-k-1}\theta^{\alpha+k-1} d\theta}.$$

Thus,

$$\text{Prob}(\theta|k) \sim Beta(\alpha+k, \beta+N-k)$$

The analytical posterior for a given conjugate beta prior is coded in the following

```
def simulate_draw(θ, n):
    """Draws a Bernoulli sample of size n with probability P(Y=1) = θ"""
    rand_draw = np.random.rand(n)
    draw = (rand_draw < θ).astype(int)
    return draw


def analytical_beta_posterior(data, α0, β0):
    """
    Computes analytically the posterior distribution
    with beta prior parametrized by (α, β)
    given # num observations

    Parameters
    ---------
    num : int.
        the number of observations after which we calculate the posterior
    α0, β0 : float.
        the parameters for the beta distribution as a prior

    Returns
    ---------
    The posterior beta distribution
    """
    num = len(data)
    up_num = data.sum()
    down_num = num - up_num
    return st.beta(α0 + up_num, β0 + down_num)
```

18.1.2 Two ways to approximate posteriors

Suppose that we don't have a conjugate prior.

Then we can't compute posteriors analytically.

Instead, we use computational tools to approximate the posterior distribution for a set of alternative prior distributions using `numpyro`.

We first use the **Markov Chain Monte Carlo** (MCMC) algorithm.

We implement the NUTS sampler to sample from the posterior.

In that way we construct a sampling distribution that approximates the posterior.

After doing that we deploy another procedure called **Variational Inference** (VI).

In particular, we implement Stochastic Variational Inference (SVI) machinery in `numpyro`.

The MCMC algorithm supposedly generates a more accurate approximation since in principle it directly samples from the posterior distribution.

But it can be computationally expensive, especially when dimension is large.

A VI approach can be cheaper, but it is likely to produce an inferior approximation to the posterior, for the simple reason that it requires guessing a parametric **guide functional form** that we use to approximate a posterior.

This guide function is likely at best to be an imperfect approximation.

By paying the cost of restricting the putative posterior to have a restricted functional form, the problem of approximating a posterior is transformed to a well-posed optimization problem that seeks parameters of the putative posterior that minimize a Kullback-Leibler (KL) divergence between true posterior and the putative posterior distribution.

- minimizing the KL divergence is equivalent to maximizing a criterion called the **Evidence Lower Bound** (ELBO), as we shall verify soon.

18.2 Prior distributions

In order to be able to apply MCMC sampling or VI, `numpyro` requires that a prior distribution satisfy special properties:

- we must be able to sample from it;
- we must be able to compute the log pdf pointwise;
- the pdf must be differentiable with respect to the parameters.

We'll want to define a distribution `class`.

We will use the following priors:

- a uniform distribution on $[\underline{\theta}, \overline{\theta}]$, where $0 \leq \underline{\theta} < \overline{\theta} \leq 1$.
- a truncated log-normal distribution with support on $[0, 1]$ with parameters (μ, σ).
  - To implement this, let $Z \sim N(\mu, \sigma)$ and $\tilde{Z}$ be truncated normal with support $[-\infty, \log(1)]$, then $\exp(Z)$ has a log normal distribution with bounded support $[0, 1]$. This can be easily coded since `numpyro` has a built-in truncated normal distribution, and `numpyro`'s `TransformedDistribution` class that includes an exponential transformation.
- a shifted von Mises distribution that has support confined to $[0, 1]$ with parameter (μ, κ).
  - Let $X \sim vonMises(0, \kappa)$. We know that X has bounded support $[-\pi, \pi]$. We can define a shifted von Mises random variable $\tilde{X} = a + bX$ where $a = 0.5, b = 1/(2\pi)$ so that $\tilde{X}$ is supported on $[0, 1]$.
  - This can be implemented using `numpyro`'s `TransformedDistribution` class with its `AffineTransform` method.
- a truncated Laplace distribution.
  - We also considered a truncated Laplace distribution because its density comes in a piece-wise non-smooth form and has a distinctive spiked shape.
  - The truncated Laplace can be created using `numpyro`'s `TruncatedDistribution` class.

```
def truncated_log_normal_trans(loc, scale):
    """
    Obtains the truncated log normal distribution
    using numpyro's TruncatedNormal and ExpTransform
    """
    base_dist = dist.TruncatedNormal(
```

(continues on next page)

(continued from previous page)

```
        low=-jnp.inf, high=jnp.log(1), loc=loc, scale=scale
    )
    return dist.TransformedDistribution(
        base_dist, dist.transforms.ExpTransform()
    )


def shifted_von_mises(κ):
    """Obtains the shifted von Mises distribution using AffineTransform"""
    base_dist = dist.VonMises(0, κ)
    return dist.TransformedDistribution(
        base_dist,
        dist.transforms.AffineTransform(loc=0.5, scale=1 / (2 * jnp.pi))
    )


def truncated_laplace(loc, scale):
    """Obtains the truncated Laplace distribution on [0,1]"""
    base_dist = dist.Laplace(loc, scale)
    return dist.TruncatedDistribution(base_dist, low=0.0, high=1.0)
```

18.2.1 Variational inference

Instead of directly sampling from the posterior, the **variational inference** method approximates an unknown posterior distribution with a family of tractable distributions/densities.

It then seeks to minimize a measure of statistical discrepancy between the approximating and true posteriors.

Thus variational inference (VI) approximates a posterior by solving a minimization problem.

Let the latent parameter/variable that we want to infer be θ.

Let the prior be $p(\theta)$ and the likelihood be $p\left(Y|\theta\right)$.

We want $p\left(\theta|Y\right)$.

Bayes' rule implies

$$p\left(\theta|Y\right)=\frac{p\left(Y,\theta\right)}{p\left(Y\right)}=\frac{p\left(Y|\theta\right)p\left(\theta\right)}{p\left(Y\right)}$$

where

$$p\left(Y\right)=\int p\left(Y\mid\theta\right)p\left(\theta\right)d\theta. \tag{18.1}$$

The integral on the right side of (18.1) is typically difficult to compute.

Consider a **guide distribution** $q_{\phi}(\theta)$ parameterized by ϕ that we'll use to approximate the posterior.

We choose parameters ϕ of the guide distribution to minimize a Kullback-Leibler (KL) divergence between the approximate posterior $q_{\phi}(\theta)$ and the posterior:

$$D_{KL}(q(\theta;\phi)\;\|\;p(\theta\mid Y))\equiv-\int q(\theta;\phi)\log\frac{p(\theta\mid Y)}{q(\theta;\phi)}d\theta$$

Thus, we want a **variational distribution** q that solves

$$\min_{\phi}\quad D_{KL}(q(\theta;\phi)\;\|\;p(\theta\mid Y))$$

Note that

$$
\begin{aligned}
D_{KL}(q(\theta;\phi)\parallel p(\theta\mid Y)) &= -\int q(\theta;\phi)\log\frac{P(\theta\mid Y)}{q(\theta;\phi)}d\theta \\
&= -\int q(\theta)\log\frac{\frac{p(\theta,Y)}{p(Y)}}{q(\theta)}d\theta \\
&= -\int q(\theta)\log\frac{p(\theta,Y)}{q(\theta)p(Y)}d\theta \\
&= -\int q(\theta)\left[\log\frac{p(\theta,Y)}{q(\theta)}-\log p(Y)\right]d\theta \\
&= -\int q(\theta)\log\frac{p(\theta,Y)}{q(\theta)}+\int q(\theta)\log p(Y)d\theta \\
&= -\int q(\theta)\log\frac{p(\theta,Y)}{q(\theta)}d\theta+\log p(Y) \\
\log p(Y) &= D_{KL}(q(\theta;\phi)\parallel p(\theta\mid Y))+\int q_\phi(\theta)\log\frac{p(\theta,Y)}{q_\phi(\theta)}d\theta
\end{aligned}
$$

For observed data Y, $p(\theta, Y)$ is a constant, so minimizing KL divergence is equivalent to maximizing

$$
ELBO\equiv\int q_\phi(\theta)\log\frac{p(\theta,Y)}{q_\phi(\theta)}d\theta=\mathbb{E}_{q_\phi(\theta)}\left[\log p(\theta,Y)-\log q_\phi(\theta)\right] \tag{18.2}
$$

Formula (18.2) is called the evidence lower bound (ELBO).

A standard optimization routine can be used to search for the optimal ϕ in our parametrized distribution $q_\phi(\theta)$.

The parameterized distribution $q_\phi(\theta)$ is called the **variational distribution**.

We can implement Stochastic Variational Inference (SVI) in numpyro using the `Adam` gradient descent algorithm to approximate the posterior.

We use two sets of variational distributions: Beta and TruncatedNormal with support $[0, 1]$

- Learnable parameters for the Beta distribution are (α, β), both of which are positive.
- Learnable parameters for the Truncated Normal distribution are (loc, scale).

Note

We restrict the truncated Normal parameter 'loc' to be in the interval $[0, 1]$

18.3 Implementation

We have constructed a Python class `BayesianInference` that requires the following arguments to be initialized:

- `param`: a tuple/scalar of parameters dependent on distribution types
- `name_dist`: a string that specifies distribution names

The (`param`, `name_dist`) pair includes:

- (α, β, 'beta')
- (lower_bound, upper_bound, 'uniform')
- (loc, scale, 'lognormal')

  - Note: This is the truncated log normal.
- (κ, 'vonMises'), where κ denotes concentration parameter, and center location is set to 0.5. Using `numpyro`, this is the **shifted** distribution.
- (loc, scale, 'laplace')
  - Note: This is the truncated Laplace

The class `BayesianInference` has several key methods :

- `sample_prior`:
  - This can be used to draw a single sample from the given prior distribution.
- `show_prior`:
  - Plots the approximate prior distribution by repeatedly drawing samples and fitting a kernel density curve.
- `mcmc_sampling`:
  - INPUT: (data, num_samples, num_warmup=1000)
  - Takes a `jnp.array` data and generates MCMC sampling of posterior of size `num_samples`.
- `svi_run`:
  - INPUT: (data, guide_dist, n_steps=10000)
  - guide_dist = 'normal' - use a **truncated** normal distribution as the parametrized guide
  - guide_dist = 'beta' - use a beta distribution as the parametrized guide
  - RETURN: (params, losses) - the learned parameters in a `dict` and the vector of loss at each step.

```
class BayesianInference(NamedTuple):
    """
    Parameters
    ----------
    param : tuple.
        a tuple object that contains all relevant parameters for the distribution
    name_dist : str.
        name of the distribution - 'beta', 'uniform', 'lognormal', 'vonMises',
↪'laplace'
    rng_key : jax.random.PRNGKey
        PRNG key for random number generation.
    """
    param: tuple
    name_dist: str
    rng_key: random.PRNGKey


def create_bayesian_inference(
    param: tuple,
    name_dist: str,
    seed: int = 0
) -> BayesianInference:
    """Factory function to create a BayesianInference instance"""

    rng_key = random.PRNGKey(seed)

    return BayesianInference(
        param=param,
        name_dist=name_dist,
```

(continues on next page)

(continued from previous page)

```
        rng_key=rng_key
    )


def sample_prior(model: BayesianInference):
    """Define the prior distribution to sample from in numpyro models."""
    if model.name_dist == "beta":
        # unpack parameters
        α0, β0 = model.param
        sample = numpyro.sample(
            "theta", dist.Beta(α0, β0), rng_key=model.rng_key
        )

    elif model.name_dist == "uniform":
        # unpack parameters
        lb, ub = model.param
        sample = numpyro.sample(
            "theta", dist.Uniform(lb, ub), rng_key=model.rng_key
        )

    elif model.name_dist == "lognormal":
        # unpack parameters
        loc, scale = model.param
        sample = numpyro.sample(
            "theta",
            truncated_log_normal_trans(loc, scale),
            rng_key=model.rng_key
        )

    elif model.name_dist == "vonMises":
        # unpack parameters
        κ = model.param
        sample = numpyro.sample(
            "theta", shifted_von_mises(κ), rng_key=model.rng_key
        )

    elif model.name_dist == "laplace":
        # unpack parameters
        loc, scale = model.param
        sample = numpyro.sample(
            "theta", truncated_laplace(loc, scale), rng_key=model.rng_key
        )

    return sample


def show_prior(
    model: BayesianInference, size=1e5, bins=20, disp_plot=1
):
    """
    Visualizes prior distribution by sampling from prior
    and plots the approximated sampling distribution
    """
    with numpyro.plate("show_prior", size=size):
        sample = sample_prior(model)
    # to JAX array
    sample_array = jnp.asarray(sample)
```

(continues on next page)

(continued from previous page)

```python
    # plot histogram and kernel density
    if disp_plot == 1:
        sns.displot(
            sample_array,
            kde=True,
            stat="density",
            bins=bins,
            height=5,
            aspect=1.5
        )
        plt.xlim(0, 1)
        plt.show()
    else:
        return sample_array


def set_model(model: BayesianInference, data):
    """
    Define the probabilistic model by specifying prior,
    conditional likelihood, and data conditioning
    """
    theta = sample_prior(model)
    output = numpyro.sample(
        "obs", dist.Binomial(len(data), theta), obs=jnp.sum(data)
    )


def mcmc_sampling(
    model: BayesianInference, data, num_samples, num_warmup=1000
):
    """
    Computes numerically the posterior distribution
    with beta prior parametrized by (α0, β0)
    given data using MCMC
    """
    data = jnp.array(data, dtype=float)
    nuts_kernel = NUTS(set_model)
    mcmc = MCMC(
        nuts_kernel,
        num_samples=num_samples,
        num_warmup=num_warmup,
        progress_bar=False,
    )
    mcmc.run(model.rng_key, model=model, data=data)

    samples = mcmc.get_samples()["theta"]
    return samples


# arguments in this function are used to align with the arguments in set_model()
# this is required by svi.run()
def beta_guide(model: BayesianInference, data):
    """
    Defines the candidate parametrized variational distribution
    that we train to approximate posterior with numpyro
    Here we use parameterized beta
```

(continues on next page)

(continued from previous page)

```
    """
    α_q = numpyro.param("alpha_q", 10, constraint=constraints.positive)
    β_q = numpyro.param("beta_q", 10, constraint=constraints.positive)

    numpyro.sample("theta", dist.Beta(α_q, β_q))


# similar with beta_guide()
def truncnormal_guide(model: BayesianInference, data):
    """
    Defines the candidate parametrized variational distribution
    that we train to approximate posterior with numpyro
    Here we use truncated normal on [0,1]
    """
    loc = numpyro.param("loc", 0.5, constraint=constraints.interval(0.0, 1.0))
    scale = numpyro.param("scale", 1, constraint=constraints.positive)
    numpyro.sample(
        "theta",
        dist.TruncatedNormal(loc, scale, low=0.0, high=1.0)
    )


def svi_init(model: BayesianInference, guide_dist, lr=0.0005):
    """Initiate SVI training mode with Adam optimizer"""
    adam_params = {"lr": lr}

    if guide_dist == "beta":
        optimizer = Adam(step_size=lr)
        svi = SVI(
            set_model, beta_guide, optimizer, loss=Trace_ELBO()
        )
    elif guide_dist == "normal":
        optimizer = Adam(step_size=lr)
        svi = SVI(
            set_model, truncnormal_guide, optimizer, loss=Trace_ELBO()
        )
    else:
        print("WARNING: Please input either 'beta' or 'normal'")
        svi = None

    return svi


def svi_run(model: BayesianInference, data, guide_dist, n_steps=10000):
    """
    Runs SVI and returns optimized parameters and losses

    Returns
    --------
    params : the learned parameters for guide
    losses : a vector of loss at each step
    """

    # initiate SVI
    svi = svi_init(model, guide_dist)

    data = jnp.array(data, dtype=float)
```

(continues on next page)

(continued from previous page)

```
    result = svi.run(
        model.rng_key, n_steps, model, data, progress_bar=False
        )
    params = dict(
        (key, jnp.asarray(value)) for key, value in result.params.items()
    )
    losses = jnp.asarray(result.losses)

    return params, losses
```

18.4 Alternative prior distributions

Let's see how well our sampling algorithm does in approximating

- a log normal distribution
- a uniform distribution

To examine our alternative prior distributions, we'll plot approximate prior distributions below by calling the `show_prior` method.

```
# truncated log normal
example_ln = create_bayesian_inference(param=(0, 2), name_dist="lognormal")
show_prior(example_ln, size=100000, bins=20)
```

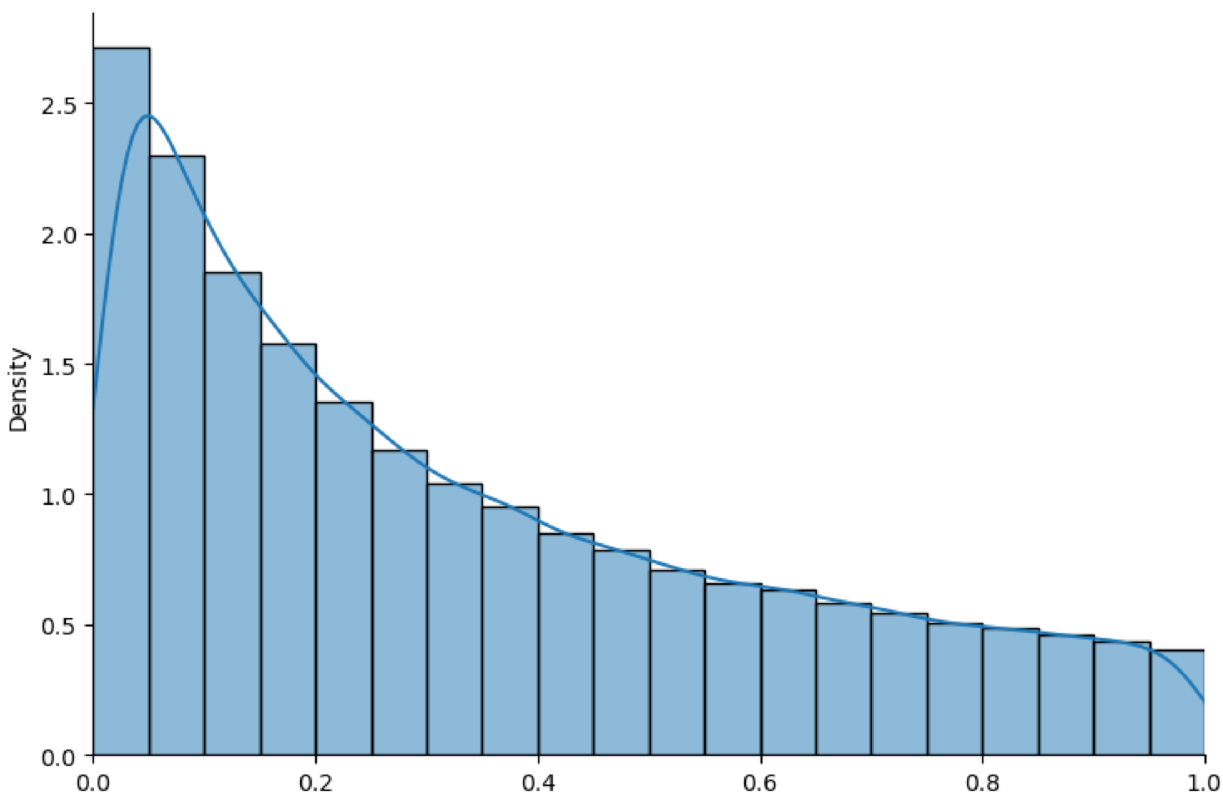


Fig. 18.1: Truncated log normal distribution

```
# truncated uniform
example_un = create_bayesian_inference(param=(0.1, 0.8), name_dist="uniform")
show_prior(example_un, size=100000, bins=20)
```

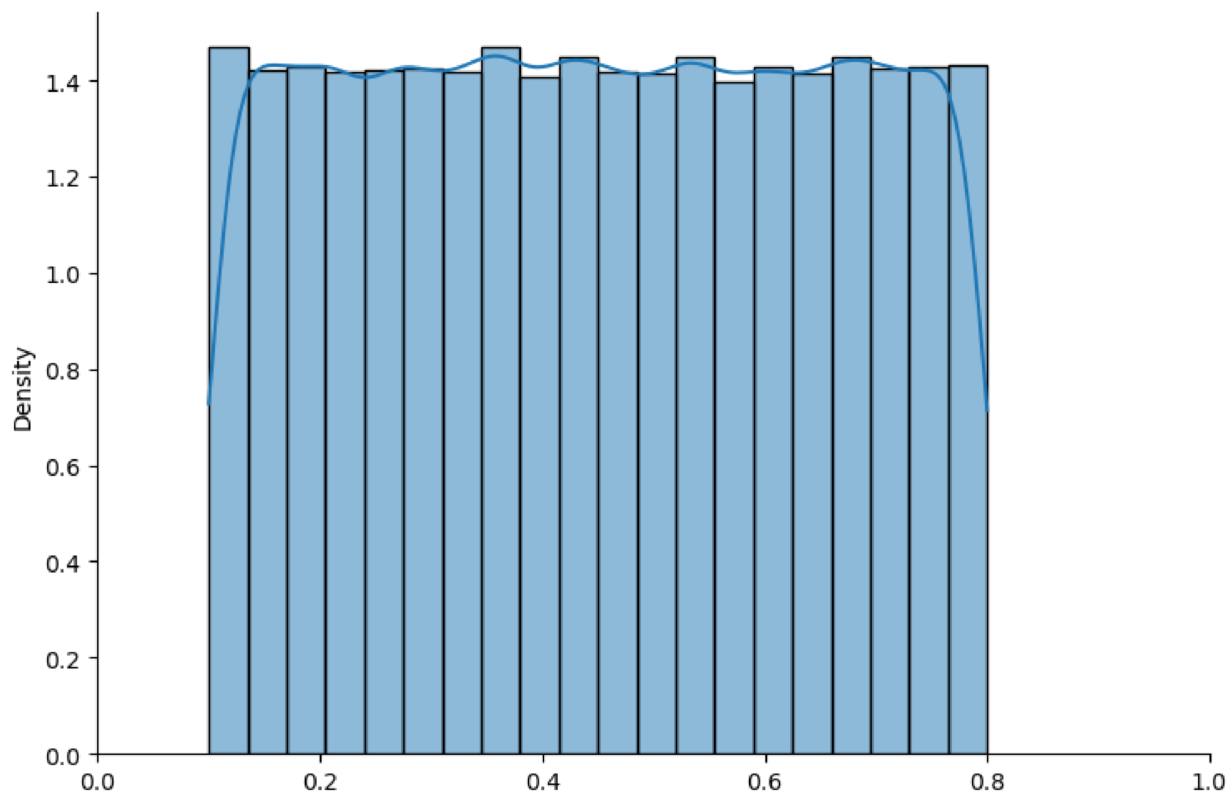


Fig. 18.2: Truncated uniform distribution

The above graphs show that sampling seems to work well with both distributions.

Now let's see how well things work with von Mises distributions.

```
# shifted von Mises
example_vm = create_bayesian_inference(param=10, name_dist="vonMises")
show_prior(example_vm, size=100000, bins=20)
```

The graphs look good too.

Now let's try with a Laplace distribution.

```
# truncated Laplace
example_lp = create_bayesian_inference(param=(0.5, 0.05), name_dist="laplace")
show_prior(example_lp, size=100000, bins=20)
```

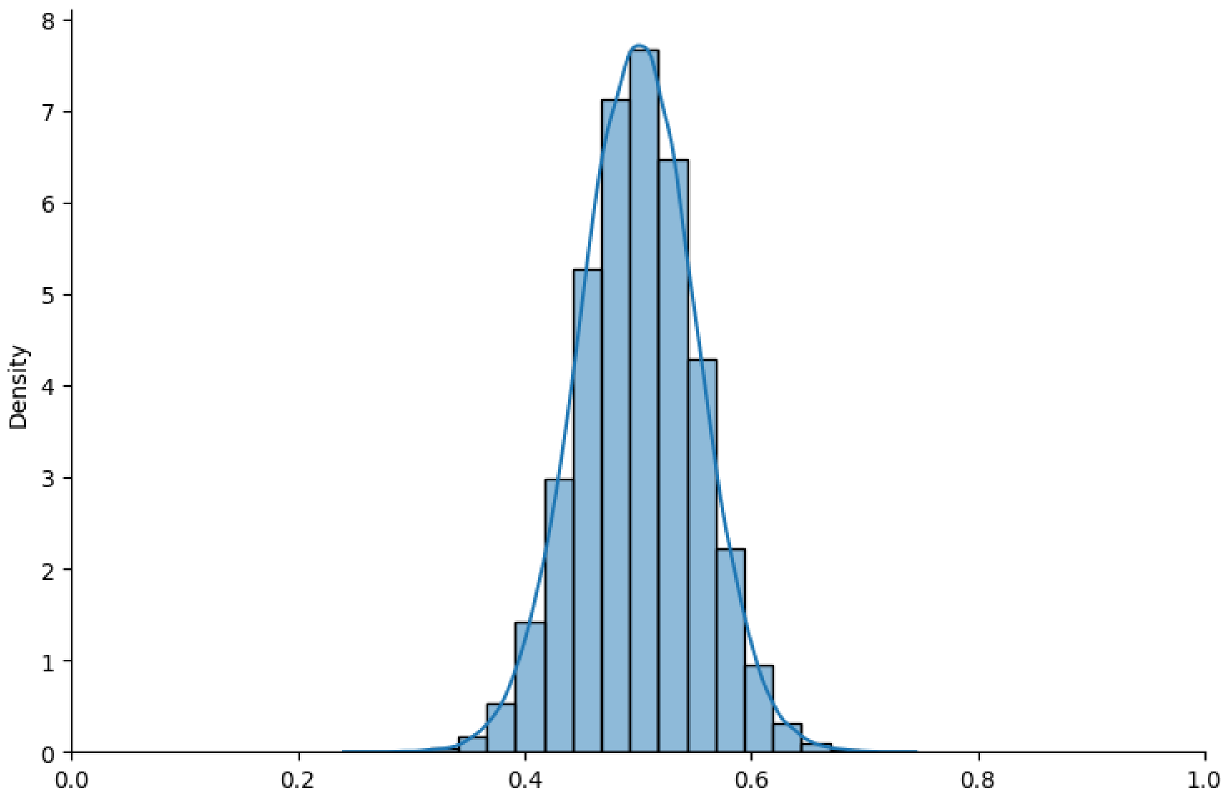


Fig. 18.3: Shifted von Mises distribution

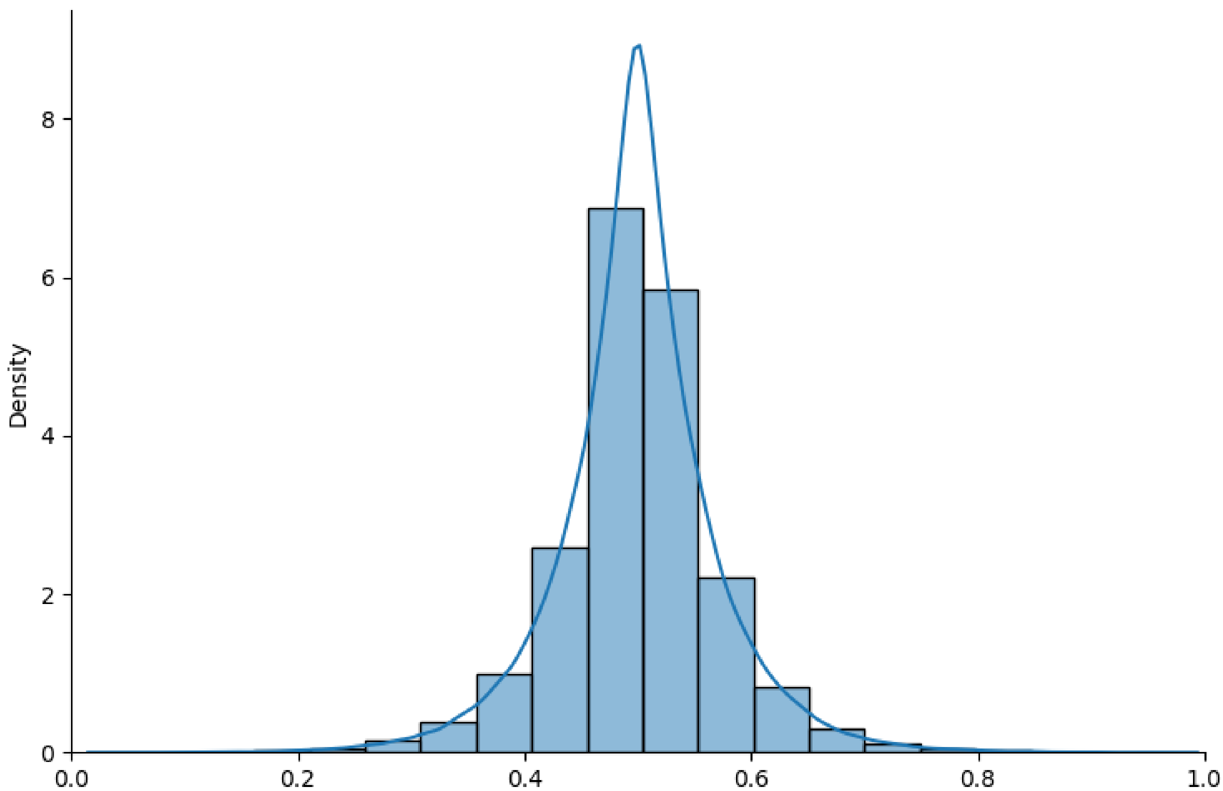


Fig. 18.4: Truncated Laplace distribution

Having assured ourselves that our sampler seems to do a good job, let's put it to work in using MCMC to compute posterior probabilities.

18.5 Posteriors via MCMC and VI

We construct a class `BayesianInferencePlot` to implement MCMC or VI algorithms and plot multiple posteriors for different updating data sizes and different possible priors.

This class takes as inputs the true data generating parameter `θ`, a list of updating data sizes for multiple posterior plotting, and a defined and parametrized `BayesianInference` class.

It has two key methods:

- `BayesianInferencePlot.mcmc_plot()` takes desired MCMC sample size as input and plots the output posteriors together with the prior defined in `BayesianInference` class.
- `BayesianInferencePlot.svi_plot()` takes desired VI distribution class ('beta' or 'normal') as input and plots the posteriors together with the prior.

```
class BayesianInferencePlot(NamedTuple):
    """
    Easily implement the MCMC and VI inference for a given instance of
    BayesianInference class and plot the prior together with multiple posteriors

    Parameters
    ----------
    θ : float.
        the true DGP parameter
    N_list : list.
        a list of sample size
    bayesian_model : BayesianInference.
        a class initiated using create_bayesian_inference()
    binwidth : float.
        plotting parameter for histogram bin width
    linewidth : float.
        plotting parameter for line width
    colorlist : list.
        list of colors for plotting
    N_max : int.
        maximum sample size
    data : np.ndarray.
        generated data array
    """
    θ: float
    N_list: Sequence[int]
    bayesian_model: BayesianInference
    binwidth: float
    linewidth: float
    colorlist: list
    N_max: int
    data: np.ndarray


def create_bayesian_inference_plot(
    θ: float,
    N_list: Sequence[int],
    bayesian_model: BayesianInference,
```

(continues on next page)

(continued from previous page)

```
    *,
    binwidth: float = 0.02,
    linewidth: float = 0.05,
) -> BayesianInferencePlot:
    """Factory function to create a BayesianInferencePlot instance"""

    colorlist = sns.color_palette(n_colors=len(N_list))
    N_max = int(max(N_list))
    data = simulate_draw(θ, N_max)
    return BayesianInferencePlot(
        θ=θ,
        N_list=list(map(int, N_list)),
        bayesian_model=bayesian_model,
        binwidth=binwidth,
        linewidth=linewidth,
        colorlist=colorlist,
        N_max=N_max,
        data=data,
    )


def mcmc_plot(
    plot_model: BayesianInferencePlot, num_samples, num_warmup=1000
):
    fig, ax = plt.subplots()

    # plot prior
    prior_sample = show_prior(
        plot_model.bayesian_model, disp_plot=0
        )
    sns.histplot(
        data=prior_sample,
        kde=True,
        stat="density",
        binwidth=plot_model.binwidth,
        color="#4C4E52",
        linewidth=plot_model.linewidth,
        alpha=0.1,
        ax=ax,
        label="Prior distribution",
    )

    # plot posteriors
    for id, n in enumerate(plot_model.N_list):
        samples = mcmc_sampling(
            plot_model.bayesian_model,
            plot_model.data[:n],
            num_samples,
            num_warmup
        )
        sns.histplot(
            samples,
            kde=True,
            stat="density",
            binwidth=plot_model.binwidth,
            linewidth=plot_model.linewidth,
            alpha=0.2,
```

(continues on next page)

(continued from previous page)

```
            color=plot_model.colorlist[id - 1],
            label=f"Posterior with $n={n}$",
        )
    ax.legend(loc="upper left")
    plt.xlim(0, 1)
    plt.show()


def svi_fitting(guide_dist, params):
    """Fit the beta/truncnormal curve using parameters trained by SVI."""
    # create x axis
    xaxis = jnp.linspace(0, 1, 1000)
    if guide_dist == "beta":
        y = st.beta.pdf(xaxis, a=params["alpha_q"], b=params["beta_q"])

    elif guide_dist == "normal":
        # rescale upper/lower bound. See Scipy's truncnorm doc
        lower, upper = (0, 1)
        loc, scale = params["loc"], params["scale"]
        a, b = (lower - loc) / scale, (upper - loc) / scale

        y = st.truncnorm.pdf(
            xaxis, a=a, b=b, loc=loc, scale=scale
        )
    return (xaxis, y)


def svi_plot(
    plot_model: BayesianInferencePlot, guide_dist, n_steps=2000
):
    fig, ax = plt.subplots()

    # plot prior
    prior_sample = show_prior(plot_model.bayesian_model, disp_plot=0)
    sns.histplot(
        data=prior_sample,
        kde=True,
        stat="density",
        binwidth=plot_model.binwidth,
        color="#4C4E52",
        linewidth=plot_model.linewidth,
        alpha=0.1,
        ax=ax,
        label="Prior distribution",
    )

    # plot posteriors
    for id, n in enumerate(plot_model.N_list):
        (params, losses) = svi_run(
            plot_model.bayesian_model, plot_model.data[:n], guide_dist, n_steps
        )
        x, y = svi_fitting(guide_dist, params)
        ax.plot(
            x,
            y,
            alpha=1,
            color=plot_model.colorlist[id - 1],
```

(continues on next page)

(continued from previous page)

```
            label=f"Posterior with $n={n}$",
        )
    ax.legend(loc="upper left")
    plt.xlim(0, 1)
    plt.show()
```

Let's set some parameters that we'll use in all of the examples below.

To save computer time at first, notice that we'll set `mcmc_num_samples = 2000` and `svi_num_steps = 5000`.

(Later, to increase accuracy of approximations, we'll want to increase these.)

```
num_list = [5, 10, 50, 100, 1000]
mcmc_num_samples = 2000
svi_num_steps = 5000

# θ is the data generating process
true_θ = 0.8
```

18.5.1 Beta prior and posteriors:

Let's compare outcomes when we use a Beta prior.

For the same Beta prior, we shall

- compute posteriors analytically
- compute posteriors using MCMC using `numpyro`.
- compute posteriors using VI using `numpyro`.

Let's start with the analytical method that we described in this *Two Meanings of Probability*

```
# first examine Beta prior
beta = create_bayesian_inference(param=(5, 5), name_dist="beta")

beta_plot = create_bayesian_inference_plot(true_θ, num_list, beta)

# plot analytical Beta prior and posteriors
xaxis = jnp.linspace(0, 1, 1000)
y_prior = st.beta.pdf(xaxis, 5, 5)

fig, ax = plt.subplots()
# plot analytical beta prior
ax.plot(xaxis, y_prior, label="Analytical Beta prior", color="#4C4E52")

data, colorlist, N_list = beta_plot.data, beta_plot.colorlist, beta_plot.N_list

# Plot analytical beta posteriors
for id, n in enumerate(N_list):
    func = analytical_beta_posterior(data[:n], α0=5, β0=5)
    y_posterior = func.pdf(xaxis)
    ax.plot(
        xaxis,
        y_posterior,
        color=colorlist[id - 1],
        label=f"Analytical Beta posterior with $n={n}$",
```

(continues on next page)

(continued from previous page)

```
    )
ax.legend(loc="upper left")
plt.xlim(0, 1)
plt.show()
```

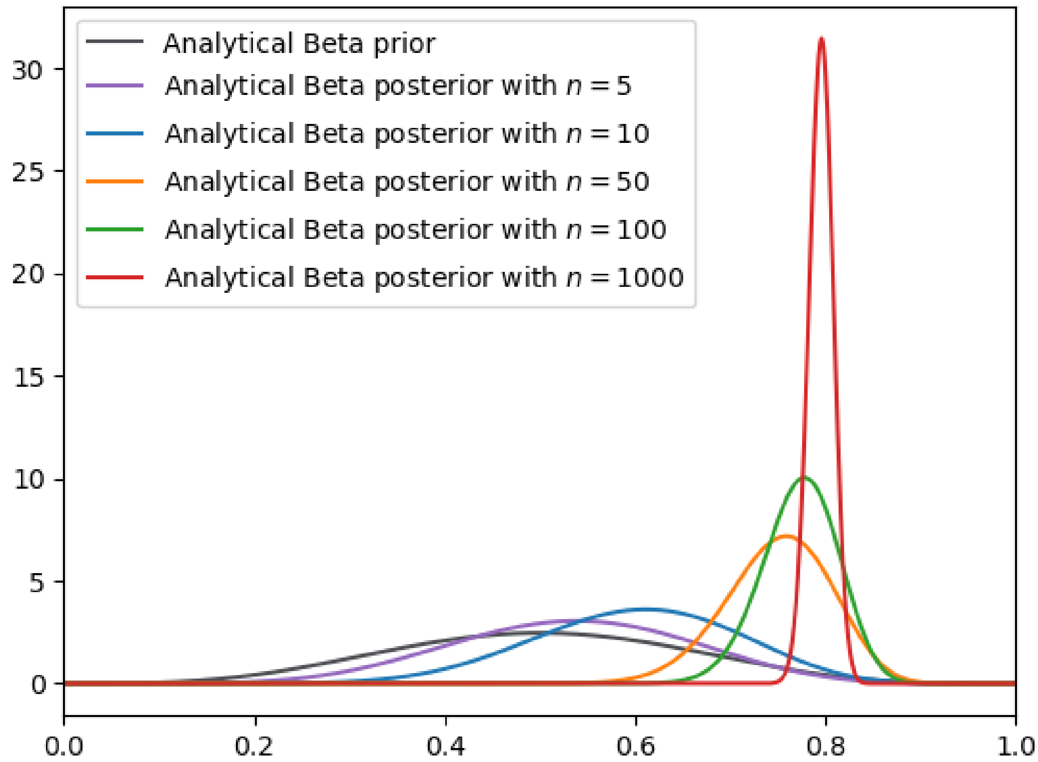


Fig. 18.5: Analytical density (Beta prior)

Now let's use MCMC while still using a beta prior.

We'll do this for both MCMC and VI.

```
mcmc_plot(
    beta_plot, num_samples=mcmc_num_samples
)
```

```
svi_plot(
    beta_plot, guide_dist="beta", n_steps=svi_num_steps
)
```

Here the MCMC approximation looks good.

But the VI approximation doesn't look so good.

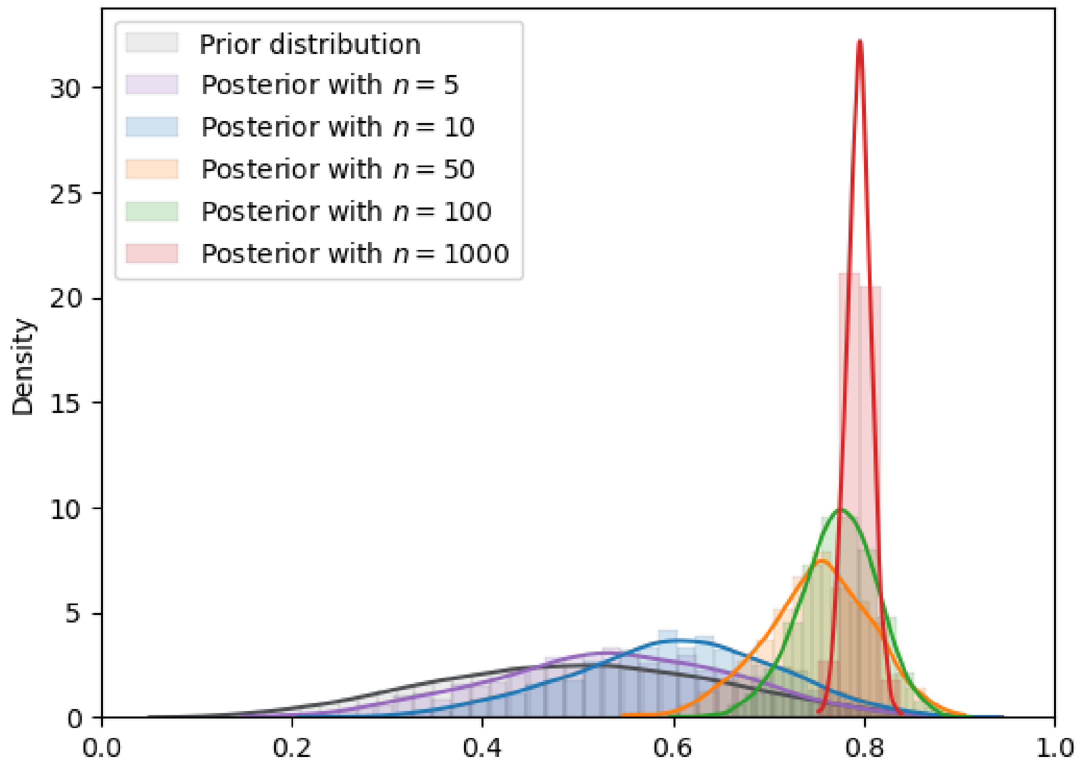


Fig. 18.6: MCMC density (Beta prior)

- even though we use the beta distribution as our guide, the VI approximated posterior distributions do not closely resemble the posteriors that we had just computed analytically.

(Here, our initial parameter for Beta guide is (0.5, 0.5).)

But if we increase the number of steps from 5000 to 100000 in VI as we now shall do, we'll get VI-approximated posteriors that will be more accurate, as we shall see next.

(Increasing the step size increases computational time though).

```
svi_plot(
    beta_plot, guide_dist="beta", n_steps=100000
)
```

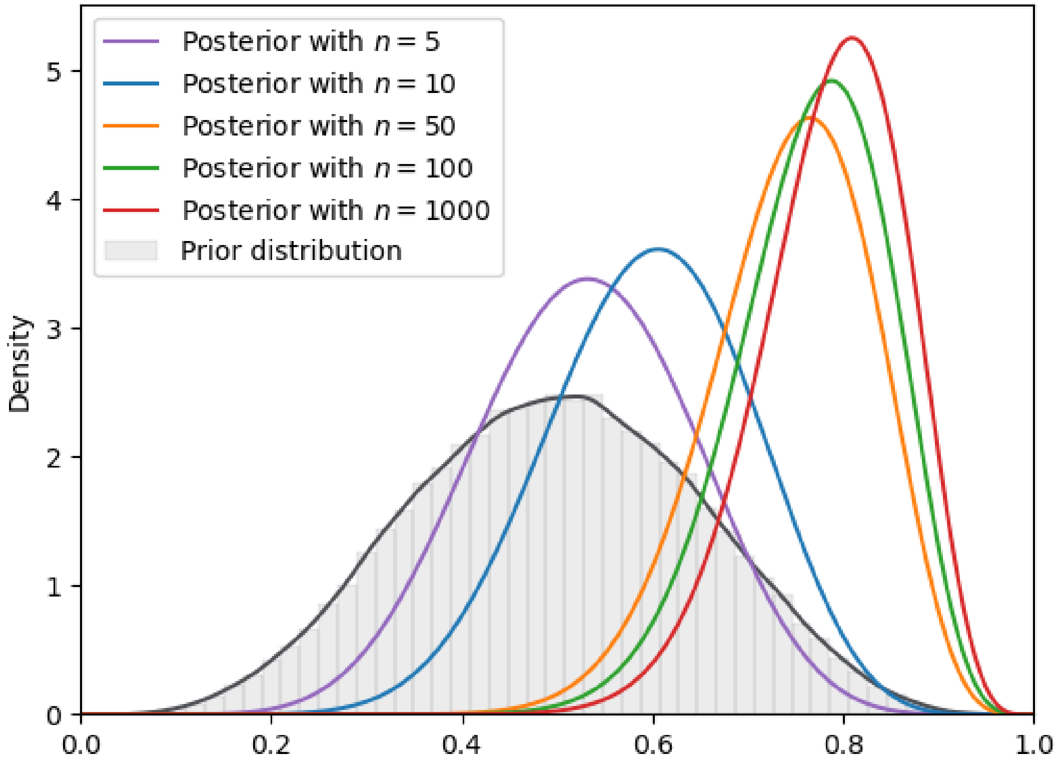
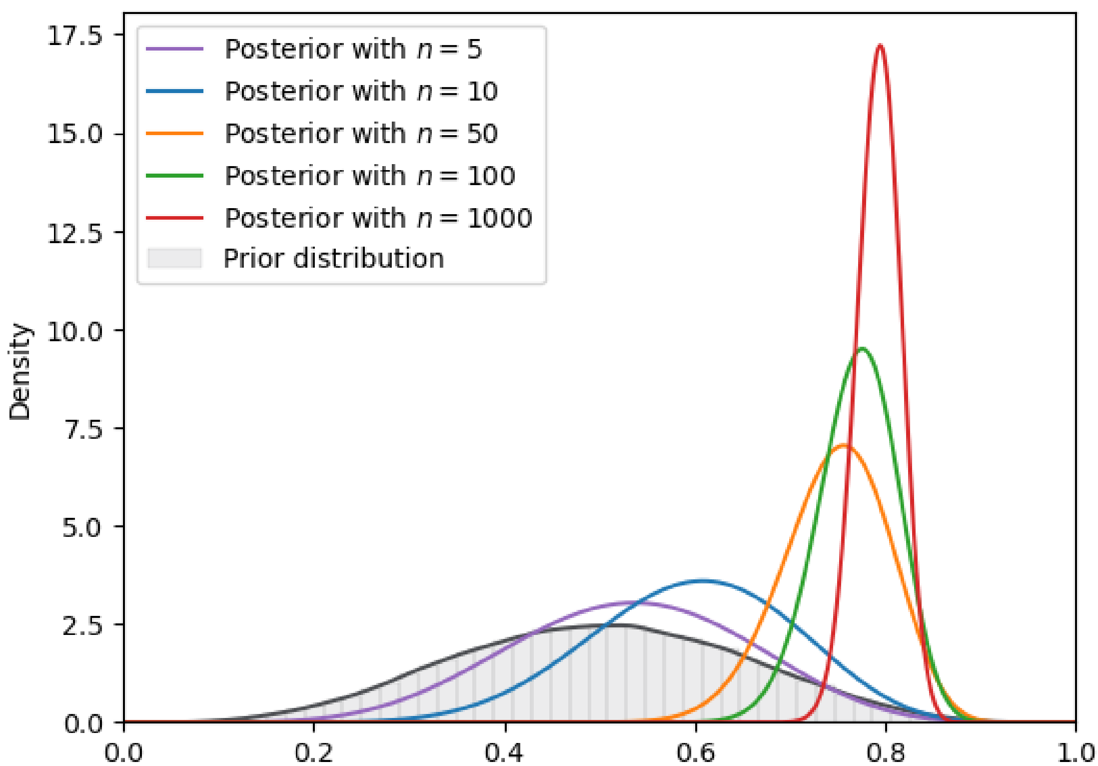


Fig. 18.7: SVI density (Beta prior, Beta guide)



18.6 Non-conjugate prior distributions

Having assured ourselves that our MCMC and VI methods can work well when we have a conjugate prior and so can also compute analytically, we next proceed to situations in which our prior is not a beta distribution, so we don't have a conjugate prior.

So we will have non-conjugate priors and are cast into situations in which we can't calculate posteriors analytically.

18.6.1 Markov chain Monte Carlo

First, we implement and display MCMC.

We first initialize the `BayesianInference` classes and then can directly call `BayesianInferencePlot` to plot both MCMC and SVI approximating posteriors.

```
# Initialize BayesianInference classes
# Try uniform
std_uniform = create_bayesian_inference(param=(0, 1), name_dist="uniform")
uniform = create_bayesian_inference(param=(0.2, 0.7), name_dist="uniform")

# Try truncated log normal
lognormal = create_bayesian_inference(param=(0, 2), name_dist="lognormal")

# Try Von Mises
vonmises = create_bayesian_inference(param=10, name_dist="vonMises")

# Try Laplace
laplace = create_bayesian_inference(param=(0.5, 0.07), name_dist="laplace")
```

To conduct our experiments more concisely, here we define two experiment functions that will print the model information and plot the result.

```
def plot_mcmc_experiment(
    bayesian_model: BayesianInference,
    true_θ: float,
    num_list: Sequence[int],
    num_samples: int,
    num_warmup: int = 1000,
    description: str = ""
):
    """
    Helper function to run and plot MCMC experiments for a given Bayesian model
    """
    print(
        f"=======INFO=======\n"
        f"Parameters: {bayesian_model.param}\n"
        f"Prior Dist: {bayesian_model.name_dist}"
    )
    if description:
        print(description)

    plot_model = create_bayesian_inference_plot(
        true_θ, num_list, bayesian_model
    )
    mcmc_plot(plot_model, num_samples=num_samples, num_warmup=num_warmup)

```

(continues on next page)

(continued from previous page)

```
def plot_svi_experiment(
    bayesian_model: BayesianInference,
    true_θ: float,
    num_list: Sequence[int],
    guide_dist: str,
    n_steps: int,
    description: str = ""
):
    """
    Helper function to run and plot SVI experiments for a given Bayesian model
    """
    print(
        f"=======INFO=======\n"
        f"Parameters: {bayesian_model.param}\n"
        f"Prior Dist: {bayesian_model.name_dist}"
    )
    if description:
        print(description)

    plot_model = create_bayesian_inference_plot(
        true_θ, num_list, bayesian_model
    )
    svi_plot(plot_model, guide_dist=guide_dist, n_steps=n_steps)
```

```
# Uniform
plot_mcmc_experiment(
    std_uniform,
    true_θ,
    num_list,
    mcmc_num_samples
)
```

```
=======INFO=======
Parameters: (0, 1)
Prior Dist: uniform
```

```
plot_mcmc_experiment(
    uniform,
    true_θ,
    num_list,
    mcmc_num_samples
)
```

```
=======INFO=======
Parameters: (0.2, 0.7)
Prior Dist: uniform
```

In the situation depicted above, we have assumed a $Uniform(\underline{\theta}, \overline{\theta})$ prior that puts zero probability outside a bounded support that excludes the true value.

Consequently, the posterior cannot put positive probability above $\overline{\theta}$ or below $\underline{\theta}$.

Note how when the true data-generating θ is located at 0.8 as it is here, when n gets large, the posterior concentrates on the upper bound of the support of the prior, 0.7 here.

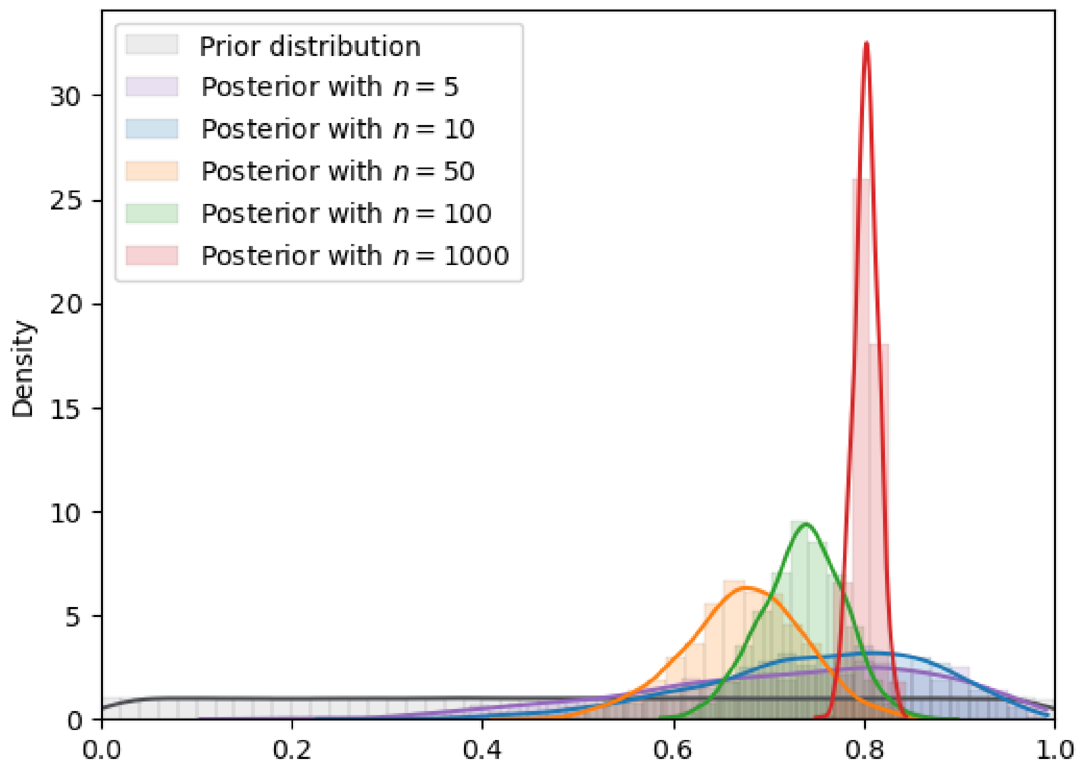


Fig. 18.8: MCMC density (uniform prior)

```
# log normal
plot_mcmc_experiment(
    lognormal,
    true_θ,
    num_list,
    mcmc_num_samples
)
```

```
=======INFO=======
Parameters: (0, 2)
Prior Dist: lognormal
```

```
# von Mises
plot_mcmc_experiment(
    vonmises,
    true_θ,
    num_list,
    mcmc_num_samples,
    description="\nNOTE: Shifted von Mises"
)
```

```
=======INFO=======
Parameters: 10
Prior Dist: vonMises
```

(continues on next page)

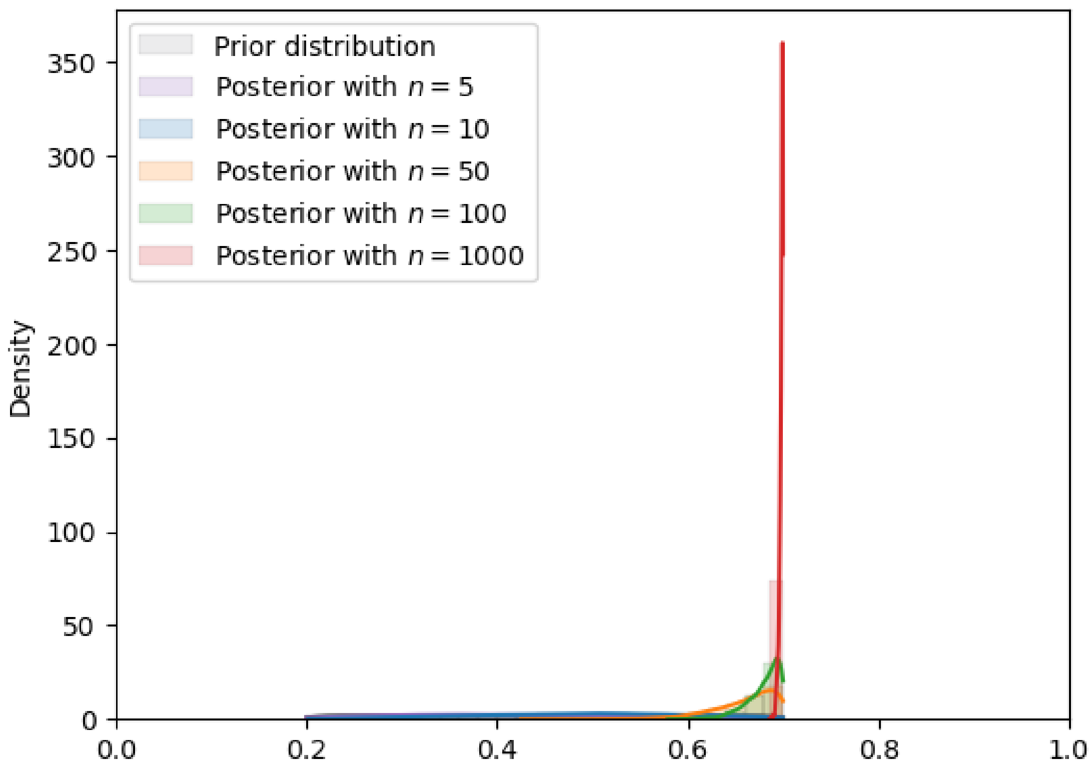


Fig. 18.9: MCMC density (uniform prior)

(continued from previous page)

```
NOTE: Shifted von Mises
```

```
# Laplace
plot_mcmc_experiment(
    laplace,
    true_θ,
    num_list,
    mcmc_num_samples
)
```

```
=======INFO=======
Parameters: (0.5, 0.07)
Prior Dist: laplace
```

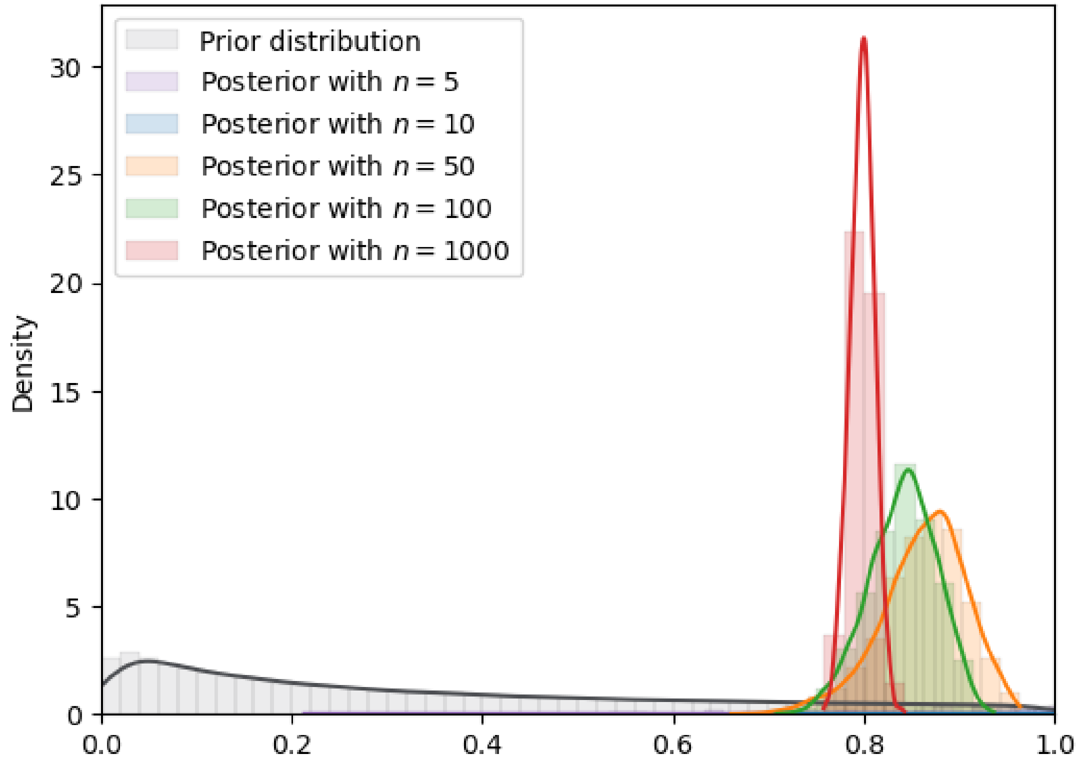


Fig. 18.10: MCMC density (log normal prior)

18.6.2 Variational inference

To get more accuracy we will now increase the number of steps for Variational Inference (VI)

```
svi_num_steps = 50000
```

VI with a truncated normal guide

```
# Uniform
plot_svi_experiment(
    create_bayesian_inference(param=(0, 1), name_dist="uniform"),
    true_θ,
    num_list,
    "normal",
    svi_num_steps
)
```

```
=======INFO=======
Parameters: (0, 1)
Prior Dist: uniform
```

```
# log normal
plot_svi_experiment(
    lognormal,
```

(continues on next page)

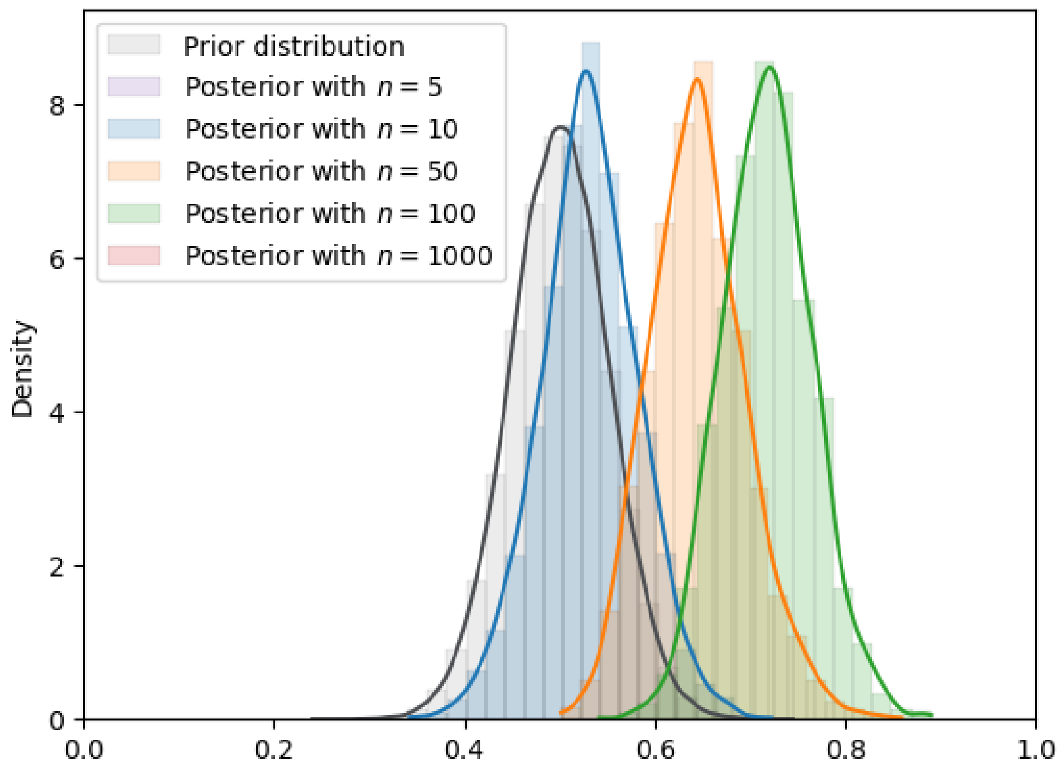


Fig. 18.11: MCMC density (von Mises prior)

(continued from previous page)

```
    true_θ,
    num_list,
    "normal",
    svi_num_steps
)
```

```
=======INFO=======
Parameters: (0, 2)
Prior Dist: lognormal
```

```
# Laplace
plot_svi_experiment(
    laplace,
    true_θ,
    num_list,
    "normal",
    svi_num_steps
)
```

```
=======INFO=======
Parameters: (0.5, 0.07)
Prior Dist: laplace
```

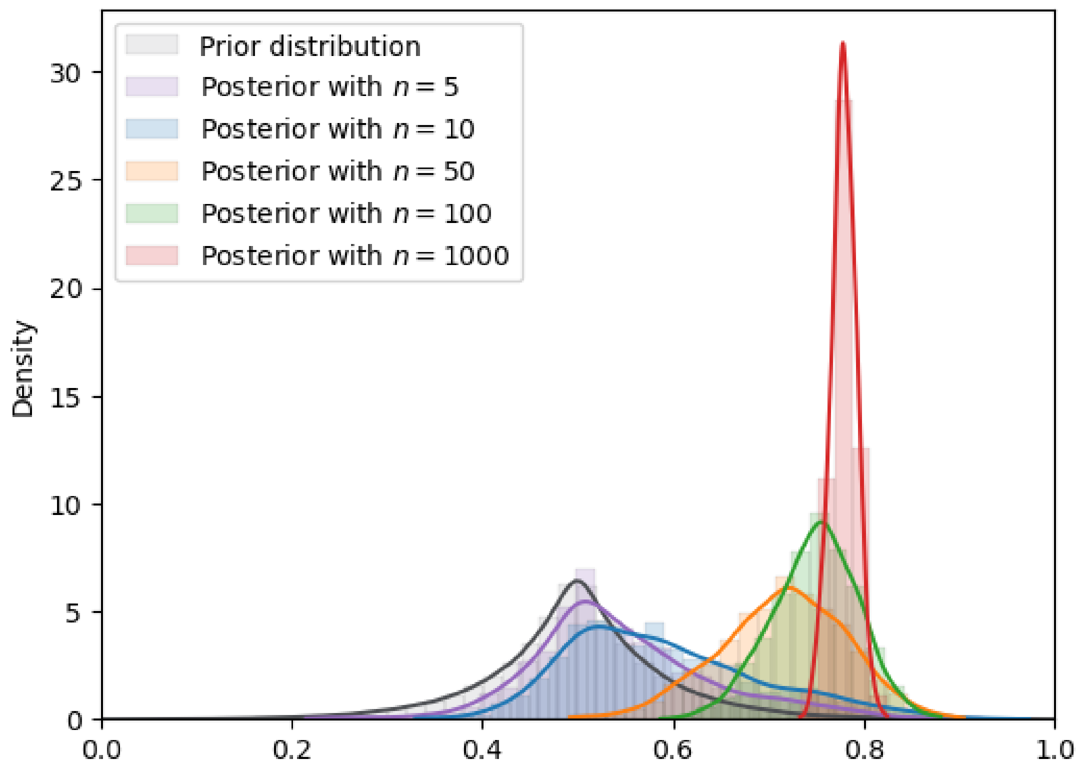


Fig. 18.12: MCMC density (Laplace prior)

Variational inference with a Beta guide distribution

```
# uniform
plot_svi_experiment(
    std_uniform,
    true_θ,
    num_list,
    "beta",
    svi_num_steps
)
```

```
=======INFO=======
Parameters: (0, 1)
Prior Dist: uniform
```

```
# log normal
plot_svi_experiment(
    lognormal,
    true_θ,
    num_list,
    "beta",
    svi_num_steps
)
```

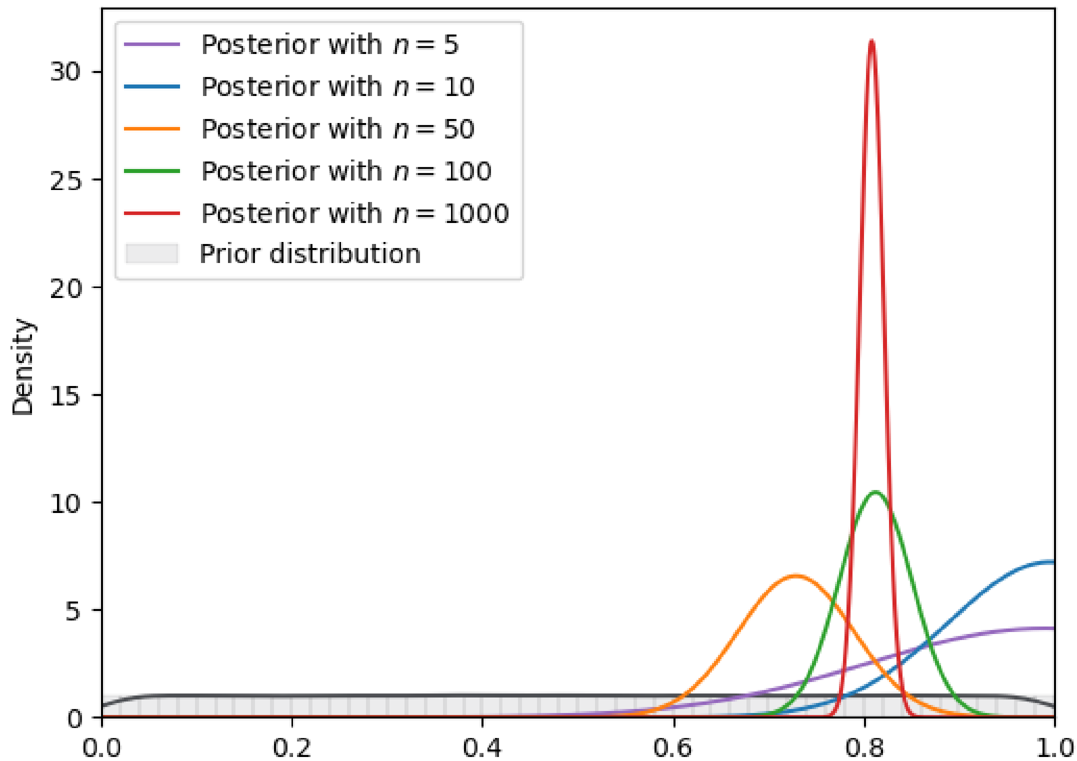


Fig. 18.13: SVI density (uniform prior, normal guide)

```
=======INFO=======
Parameters: (0, 2)
Prior Dist: lognormal
```

```
# von Mises
plot_svi_experiment(
    vonmises,
    true_θ,
    num_list,
    "beta",
    svi_num_steps,
    description="Shifted von Mises"
)
```

```
=======INFO=======
Parameters: 10
Prior Dist: vonMises
Shifted von Mises
```

```
# Laplace
plot_svi_experiment(
    laplace,
    true_θ,
    num_list,
    "beta",
```

(continues on next page)

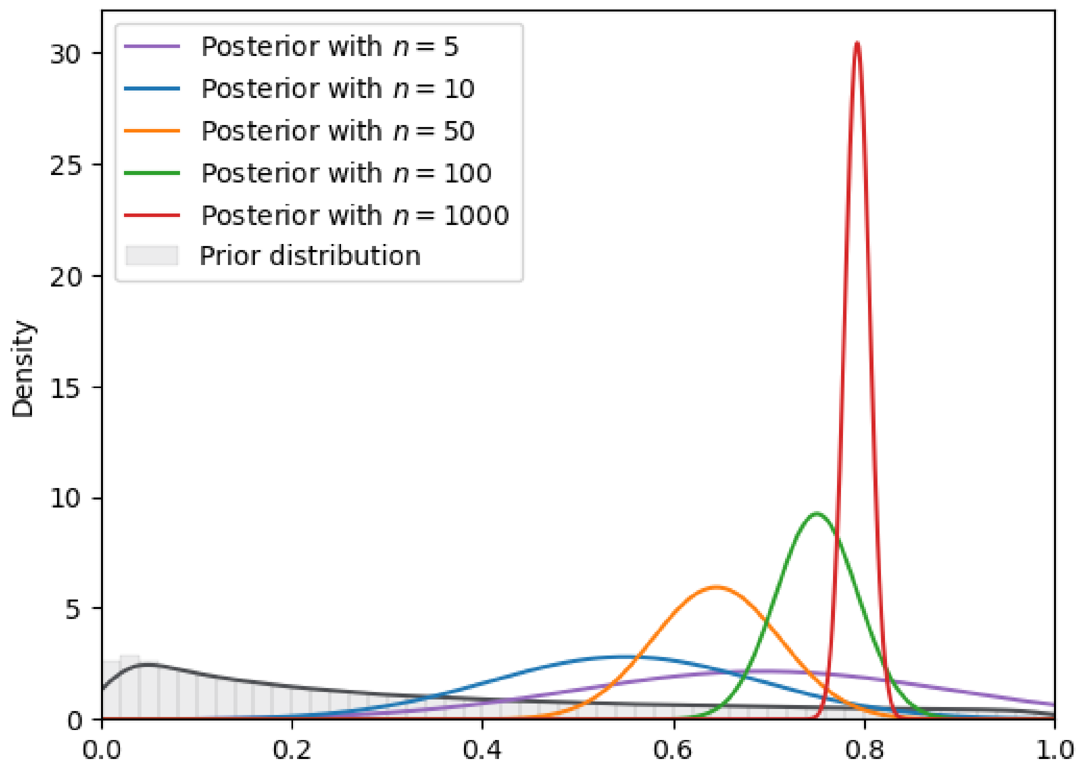


Fig. 18.14: SVI density (log normal prior, normal guide)

(continued from previous page)

```
    svi_num_steps
)
```

```
=======INFO=======
Parameters: (0.5, 0.07)
Prior Dist: laplace
```

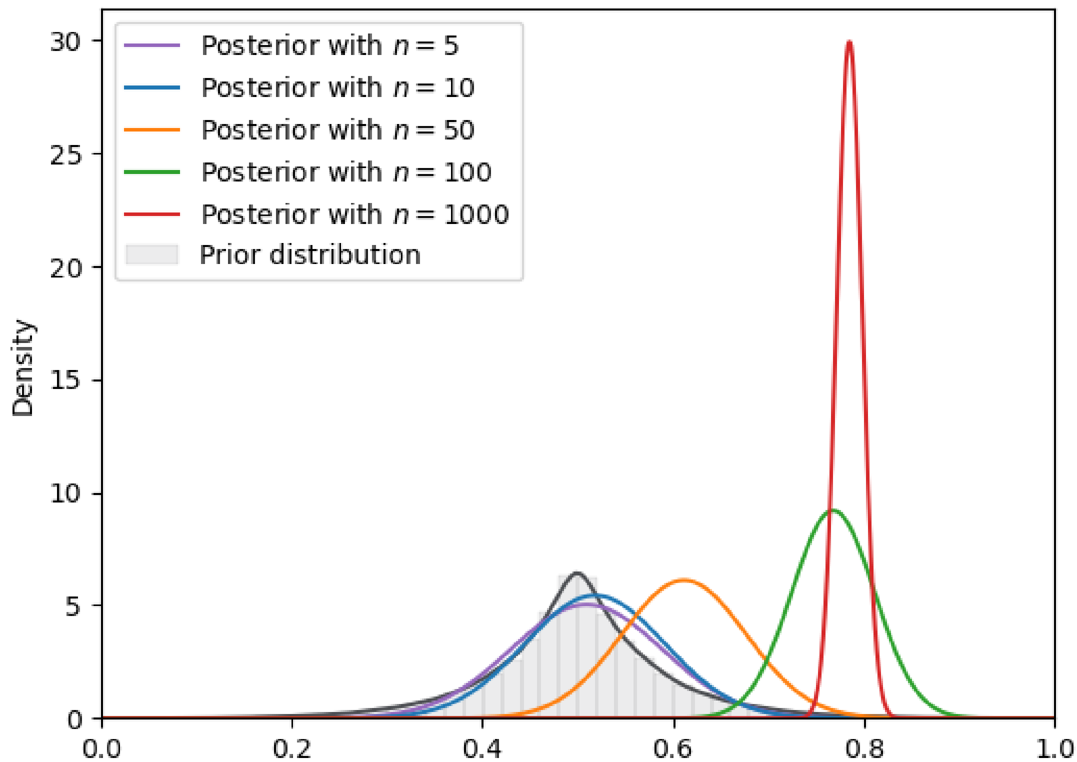


Fig. 18.15: SVI density (Laplace prior, normal guide)

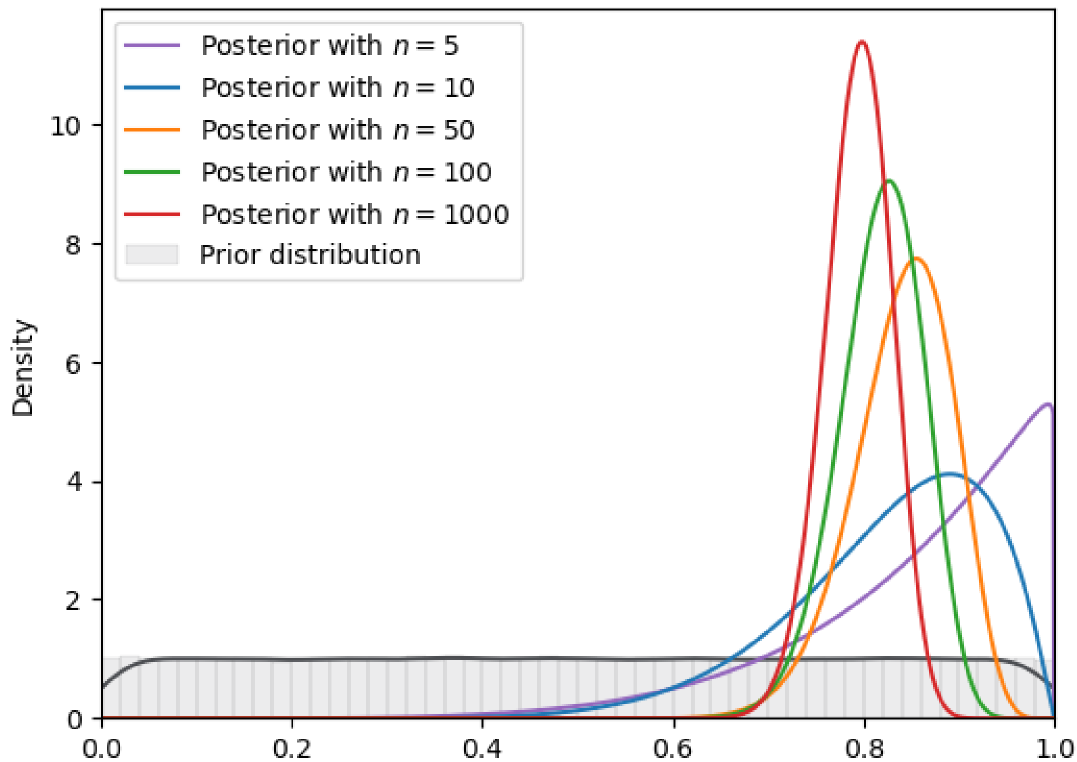


Fig. 18.16: SVI density (uniform prior, Beta guide)

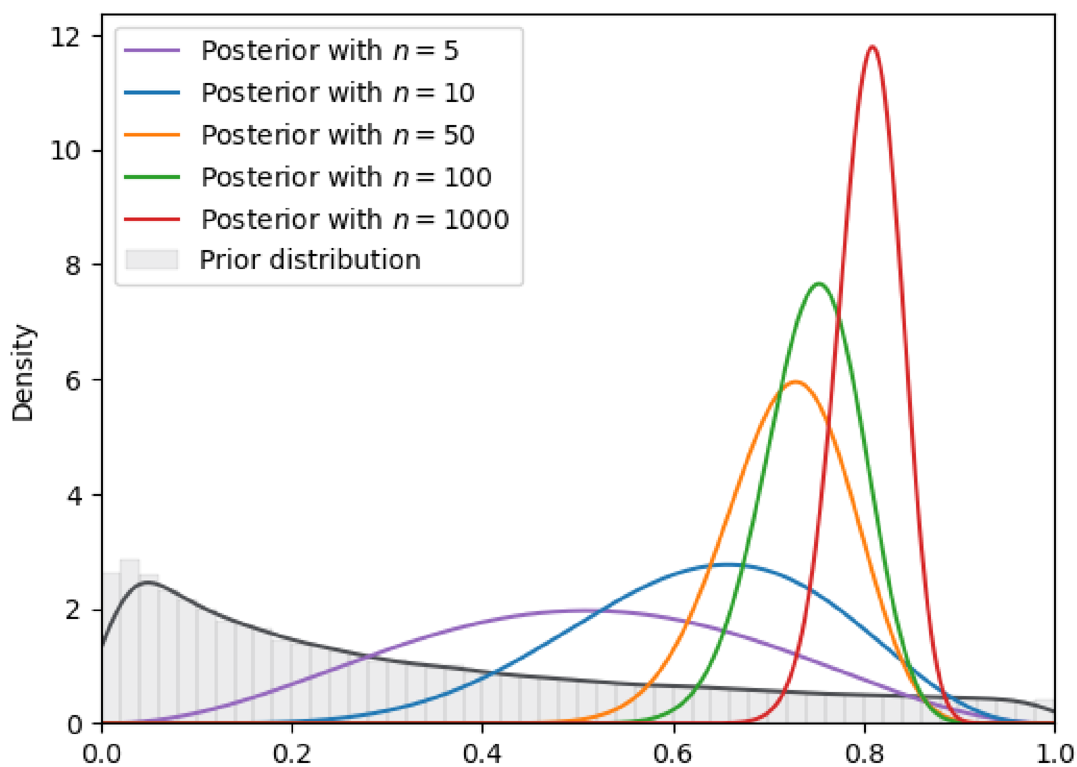


Fig. 18.17: SVI density (log normal prior, Beta guide)

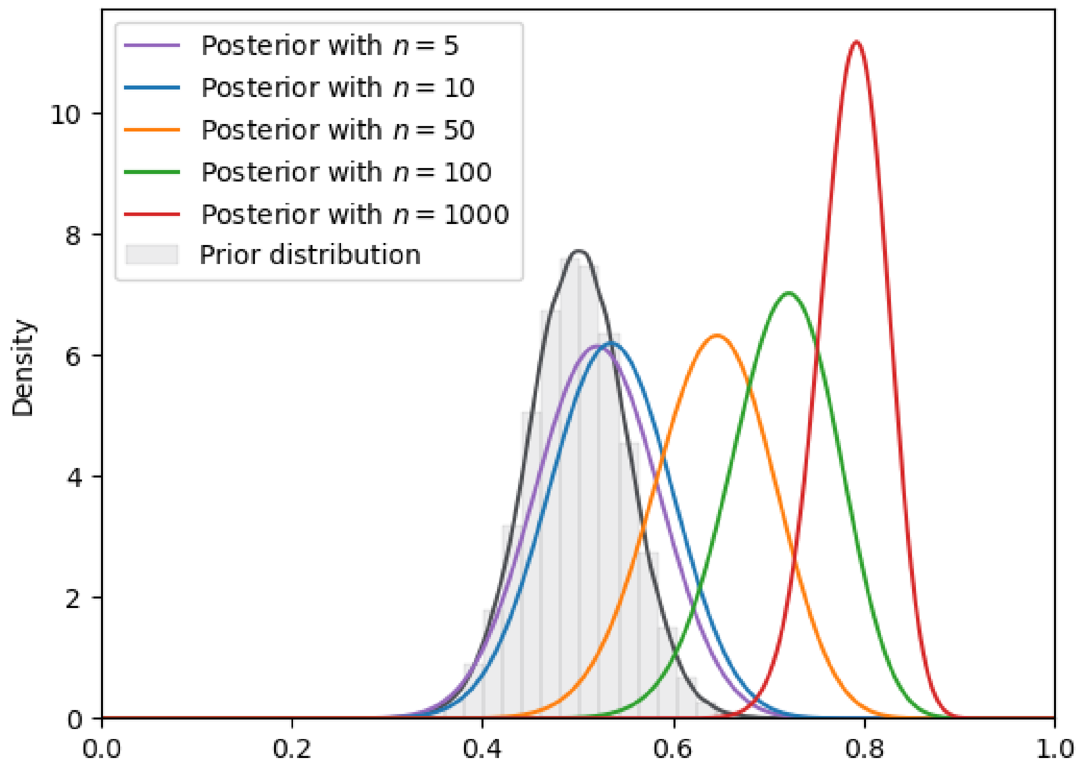


Fig. 18.18: SVI density (von Mises prior, Beta guide)

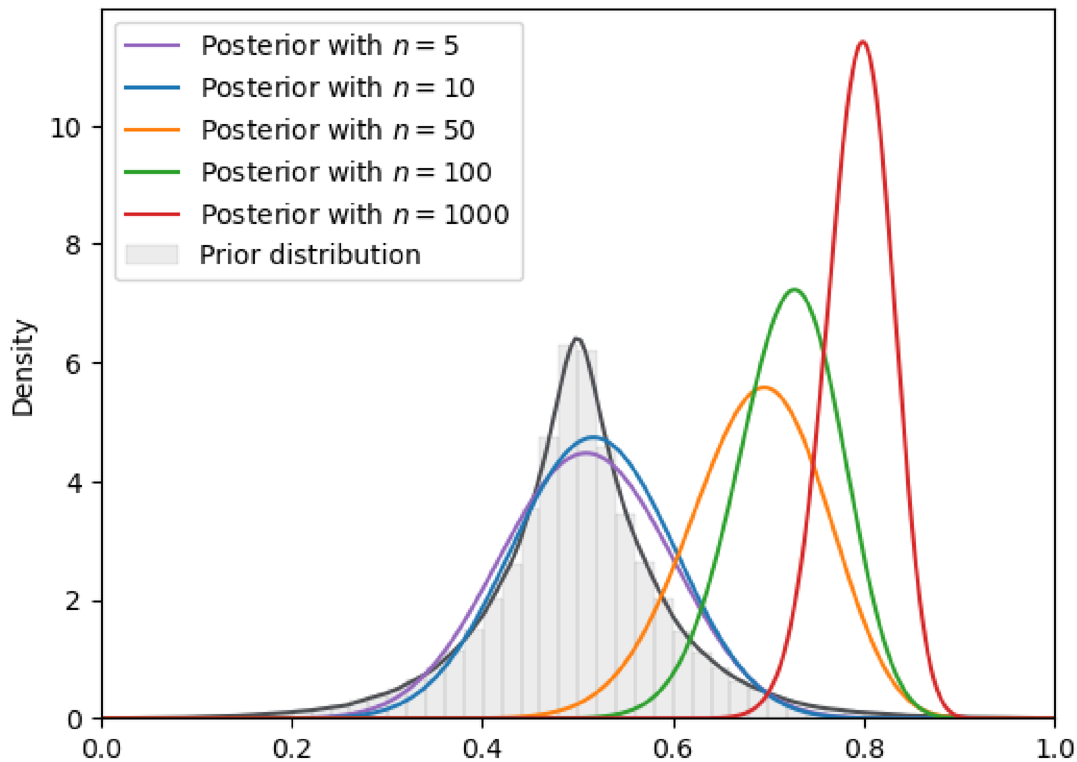


Fig. 18.19: SVI density (Laplace prior, Beta guide)

CHAPTER

NINETEEN

POSTERIOR DISTRIBUTIONS FOR AR(1) PARAMETERS

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

```
!pip install numpyro jax
```

In addition to what's included in base Anaconda, we need to install the following packages

```
!pip install arviz pymc
```

We'll begin with some Python imports.

```
import arviz as az
import pymc as pmc
import numpyro
from numpyro import distributions as dist

import numpy as np
import jax.numpy as jnp
from jax import random
import matplotlib.pyplot as plt

import logging
logging.basicConfig()
logger = logging.getLogger('pymc')
logger.setLevel(logging.CRITICAL)
```

```
/home/runner/miniconda3/envs/quantecon/lib/python3.13/site-packages/arviz/__init__.
↪py:50: FutureWarning:
ArviZ is undergoing a major refactor to improve flexibility and extensibility␣
↪while maintaining a user-friendly interface.
Some upcoming changes may be backward incompatible.
For details and migration guidance, visit: https://python.arviz.org/en/latest/user_
↪guide/migration_guide.html
  warn(
```

This lecture uses Bayesian methods offered by pymc and numpyro to make statistical inferences about two parameters of a univariate first-order autoregression.

The model is a good laboratory for illustrating consequences of alternative ways of modeling the distribution of the initial y_0:

- As a fixed number
- As a random variable drawn from the stationary distribution of the $\{y_t\}$ stochastic process

The first component of the statistical model is

$$y_{t+1} = \rho y_t + \sigma_x \epsilon_{t+1}, \quad t \geq 0 \tag{19.1}$$

where the scalars ρ and σ_x satisfy $|\rho| < 1$ and $\sigma_x > 0$; $\{\epsilon_{t+1}\}$ is a sequence of i.i.d. normal random variables with mean 0 and variance 1.

The second component of the statistical model is

$$y_0 \sim N(\mu_0, \sigma_0^2) \tag{19.2}$$

Consider a sample $\{y_t\}_{t=0}^T$ governed by this statistical model.

The model implies that the likelihood function of $\{y_t\}_{t=0}^T$ can be **factored**:

$$f(y_T, y_{T-1}, \ldots, y_0) = f(y_T|y_{T-1}) f(y_{T-1}|y_{T-2}) \cdots f(y_1|y_0) f(y_0)$$

where we use f to denote a generic probability density.

The statistical model (19.1)-(19.2) implies

$$\begin{aligned} f(y_t|y_{t-1}) &\sim \mathcal{N}(\rho y_{t-1}, \sigma_x^2) \\ f(y_0) &\sim \mathcal{N}(\mu_0, \sigma_0^2) \end{aligned}$$

We want to study how inferences about the unknown parameters (ρ, σ_x) depend on what is assumed about the parameters μ_0, σ_0 of the distribution of y_0.

Below, we study two widely used alternative assumptions:

- $(\mu_0, \sigma_0) = (y_0, 0)$ which means that y_0 is drawn from the distribution $\mathcal{N}(y_0, 0)$; in effect, we are **conditioning on an observed initial value**.
- μ_0, σ_0 are functions of ρ, σ_x because y_0 is drawn from the stationary distribution implied by ρ, σ_x.

Note: We do **not** treat a third possible case in which μ_0, σ_0 are free parameters to be estimated.

Unknown parameters are ρ, σ_x.

We have independent **prior probability distributions** for ρ, σ_x and want to compute a posterior probability distribution after observing a sample $\{y_t\}_{t=0}^T$.

The notebook uses `pymc4` and `numpyro` to compute a posterior distribution of ρ, σ_x. We will use NUTS samplers to generate samples from the posterior in a chain. Both of these libraries support NUTS samplers.

NUTS is a form of Monte Carlo Markov Chain (MCMC) algorithm that bypasses random walk behaviour and allows for convergence to a target distribution more quickly. This not only has the advantage of speed, but allows for complex models to be fitted without having to employ specialised knowledge regarding the theory underlying those fitting methods.

Thus, we explore consequences of making these alternative assumptions about the distribution of y_0:

- A first procedure is to condition on whatever value of y_0 is observed. This amounts to assuming that the probability distribution of the random variable y_0 is a Dirac delta function that puts probability one on the observed value of y_0.

- A second procedure assumes that y_0 is drawn from the stationary distribution of a process described by (19.1) so that $y_0 \sim N\left(0, \frac{\sigma_x^2}{(1-\rho)^2}\right)$

When the initial value y_0 is far out in a tail of the stationary distribution, conditioning on an initial value gives a posterior that is **more accurate** in a sense that we'll explain.

Basically, when y_0 happens to be in a tail of the stationary distribution and we **don't condition on** y_0, the likelihood function for $\{y_t\}_{t=0}^T$ adjusts the posterior distribution of the parameter pair ρ, σ_x to make the observed value of y_0 more likely than it really is under the stationary distribution, thereby adversely twisting the posterior in short samples.

An example below shows how not conditioning on y_0 adversely shifts the posterior probability distribution of ρ toward larger values.

We begin by solving a **direct problem** that simulates an AR(1) process.

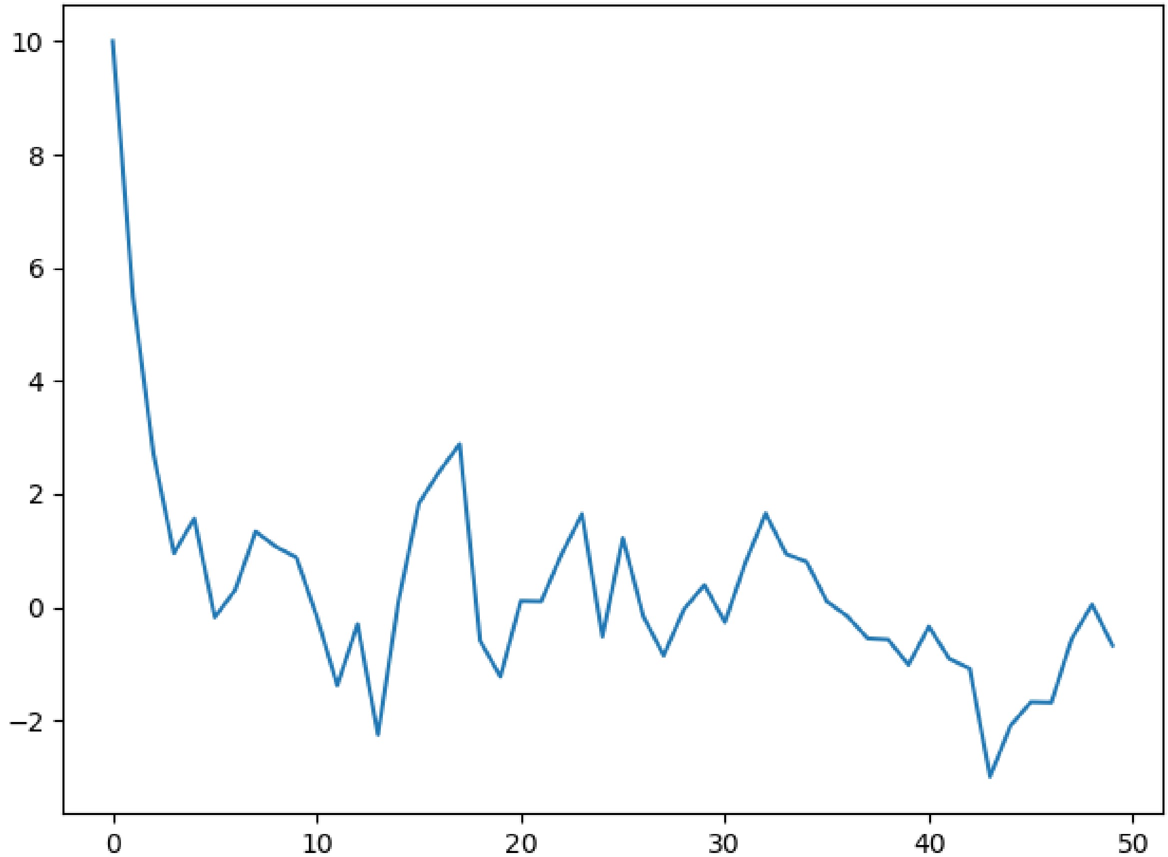
How we select the initial value y_0 matters.

- If we think y_0 is drawn from the stationary distribution $\mathcal{N}(0, \frac{\sigma_x^2}{1-\rho^2})$, then it is a good idea to use this distribution as $f(y_0)$. Why? Because y_0 contains information about ρ, σ_x.
- If we suspect that y_0 is far in the tails of the stationary distribution – so that variation in early observations in the sample have a significant **transient component** – it is better to condition on y_0 by setting $f(y_0) = 1$.

To illustrate the issue, we'll begin by choosing an initial y_0 that is far out in a tail of the stationary distribution.

```
def ar1_simulate(rho, sigma, y0, T):

    # Allocate space and draw epsilons
    y = np.empty(T)
    eps = np.random.normal(0.,sigma,T)

    # Initial condition and step forward
    y[0] = y0
    for t in range(1, T):
        y[t] = rho*y[t-1] + eps[t]

    return y

sigma =  1.
rho = 0.5
T = 50

np.random.seed(145353452)
y = ar1_simulate(rho, sigma, 10, T)
```

```
plt.plot(y)
plt.tight_layout()
```



Now we shall use Bayes' law to construct a posterior distribution, conditioning on the initial value of y_0.

(Later we'll assume that y_0 is drawn from the stationary distribution, but not now.)

First we'll use **pymc4**.

19.1 PyMC Implementation

For a normal distribution in pymc, $var = 1/\tau = \sigma^2$.

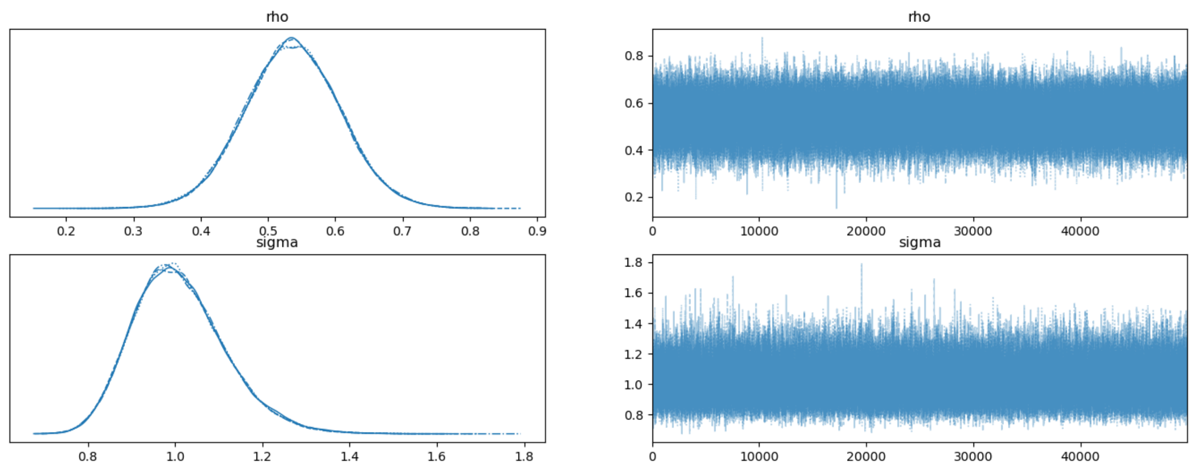
```
AR1_model = pmc.Model()

with AR1_model:

    # Start with priors
    rho = pmc.Uniform('rho', lower=-1., upper=1.) # Assume stable rho
    sigma = pmc.HalfNormal('sigma', sigma = np.sqrt(10))

    # Expected value of y at the next period (rho * y)
    yhat = rho * y[:-1]

    # Likelihood of the actual realization
    y_like = pmc.Normal('y_obs', mu=yhat, sigma=sigma, observed=y[1:])
```

pmc.sample by default uses the NUTS samplers to generate samples as shown in the below cell:

```
with AR1_model:
    trace = pmc.sample(50000, tune=10000, return_inferencedata=True)
```

```
Output()
```

```
with AR1_model:
    az.plot_trace(trace, figsize=(17,6))
```



Evidently, the posteriors aren't centered on the true values of $.5, 1$ that we used to generate the data.

This is a symptom of the classic **Hurwicz bias** for first order autoregressive processes (see Leonid Hurwicz [Hurwicz, 1950].)

The Hurwicz bias is worse the smaller is the sample (see [Orcutt and Winokur, 1969]).

Be that as it may, here is more information about the posterior.

```
with AR1_model:
    summary = az.summary(trace, round_to=4)

summary
```

```
         mean      sd  hdi_3%  hdi_97%  mcse_mean  mcse_sd     ess_bulk  \
rho    0.5363  0.0712  0.4015   0.6696     0.0002   0.0002  164842.2451
sigma  1.0100  0.1061  0.8184   1.2104     0.0003   0.0003  173296.3093

          ess_tail   r_hat
rho    117203.2369  1.0001
sigma  136956.3795  1.0001
```

Now we shall compute a posterior distribution after seeing the same data but instead assuming that y_0 is drawn from the stationary distribution.

This means that

$$y_0 \sim N\left(0, \frac{\sigma_x^2}{1-\rho^2}\right)$$

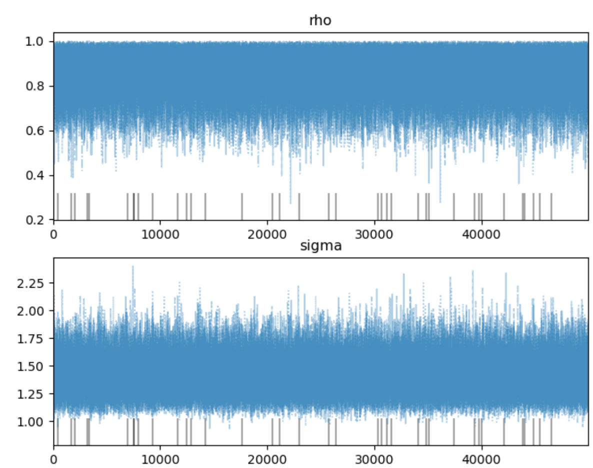
We alter the code as follows:

```python
AR1_model_y0 = pmc.Model()

with AR1_model_y0:

    # Start with priors
    rho = pmc.Uniform('rho', lower=-1., upper=1.) # Assume stable rho
    sigma = pmc.HalfNormal('sigma', sigma=np.sqrt(10))

    # Standard deviation of ergodic y
    y_sd = sigma / np.sqrt(1 - rho**2)

    # yhat
    yhat = rho * y[:-1]
    y_data = pmc.Normal('y_obs', mu=yhat, sigma=sigma, observed=y[1:])
    y0_data = pmc.Normal('y0_obs', mu=0., sigma=y_sd, observed=y[0])
```

```python
with AR1_model_y0:
    trace_y0 = pmc.sample(50000, tune=10000, return_inferencedata=True)

# Grey vertical lines are the cases of divergence
```

```
Output()
```

```python
with AR1_model_y0:
    az.plot_trace(trace_y0, figsize=(17,6))
```




```python
with AR1_model:
    summary_y0 = az.summary(trace_y0, round_to=4)

summary_y0
```

```
        mean      sd  hdi_3%  hdi_97%  mcse_mean  mcse_sd     ess_bulk  \
rho    0.876  0.0812  0.7321   0.9983     0.0002   0.0002  108535.0523
sigma  1.405  0.1470  1.1415   1.6842     0.0004   0.0004  133238.7241

         ess_tail  r_hat
```

(continues on next page)

(continued from previous page)

```
rho      80388.3858    1.0
sigma   112170.4237    1.0
```

Please note how the posterior for ρ has shifted to the right relative to when we conditioned on y_0 instead of assuming that y_0 is drawn from the stationary distribution.

Think about why this happens.

Hint

It is connected to how Bayes Law (conditional probability) solves an **inverse problem** by putting high probability on parameter values that make observations more likely.

We'll return to this issue after we use `numpyro` to compute posteriors under our two alternative assumptions about the distribution of y_0.

We'll now repeat the calculations using `numpyro`.

19.2 Numpyro Implementation

```
def plot_posterior(sample):
    """
    Plot trace and histogram
    """
    # To np array
    rhos = sample['rho']
    sigmas = sample['sigma']
    rhos, sigmas, = np.array(rhos), np.array(sigmas)

    fig, axs = plt.subplots(2, 2, figsize=(17, 6))
    # Plot trace
    axs[0, 0].plot(rhos)   # rho
    axs[1, 0].plot(sigmas) # sigma

    # Plot posterior
    axs[0, 1].hist(rhos, bins=50, density=True, alpha=0.7)
    axs[0, 1].set_xlim([0, 1])
    axs[1, 1].hist(sigmas, bins=50, density=True, alpha=0.7)

    axs[0, 0].set_title("rho")
    axs[0, 1].set_title("rho")
    axs[1, 0].set_title("sigma")
    axs[1, 1].set_title("sigma")
    plt.show()
```

```
def AR1_model(data):
    # set prior
    rho = numpyro.sample('rho', dist.Uniform(low=-1., high=1.))
    sigma = numpyro.sample('sigma', dist.HalfNormal(scale=np.sqrt(10)))

    # Expected value of y at the next period (rho * y)
    yhat = rho * data[:-1]
```

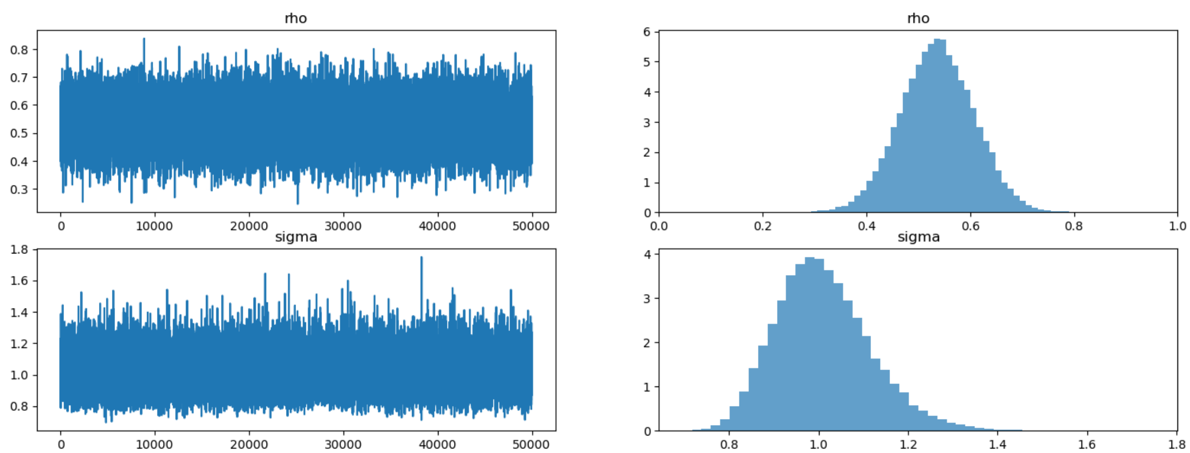
(continues on next page)

(continued from previous page)

```
    # Likelihood of the actual realization.
    y_data = numpyro.sample('y_obs', dist.Normal(loc=yhat, scale=sigma), obs=data[1:])
```

```
# Make jnp array
y = jnp.array(y)

# Set NUTS kernal
NUTS_kernel = numpyro.infer.NUTS(AR1_model)

# Run MCMC
mcmc = numpyro.infer.MCMC(NUTS_kernel, num_samples=50000, num_warmup=10000, progress_
↪bar=False)
mcmc.run(rng_key=random.PRNGKey(1), data=y)
```

```
plot_posterior(mcmc.get_samples())
```




```
mcmc.print_summary()
```

```
                mean       std    median      5.0%     95.0%     n_eff     r_hat
       rho      0.54      0.07      0.54      0.42      0.65  44430.72      1.00
     sigma      1.01      0.11      1.00      0.84      1.18  42626.35      1.00

Number of divergences: 0
```

Next, we again compute the posterior under the assumption that y_0 is drawn from the stationary distribution, so that

$$y_0 \sim N\left(0, \frac{\sigma_x^2}{1-\rho^2}\right)$$

Here's the new code to achieve this.

```
def AR1_model_y0(data):
    # Set prior
    rho = numpyro.sample('rho', dist.Uniform(low=-1., high=1.))
    sigma = numpyro.sample('sigma', dist.HalfNormal(scale=np.sqrt(10)))

    # Standard deviation of ergodic y
    y_sd = sigma / jnp.sqrt(1 - rho**2)
```

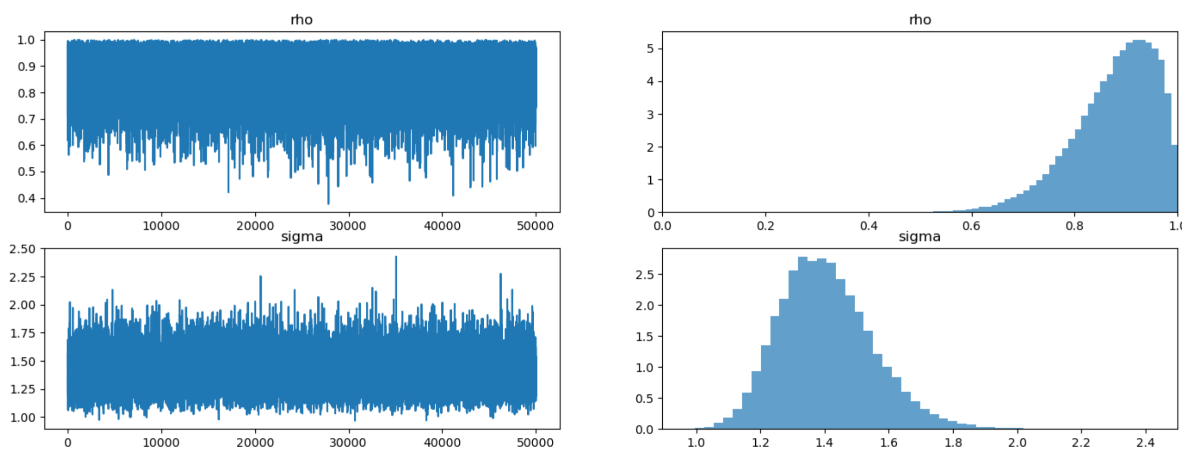
(continues on next page)

(continued from previous page)

```
    # Expected value of y at the next period (rho * y)
    yhat = rho * data[:-1]

    # Likelihood of the actual realization.
    y_data = numpyro.sample('y_obs', dist.Normal(loc=yhat, scale=sigma), obs=data[1:])
    y0_data = numpyro.sample('y0_obs', dist.Normal(loc=0., scale=y_sd), obs=data[0])
```

```
# Make jnp array
y = jnp.array(y)

# Set NUTS kernal
NUTS_kernel = numpyro.infer.NUTS(AR1_model_y0)

# Run MCMC
mcmc2 = numpyro.infer.MCMC(NUTS_kernel, num_samples=50000, num_warmup=10000, progress_
↪bar=False)
mcmc2.run(rng_key=random.PRNGKey(1), data=y)
```

```
plot_posterior(mcmc2.get_samples())
```




```
mcmc2.print_summary()
```

```
                mean       std    median      5.0%     95.0%     n_eff     r_hat
       rho      0.88      0.08      0.89      0.76      1.00  31419.08      1.00
     sigma      1.41      0.15      1.39      1.17      1.64  26542.09      1.00

Number of divergences: 0
```

Look what happened to the posterior!

It has moved far from the true values of the parameters used to generate the data because of how Bayes' Law (i.e., conditional probability) is telling `numpyro` to explain what it interprets as "explosive" observations early in the sample.

Bayes' Law is able to generate a plausible likelihood for the first observation by driving $\rho \to 1$ and $\sigma \uparrow$ in order to raise the variance of the stationary distribution.

Our example illustrates the importance of what you assume about the distribution of initial conditions.

CHAPTER

TWENTY

FORECASTING AN AR(1) PROCESS

```
!pip install arviz pymc
```

This lecture describes methods for forecasting statistics that are functions of future values of a univariate autogressive process.

The methods are designed to take into account two possible sources of uncertainty about these statistics:

- random shocks that impinge of the transition law
- uncertainty about the parameter values of the AR(1) process

We consider two sorts of statistics:

- prospective values y_{t+j} of a random process $\{y_t\}$ that is governed by the AR(1) process
- sample path properties that are defined as non-linear functions of future values $\{y_{t+j}\}_{j\geq 1}$ at time t

Sample path properties are things like "time to next turning point" or "time to next recession".

To investigate sample path properties we'll use a simulation procedure recommended by Wecker [Wecker, 1979].

To acknowledge uncertainty about parameters, we'll deploy `pymc` to construct a Bayesian joint posterior distribution for unknown parameters.

Let's start with some imports.

```
import numpy as np
import arviz as az
import pymc as pmc
import matplotlib.pyplot as plt
import seaborn as sns

sns.set_style('white')
colors = sns.color_palette()

import logging
logging.basicConfig()
logger = logging.getLogger('pymc')
logger.setLevel(logging.CRITICAL)
```

20.1 A Univariate First-Order Autoregressive Process

Consider the univariate AR(1) model:

$$y_{t+1} = \rho y_t + \sigma \epsilon_{t+1}, \quad t \geq 0 \tag{20.1}$$

where the scalars ρ and σ satisfy $|\rho| < 1$ and $\sigma > 0$; $\{\epsilon_{t+1}\}$ is a sequence of i.i.d. normal random variables with mean 0 and variance 1.

The initial condition y_0 is a known number.

Equation (20.1) implies that for $t \geq 0$, the conditional density of y_{t+1} is

$$f(y_{t+1}|y_t; \rho, \sigma) \sim \mathcal{N}(\rho y_t, \sigma^2) \tag{20.2}$$

Further, equation (20.1) also implies that for $t \geq 0$, the conditional density of y_{t+j} for $j \geq 1$ is

$$f(y_{t+j}|y_t; \rho, \sigma) \sim \mathcal{N}\left(\rho^j y_t, \sigma^2 \frac{1-\rho^{2j}}{1-\rho^2}\right) \tag{20.3}$$

The predictive distribution (20.3) that assumes that the parameters ρ, σ are known, which we express by conditioning on them.

We also want to compute a predictive distribution that does not condition on ρ, σ but instead takes account of our uncertainty about them.

We form this predictive distribution by integrating (20.3) with respect to a joint posterior distribution $\pi_t(\rho, \sigma|y^t)$ that conditions on an observed history $y^t = \{y_s\}_{s=0}^t$:

$$f(y_{t+j}|y^t) = \int f(y_{t+j}|y_t; \rho, \sigma)\pi_t(\rho, \sigma|y^t) d\rho d\sigma \tag{20.4}$$

Predictive distribution (20.3) assumes that parameters (ρ, σ) are known.

Predictive distribution (20.4) assumes that parameters (ρ, σ) are uncertain, but have known probability distribution $\pi_t(\rho, \sigma|y^t)$.

We also want to compute some predictive distributions of "sample path statistics" that might include, for example

- the time until the next "recession",
- the minimum value of Y over the next 8 periods,
- "severe recession", and
- the time until the next turning point (positive or negative).

To accomplish that for situations in which we are uncertain about parameter values, we shall extend Wecker's [Wecker, 1979] approach in the following way.

- first simulate an initial path of length T_0;
- for a given prior, draw a sample of size N from the posterior joint distribution of parameters (ρ, σ) after observing the initial path;
- for each draw $n = 0, 1, ..., N$, simulate a "future path" of length T_1 with parameters (ρ_n, σ_n) and compute our three "sample path statistics";
- finally, plot the desired statistics from the N samples as an empirical distribution.

20.2 Implementation

First, we'll simulate a sample path from which to launch our forecasts.

In addition to plotting the sample path, under the assumption that the true parameter values are known, we'll plot .9 and .95 coverage intervals using conditional distribution (20.3) described above.

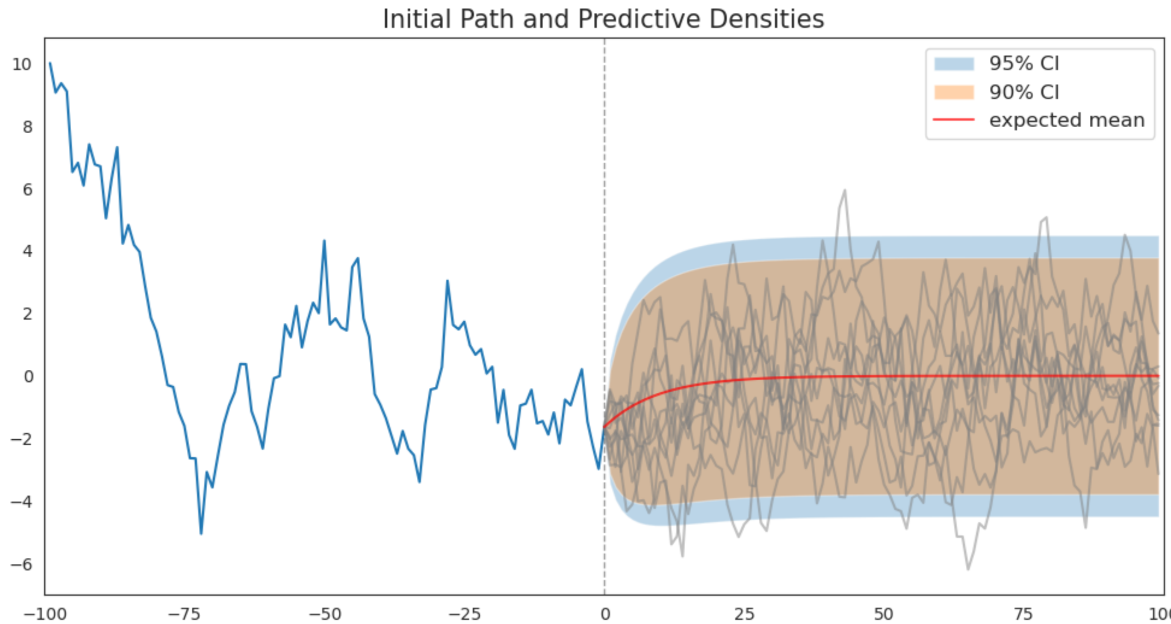
We'll also plot a bunch of samples of sequences of future values and watch where they fall relative to the coverage interval.

```python
def AR1_simulate(rho, sigma, y0, T):

    # Allocate space and draw epsilons
    y = np.empty(T)
    eps = np.random.normal(0, sigma, T)

    # Initial condition and step forward
    y[0] = y0
    for t in range(1, T):
        y[t] = rho * y[t-1] + eps[t]

    return y


def plot_initial_path(initial_path):
    """
    Plot the initial path and the preceding predictive densities
    """
    # Compute .9 confidence interval]
    y0 = initial_path[-1]
    center = np.array([rho**j * y0 for j in range(T1)])
    vars = np.array([sigma**2 * (1 - rho**(2 * j)) / (1 - rho**2) for j in range(T1)])
    y_bounds1_c95, y_bounds2_c95 = center + 1.96 * np.sqrt(vars), center - 1.96 * np.
↪sqrt(vars)
    y_bounds1_c90, y_bounds2_c90 = center + 1.65 * np.sqrt(vars), center - 1.65 * np.
↪sqrt(vars)

    # Plot
    fig, ax = plt.subplots(1, 1, figsize=(12, 6))
    ax.set_title("Initial Path and Predictive Densities", fontsize=15)
    ax.plot(np.arange(-T0 + 1, 1), initial_path)
    ax.set_xlim([-T0, T1])
    ax.axvline(0, linestyle='--', alpha=.4, color='k', lw=1)

    # Simulate future paths
    for i in range(10):
        y_future = AR1_simulate(rho, sigma, y0, T1)
        ax.plot(np.arange(T1), y_future, color='grey', alpha=.5)

    # Plot 90% CI
    ax.fill_between(np.arange(T1), y_bounds1_c95, y_bounds2_c95, alpha=.3, label='95%␣
↪CI')
    ax.fill_between(np.arange(T1), y_bounds1_c90, y_bounds2_c90, alpha=.35, label='90
↪% CI')
    ax.plot(np.arange(T1), center, color='red', alpha=.7, label='expected mean')
    ax.legend(fontsize=12)
    plt.show()
```

(continues on next page)

(continued from previous page)

```
sigma =  1
rho = 0.9
T0, T1 = 100, 100
y0 = 10

# Simulate
np.random.seed(145)
initial_path = AR1_simulate(rho, sigma, y0, T0)

# Plot
plot_initial_path(initial_path)
```



As functions of forecast horizon, the coverage intervals have shapes like those described in https://python.quantecon.org/perm_income_cons.html

20.3 Predictive Distributions of Path Properties

Wecker [Wecker, 1979] proposed using simulation techniques to characterize predictive distribution of some statistics that are non-linear functions of y.

He called these functions "path properties" to contrast them with properties of single data points.

He studied two special prospective path properties of a given series $\{y_t\}$.

The first was **time until the next turning point**.

- he defined a **"turning point"** to be the date of the second of two successive declines in y.

To examine this statistic, let Z be an indicator process

$$Z_t(Y(\omega)) := \begin{cases} 1 & \text{if } Y_t(\omega) < Y_{t-1}(\omega) < Y_{t-2}(\omega) \geq Y_{t-3}(\omega) \\ 0 & \text{otherwise} \end{cases}$$

Then the random variable **time until the next turning point** is defined as the following **stopping time** with respect to Z:

$$W_t(\omega) := \inf\{k \geq 1 \mid Z_{t+k}(\omega) = 1\}$$

Wecker [Wecker, 1979] also studied **the minimum value of** Y **over the next 8 quarters** which can be defined as the random variable.

$$M_t(\omega) := \min\{Y_{t+1}(\omega); Y_{t+2}(\omega); \dots; Y_{t+8}(\omega)\}$$

It is interesting to study yet another possible concept of a **turning point**.

Thus, let

$$T_t(Y(\omega)) := \begin{cases} 1 & \text{if } Y_{t-2}(\omega) > Y_{t-1}(\omega) > Y_t(\omega) \text{ and } \ Y_t(\omega) < Y_{t+1}(\omega) < Y_{t+2}(\omega) \\ -1 & \text{if } Y_{t-2}(\omega) < Y_{t-1}(\omega) < Y_t(\omega) \text{ and } \ Y_t(\omega) > Y_{t+1}(\omega) > Y_{t+2}(\omega) \\ 0 & \text{otherwise} \end{cases}$$

Define a **positive turning point today or tomorrow** statistic as

$$P_t(\omega) := \begin{cases} 1 & \text{if } T_t(\omega) = 1 \text{ or } T_{t+1}(\omega) = 1 \\ 0 & \text{otherwise} \end{cases}$$

This is designed to express the event

- "after one or two decrease(s), Y will grow for two consecutive quarters"

Following [Wecker, 1979], we can use simulations to calculate probabilities of P_t and N_t for each period t.

20.4 A Wecker-Like Algorithm

The procedure consists of the following steps:

- index a sample path by ω_i
- for a given date t, simulate I sample paths of length N

$$Y(\omega_i) = \{Y_{t+1}(\omega_i), Y_{t+2}(\omega_i), \dots, Y_{t+N}(\omega_i)\}_{i=1}^{I}$$

- for each path ω_i, compute the associated value of $W_t(\omega_i), W_{t+1}(\omega_i), \dots$
- consider the sets $\{W_t(\omega_i)\}_{i=1}^{T}$, $\{W_{t+1}(\omega_i)\}_{i=1}^{T}$, … , $\{W_{t+N}(\omega_i)\}_{i=1}^{T}$ as samples from the predictive distributions $f(W_{t+1} \mid y_t, \dots)$, $f(W_{t+2} \mid y_t, y_{t-1}, \dots)$, …, $f(W_{t+N} \mid y_t, y_{t-1}, \dots)$.

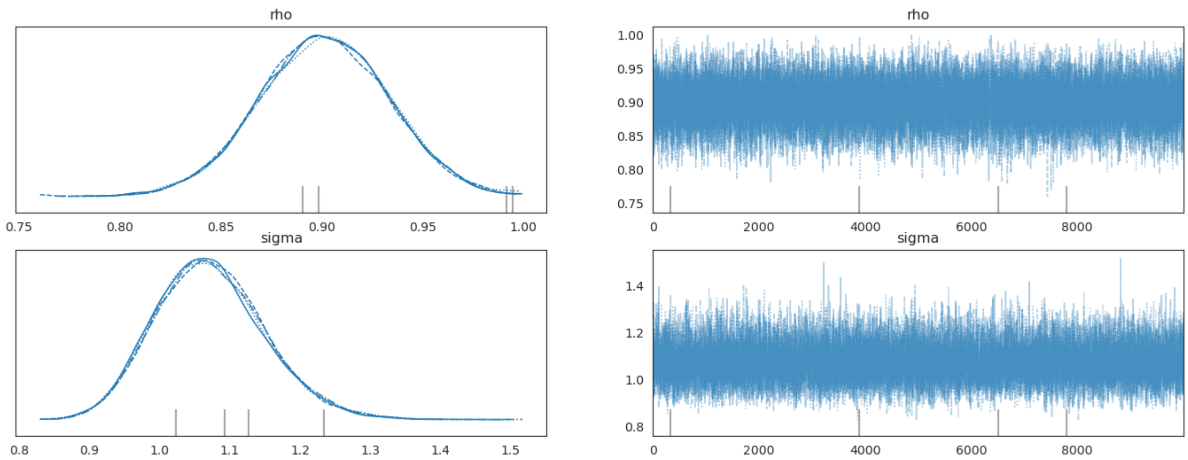
20.5 Using Simulations to Approximate a Posterior Distribution

The next code cells use `pymc` to compute the time t posterior distribution of ρ, σ.

Note that in defining the likelihood function, we choose to condition on the initial value y_0.

```python
def draw_from_posterior(sample):
    """
    Draw a sample of size N from the posterior distribution.
    """

    AR1_model = pmc.Model()

    with AR1_model:

        # Start with priors
        rho = pmc.Uniform('rho',lower=-1.,upper=1.)  # Assume stable rho
        sigma = pmc.HalfNormal('sigma', sigma = np.sqrt(10))

        # Expected value of y at the next period (rho * y)
        yhat = rho * sample[:-1]

        # Likelihood of the actual realization.
        y_like = pmc.Normal('y_obs', mu=yhat, sigma=sigma, observed=sample[1:])

    with AR1_model:
        trace = pmc.sample(10000, tune=5000)

    # check condition
    with AR1_model:
        az.plot_trace(trace, figsize=(17, 6))

    rhos = trace.posterior.rho.values.flatten()
    sigmas = trace.posterior.sigma.values.flatten()

    post_sample = {
        'rho': rhos,
        'sigma': sigmas
    }

    return post_sample

post_samples = draw_from_posterior(initial_path)
```

```
Output()
```



The graphs on the left portray posterior marginal distributions.

20.6 Calculating Sample Path Statistics

Our next step is to prepare Python code to compute our sample path statistics.

```python
# define statistics
def next_recession(omega):
    n = omega.shape[0] - 3
    z = np.zeros(n, dtype=int)

    for i in range(n):
        z[i] = int(omega[i] <= omega[i+1] and omega[i+1] > omega[i+2] and omega[i+2] >
 ↪ omega[i+3])

    if np.any(z) == False:
        return 500
    else:
        return np.where(z==1)[0][0] + 1

def minimum_value(omega):
    return min(omega[:8])

def severe_recession(omega):
    z = np.diff(omega)
    n = z.shape[0]

    sr = (z < -.02).astype(int)
    indices = np.where(sr == 1)[0]

    if len(indices) == 0:
        return T1
    else:
        return indices[0] + 1

def next_turning_point(omega):
    """
    Suppose that omega is of length 6
```

(continues on next page)

(continued from previous page)

```
        y_{t-2}, y_{t-1}, y_{t}, y_{t+1}, y_{t+2}, y_{t+3}

    that is sufficient for determining the value of P/N
    """

    n = np.asarray(omega).shape[0] - 4
    T = np.zeros(n, dtype=int)

    for i in range(n):
        if ((omega[i] > omega[i+1]) and (omega[i+1] > omega[i+2]) and
            (omega[i+2] < omega[i+3]) and (omega[i+3] < omega[i+4])):
            T[i] = 1
        elif ((omega[i] < omega[i+1]) and (omega[i+1] < omega[i+2]) and
            (omega[i+2] > omega[i+3]) and (omega[i+3] > omega[i+4])):
            T[i] = -1

    up_turn = np.where(T == 1)[0][0] + 1 if (1 in T) == True else T1
    down_turn = np.where(T == -1)[0][0] + 1 if (-1 in T) == True else T1

    return up_turn, down_turn
```

20.7 Original Wecker Method

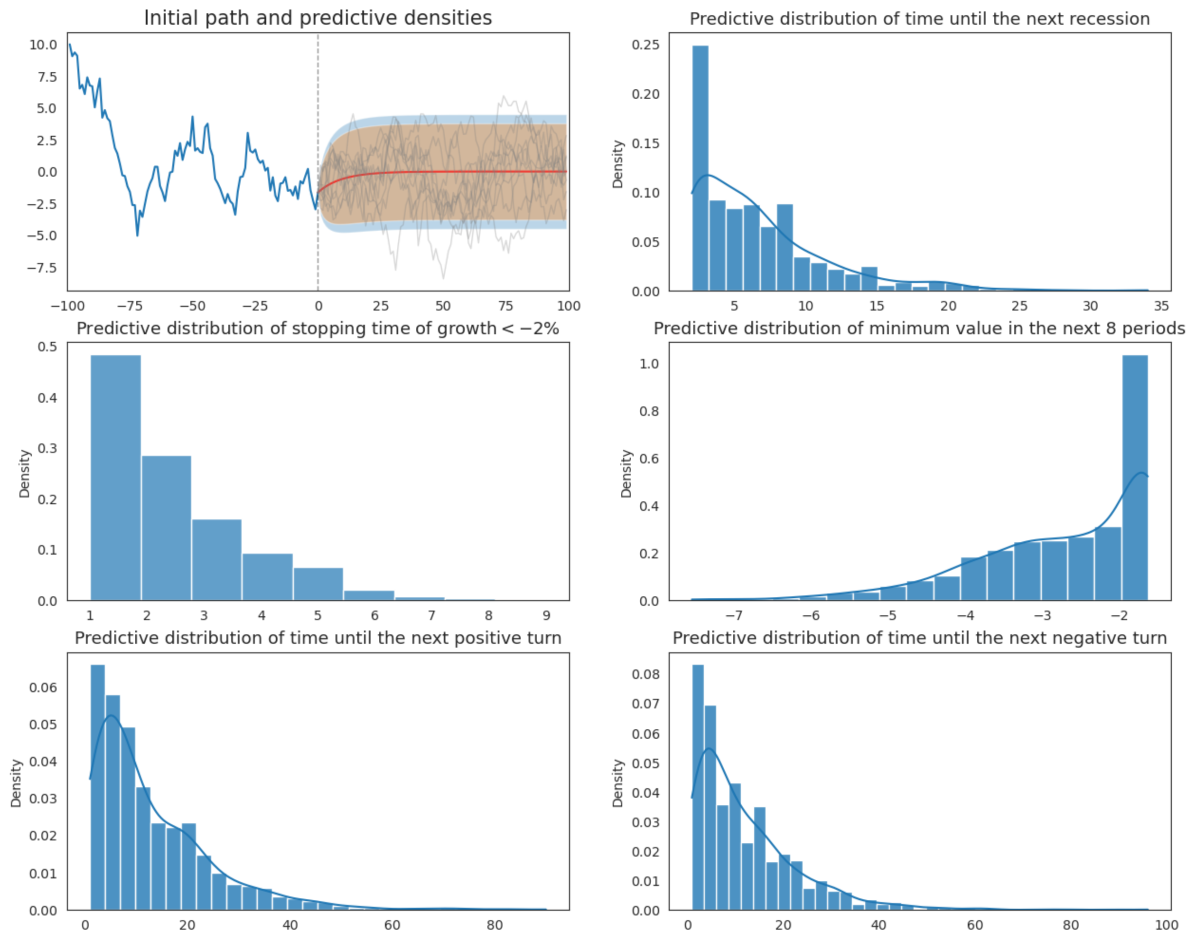
Now we apply Wecker's original method by simulating future paths and compute predictive distributions, conditioning on the true parameters associated with the data-generating model.

```
def plot_Wecker(initial_path, N, ax):
    """
    Plot the predictive distributions from "pure" Wecker's method.
    """
    # Store outcomes
    next_reces = np.zeros(N)
    severe_rec = np.zeros(N)
    min_vals = np.zeros(N)
    next_up_turn, next_down_turn = np.zeros(N), np.zeros(N)

    # Compute .9 confidence interval]
    y0 = initial_path[-1]
    center = np.array([rho**j * y0 for j in range(T1)])
    vars = np.array([sigma**2 * (1 - rho**(2 * j)) / (1 - rho**2) for j in range(T1)])
    y_bounds1_c95, y_bounds2_c95 = center + 1.96 * np.sqrt(vars), center - 1.96 * np.
↪sqrt(vars)
    y_bounds1_c90, y_bounds2_c90 = center + 1.65 * np.sqrt(vars), center - 1.65 * np.
↪sqrt(vars)

    # Plot
    ax[0, 0].set_title("Initial path and predictive densities", fontsize=15)
    ax[0, 0].plot(np.arange(-T0 + 1, 1), initial_path)
    ax[0, 0].set_xlim([-T0, T1])
    ax[0, 0].axvline(0, linestyle='--', alpha=.4, color='k', lw=1)

    # Plot 90% CI
    ax[0, 0].fill_between(np.arange(T1), y_bounds1_c95, y_bounds2_c95, alpha=.3)
    ax[0, 0].fill_between(np.arange(T1), y_bounds1_c90, y_bounds2_c90, alpha=.35)
```

(continues on next page)

(continued from previous page)

```
    ax[0, 0].plot(np.arange(T1), center, color='red', alpha=.7)

    # Simulate future paths
    for n in range(N):
        sim_path = AR1_simulate(rho, sigma, initial_path[-1], T1)
        next_reces[n] = next_recession(np.hstack([initial_path[-3:-1], sim_path]))
        severe_rec[n] = severe_recession(sim_path)
        min_vals[n] = minimum_value(sim_path)
        next_up_turn[n], next_down_turn[n] = next_turning_point(sim_path)

        if n%(N/10) == 0:
            ax[0, 0].plot(np.arange(T1), sim_path, color='gray', alpha=.3, lw=1)

    # Return next_up_turn, next_down_turn
    sns.histplot(next_reces, kde=True, stat='density', ax=ax[0, 1], alpha=.8, label=
↪'True parameters')
    ax[0, 1].set_title("Predictive distribution of time until the next recession",␣
↪fontsize=13)

    sns.histplot(severe_rec, kde=False, stat='density', ax=ax[1, 0], binwidth=0.9,␣
↪alpha=.7, label='True parameters')
    ax[1, 0].set_title(r"Predictive distribution of stopping time of growth$<-2\%$",␣
↪fontsize=13)

    sns.histplot(min_vals, kde=True, stat='density', ax=ax[1, 1], alpha=.8, label=
↪'True parameters')
    ax[1, 1].set_title("Predictive distribution of minimum value in the next 8 periods
↪", fontsize=13)

    sns.histplot(next_up_turn, kde=True, stat='density', ax=ax[2, 0], alpha=.8, label=
↪'True parameters')
    ax[2, 0].set_title("Predictive distribution of time until the next positive turn",
↪ fontsize=13)

    sns.histplot(next_down_turn, kde=True, stat='density', ax=ax[2, 1], alpha=.8,␣
↪label='True parameters')
    ax[2, 1].set_title("Predictive distribution of time until the next negative turn",
↪ fontsize=13)

fig, ax = plt.subplots(3, 2, figsize=(15,12))
plot_Wecker(initial_path, 1000, ax)
plt.show()
```



20.8 Extended Wecker Method

Now we apply we apply our "extended" Wecker method based on predictive densities of y defined by (20.4) that acknowledge posterior uncertainty in the parameters ρ, σ.

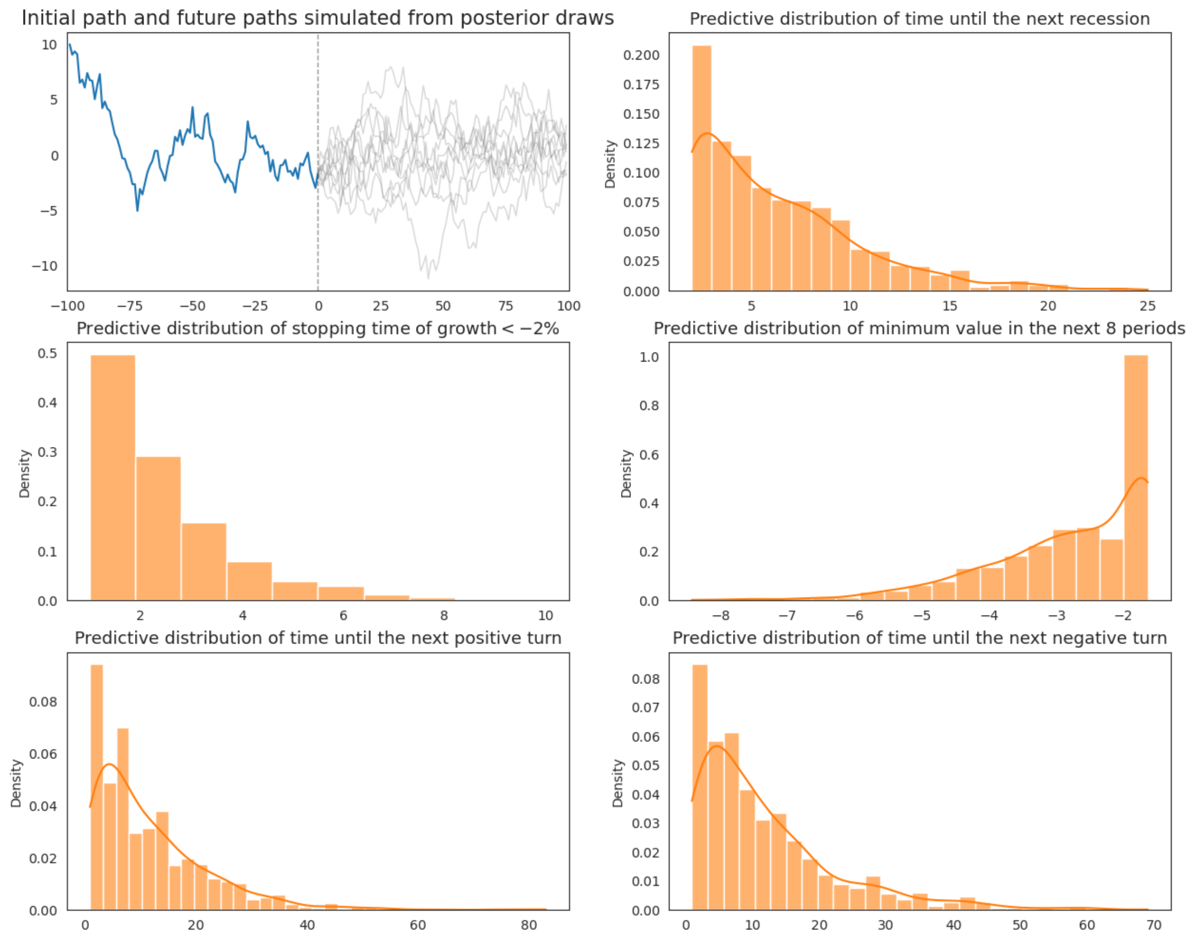
To approximate the intergration on the right side of (20.4), we repeatedly draw parameters from the joint posterior distribution each time we simulate a sequence of future values from model (20.1).

```
def plot_extended_Wecker(post_samples, initial_path, N, ax):
    """
    Plot the extended Wecker's predictive distribution
    """
    # Select a sample
    index = np.random.choice(np.arange(len(post_samples['rho'])), N + 1,␣
↪replace=False)
    rho_sample = post_samples['rho'][index]
    sigma_sample = post_samples['sigma'][index]

    # Store outcomes
    next_reces = np.zeros(N)
    severe_rec = np.zeros(N)
    min_vals = np.zeros(N)
    next_up_turn, next_down_turn = np.zeros(N), np.zeros(N)
```

(continues on next page)

(continued from previous page)

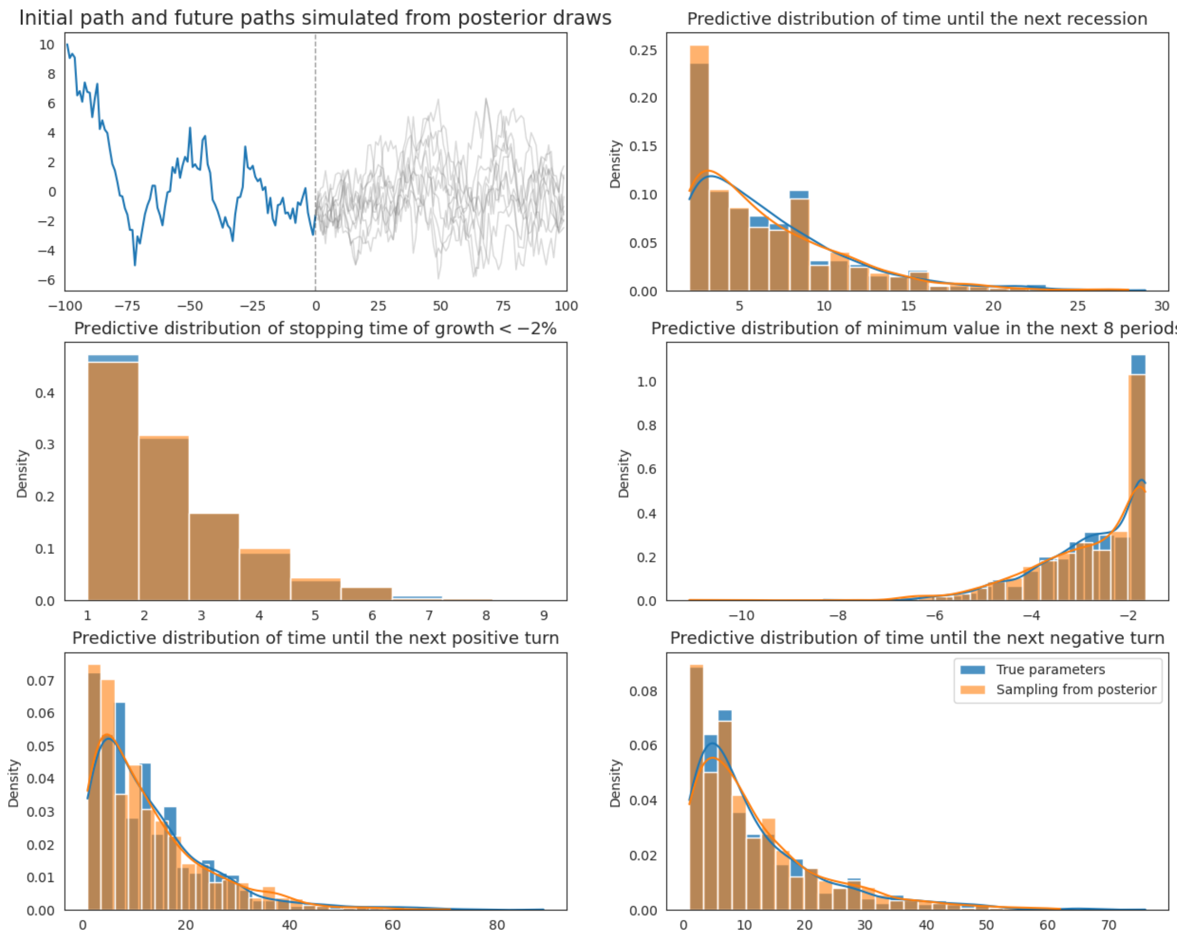
```
    # Plot
    ax[0, 0].set_title("Initial path and future paths simulated from posterior draws",
↪ fontsize=15)
    ax[0, 0].plot(np.arange(-T0 + 1, 1), initial_path)
    ax[0, 0].set_xlim([-T0, T1])
    ax[0, 0].axvline(0, linestyle='--', alpha=.4, color='k', lw=1)

    # Simulate future paths
    for n in range(N):
        sim_path = AR1_simulate(rho_sample[n], sigma_sample[n], initial_path[-1], T1)
        next_reces[n] = next_recession(np.hstack([initial_path[-3:-1], sim_path]))
        severe_rec[n] = severe_recession(sim_path)
        min_vals[n] = minimum_value(sim_path)
        next_up_turn[n], next_down_turn[n] = next_turning_point(sim_path)

        if n % (N / 10) == 0:
            ax[0, 0].plot(np.arange(T1), sim_path, color='gray', alpha=.3, lw=1)

    # Return next_up_turn, next_down_turn
    sns.histplot(next_reces, kde=True, stat='density', ax=ax[0, 1], alpha=.6,␣
↪color=colors[1], label='Sampling from posterior')
    ax[0, 1].set_title("Predictive distribution of time until the next recession",␣
↪fontsize=13)

    sns.histplot(severe_rec, kde=False, stat='density', ax=ax[1, 0], binwidth=.9,␣
↪alpha=.6, color=colors[1], label='Sampling from posterior')
    ax[1, 0].set_title(r"Predictive distribution of stopping time of growth$<-2\%$",␣
↪fontsize=13)

    sns.histplot(min_vals, kde=True, stat='density', ax=ax[1, 1], alpha=.6,␣
↪color=colors[1], label='Sampling from posterior')
    ax[1, 1].set_title("Predictive distribution of minimum value in the next 8 periods
↪", fontsize=13)

    sns.histplot(next_up_turn, kde=True, stat='density', ax=ax[2, 0], alpha=.6,␣
↪color=colors[1], label='Sampling from posterior')
    ax[2, 0].set_title("Predictive distribution of time until the next positive turn",
↪ fontsize=13)

    sns.histplot(next_down_turn, kde=True, stat='density', ax=ax[2, 1], alpha=.6,␣
↪color=colors[1], label='Sampling from posterior')
    ax[2, 1].set_title("Predictive distribution of time until the next negative turn",
↪ fontsize=13)

fig, ax = plt.subplots(3, 2, figsize=(15, 12))
plot_extended_Wecker(post_samples, initial_path, 1000, ax)
plt.show()
```



20.9 Comparison

Finally, we plot both the original Wecker method and the extended method with parameter values drawn from the posterior together to compare the differences that emerge from pretending to know parameter values when they are actually uncertain.

```
fig, ax = plt.subplots(3, 2, figsize=(15,12))
plot_Wecker(initial_path, 1000, ax)
ax[0, 0].clear()
plot_extended_Wecker(post_samples, initial_path, 1000, ax)
plt.legend()
plt.show()
```


Initial path and future paths simulated from posterior draws
Predictive distribution of time until the next recession
Predictive distribution of stopping time of growth < −2%
Predictive distribution of minimum value in the next 8 periods
Predictive distribution of time until the next positive turn
Predictive distribution of time until the next negative turn
Density
True parameters
Sampling from posterior

Part IV

Statistics and Information

CHAPTER

TWENTYONE

STATISTICAL DIVERGENCE MEASURES

Contents

21.1 Overview

A statistical divergence quantifies discrepancies between two distinct probability distributions that can be challenging to distinguish for the following reason:

- every event that has positive probability under one of the distributions also has positive probability under the other distribution
- this means that there is no "smoking gun" event whose occurrence tells a statistician that one of the probability distributions surely governs the data

A statistical divergence is a **function** that maps two probability distributions into a nonnegative real number.

Statistical divergence functions play important roles in statistics, information theory, and what many people now call "machine learning".

This lecture describes three divergence measures:

- **Kullback–Leibler (KL) divergence**
- **Jensen–Shannon (JS) divergence**
- **Chernoff entropy**

These will appear in several quantecon lectures.

Let's start by importing the necessary Python tools.

```
import matplotlib.pyplot as plt
import numpy as np
from numba import vectorize, jit
from math import gamma
from scipy.integrate import quad
from scipy.optimize import minimize_scalar
import pandas as pd
from IPython.display import display, Math
```

21.2 Primer on entropy, cross-entropy, KL divergence

Before diving in, we'll introduce some useful concepts in a simple setting.

We'll temporarily assume that f and g are two probability mass functions for discrete random variables on state space $I = \{1, 2, \ldots, n\}$ that satisfy $f_i \geq 0, \sum_i f_i = 1, g_i \geq 0, \sum_i g_i = 1$.

We follow some statisticians and information theorists who define the **surprise** or **surprisal** associated with having observed a single draw $x = i$ from distribution f as

$$\log\left(\frac{1}{f_i}\right)$$

They then define the **information** that you can anticipate to gather from observing a single realization as the expected surprisal

$$H(f) = \sum_i f_i \log\left(\frac{1}{f_i}\right).$$

Claude Shannon [Shannon, 1948] called $H(f)$ the **entropy** of distribution f.

Note

By maximizing $H(f)$ with respect to $\{f_1, f_2, \ldots, f_n\}$ subject to $\sum_i f_i = 1$, we can verify that the distribution that maximizes entropy is the uniform distribution $f_i = \frac{1}{n}$. Entropy $H(f)$ for the uniform distribution evidently equals $-\log(n)$.

Kullback and Leibler [Kullback and Leibler, 1951] define the amount of information that a single draw of x provides for distinguishing f from g as the log likelihood ratio

$$\log \frac{f(x)}{g(x)}$$

The following two concepts are widely used to compare two distributions f and g.

Cross-Entropy:

$$H(f, g) = -\sum_i f_i \log g_i \tag{21.1}$$

Kullback-Leibler (KL) Divergence:

$$D_{KL}(f \parallel g) = \sum_i f_i \log\left[\frac{f_i}{g_i}\right] \tag{21.2}$$

These concepts are related by the following equality.

$$D_{KL}(f \parallel g) = H(f,g) - H(f) \tag{21.3}$$

To prove (21.3), note that

$$\begin{aligned}
D_{KL}(f \parallel g) &= \sum_i f_i \log \left[\frac{f_i}{g_i}\right] && (21.4)\\
&= \sum_i f_i \left[\log f_i - \log g_i\right] && (21.5)\\
&= \sum_i f_i \log f_i - \sum_i f_i \log g_i && (21.6)\\
&= -H(f) + H(f,g) && (21.7)\\
&= H(f,g) - H(f) && (21.8)
\end{aligned}$$

Remember that $H(f)$ is the anticipated surprisal from drawing x from f.

Then the above equation tells us that the KL divergence is an anticipated "excess surprise" that comes from anticipating that x is drawn from f when it is actually drawn from g.

21.3 Two Beta distributions: running example

We'll use Beta distributions extensively to illustrate concepts.

The Beta distribution is particularly convenient as it's defined on $[0,1]$ and exhibits diverse shapes by appropriately choosing its two parameters.

The density of a Beta distribution with parameters a and b is given by

$$f(z;a,b) = \frac{\Gamma(a+b)z^{a-1}(1-z)^{b-1}}{\Gamma(a)\Gamma(b)} \quad \text{where} \quad \Gamma(p) := \int_0^\infty x^{p-1}e^{-x}dx$$

We introduce two Beta distributions $f(x)$ and $g(x)$, which we will use to illustrate the different divergence measures.

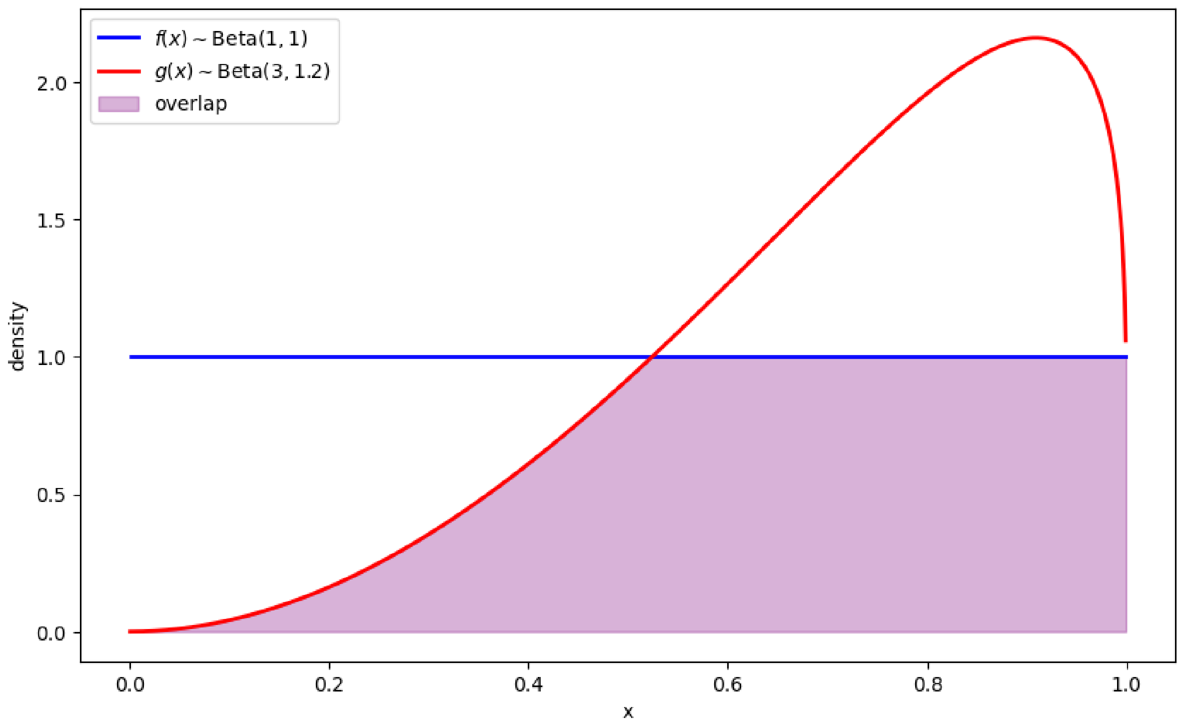
Let's define parameters and density functions in Python

```
# Parameters in the two Beta distributions
F_a, F_b = 1, 1
G_a, G_b = 3, 1.2

@vectorize
def p(x, a, b):
    r = gamma(a + b) / (gamma(a) * gamma(b))
    return r * x** (a-1) * (1 - x) ** (b-1)

# The two density functions
f = jit(lambda x: p(x, F_a, F_b))
g = jit(lambda x: p(x, G_a, G_b))

# Plot the distributions
x_range = np.linspace(0.001, 0.999, 1000)
f_vals = [f(x) for x in x_range]
g_vals = [g(x) for x in x_range]

plt.figure(figsize=(10, 6))
```

(continues on next page)

(continued from previous page)

```
plt.plot(x_range, f_vals, 'b-', linewidth=2, label=r'$f(x) \sim \text{Beta}(1,1)$')
plt.plot(x_range, g_vals, 'r-', linewidth=2, label=r'$g(x) \sim \text{Beta}(3,1.2)$')

# Fill overlap region
overlap = np.minimum(f_vals, g_vals)
plt.fill_between(x_range, 0, overlap, alpha=0.3, color='purple', label='overlap')

plt.xlabel('x')
plt.ylabel('density')
plt.legend()
plt.show()
```



21.4 Kullback–Leibler divergence

Our first divergence function is the **Kullback–Leibler (KL) divergence**.

For probability densities (or pmfs) f and g it is defined by

$$D_{KL}(f\|g) = KL(f,g) = \int f(x) \log \frac{f(x)}{g(x)}\, dx.$$

We can interpret $D_{KL}(f\|g)$ as the expected excess log loss (expected excess surprisal) incurred when we use g while the data are generated by f.

It has several important properties:

- Non-negativity (Gibbs' inequality): $D_{KL}(f\|g) \geq 0$ with equality if and only if $f = g$ almost everywhere.
- Asymmetry: $D_{KL}(f\|g) \neq D_{KL}(g\|f)$ in general (hence it is not a metric)

- Information decomposition: $D_{KL}(f\|g) = H(f,g) - H(f)$, where $H(f,g)$ is the cross entropy and $H(f)$ is the Shannon entropy of f.
- Chain rule: For joint distributions $f(x,y)$ and $g(x,y)$, $D_{KL}(f(x,y)\|g(x,y)) = D_{KL}(f(x)\|g(x)) + E_f\left[D_{KL}(f(y|x)\|g(y|x))\right]$

KL divergence plays a central role in statistical inference, including model selection and hypothesis testing.

Likelihood Ratio Processes describes a link between KL divergence and the expected log likelihood ratio, and the lecture *A Problem that Stumped Milton Friedman* connects it to the test performance of the sequential probability ratio test.

Let's compute the KL divergence between our example distributions f and g.

```python
def compute_KL(f, g):
    """
    Compute KL divergence KL(f, g) via numerical integration
    """
    def integrand(w):
        fw = f(w)
        gw = g(w)
        return fw * np.log(fw / gw)
    val, _ = quad(integrand, 1e-5, 1-1e-5)
    return val

# Compute KL divergences between our example distributions
kl_fg = compute_KL(f, g)
kl_gf = compute_KL(g, f)

print(f"KL(f, g) = {kl_fg:.4f}")
print(f"KL(g, f) = {kl_gf:.4f}")
```

```
KL(f, g) = 0.7590
KL(g, f) = 0.3436
```

The asymmetry of KL divergence has important practical implications.

$D_{KL}(f\|g)$ penalizes regions where $f > 0$ but g is close to zero, reflecting the cost of using g to model f and vice versa.

21.5 Jensen-Shannon divergence

Sometimes we want a symmetric measure of divergence that captures the difference between two distributions without favoring one over the other.

This often arises in applications like clustering, where we want to compare distributions without assuming one is the true model.

The **Jensen-Shannon (JS) divergence** symmetrizes KL divergence by comparing both distributions to their mixture:

$$JS(f,g) = \frac{1}{2}D_{KL}(f\|m) + \frac{1}{2}D_{KL}(g\|m), \quad m = \frac{1}{2}(f+g).$$

where m is a mixture distribution that averages f and g

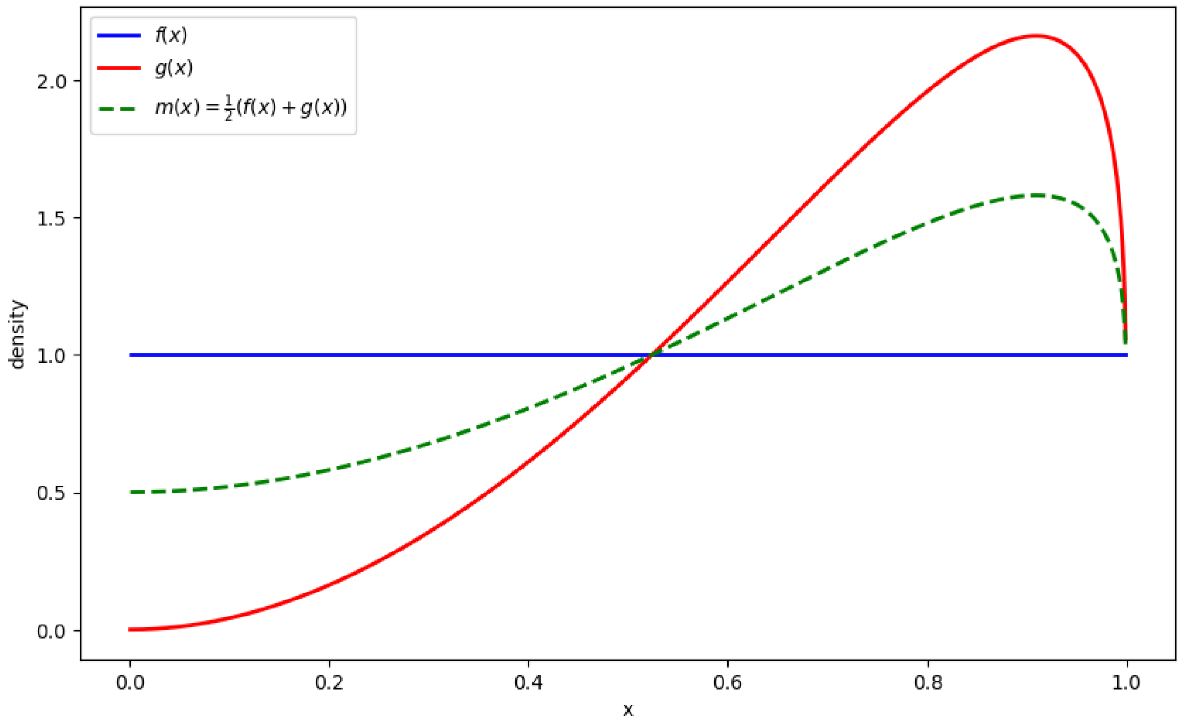
Let's also visualize the mixture distribution m:

```python
def m(x):
    return 0.5 * (f(x) + g(x))
```

(continues on next page)

(continued from previous page)

```
m_vals = [m(x) for x in x_range]

plt.figure(figsize=(10, 6))
plt.plot(x_range, f_vals, 'b-', linewidth=2, label=r'$f(x)$')
plt.plot(x_range, g_vals, 'r-', linewidth=2, label=r'$g(x)$')
plt.plot(x_range, m_vals, 'g--', linewidth=2, label=r'$m(x) = \frac{1}{2}(f(x) + g(x))
↪$')

plt.xlabel('x')
plt.ylabel('density')
plt.legend()
plt.show()
```



The JS divergence has several useful properties:

- Symmetry: $JS(f, g) = JS(g, f)$.
- Boundedness: $0 \leq JS(f, g) \leq \log 2$.
- Its square root $\sqrt{JS}$ is a metric (Jensen–Shannon distance) on the space of probability distributions.
- JS divergence equals the mutual information between a binary random variable $Z \sim \text{Bernoulli}(1/2)$ indicating the source and a sample X drawn from f if $Z = 0$ or from g if $Z = 1$.

The Jensen–Shannon divergence plays a key role in the optimization of certain generative models, as it is bounded, symmetric, and smoother than KL divergence, often providing more stable gradients for training.

Let's compute the JS divergence between our example distributions f and g

```
def compute_JS(f, g):
    """Compute Jensen-Shannon divergence."""
    def m(w):
        return 0.5 * (f(w) + g(w))
```

(continues on next page)

(continued from previous page)

```
    js_div = 0.5 * compute_KL(f, m) + 0.5 * compute_KL(g, m)
    return js_div

js_div = compute_JS(f, g)
print(f"Jensen-Shannon divergence JS(f,g) = {js_div:.4f}")
```

```
Jensen-Shannon divergence JS(f,g) = 0.0984
```

We can easily generalize to more than two distributions using the generalized Jensen-Shannon divergence with weights $\alpha = (\alpha_i)_{i=1}^n$:

$$JS_\alpha(f_1, \ldots, f_n) = H\left(\sum_{i=1}^n \alpha_i f_i\right) - \sum_{i=1}^n \alpha_i H(f_i)$$

where:

- $\alpha_i \geq 0$ and $\sum_{i=1}^n \alpha_i = 1$, and
- $H(f) = -\int f(x) \log f(x) dx$ is the **Shannon entropy** of distribution f

21.6 Chernoff entropy

Chernoff entropy originates from early applications of the theory of large deviations, which refines central limit approximations by providing exponential decay rates for rare events.

For densities f and g the Chernoff entropy is

$$C(f,g) = -\log \min_{\phi \in (0,1)} \int f^\phi(x) g^{1-\phi}(x)\, dx.$$

Remarks:

- The inner integral is the **Chernoff coefficient**.
- At $\phi = 1/2$ it becomes the **Bhattacharyya coefficient** $\int \sqrt{fg}$.
- In binary hypothesis testing with T iid observations, the optimal error probability decays as $e^{-C(f,g)T}$.

We will see an example of the third point in the lecture *Likelihood Ratio Processes*, where we study the Chernoff entropy in the context of model selection.

Let's compute the Chernoff entropy between our example distributions f and g.

```
def chernoff_integrand(ϕ, f, g):
    """Integral entering Chernoff entropy for a given ϕ."""
    def integrand(w):
        return f(w)**ϕ * g(w)**(1-ϕ)
    result, _ = quad(integrand, 1e-5, 1-1e-5)
    return result

def compute_chernoff_entropy(f, g):
    """Compute Chernoff entropy C(f,g)."""
    def objective(ϕ):
        return chernoff_integrand(ϕ, f, g)
    result = minimize_scalar(objective, bounds=(1e-5, 1-1e-5), method='bounded')
    min_value = result.fun
```

(continues on next page)

(continued from previous page)

```
    ϕ_optimal = result.x
    chernoff_entropy = -np.log(min_value)
    return chernoff_entropy, ϕ_optimal

C_fg, ϕ_optimal = compute_chernoff_entropy(f, g)
print(f"Chernoff entropy C(f,g) = {C_fg:.4f}")
print(f"Optimal ϕ = {ϕ_optimal:.4f}")
```

```
Chernoff entropy C(f,g) = 0.1212
Optimal ϕ = 0.5969
```

21.7 Comparing divergence measures

We now compare these measures across several pairs of Beta distributions

Pair (f, g)	KL(f, g)	KL(g, f)	JS	C
$\mathrm{Beta}(1,1), \mathrm{Beta}(1.1,1.05)$	0.0028	0.0026	0.0007	0.0007
$\mathrm{Beta}(1,1), \mathrm{Beta}(1.2,1.1)$	0.0105	0.0092	0.0024	0.0025
$\mathrm{Beta}(1,1), \mathrm{Beta}(0.9,0.8)$	0.0143	0.0166	0.0038	0.0039
$\mathrm{Beta}(1,1), \mathrm{Beta}(1.5,1.2)$	0.0589	0.0437	0.0121	0.0126
$\mathrm{Beta}(1,1), \mathrm{Beta}(0.7,0.6)$	0.0673	0.0924	0.0186	0.0201
$\mathrm{Beta}(1,1), \mathrm{Beta}(2,1.5)$	0.1781	0.1081	0.0309	0.0339
$\mathrm{Beta}(1,1), \mathrm{Beta}(0.5,0.5)$	0.1448	0.2190	0.0400	0.0461
$\mathrm{Beta}(1,1), \mathrm{Beta}(2.5,1.8)$	0.3323	0.1731	0.0502	0.0577
$\mathrm{Beta}(1,1), \mathrm{Beta}(0.3,0.4)$	0.3317	0.5572	0.0869	0.1203
$\mathrm{Beta}(1,1), \mathrm{Beta}(3,1.2)$	0.7590	0.3436	0.0984	0.1212
$\mathrm{Beta}(1,1), \mathrm{Beta}(0.3,0.3)$	0.3935	0.6516	0.1008	0.1456
$\mathrm{Beta}(1,1), \mathrm{Beta}(4,1)$	1.6134	0.6362	0.1733	0.2341
$\mathrm{Beta}(1,1), \mathrm{Beta}(0.1,0.2)$	0.9811	1.0036	0.1783	0.4556
$\mathrm{Beta}(1,1), \mathrm{Beta}(5,1)$	2.3901	0.8094	0.2162	0.3128

We can clearly see co-movement across the divergence measures as we vary the parameters of the Beta distributions.

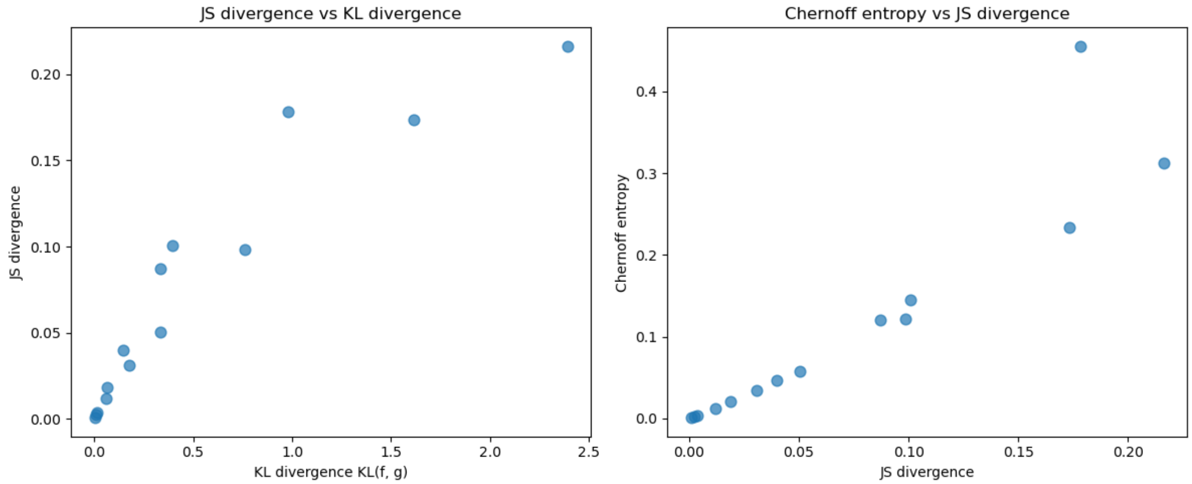
Next we visualize relationships among KL, JS, and Chernoff entropy.

```
kl_fg_values = [float(result['KL(f, g)']) for result in results]
js_values = [float(result['JS']) for result in results]
chernoff_values = [float(result['C']) for result in results]

fig, axes = plt.subplots(1, 2, figsize=(12, 5))

axes[0].scatter(kl_fg_values, js_values, alpha=0.7, s=60)
axes[0].set_xlabel('KL divergence KL(f, g)')
axes[0].set_ylabel('JS divergence')
axes[0].set_title('JS divergence vs KL divergence')

axes[1].scatter(js_values, chernoff_values, alpha=0.7, s=60)
axes[1].set_xlabel('JS divergence')
axes[1].set_ylabel('Chernoff entropy')
axes[1].set_title('Chernoff entropy vs JS divergence')
```

(continues on next page)

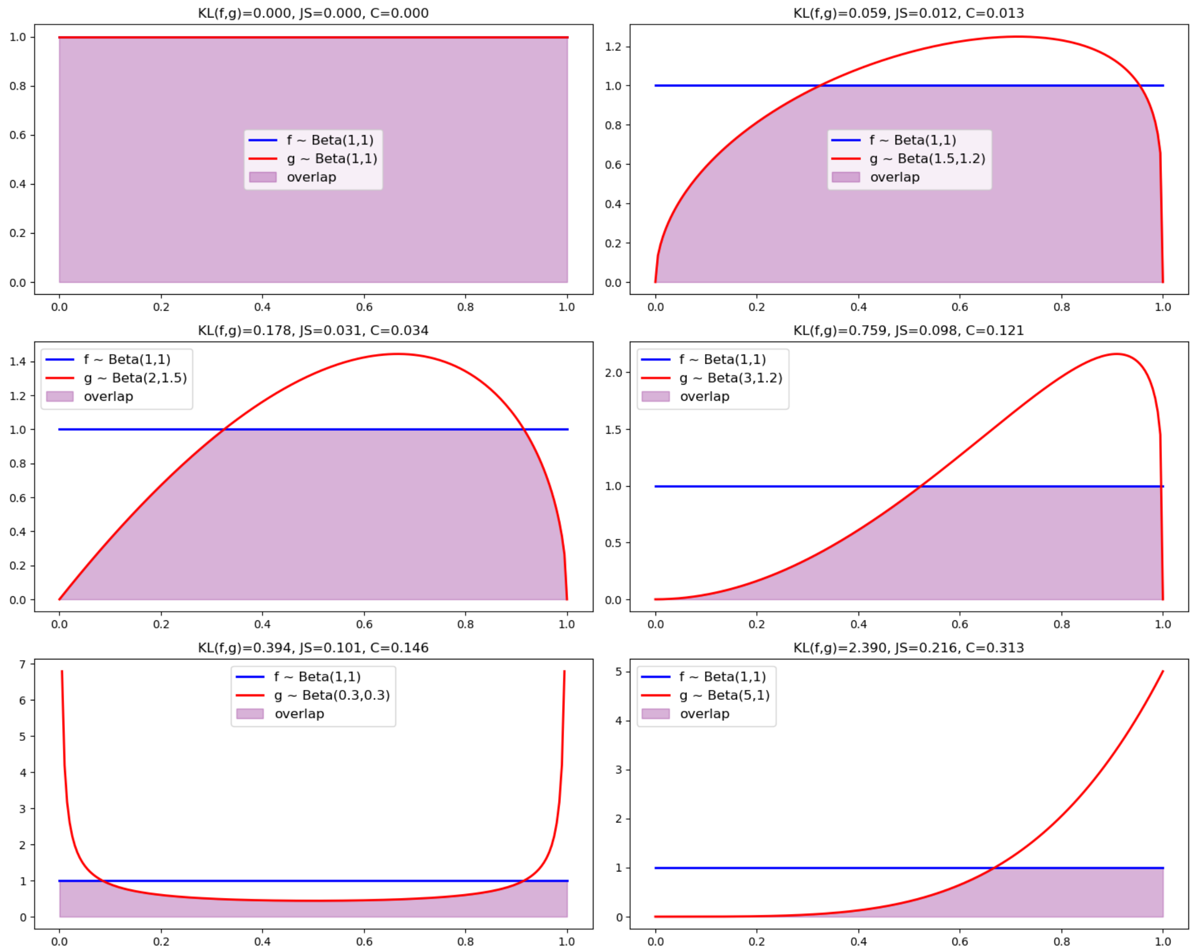
(continued from previous page)

```
plt.tight_layout()
plt.show()
```



We now generate plots illustrating how overlap visually diminishes as divergence measures increase.

```
param_grid = [
    ((1, 1), (1, 1)),
    ((1, 1), (1.5, 1.2)),
    ((1, 1), (2, 1.5)),
    ((1, 1), (3, 1.2)),
    ((1, 1), (0.3, 0.3)),
    ((1, 1), (5, 1))
]
```



21.8 KL divergence and maximum-likelihood estimation

Given a sample of n observations $X = \{x_1, x_2, \ldots, x_n\}$, the **empirical distribution** is

$$p_e(x) = \frac{1}{n} \sum_{i=1}^{n} \delta(x - x_i)$$

where $\delta(x - x_i)$ is the Dirac delta function centered at x_i:

$$\delta(x - x_i) = \begin{cases} +\infty & \text{if } x = x_i \\ 0 & \text{if } x \neq x_i \end{cases}$$

- **Discrete probability measure**: Assigns probability $\frac{1}{n}$ to each observed data point
- **Empirical expectation**: $\langle X \rangle_{p_e} = \frac{1}{n} \sum_{i=1}^{n} x_i = \bar{\mu}$
- **Support**: Only on the observed data points $\{x_1, x_2, \ldots, x_n\}$

The KL divergence from the empirical distribution p_e to a parametric model $p_\theta(x)$ is:

$$D_{KL}(p_e \parallel p_\theta) = \int p_e(x) \log \frac{p_e(x)}{p_\theta(x)} dx$$

Using the mathematics of the Dirac delta function, it follows that

$$D_{KL}(p_e \parallel p_\theta) = \sum_{i=1}^{n} \frac{1}{n} \log \frac{\left(\frac{1}{n}\right)}{p_\theta(x_i)}$$

$$= \frac{1}{n} \sum_{i=1}^{n} \log \frac{1}{n} - \frac{1}{n} \sum_{i=1}^{n} \log p_\theta(x_i)$$

$$= -\log n - \frac{1}{n} \sum_{i=1}^{n} \log p_\theta(x_i)$$

Since the log-likelihood function for parameter θ is:

$$\ell(\theta; X) = \sum_{i=1}^{n} \log p_\theta(x_i),$$

it follows that maximum likelihood chooses parameters to minimize

$$D_{KL}(p_e \parallel p_\theta)$$

Thus, MLE is equivalent to minimizing the KL divergence from the empirical distribution to the statistical model p_θ.

21.9 Related lectures

This lecture has introduced tools that we'll encounter elsewhere.

- Other quantecon lectures that apply connections between divergence measures and statistical inference include *Likelihood Ratio Processes*, *A Problem that Stumped Milton Friedman*, and *Incorrect Models*.
- Statistical divergence functions also take center stage in *Heterogeneous Beliefs and Financial Markets* that studies Lawrence Blume and David Easley's model of heterogeneous beliefs and financial markets.

CHAPTER

TWENTYTWO

LIKELIHOOD RATIO PROCESSES

Contents

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install --upgrade quantecon
```

22.1 Overview

This lecture describes likelihood ratio processes and some of their uses.

We'll study the same setting that is also used in *this lecture on exchangeability*.

Among the things that we'll learn are

- How a likelihood ratio process is a key ingredient in frequentist hypothesis testing
- How a **receiver operator characteristic curve** summarizes information about a false alarm probability and power in frequentist hypothesis testing
- How a statistician can combine frequentist probabilities of type I and type II errors to form posterior probabilities of mistakes in a model selection or in an individual-classification problem

- How to use a Kullback-Leibler divergence to quantify the difference between two probability distributions with the same support
- How during World War II the United States Navy devised a decision rule for doing quality control on lots of ammunition, a topic that sets the stage for *this lecture*
- A peculiar property of likelihood ratio processes

Let's start by importing some Python tools.

```
import matplotlib.pyplot as plt
import numpy as np
from numba import vectorize, jit
from math import gamma
from scipy.integrate import quad
from scipy.optimize import brentq, minimize_scalar
from scipy.stats import beta as beta_dist
import pandas as pd
from IPython.display import display, Math
import quantecon as qe
```

22.2 Likelihood Ratio Process

A nonnegative random variable W has one of two probability density functions, either f or g.

Before the beginning of time, nature once and for all decides whether she will draw a sequence of IID draws from either f or g.

We will sometimes let q be the density that nature chose once and for all, so that q is either f or g, permanently.

Nature knows which density it permanently draws from, but we the observers do not.

We know both f and g but we don't know which density nature chose.

But we want to know.

To do that, we use observations.

We observe a sequence $\{w_t\}_{t=1}^T$ of T IID draws that we know came from either f or g.

We want to use these observations to infer whether nature chose f or g.

A **likelihood ratio process** is a useful tool for this task.

To begin, we define a key component of a likelihood ratio process, namely, the time t likelihood ratio as the random variable

$$\ell(w_t) = \frac{f\left(w_t\right)}{g\left(w_t\right)}, \quad t \geq 1.$$

We assume that f and g both put positive probabilities on the same intervals of possible realizations of the random variable W.

That means that under the g density, $\ell(w_t) = \frac{f(w_t)}{g(w_t)}$ is a nonnegative random variable with mean 1.

A **likelihood ratio process** for sequence $\{w_t\}_{t=1}^{\infty}$ is defined as

$$L\left(w^t\right) = \prod_{i=1}^{t} \ell(w_i),$$

where $w^t = \{w_1, \ldots, w_t\}$ is a history of observations up to and including time t.

Sometimes for shorthand we'll write $L_t = L(w^t)$.

Notice that the likelihood process satisfies the *recursion*

$$L(w^t) = \ell(w_t)L(w^{t-1}).$$

The likelihood ratio and its logarithm are key tools for making inferences using a classic frequentist approach due to Neyman and Pearson [Neyman and Pearson, 1933].

To help us appreciate how things work, the following Python code evaluates f and g as two different Beta distributions, then computes and simulates an associated likelihood ratio process by generating a sequence w^t from one of the two probability distributions, for example, a sequence of IID draws from g.

```
# Parameters for the two Beta distributions
F_a, F_b = 1, 1
G_a, G_b = 3, 1.2

@vectorize
def p(x, a, b):
    """Beta distribution density function."""
    r = gamma(a + b) / (gamma(a) * gamma(b))
    return r * x** (a-1) * (1 - x) ** (b-1)

f = jit(lambda x: p(x, F_a, F_b))
g = jit(lambda x: p(x, G_a, G_b))

def create_beta_density(a, b):
    """Create a beta density function with specified parameters."""
    return jit(lambda x: p(x, a, b))

def likelihood_ratio(w, f_func, g_func):
    """Compute likelihood ratio for observation(s) w."""
    return f_func(w) / g_func(w)

@jit
def simulate_likelihood_ratios(a, b, f_func, g_func, T=50, N=500):
    """
    Generate N sets of T observations of the likelihood ratio.
    """
    l_arr = np.empty((N, T))
    for i in range(N):
        for j in range(T):
            w = np.random.beta(a, b)
            l_arr[i, j] = f_func(w) / g_func(w)
    return l_arr

def simulate_sequences(distribution, f_func, g_func,
        F_params=(1, 1), G_params=(3, 1.2), T=50, N=500):
    """
    Generate N sequences of T observations from specified distribution.
    """
    if distribution == 'f':
        a, b = F_params
    elif distribution == 'g':
        a, b = G_params
    else:
        raise ValueError("distribution must be 'f' or 'g'")

    l_arr = simulate_likelihood_ratios(a, b, f_func, g_func, T, N)
```

(continues on next page)

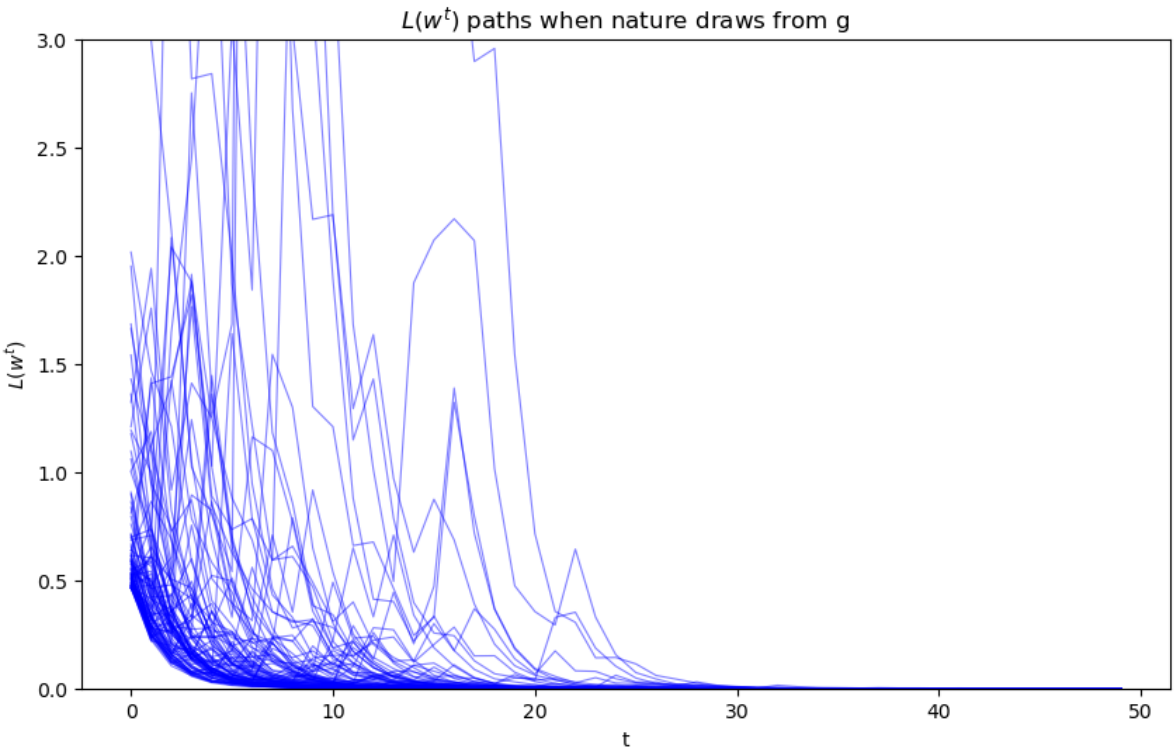
(continued from previous page)

```
    l_seq = np.cumprod(l_arr, axis=1)
    return l_arr, l_seq

def plot_likelihood_paths(l_seq, title="Likelihood ratio paths",
                        ylim=None, n_paths=None):
    """Plot likelihood ratio paths."""
    N, T = l_seq.shape
    n_show = n_paths or min(N, 100)

    plt.figure(figsize=(10, 6))
    for i in range(n_show):
        plt.plot(range(T), l_seq[i, :], color='b', lw=0.8, alpha=0.5)

    if ylim:
        plt.ylim(ylim)
    plt.title(title)
    plt.xlabel('t')
    plt.ylabel('$L(w^t)$')
    plt.show()
```

22.3 Nature permanently draws from density g

We first simulate the likelihood ratio process when nature permanently draws from g.

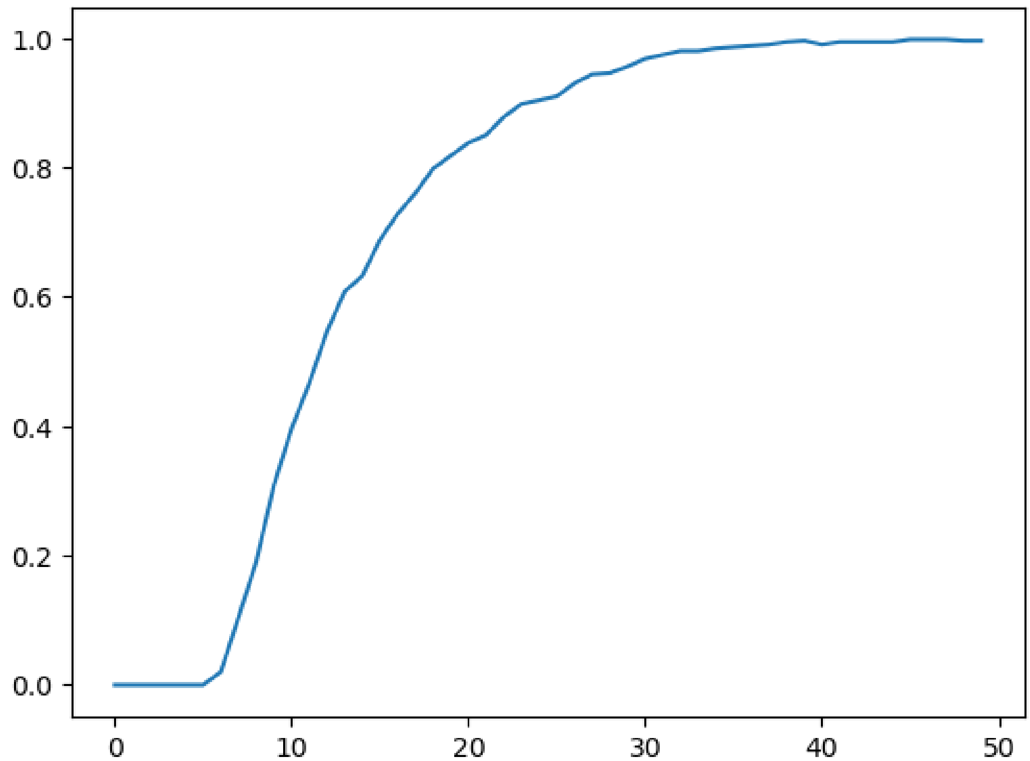
```
# Simulate when nature draws from g
l_arr_g, l_seq_g = simulate_sequences('g', f, g, (F_a, F_b), (G_a, G_b))
plot_likelihood_paths(l_seq_g,
                     title="$L(w^{t})$ paths when nature draws from g",
                     ylim=[0, 3])
```



Evidently, as sample length T grows, most probability mass shifts toward zero

To see this more clearly, we plot over time the fraction of paths $L\left(w^t\right)$ that fall in the interval $[0, 0.01]$.

```
N, T = l_arr_g.shape
plt.plot(range(T), np.sum(l_seq_g <= 0.01, axis=0) / N)
plt.show()
```



Despite the evident convergence of most probability mass to a very small interval near 0, the unconditional mean of $L(w^t)$ under probability density g is identically 1 for all t.

To verify this assertion, first notice that as mentioned earlier the unconditional mean $E[\ell(w_t) \mid q = g]$ is 1 for all t:

$$\begin{aligned} E[\ell(w_t) \mid q = g] &= \int \frac{f(w_t)}{g(w_t)} g(w_t) \, dw_t \\ &= \int f(w_t) \, dw_t \\ &= 1, \end{aligned}$$

which immediately implies

$$\begin{aligned} E[L(w^1) \mid q = g] &= E[\ell(w_1) \mid q = g] \\ &= 1. \end{aligned}$$

Because $L(w^t) = \ell(w_t) L(w^{t-1})$ and $\{w_t\}_{t=1}^t$ is an IID sequence, we have

$$\begin{aligned} E[L(w^t) \mid q = g] &= E[L(w^{t-1}) \ell(w_t) \mid q = g] \\ &= E[L(w^{t-1}) E[\ell(w_t) \mid q = g, w^{t-1}] \mid q = g] \\ &= E[L(w^{t-1}) E[\ell(w_t) \mid q = g] \mid q = g] \\ &= E[L(w^{t-1}) \mid q = g] \end{aligned}$$

for any $t \geq 1$.

Mathematical induction implies $E[L(w^t) \mid q = g] = 1$ for all $t \geq 1$.

22.4 Peculiar property

How can $E\left[L\left(w^t\right) \bigm| q=g\right]=1$ possibly be true when most probability mass of the likelihood ratio process is piling up near 0 as $t \rightarrow +\infty$?

The answer is that as $t \rightarrow +\infty$, the distribution of L_t becomes more and more fat-tailed: enough mass shifts to larger and larger values of L_t to make the mean of L_t continue to be one despite most of the probability mass piling up near 0.

To illustrate this peculiar property, we simulate many paths and calculate the unconditional mean of $L\left(w^t\right)$ by averaging across these many paths at each t.

```
l_arr_g, l_seq_g = simulate_sequences('g',
                   f, g, (F_a, F_b), (G_a, G_b), N=50000)
```

It would be useful to use simulations to verify that unconditional means $E\left[L\left(w^t\right)\right]$ equal unity by averaging across sample paths.

But it would be too computer-time-consuming for us to do that here simply by applying a standard Monte Carlo simulation approach.

The reason is that the distribution of $L\left(w^t\right)$ is extremely skewed for large values of t.

Because the probability density in the right tail is close to 0, it just takes too much computer time to sample enough points from the right tail.

We explain the problem in more detail in *this lecture*.

There we describe an alternative way to compute the mean of a likelihood ratio by computing the mean of a *different* random variable by sampling from a *different* probability distribution.

22.5 Nature permanently draws from density f

Now suppose that before time 0 nature permanently decided to draw repeatedly from density f.

While the mean of the likelihood ratio $\ell\left(w_t\right)$ under density g is 1, its mean under the density f exceeds one.

To see this, we compute

$$
\begin{aligned}
E\left[\ell\left(w_t\right) \bigm| q=f\right] &= \int \frac{f\left(w_t\right)}{g\left(w_t\right)} f\left(w_t\right) dw_t \\
&= \int \frac{f\left(w_t\right)}{g\left(w_t\right)} \frac{f\left(w_t\right)}{g\left(w_t\right)} g\left(w_t\right) dw_t \\
&= \int \ell\left(w_t\right)^2 g\left(w_t\right) dw_t \\
&= E\left[\ell\left(w_t\right)^2 \mid q=g\right] \\
&= E\left[\ell\left(w_t\right) \mid q=g\right]^2 + Var\left(\ell\left(w_t\right) \mid q=g\right) \\
&> E\left[\ell\left(w_t\right) \mid q=g\right]^2 = 1
\end{aligned}
$$

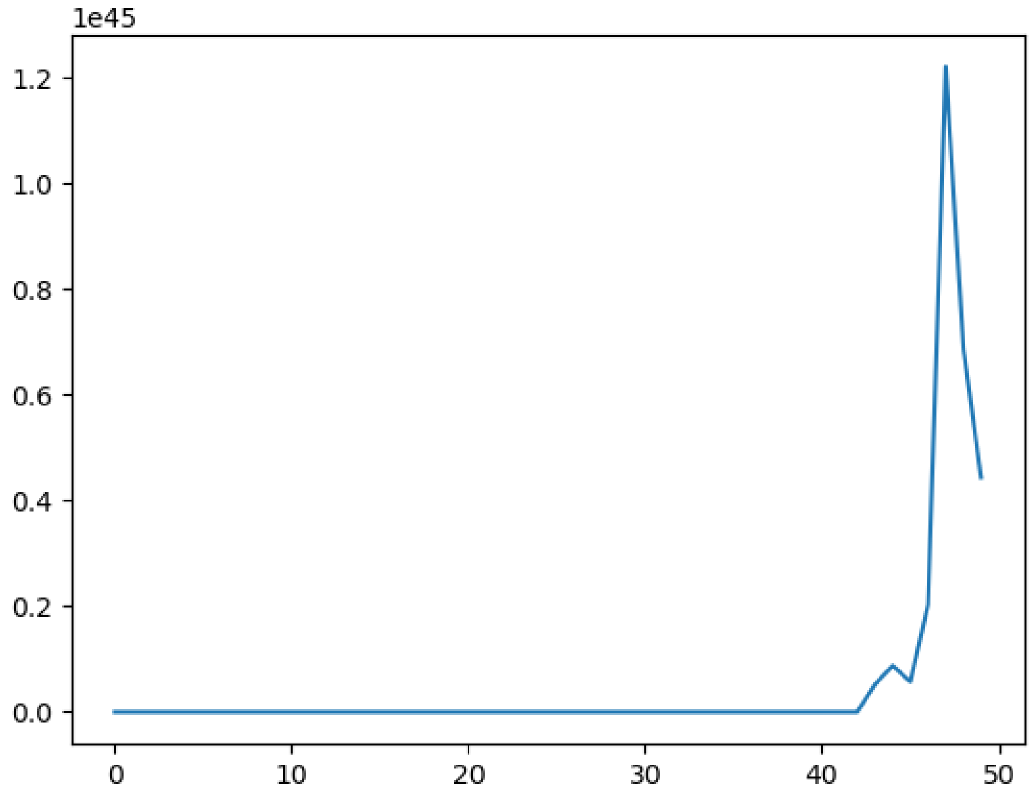
This in turn implies that the unconditional mean of the likelihood ratio process $L(w^t)$ diverges toward $+\infty$.

Simulations below confirm this conclusion.

Please note the scale of the y axis.

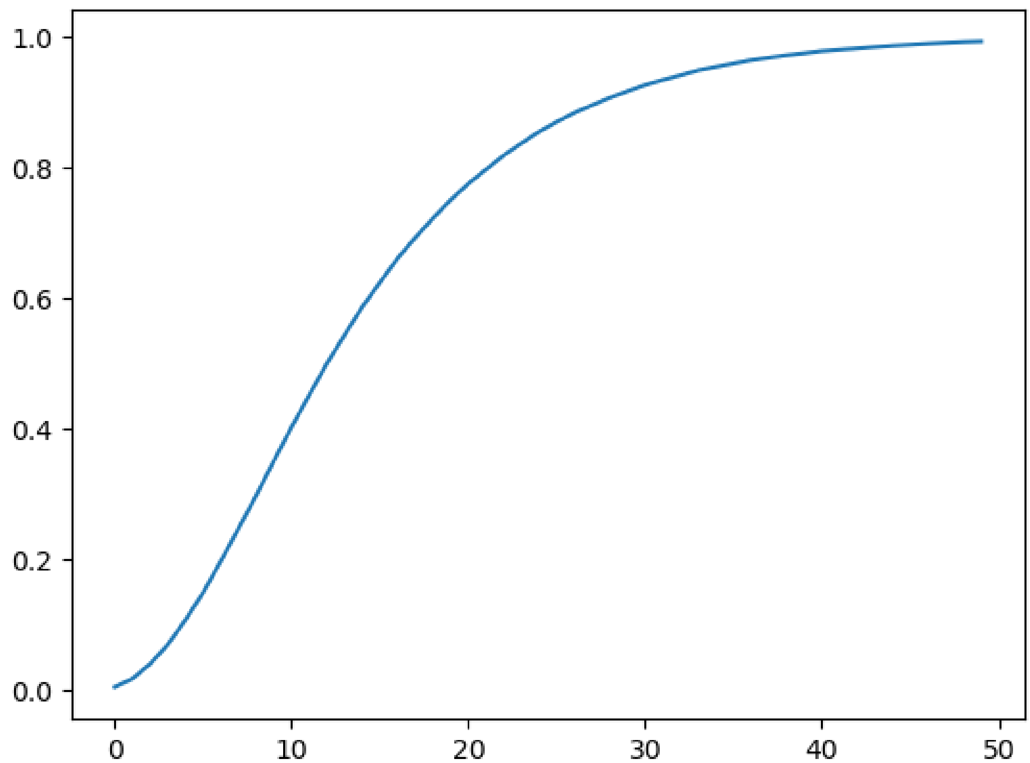
```
# Simulate when nature draws from f
l_arr_f, l_seq_f = simulate_sequences('f', f, g,
                            (F_a, F_b), (G_a, G_b), N=50000)
```

```
N, T = l_arr_f.shape
plt.plot(range(T), np.mean(l_seq_f, axis=0))
plt.show()
```



We also plot the probability that $L(w^t)$ falls into the interval $[10000, \infty)$ as a function of time and watch how fast probability mass diverges to $+\infty$.

```
plt.plot(range(T), np.sum(l_seq_f > 10000, axis=0) / N)
plt.show()
```



22.6 Likelihood ratio test

We now describe how to employ the machinery of Neyman and Pearson [Neyman and Pearson, 1933] to test the hypothesis that history w^t is generated by repeated IID draws from density f.

Denote q as the data generating process, so that $q = f$ or g.

Upon observing a sample $\{W_i\}_{i=1}^t$, we want to decide whether nature is drawing from g or from f by performing a (frequentist) hypothesis test.

We specify

- Null hypothesis H_0: $q = f$,
- Alternative hypothesis H_1: $q = g$.

Neyman and Pearson proved that the best way to test this hypothesis is to use a **likelihood ratio test** that takes the form:

- accept H_0 if $L(W^t) > c$,
- reject H_0 if $L(W^t) < c$,

where c is a given discrimination threshold.

Setting $c = 1$ is a common choice.

We'll discuss consequences of other choices of c below.

This test is *best* in the sense that it is **uniformly most powerful**.

To understand what this means, we have to define probabilities of two important events that allow us to characterize a test associated with a given threshold c.

The two probabilities are:

- Probability of a Type I error in which we reject H_0 when it is true:

$$\alpha \equiv \Pr \left\{ L\left(w^t\right) < c \mid q = f \right\}$$

- Probability of a Type II error in which we accept H_0 when it is false:

$$\beta \equiv \Pr \left\{ L\left(w^t\right) > c \mid q = g \right\}$$

These two probabilities underlie the following two concepts:

- Probability of false alarm (= significance level = probability of Type I error):

$$\alpha \equiv \Pr \left\{ L\left(w^t\right) < c \mid q = f \right\}$$

- Probability of detection (= power = 1 minus probability of Type II error):

$$1 - \beta \equiv \Pr \left\{ L\left(w^t\right) < c \mid q = g \right\}$$

The Neyman-Pearson Lemma states that among all possible tests, a likelihood ratio test maximizes the probability of detection for a given probability of false alarm.

Another way to say the same thing is that among all possible tests, a likelihood ratio test maximizes **power** for a given **significance level**.

We want a small probability of false alarm and a large probability of detection.

With sample size t fixed, we can change our two probabilities by adjusting c.

A troublesome "that's life" fact is that these two probabilities move in the same direction as we vary the critical value c.

Without specifying quantitative losses from making Type I and Type II errors, there is little that we can say about how we *should* trade off probabilities of the two types of mistakes.

We do know that increasing sample size t improves statistical inference.

Below we plot some informative figures that illustrate this.

We also present a classical frequentist method for choosing a sample size t.

Let's start with a case in which we fix the threshold c at 1.

```
c = 1
```

Below we plot empirical distributions of logarithms of the cumulative likelihood ratios simulated above, which are generated by either f or g.

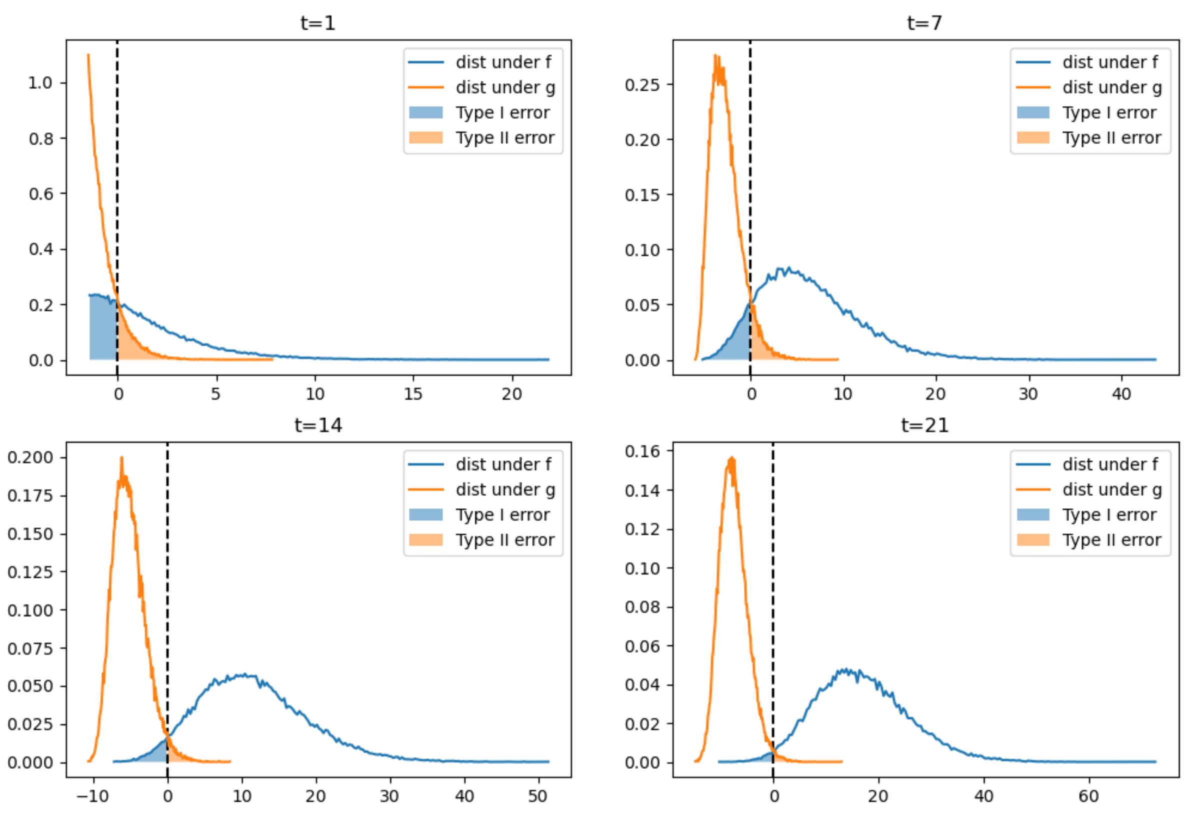
Taking logarithms has no effect on calculating the probabilities because the log is a monotonic transformation.

As t increases, the probabilities of making Type I and Type II errors both decrease, which is good.

This is because most of the probability mass of $\log(L(w^t))$ moves toward $-\infty$ when g is the data generating process, while $\log(L(w^t))$ goes to ∞ when data are generated by f.

That disparate behavior of $\log(L(w^t))$ under f and g is what makes it possible eventually to distinguish $q = f$ from $q = g$.

```
def plot_log_histograms(l_seq_f, l_seq_g, c=1, time_points=[1, 7, 14, 21]):
    """Plot log likelihood ratio histograms."""
    fig, axs = plt.subplots(2, 2, figsize=(12, 8))

    for i, t in enumerate(time_points):
        nr, nc = i // 2, i % 2

        axs[nr, nc].axvline(np.log(c), color="k", ls="--")

        hist_f, x_f = np.histogram(np.log(l_seq_f[:, t]), 200, density=True)
        hist_g, x_g = np.histogram(np.log(l_seq_g[:, t]), 200, density=True)

        axs[nr, nc].plot(x_f[1:], hist_f, label="dist under f")
        axs[nr, nc].plot(x_g[1:], hist_g, label="dist under g")

        # Fill error regions
        for j, (x, hist, label) in enumerate(
            zip([x_f, x_g], [hist_f, hist_g],
            ["Type I error", "Type II error"])):
            ind = x[1:] <= np.log(c) if j == 0 else x[1:] > np.log(c)
            axs[nr, nc].fill_between(x[1:][ind], hist[ind],
                                     alpha=0.5, label=label)

        axs[nr, nc].legend()
        axs[nr, nc].set_title(f"t={t}")

    plt.show()

plot_log_histograms(l_seq_f, l_seq_g, c=c)
```



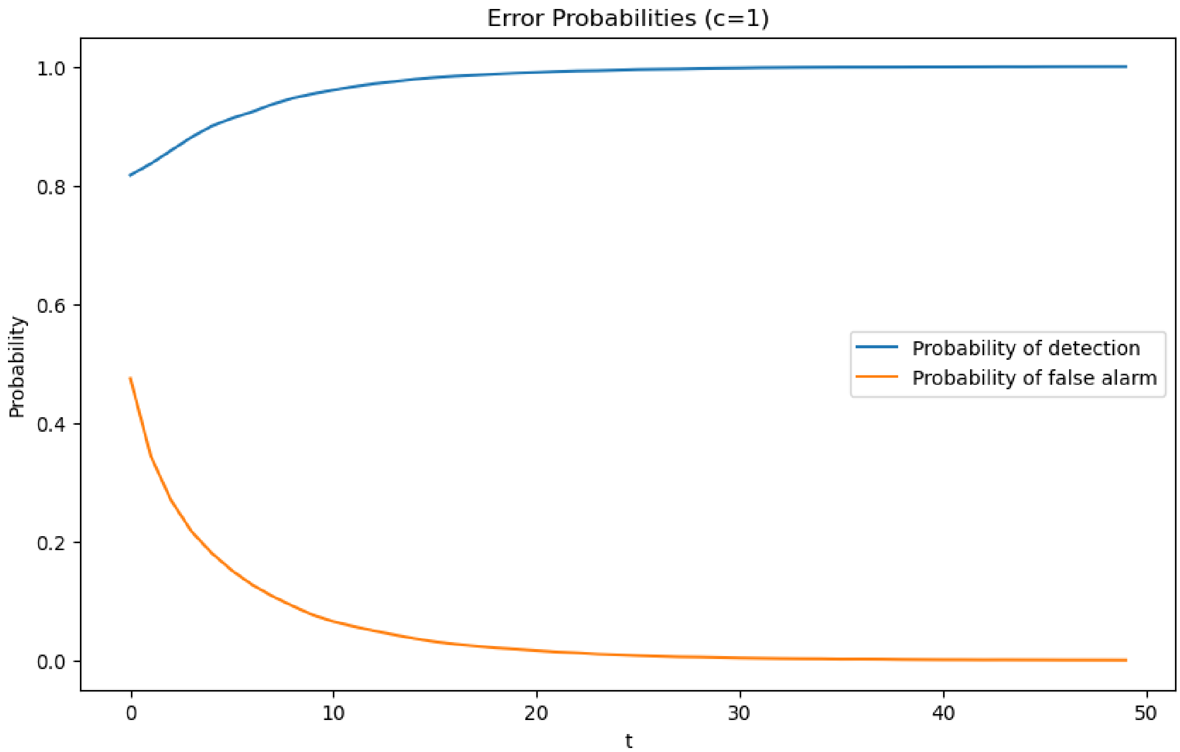
In the above graphs,

- the blue areas are related to but not equal to probabilities α of a type I error because they are integrals of $\log L_t$, not integrals of L_t, over rejection region $L_t < 1$
- the orange areas are related to but not equal to probabilities β of a type II error because they are integrals of $\log L_t$, not integrals of L_t, over acceptance region $L_t > 1$

When we hold c fixed at $c = 1$, the following graph shows that

- the probability of detection monotonically increases with increases in t
- the probability of a false alarm monotonically decreases with increases in t.

```
def compute_error_probabilities(l_seq_f, l_seq_g, c=1):
    """
    Compute Type I and Type II error probabilities.
    """
    N, T = l_seq_f.shape

    # Type I error (false alarm) - reject H0 when true
    PFA = np.array([np.sum(l_seq_f[:, t] < c) / N for t in range(T)])

    # Type II error - accept H0 when false
    beta = np.array([np.sum(l_seq_g[:, t] >= c) / N for t in range(T)])

    # Probability of detection (power)
    PD = np.array([np.sum(l_seq_g[:, t] < c) / N for t in range(T)])

    return {
        'alpha': PFA,
        'beta': beta,
        'PD': PD,
        'PFA': PFA
    }

def plot_error_probabilities(error_dict, T, c=1, title_suffix=""):
    """Plot error probabilities over time."""
    plt.figure(figsize=(10, 6))
    plt.plot(range(T), error_dict['PD'], label="Probability of detection")
    plt.plot(range(T), error_dict['PFA'], label="Probability of false alarm")
    plt.xlabel("t")
    plt.ylabel("Probability")
    plt.title(f"Error Probabilities (c={c}){title_suffix}")
    plt.legend()
    plt.show()

error_probs = compute_error_probabilities(l_seq_f, l_seq_g, c=c)
N, T = l_seq_f.shape
plot_error_probabilities(error_probs, T, c)
```



For a given sample size t, the threshold c uniquely pins down probabilities of both types of error.

If for a fixed t we now free up and move c, we will sweep out the probability of detection as a function of the probability of false alarm.

This produces a receiver operating characteristic curve (ROC curve).

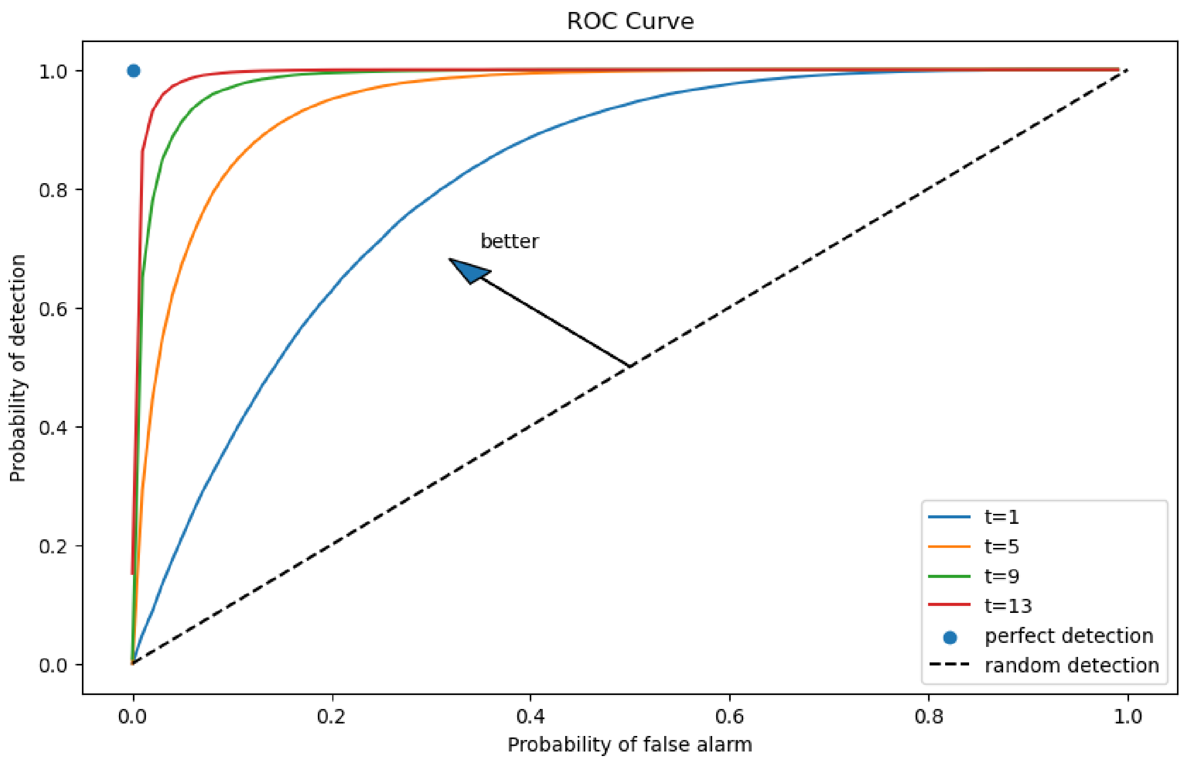
Below, we plot receiver operating characteristic curves for different sample sizes t.

```
def plot_roc_curves(l_seq_f, l_seq_g, t_values=[1, 5, 9, 13], N=None):
    """Plot ROC curves for different sample sizes."""
    if N is None:
        N = l_seq_f.shape[0]

    PFA = np.arange(0, 100, 1)

    plt.figure(figsize=(10, 6))
    for t in t_values:
        percentile = np.percentile(l_seq_f[:, t], PFA)
        PD = [np.sum(l_seq_g[:, t] < p) / N for p in percentile]
        plt.plot(PFA / 100, PD, label=f"t={t}")

    plt.scatter(0, 1, label="perfect detection")
    plt.plot([0, 1], [0, 1], color='k', ls='--', label="random detection")

    plt.arrow(0.5, 0.5, -0.15, 0.15, head_width=0.03)
    plt.text(0.35, 0.7, "better")
    plt.xlabel("Probability of false alarm")
    plt.ylabel("Probability of detection")
    plt.legend()
    plt.title("ROC Curve")
    plt.show()
```

(continues on next page)

(continued from previous page)

```
plot_roc_curves(l_seq_f, l_seq_g, t_values=range(1, 15, 4), N=N)
```



Notice that as t increases, we are assured a larger probability of detection and a smaller probability of false alarm associated with a given discrimination threshold c.

For a given sample size t, both α and β change as we vary c.

As we increase c

- $\alpha \equiv \Pr\left\{L\left(w^t\right) < c \mid q = f\right\}$ increases
- $\beta \equiv \Pr\left\{L\left(w^t\right) > c \mid q = g\right\}$ decreases

As $t \to +\infty$, we approach the perfect detection curve that is indicated by a right angle hinging on the blue dot.

For a given sample size t, the discrimination threshold c determines a point on the receiver operating characteristic curve.

It is up to the test designer to trade off probabilities of making the two types of errors.

But we know how to choose the smallest sample size to achieve given targets for the probabilities.

Typically, frequentists aim for a high probability of detection that respects an upper bound on the probability of false alarm.

Below we show an example in which we fix the probability of false alarm at 0.05.

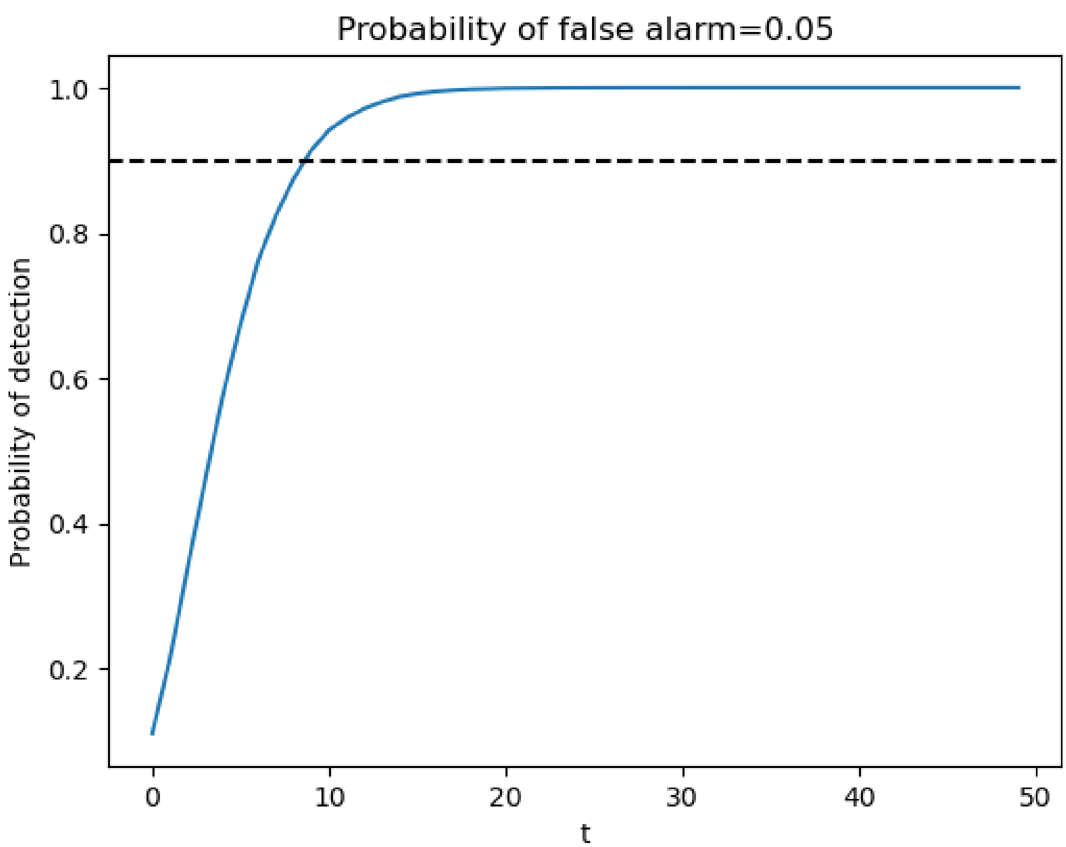
The required sample size for making a decision is then determined by a target probability of detection, for example, 0.9, as depicted in the following graph.

```
PFA = 0.05
PD = np.empty(T)
```

(continues on next page)

(continued from previous page)

```
for t in range(T):

    c = np.percentile(l_seq_f[:, t], PFA * 100)
    PD[t] = np.sum(l_seq_g[:, t] < c) / N

plt.plot(range(T), PD)
plt.axhline(0.9, color="k", ls="--")

plt.xlabel("t")
plt.ylabel("Probability of detection")
plt.title(f"Probability of false alarm={PFA}")
plt.show()
```



The United States Navy evidently used a procedure like this to select a sample size t for doing quality control tests during World War II.

A Navy Captain who had been ordered to perform tests of this kind had doubts about it that he presented to Milton Friedman, as we describe in *this lecture*.

22.6.1 A third distribution h

Now let's consider a case in which neither g nor f generates the data.

Instead, a third distribution h does.

Let's study how accumulated likelihood ratios L behave when h governs the data.

A key tool here is called **Kullback–Leibler divergence** we studied in *Statistical Divergence Measures*.

In our application, we want to measure how much f or g diverges from h

Two Kullback–Leibler divergences pertinent for us are K_f and K_g defined as

$$
\begin{aligned}
K_f = D_{KL}(h\|f) = KL(h, f) &= E_h \left[\log \frac{h(w)}{f(w)}\right] \\
&= \int \log \left(\frac{h(w)}{f(w)}\right) h(w) dw.
\end{aligned}
$$

$$
\begin{aligned}
K_g = D_{KL}(h\|g) = KL(h, g) &= E_h \left[\log \frac{h(w)}{g(w)}\right] \\
&= \int \log \left(\frac{h(w)}{g(w)}\right) h(w) dw.
\end{aligned}
$$

Let's compute the Kullback–Leibler discrepancies using the same code in *Statistical Divergence Measures*.

```
def compute_KL(f, g):
    """
    Compute KL divergence KL(f, g)
    """
    integrand = lambda w: f(w) * np.log(f(w) / g(w))
    val, _ = quad(integrand, 1e-5, 1-1e-5)
    return val

def compute_KL_h(h, f, g):
    """
    Compute KL divergences with respect to reference distribution h
    """
    Kf = compute_KL(h, f)
    Kg = compute_KL(h, g)
    return Kf, Kg
```

22.6.2 A helpful formula

There is a mathematical relationship between likelihood ratios and KL divergence.

When data is generated by distribution h, the expected log likelihood ratio is:

$$
\frac{1}{t} E_h[\log L_t] = K_g - K_f \tag{22.1}
$$

where $L_t = \prod_{j=1}^{t} \frac{f(w_j)}{g(w_j)}$ is the likelihood ratio process.

Equation (22.1) tells us that:

- When $K_g < K_f$ (i.e., g is closer to h than f is), the expected log likelihood ratio is negative, so $L(w^t) \rightarrow 0$.
- When $K_g > K_f$ (i.e., f is closer to h than g is), the expected log likelihood ratio is positive, so $L(w^t) \rightarrow +\infty$.

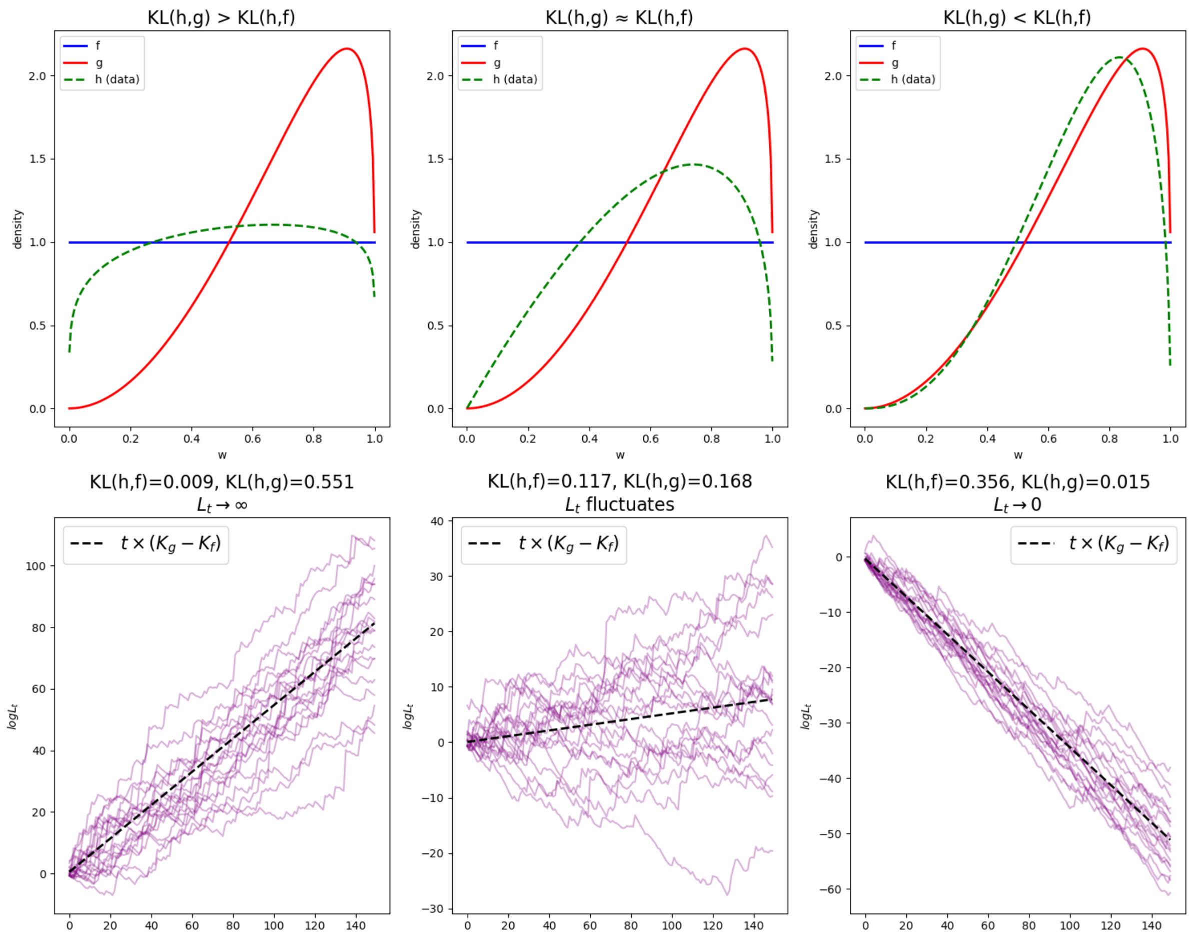
Let's verify this using simulation.

In the simulation, we generate multiple paths using Beta distributions f, g, and h, and compute the paths of $\log(L(w^t))$.

First, we write a function to compute the likelihood ratio process

```python
def compute_likelihood_ratios(sequences, f, g):
    """Compute likelihood ratios and cumulative products."""
    l_ratios = f(sequences) / g(sequences)
    L_cumulative = np.cumprod(l_ratios, axis=1)
    return l_ratios, L_cumulative
```

We consider three cases: (1) h is closer to f, (2) f and g are approximately equidistant from h, and (3) h is closer to g.



Note that

- In the first figure, $\log L(w^t)$ diverges to ∞ because $K_g > K_f$.
- In the second figure, we still have $K_g > K_f$, but the difference is smaller, so $L(w^t)$ diverges to infinity at a slower pace.
- In the last figure, $\log L(w^t)$ diverges to $-\infty$ because $K_g < K_f$.
- The black dotted line, $t\left(D_{KL}(h\|g) - D_{KL}(h\|f)\right)$, closely fits the paths verifying (22.1).

These observations align with the theory.

In *Heterogeneous Beliefs and Financial Markets*, we will see an application of these ideas.

22.7 Hypothesis testing and classification

This section discusses another application of likelihood ratio processes.

We describe how a statistician can combine frequentist probabilities of type I and type II errors in order to

- compute an anticipated frequency of selecting a wrong model based on a sample length T
- compute an anticipated error rate in a classification problem

We consider a situation in which nature generates data by mixing known densities f and g with known mixing parameter $\pi_{-1} \in (0, 1)$ so that the random variable w is drawn from the density

$$h(w) = \pi_{-1} f(w) + (1 - \pi_{-1}) g(w)$$

We assume that the statistician knows the densities f and g and also the mixing parameter π_{-1}.

Below, we'll set $\pi_{-1} = .5$, although much of the analysis would follow through with other settings of $\pi_{-1} \in (0, 1)$.

We assume that f and g both put positive probabilities on the same intervals of possible realizations of the random variable W.

In the simulations below, we specify that f is a Beta$(1, 1)$ distribution and that g is Beta$(3, 1.2)$ distribution.

We consider two alternative timing protocols.

- Timing protocol 1 is for the model selection problem
- Timing protocol 2 is for the individual classification problem

Timing Protocol 1: Nature flips a coin only **once** at time $t = -1$ and with probability π_{-1} generates a sequence $\{w_t\}_{t=1}^T$ of IID draws from f and with probability $1 - \pi_{-1}$ generates a sequence $\{w_t\}_{t=1}^T$ of IID draws from g.

Timing Protocol 2. Nature flips a coin **often**. At each time $t \geq 0$, nature flips a coin and with probability π_{-1} draws w_t from f and with probability $1 - \pi_{-1}$ draws w_t from g.

Here is Python code that we'll use to implement timing protocol 1 and 2

```
def protocol_1(π_minus_1, T, N=1000, F_params=(1, 1), G_params=(3, 1.2)):
    """
    Simulate Protocol 1: Nature decides once at t=-1 which model to use.
    """
    F_a, F_b = F_params
    G_a, G_b = G_params

    # Single coin flip for the true model
    true_models_F = np.random.rand(N) < π_minus_1
    sequences = np.empty((N, T))

    n_f = np.sum(true_models_F)
    n_g = N - n_f

    if n_f > 0:
        sequences[true_models_F, :] = np.random.beta(F_a, F_b, (n_f, T))
    if n_g > 0:
        sequences[~true_models_F, :] = np.random.beta(G_a, G_b, (n_g, T))

    return sequences, true_models_F

def protocol_2(π_minus_1, T, N=1000, F_params=(1, 1), G_params=(3, 1.2)):
    """
```

(continues on next page)

(continued from previous page)

```
    Simulate Protocol 2: Nature decides at each time step which model to use.
    """
    F_a, F_b = F_params
    G_a, G_b = G_params

    # Coin flips for each time step
    true_models_F = np.random.rand(N, T) < π_minus_1
    sequences = np.empty((N, T))

    n_f = np.sum(true_models_F)
    n_g = N * T - n_f

    if n_f > 0:
        sequences[true_models_F] = np.random.beta(F_a, F_b, n_f)
    if n_g > 0:
        sequences[~true_models_F] = np.random.beta(G_a, G_b, n_g)

    return sequences, true_models_F
```

Remark: Under timing protocol 2, the $\{w_t\}_{t=1}^T$ is a sequence of IID draws from $h(w)$. Under timing protocol 1, the $\{w_t\}_{t=1}^T$ is not IID. It is **conditionally IID** – meaning that with probability π_{-1} it is a sequence of IID draws from $f(w)$ and with probability $1-\pi_{-1}$ it is a sequence of IID draws from $g(w)$. For more about this, see *this lecture about exchangeability*.

We again deploy a **likelihood ratio process** with time t component being the likelihood ratio

$$\ell(w_t) = \frac{f(w_t)}{g(w_t)}, \quad t \geq 1.$$

The **likelihood ratio process** for sequence $\{w_t\}_{t=1}^{\infty}$ is

$$L(w^t) = \prod_{i=1}^{t} \ell(w_i),$$

For shorthand we'll write $L_t = L(w^t)$.

22.7.1 Model selection mistake probability

We first study a problem that assumes timing protocol 1.

Consider a decision maker who wants to know whether model f or model g governs a data set of length T observations.

The decision makers has observed a sequence $\{w_t\}_{t=1}^T$.

On the basis of that observed sequence, a likelihood ratio test selects model f when $L_T \geq 1$ and model g when $L_T < 1$.

When model f generates the data, the probability that the likelihood ratio test selects the wrong model is

$$p_f = \text{Prob}\left(L_T < 1 \middle| f\right) = \alpha_T.$$

When model g generates the data, the probability that the likelihood ratio test selects the wrong model is

$$p_g = \text{Prob}\left(L_T \geq 1 \middle| g\right) = \beta_T.$$

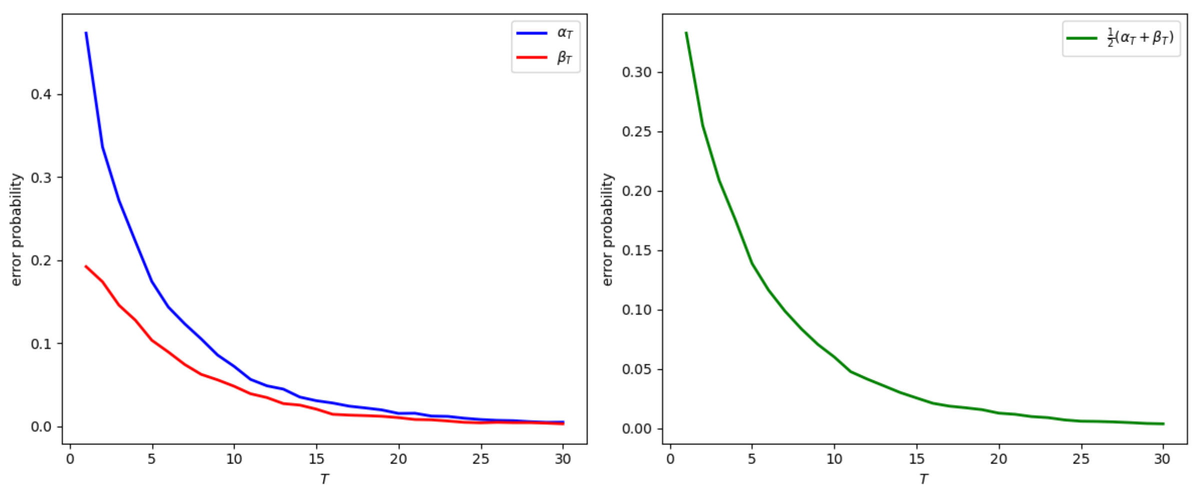
We can construct a probability that the likelihood ratio selects the wrong model by assigning a Bayesian prior probability of $\pi_{-1} = .5$ that nature selects model f then averaging p_f and p_g to form the Bayesian posterior probability of a detection error equal to

$$p(\text{wrong decision}) = \frac{1}{2}(\alpha_T + \beta_T). \tag{22.2}$$

Now let's simulate timing protocol 1 and compute the error probabilities

```
def compute_protocol_1_errors(π_minus_1, T_max, N_simulations, f_func, g_func,
                              F_params=(1, 1), G_params=(3, 1.2)):
    """
    Compute error probabilities for Protocol 1.
    """
    sequences, true_models = protocol_1(
        π_minus_1, T_max, N_simulations, F_params, G_params)
    l_ratios, L_cumulative = compute_likelihood_ratios(sequences,
                                                       f_func, g_func)

    T_range = np.arange(1, T_max + 1)

    mask_f = true_models
    mask_g = ~true_models

    L_f = L_cumulative[mask_f, :]
    L_g = L_cumulative[mask_g, :]

    α_T = np.mean(L_f < 1, axis=0)
    β_T = np.mean(L_g >= 1, axis=0)
    error_prob = 0.5 * (α_T + β_T)

    return {
        'T_range': T_range,
        'alpha': α_T,
        'beta': β_T,
        'error_prob': error_prob,
        'L_cumulative': L_cumulative,
        'true_models': true_models
    }
```

The following code visualizes the error probabilities for timing protocol 1



```
At T=30:
α_30 = 0.0048
β_30 = 0.0028
Model selection error probability = 0.0038
```

Notice how the model selection error probability approaches zero as T grows.

22.7.2 Classification

We now consider a problem that assumes timing protocol 2.

A decision maker wants to classify components of an observed sequence $\{w_t\}_{t=1}^T$ as having been drawn from either f or g.

The decision maker uses the following classification rule:

$$\begin{aligned} w_t \text{ is from } f \text{ if } l_t > 1 \\ w_t \text{ is from } g \text{ if } l_t \leq 1. \end{aligned}$$

Under this rule, the expected misclassification rate is

$$p(\text{misclassification}) = \frac{1}{2}(\tilde{\alpha}_t + \tilde{\beta}_t) \tag{22.3}$$

where $\tilde{\alpha}_t = \text{Prob}(l_t < 1 \mid f)$ and $\tilde{\beta}_t = \text{Prob}(l_t \geq 1 \mid g)$.

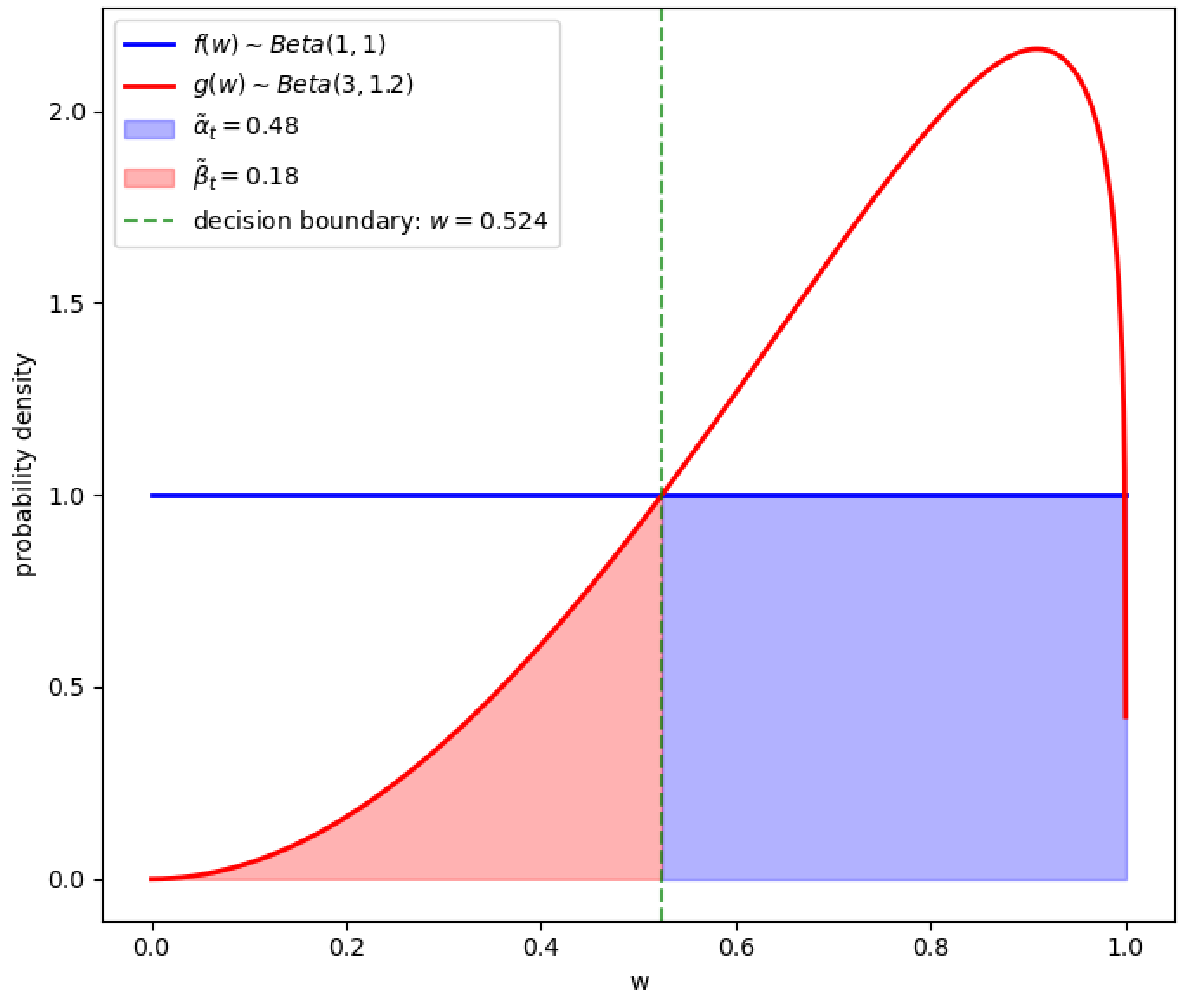
Now let's write some code to simulate it

```python
def compute_protocol_2_errors(π_minus_1, T_max, N_simulations, f_func, g_func,
                              F_params=(1, 1), G_params=(3, 1.2)):
    """
    Compute error probabilities for Protocol 2.
    """
    sequences, true_models = protocol_2(π_minus_1,
                        T_max, N_simulations, F_params, G_params)
    l_ratios, _ = compute_likelihood_ratios(sequences, f_func, g_func)

    T_range = np.arange(1, T_max + 1)

    accuracy = np.empty(T_max)
    for t in range(T_max):
        predictions = (l_ratios[:, t] >= 1)
        actual = true_models[:, t]
        accuracy[t] = np.mean(predictions == actual)

    return {
        'T_range': T_range,
        'accuracy': accuracy,
        'l_ratios': l_ratios,
        'true_models': true_models
    }
```

Since for each t, the decision boundary is the same, the decision boundary can be computed as

```python
root = brentq(lambda w: f(w) / g(w) - 1, 0.001, 0.999)
```

we can plot the distributions of f and g and the decision boundary



To the left of the green vertical line $g < f$, so $l_t > 1$; therefore a w_t that falls to the left of the green line is classified as a type f individual.

- The shaded red area equals β – the probability of classifying someone as a type g individual when it is really a type f individual.

To the right of the green vertical line $g > f$, so $l_t < 1$; therefore a w_t that falls to the right of the green line is classified as a type g individual.

- The shaded blue area equals α – the probability of classifying someone as a type f when it is really a type g individual.

This gives us clues about how to compute the theoretical classification error probability

```
# Compute theoretical tilde α_t and tilde β_t
def α_integrand(w):
    """Integrand for tilde α_t = P(l_t < 1 | f)"""
    return f(w) if f(w) / g(w) < 1 else 0

def β_integrand(w):
    """Integrand for tilde β_t = P(l_t >= 1 | g)"""
    return g(w) if f(w) / g(w) >= 1 else 0
```

(continues on next page)

(continued from previous page)

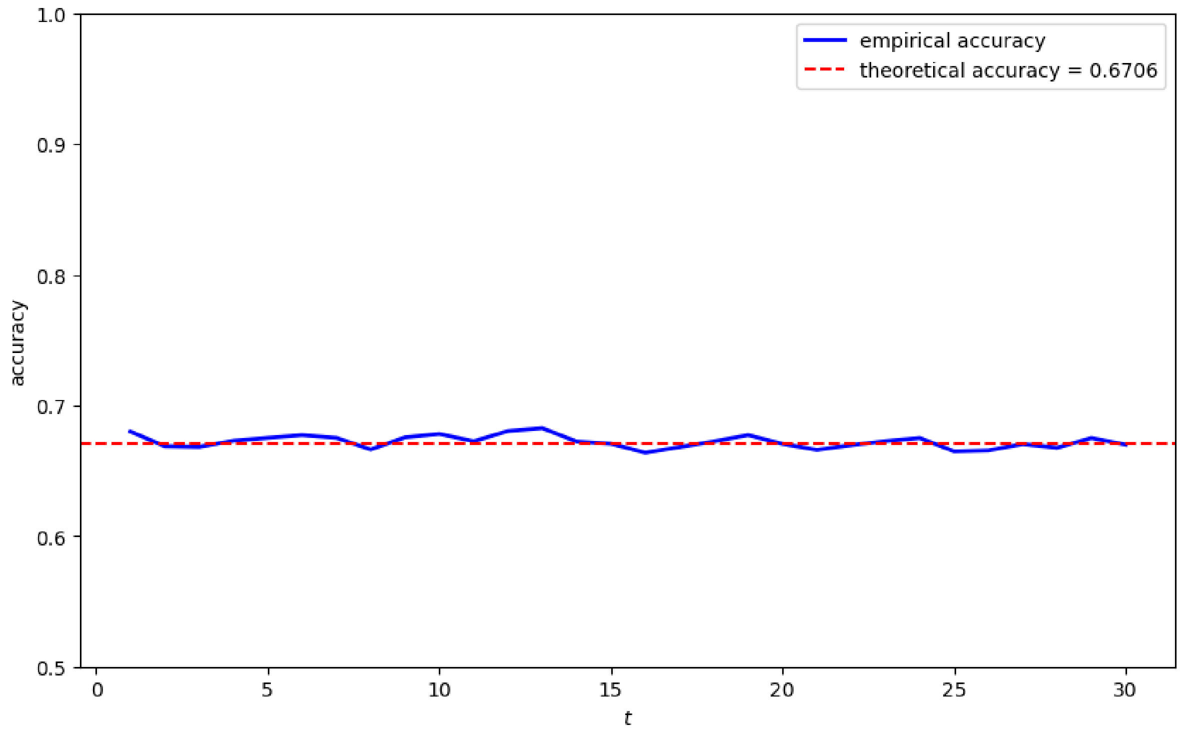
```
# Compute the integrals
α_theory, _ = quad(α_integrand, 0, 1, limit=100)
β_theory, _ = quad(β_integrand, 0, 1, limit=100)

theory_error = 0.5 * (α_theory + β_theory)

print(f"theoretical tilde α_t = {α_theory:.4f}")
print(f"theoretical tilde β_t = {β_theory:.4f}")
print(f"theoretical classification error probability = {theory_error:.4f}")
```

```
theoretical tilde α_t = 0.4752
theoretical tilde β_t = 0.1836
theoretical classification error probability = 0.3294
```

Now we simulate timing protocol 2 and compute the classification error probability.

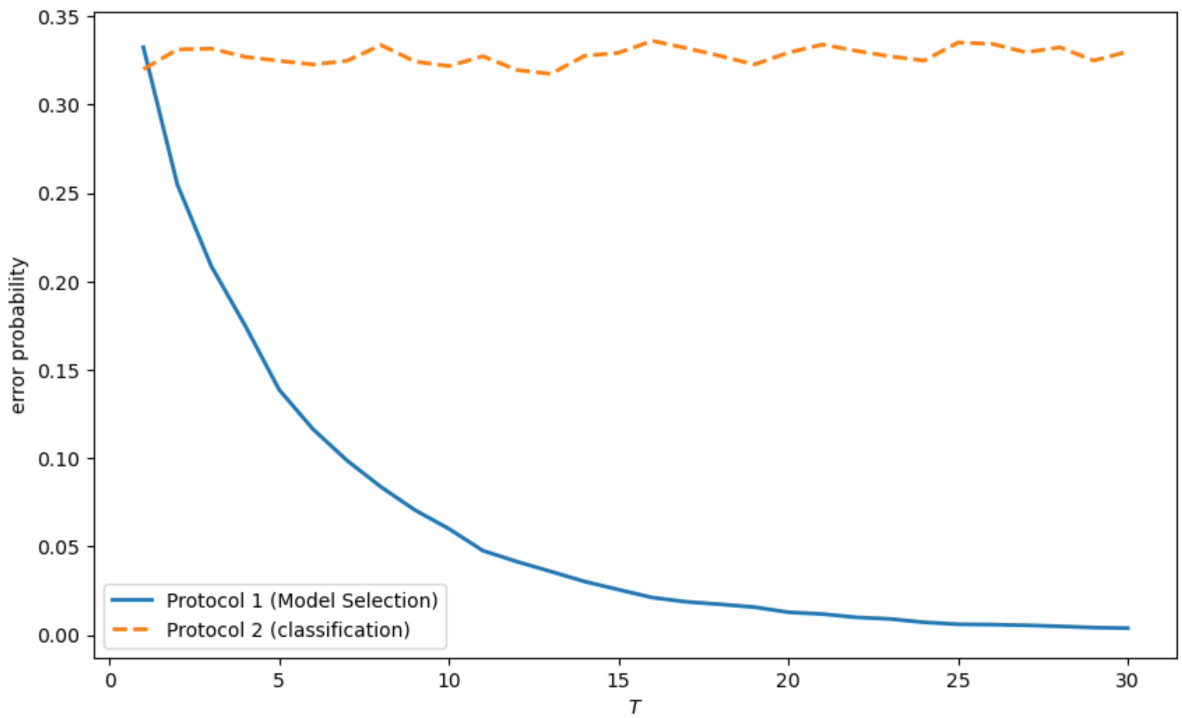
In the next cell, we also compare the theoretical classification accuracy to the empirical classification accuracy

```
def analyze_protocol_2(π_minus_1, T_max, N_simulations, f_func, g_func,
                       theory_error=None, F_params=(1, 1), G_params=(3, 1.2)):
    """Analyze Protocol 2."""
    result = compute_protocol_2_errors(π_minus_1, T_max, N_simulations,
                                       f_func, g_func, F_params, G_params)

    # Plot results
    plt.figure(figsize=(10, 6))
    plt.plot(result['T_range'], result['accuracy'],
            'b-', linewidth=2, label='empirical accuracy')

    if theory_error is not None:
        plt.axhline(1 - theory_error, color='r', linestyle='--',
                   label=f'theoretical accuracy = {1 - theory_error:.4f}')

    plt.xlabel('$t$')
    plt.ylabel('accuracy')
    plt.legend()
    plt.ylim(0.5, 1.0)
    plt.show()

    return result

# Analyze Protocol 2
result_p2 = analyze_protocol_2(π_minus_1, T_max, N_simulations, f, g,
                              theory_error, (F_a, F_b), (G_a, G_b))
```



Let's watch decisions made by the two timing protocols as more and more observations accrue.

```
def compare_protocols(result1, result2):
    """Compare results from both protocols."""
    plt.figure(figsize=(10, 6))

    plt.plot(result1['T_range'], result1['error_prob'], linewidth=2,
            label='Protocol 1 (Model Selection)')
    plt.plot(result2['T_range'], 1 - result2['accuracy'],
            linestyle='--', linewidth=2,
            label='Protocol 2 (classification)')

    plt.xlabel('$T$')
    plt.ylabel('error probability')
    plt.legend()
    plt.show()

compare_protocols(result_p1, result_p2)
```



From the figure above, we can see:

- For both timing protocols, the error probability starts at the same level, subject to a little randomness.
- For timing protocol 1, the error probability decreases as the sample size increases because we are making just **one** decision – i.e., selecting whether f or g governs **all** individuals. More data provides better evidence.
- For timing protocol 2, the error probability remains constant because we are making **many** decisions – one classification decision for each observation.

Remark: Think about how laws of large numbers are applied to compute error probabilities for the model selection problem and the classification problem.

22.7.3 Error probability and divergence measures

A plausible guess is that the ability of a likelihood ratio to distinguish distributions f and g depends on how "different" they are.

We have learnt some measures of "difference" between distributions in *Statistical Divergence Measures*.

Let's now study two more measures of "difference" between distributions that are useful in the context of model selection and classification.

Recall that Chernoff entropy between probability densities f and g is defined as:

$$C(f,g) = -\log \min_{\phi \in (0,1)} \int f^{\phi}(x) g^{1-\phi}(x) dx$$

An upper bound on model selection error probability is

$$e^{-C(f,g)T}.$$

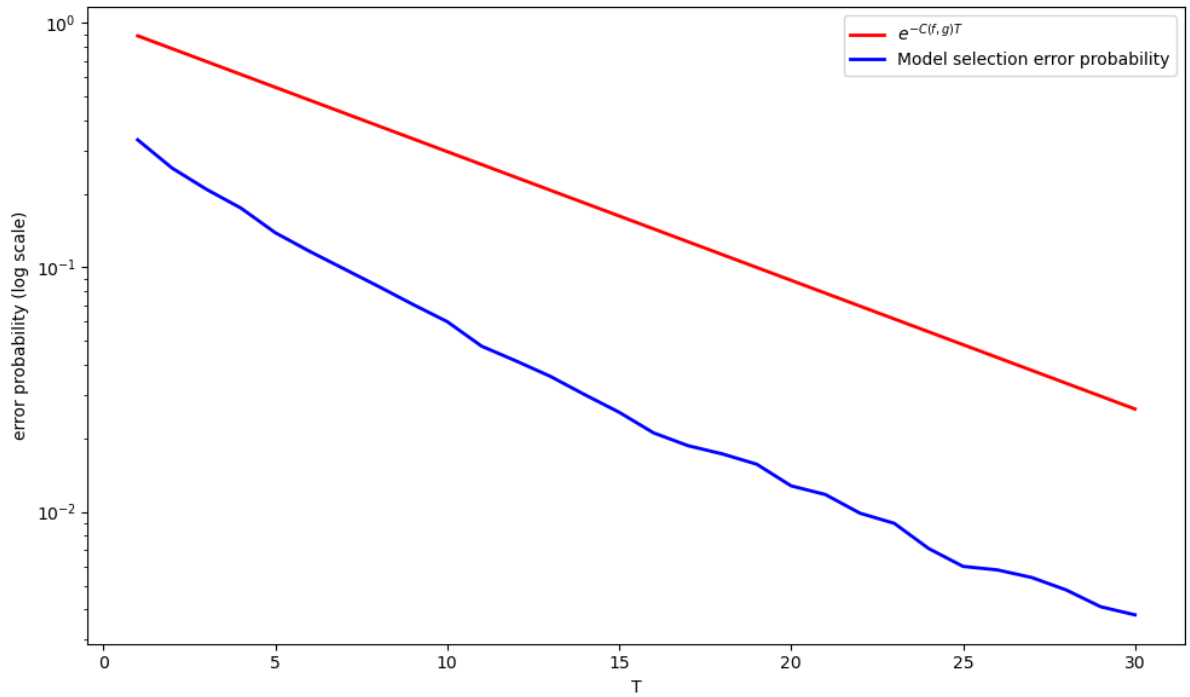
Let's compute Chernoff entropy numerically with some Python code

```
def chernoff_integrand(ϕ, f, g):
    """
    Compute the integrand for Chernoff entropy
    """
    def integrand(w):
        return f(w)**ϕ * g(w)**(1-ϕ)

    result, _ = quad(integrand, 1e-5, 1-1e-5)
    return result

def compute_chernoff_entropy(f, g):
    """
    Compute Chernoff entropy C(f,g)
    """
    def objective(ϕ):
        return chernoff_integrand(ϕ, f, g)

    # Find the minimum over ϕ in (0,1)
    result = minimize_scalar(objective,
                             bounds=(1e-5, 1-1e-5),
                             method='bounded')
    min_value = result.fun
    ϕ_optimal = result.x

    chernoff_entropy = -np.log(min_value)
    return chernoff_entropy, ϕ_optimal
C_fg, ϕ_optimal = compute_chernoff_entropy(f, g)
print(f"Chernoff entropy C(f,g) = {C_fg:.4f}")
print(f"Optimal ϕ = {ϕ_optimal:.4f}")
```

```
Chernoff entropy C(f,g) = 0.1212
Optimal ϕ = 0.5969
```

Now let's examine how $e^{-C(f,g)T}$ behaves as a function of T and compare it to the model selection error probability

```
T_range = np.arange(1, T_max+1)
chernoff_bound = np.exp(-C_fg * T_range)

# Plot comparison
fig, ax = plt.subplots(figsize=(10, 6))

ax.semilogy(T_range, chernoff_bound, 'r-', linewidth=2,
           label=f'$e^{{-C(f,g)T}}$')
ax.semilogy(T_range, result_p1['error_prob'], 'b-', linewidth=2,
           label='Model selection error probability')

ax.set_xlabel('T')
ax.set_ylabel('error probability (log scale)')
ax.legend()
plt.tight_layout()
plt.show()
```



Evidently, $e^{-C(f,g)T}$ is an upper bound on the error rate.

In {doc}divergence_measures`, we also studied **Jensen-Shannon divergence** as a symmetric measure of distance between distributions.

We can use Jensen-Shannon divergence to measure the distance between distributions f and g and compute how it covaries with the model selection error probability.

We also compute Jensen-Shannon divergence numerically with some Python code

```
def compute_JS(f, g):
    """
    Compute Jensen-Shannon divergence
    """
    def m(w):
        return 0.5 * (f(w) + g(w))

    js_div = 0.5 * compute_KL(f, m) + 0.5 * compute_KL(g, m)
    return js_div
```

Now let's return to our guess that the error probability at large sample sizes is related to the Chernoff entropy between two distributions.

We verify this by computing the correlation between the log of the error probability at $T = 50$ under Timing Protocol 1 and the divergence measures.

In the simulation below, nature draws $N/2$ sequences from g and $N/2$ sequences from f.

Note

Nature does this rather than flipping a fair coin to decide whether to draw from g or f once and for all before each simulation of length T.

We use the following pairs of Beta distributions for f and g as test cases

```
distribution_pairs = [
    # (f_params, g_params)
    ((1, 1), (0.1, 0.2)),
    ((1, 1), (0.3, 0.3)),
    ((1, 1), (0.3, 0.4)),
    ((1, 1), (0.5, 0.5)),
    ((1, 1), (0.7, 0.6)),
    ((1, 1), (0.9, 0.8)),
    ((1, 1), (1.1, 1.05)),
    ((1, 1), (1.2, 1.1)),
    ((1, 1), (1.5, 1.2)),
    ((1, 1), (2, 1.5)),
    ((1, 1), (2.5, 1.8)),
    ((1, 1), (3, 1.2)),
    ((1, 1), (4, 1)),
    ((1, 1), (5, 1))
]
```

Now let's run the simmulation

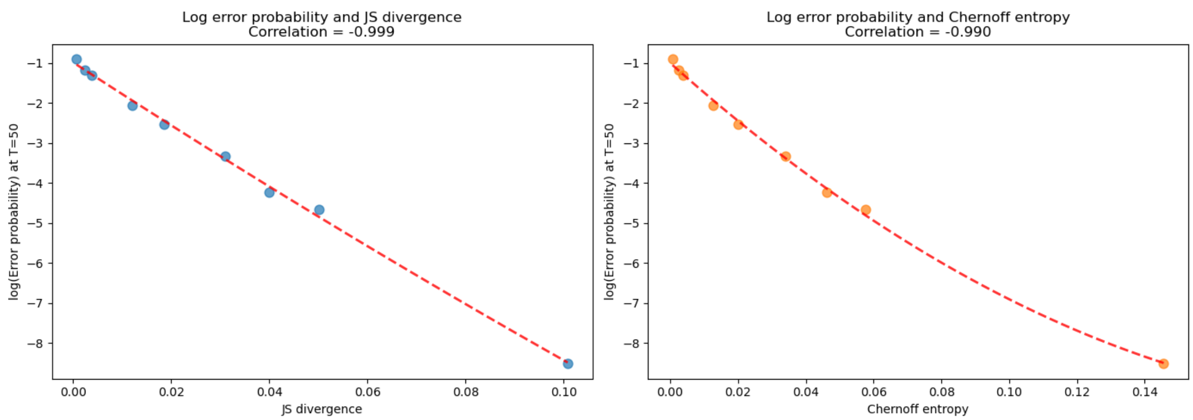
```
# Parameters for simulation
T_large = 50
N_sims = 5000
N_half = N_sims // 2

# Initialize arrays
n_pairs = len(distribution_pairs)
kl_fg_vals = np.zeros(n_pairs)
kl_gf_vals = np.zeros(n_pairs)
js_vals = np.zeros(n_pairs)
chernoff_vals = np.zeros(n_pairs)
error_probs = np.zeros(n_pairs)
pair_names = []

for i, ((f_a, f_b), (g_a, g_b)) in enumerate(distribution_pairs):
    # Create density functions
    f = jit(lambda x, a=f_a, b=f_b: p(x, a, b))
    g = jit(lambda x, a=g_a, b=g_b: p(x, a, b))

    # Compute divergence measures
    kl_fg_vals[i] = compute_KL(f, g)
    kl_gf_vals[i] = compute_KL(g, f)
    js_vals[i] = compute_JS(f, g)
    chernoff_vals[i], _ = compute_chernoff_entropy(f, g)

    # Generate samples
    sequences_f = np.random.beta(f_a, f_b, (N_half, T_large))
    sequences_g = np.random.beta(g_a, g_b, (N_half, T_large))

    # Compute likelihood ratios and cumulative products
    _, L_cumulative_f = compute_likelihood_ratios(sequences_f, f, g)
    _, L_cumulative_g = compute_likelihood_ratios(sequences_g, f, g)

    # Get final values
    L_cumulative_f = L_cumulative_f[:, -1]
    L_cumulative_g = L_cumulative_g[:, -1]
```

(continues on next page)

(continued from previous page)

```
    # Calculate error probabilities
    error_probs[i] = 0.5 * (np.mean(L_cumulative_f < 1) +
                            np.mean(L_cumulative_g >= 1))
    pair_names.append(f"Beta({f_a},{f_b}) and Beta({g_a},{g_b})")

cor_data =  {
    'kl_fg': kl_fg_vals,
    'kl_gf': kl_gf_vals,
    'js': js_vals,
    'chernoff': chernoff_vals,
    'error_prob': error_probs,
    'names': pair_names,
    'T': T_large}
```

Now let's visualize the correlations



Evidently, Chernoff entropy and Jensen-Shannon entropy each covary tightly with the model selection error probability.

We'll encounter related ideas in *A Problem that Stumped Milton Friedman* very soon.

22.8 Markov chains

Let's now look at a likelihood ratio process for a sequence of random variables that is not independently and identically distributed.

Here we assume that the sequence is generated by a Markov chain on a finite state space.

We consider two n-state irreducible and aperiodic Markov chain models on the same state space $\{1, 2, \ldots, n\}$ with positive transition matrices $P^{(f)}$, $P^{(g)}$ and initial distributions $\pi_0^{(f)}$, $\pi_0^{(g)}$.

We assume that nature samples from chain f.

For a sample path $(x_0, x_1, \ldots, x_T)$, let N_{ij} count transitions from state i to j.

The likelihood process under model $m \in \{f, g\}$ is

$$L_T^{(m)} = \pi_{0,x_0}^{(m)} \prod_{i=1}^{n} \prod_{j=1}^{n} \left(P_{ij}^{(m)}\right)^{N_{ij}}$$

Hence,

$$\log L_T^{(m)} = \log \pi_{0,x_0}^{(m)} + \sum_{i,j} N_{ij} \log P_{ij}^{(m)}$$

The log-likelihood ratio is

$$\log \frac{L_T^{(f)}}{L_T^{(g)}} = \log \frac{\pi_{0,x_0}^{(f)}}{\pi_{0,x_0}^{(g)}} + \sum_{i,j} N_{ij} \log \frac{P_{ij}^{(f)}}{P_{ij}^{(g)}} \tag{22.4}$$

22.8.1 KL divergence rate

By the ergodic theorem for irreducible, aperiodic Markov chains, we have

$$\frac{N_{ij}}{T} \xrightarrow{a.s.} \pi_i^{(f)} P_{ij}^{(f)} \quad \text{as } T \to \infty$$

where $\pi^{(f)}$ is the stationary distribution satisfying $\pi^{(f)} = \pi^{(f)} P^{(f)}$.

Therefore,

$$\frac{1}{T} \log \frac{L_T^{(f)}}{L_T^{(g)}} = \frac{1}{T} \log \frac{\pi_{0,x_0}^{(f)}}{\pi_{0,x_0}^{(g)}} + \frac{1}{T} \sum_{i,j} N_{ij} \log \frac{P_{ij}^{(f)}}{P_{ij}^{(g)}}$$

Taking the limit as $T \to \infty$, we have:

- The first term: $\frac{1}{T} \log \frac{\pi_{0,x_0}^{(f)}}{\pi_{0,x_0}^{(g)}} \to 0$
- The second term: $\frac{1}{T} \sum_{i,j} N_{ij} \log \frac{P_{ij}^{(f)}}{P_{ij}^{(g)}} \xrightarrow{a.s.} \sum_{i,j} \pi_i^{(f)} P_{ij}^{(f)} \log \frac{P_{ij}^{(f)}}{P_{ij}^{(g)}}$

Define the **KL divergence rate** as

$$h_{KL}(f,g) = \sum_{i=1}^{n} \pi_i^{(f)} \underbrace{\sum_{j=1}^{n} P_{ij}^{(f)} \log \frac{P_{ij}^{(f)}}{P_{ij}^{(g)}}}_{=:KL(P_{i\cdot}^{(f)}, P_{i\cdot}^{(g)})}$$

where $KL(P_{i\cdot}^{(f)}, P_{i\cdot}^{(g)})$ is the row-wise KL divergence.

By the ergodic theorem, we have

$$\frac{1}{T} \log \frac{L_T^{(f)}}{L_T^{(g)}} \xrightarrow{a.s.} h_{KL}(f,g) \quad \text{as } T \to \infty$$

Taking expectations and using the dominated convergence theorem, we obtain

$$\frac{1}{T} E_f \left[\log \frac{L_T^{(f)}}{L_T^{(g)}} \right] \to h_{KL}(f,g) \quad \text{as } T \to \infty$$

Here we invite readers to pause and compare this result with (22.1).

Let's confirm this in the simulation below.

22.8.2 Simulations

Let's implement simulations to illustrate these concepts with a three-state Markov chain.

We start by writing functions to compute the stationary distribution and the KL divergence rate for Markov chain models.

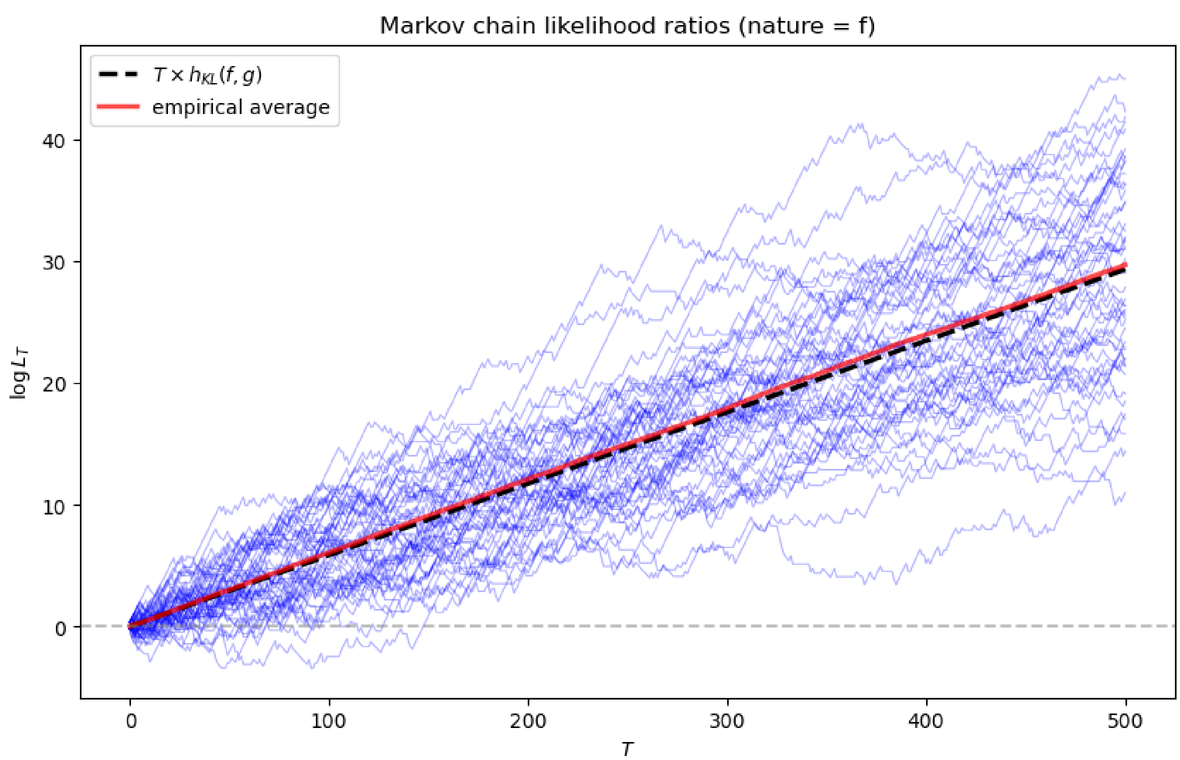
Now let's create an example with two different 3-state Markov chains.

We are now ready to simulate paths and visualize how likelihood ratios evolve.

We verify $\frac{1}{T}E_f\left[\log\frac{L_T^{(f)}}{L_T^{(g)}}\right] = h_{KL}(f,g)$ starting from the stationary distribution by plotting both the empirical average and the line predicted by the theory

```
# Define example Markov chain transition matrices
P_f = np.array([[0.7, 0.2, 0.1],
                [0.3, 0.5, 0.2],
                [0.1, 0.3, 0.6]])

P_g = np.array([[0.5, 0.3, 0.2],
                [0.2, 0.6, 0.2],
                [0.2, 0.2, 0.6]])

markov_results = analyze_markov_chains(P_f, P_g)
```

```
Stationary distribution (f): [0.41176471 0.32352941 0.26470588]
Stationary distribution (g): [0.28571429 0.38095238 0.33333333]

KL divergence rate h(f, g): 0.0588
KL divergence rate h(g, f): 0.0563
```



22.9 Related lectures

Likelihood processes play an important role in Bayesian learning, as described in *Likelihood Ratio Processes and Bayesian Learning* and as applied in *Job Search VIII: Search with Learning*.

Likelihood ratio processes are central to Lawrence Blume and David Easley's answer to their question "If you're so smart, why aren't you rich?" [Blume and Easley, 2006], the subject of the lecture*Heterogeneous Beliefs and Financial Markets*.

Likelihood ratio processes also appear in Additive and Multiplicative Functionals, which contains another illustration of the **peculiar property** of likelihood ratio processes described above.

22.10 Exercises

Exercise 22.10.1

Consider the setting where nature generates data from a third density h.

Let $\{w_t\}_{t=1}^T$ be IID draws from h, and let $L_t = L(w^t)$ be the likelihood ratio process defined as in the lecture.

Show that:

$$\frac{1}{t}E_h[\log L_t] = K_g - K_f$$

with finite K_g, K_f, $E_h|\log f(W)| < \infty$ and $E_h|\log g(W)| < \infty$.

Hint: Start by expressing $\log L_t$ as a sum of $\log \ell(w_i)$ terms and compare with the definition of K_f and K_g.

Solution

Since $w_1, \dots, w_t$ are IID draws from h, we can write

$$\log L_t = \log \prod_{i=1}^t \ell(w_i) = \sum_{i=1}^t \log \ell(w_i) = \sum_{i=1}^t \log \frac{f(w_i)}{g(w_i)}$$

Taking the expectation under h

$$E_h[\log L_t] = E_h\left[\sum_{i=1}^t \log \frac{f(w_i)}{g(w_i)}\right] = \sum_{i=1}^t E_h\left[\log \frac{f(w_i)}{g(w_i)}\right]$$

Since the w_i are identically distributed

$$E_h[\log L_t] = t \cdot E_h\left[\log \frac{f(w)}{g(w)}\right]$$

where $w \sim h$.

Therefore

$$\frac{1}{t}E_h[\log L_t] = E_h\left[\log \frac{f(w)}{g(w)}\right] = E_h[\log f(w)] - E_h[\log g(w)]$$

Now, from the definition of Kullback-Leibler divergence

$$K_f = \int h(w) \log \frac{h(w)}{f(w)} dw = E_h[\log h(w)] - E_h[\log f(w)]$$

This gives us

$$E_h[\log f(w)] = E_h[\log h(w)] - K_f$$

Similarly

$$E_h[\log g(w)] = E_h[\log h(w)] - K_g$$

Substituting back

$$\begin{aligned}
\frac{1}{t}E_h[\log L_t] &= E_h[\log f(w)] - E_h[\log g(w)] \\
&= [E_h[\log h(w)] - K_f] - [E_h[\log h(w)] - K_g] \\
&= K_g - K_f
\end{aligned}$$

Exercise 22.10.2

Building on Exercise 22.10.1, use the result to explain what happens to L_t as $t \to \infty$ in the following cases:

1. When $K_g > K_f$ (i.e., f is "closer" to h than g is)
2. When $K_g < K_f$ (i.e., g is "closer" to h than f is)

Relate your answer to the simulation results shown in *this section*.

Solution

From Exercise 22.10.1, we know that:

$$\frac{1}{t}E_h[\log L_t] = K_g - K_f$$

Case 1: When $K_g > K_f$

Here, f is "closer" to h than g is. Since $K_g - K_f > 0$

$$E_h[\log L_t] = t \cdot (K_g - K_f) \to +\infty \text{ as } t \to \infty$$

By the Law of Large Numbers, $\frac{1}{t}\log L_t \to K_g - K_f > 0$ almost surely.

Therefore $L_t \to +\infty$ almost surely.

Case 2: When $K_g < K_f$

Here, g is "closer" to h than f is. Since $K_g - K_f < 0$

$$E_h[\log L_t] = t \cdot (K_g - K_f) \to -\infty \text{ as } t \to \infty$$

Therefore by similar reasoning $L_t \to 0$ almost surely.

CHAPTER

TWENTYTHREE

HETEROGENEOUS BELIEFS AND FINANCIAL MARKETS

Contents

23.1 Overview

A likelihood ratio process lies behind Lawrence Blume and David Easley's answer to their question "If you're so smart, why aren't you rich?" [Blume and Easley, 2006].

Blume and Easley constructed formal models to study how differences of opinions about probabilities governing risky income processes would influence outcomes and be reflected in prices of stocks, bonds, and insurance policies that individuals use to share and hedge risks.

Note

[Alchian, 1950] and [Friedman, 1953] conjectured that, by rewarding traders with more realistic probability models, competitive markets in financial securities put wealth in the hands of better informed traders and help make prices of risky assets reflect realistic probability assessments.

Here we'll provide an example that illustrates basic components of Blume and Easley's analysis.

We'll focus only on their analysis of an environment with complete markets in which trades in all conceivable risky securities are possible.

We'll study two alternative arrangements:

- perfect socialism in which individuals surrender their endowments of consumption goods each period to a central planner who then dictatorially allocates those goods
- a decentralized system of competitive markets in which selfish price-taking individuals voluntarily trade with each other in competitive markets

The fundamental theorems of welfare economics will apply and assure us that these two arrangements end up producing exactly the same allocation of consumption goods to individuals **provided** that the social planner assigns an appropriate set of **Pareto weights**.

Note

You can learn about how the two welfare theorems are applied in modern macroeconomic models in *this lecture on a planning problem* and *this lecture on a related competitive equilibrium*. *This quantecon lecture* presents a recursive formulation of complete markets models with homogeneous beliefs.

Let's start by importing some Python tools.

```
import matplotlib.pyplot as plt
import numpy as np
from numba import vectorize, jit, prange
from math import gamma
from scipy.integrate import quad
```

23.2 Review: likelihood ratio processes

We'll begin by reminding ourselves definitions and properties of likelihood ratio processes.

A nonnegative random variable W has one of two probability density functions, either f or g.

Before the beginning of time, nature once and for all decides whether she will draw a sequence of IID draws from either f or g.

We will sometimes let q be the density that nature chose once and for all, so that q is either f or g, permanently.

Nature knows which density it permanently draws from, but we the observers do not.

We know both f and g but we don't know which density nature chose.

But we want to know.

To do that, we use observations.

We observe a sequence $\{w_t\}_{t=1}^T$ of T IID draws that we know came from either f or g.

We want to use these observations to infer whether nature chose f or g.

A **likelihood ratio process** is a useful tool for this task.

To begin, we define a key component of a likelihood ratio process, namely, the time t likelihood ratio as the random variable

$$\ell(w_t) = \frac{f(w_t)}{g(w_t)}, \quad t \geq 1.$$

We assume that f and g both put positive probabilities on the same intervals of possible realizations of the random variable W.

That means that under the g density, $\ell(w_t) = \frac{f(w_t)}{g(w_t)}$ is a nonnegative random variable with mean 1.

A **likelihood ratio process** for sequence $\{w_t\}_{t=1}^{\infty}$ is defined as

$$L(w^t) = \prod_{i=1}^{t} \ell(w_i),$$

where $w^t = \{w_1, \dots, w_t\}$ is a history of observations up to and including time t.

Sometimes for shorthand we'll write $L_t = L(w^t)$.

Notice that the likelihood process satisfies the *recursion*

$$L(w^t) = \ell(w_t) L(w^{t-1}).$$

The likelihood ratio and its logarithm are key tools for making inferences using a classic frequentist approach due to Neyman and Pearson [Neyman and Pearson, 1933].

To help us appreciate how things work, the following Python code evaluates f and g as two different Beta distributions, then computes and simulates an associated likelihood ratio process by generating a sequence w^t from one of the two probability distributions, for example, a sequence of IID draws from g.

```python
# Parameters in the two Beta distributions.
F_a, F_b = 1, 1
G_a, G_b = 3, 1.2

@vectorize
def p(x, a, b):
    r = gamma(a + b) / (gamma(a) * gamma(b))
    return r * x** (a-1) * (1 - x) ** (b-1)

# The two density functions.
f = jit(lambda x: p(x, F_a, F_b))
g = jit(lambda x: p(x, G_a, G_b))
```

```python
@jit
def simulate(a, b, T=50, N=500):
    '''
    Generate N sets of T observations of the likelihood ratio,
    return as N x T matrix.
    '''

    l_arr = np.empty((N, T))

    for i in range(N):
        for j in range(T):
            w = np.random.beta(a, b)
            l_arr[i, j] = f(w) / g(w)

    return l_arr
```

23.3 Blume and Easley's setting

Let the random variable $s_t \in (0, 1)$ at time $t = 0, 1, 2, \ldots$ be distributed according to the same Beta distribution with parameters $\theta = \{\theta_1, \theta_2\}$.

We'll denote this probability density as

$$\pi(s_t|\theta)$$

Below, we'll often just write $\pi(s_t)$ instead of $\pi(s_t|\theta)$ to save space.

Let $s_t \equiv y_t^1$ be the endowment of a nonstorable consumption good that a person we'll call "agent 1" receives at time t.

Let a history $s^t = [s_t, s_{t-1}, \ldots, s_0]$ be a sequence of i.i.d. random variables with joint distribution

$$\pi_t(s^t) = \pi(s_t)\pi(s_{t-1}) \cdots \pi(s_0)$$

So in our example, the history s^t is a comprehensive record of agent 1's endowments of the consumption good from time 0 up to time t.

If agent 1 were to live on an island by himself, agent 1's consumption $c^1(s_t)$ at time t is

$$c^1(s_t) = y_t^1 = s_t.$$

But in our model, agent 1 is not alone.

23.4 Nature and agents' beliefs

Nature draws i.i.d. sequences $\{s_t\}_{t=0}^{\infty}$ from $\pi_t(s^t)$.

- so π without a superscript is nature's model
- but in addition to nature, there are other entities inside our model – artificial people that we call "agents"
- each agent has a sequence of probability distributions over s^t for $t = 0, \ldots$
- agent i thinks that nature draws i.i.d. sequences $\{s_t\}_{t=0}^{\infty}$ from $\{\pi_t^i(s^t)\}_{t=0}^{\infty}$
  - agent i is mistaken unless $\pi_t^i(s^t) = \pi_t(s^t)$

Note

A **rational expectations** model would set $\pi_t^i(s^t) = \pi_t(s^t)$ for all agents i.

There are two agents named $i = 1$ and $i = 2$.

At time t, agent 1 receives an endowment

$$y_t^1 = s_t$$

of a nonstorable consumption good, while agent 2 receives an endowment of

$$y_t^2 = 1 - s_t$$

The aggregate endowment of the consumption good is

$$y_t^1 + y_t^2 = 1$$

at each date $t \geq 0$.

At date t agent i consumes $c_t^i(s^t)$ of the good.

A (non wasteful) feasible allocation of the aggregate endowment of 1 each period satisfies

$$c_t^1 + c_t^2 = 1.$$

23.5 A socialist risk-sharing arrangement

In order to share risks, a benevolent social planner dictates a history-dependent consumption allocation that takes the form of a sequence of functions

$$c_t^i = c_t^i(s^t)$$

that satisfy

$$c_t^1(s^t) + c_t^2(s^t) = 1 \tag{23.1}$$

for all s^t for all $t \geq 0$.

To design a socially optimal allocation, the social planner wants to know what each agent i believes about the endowment sequence and how they feel about bearing risks.

As for the endowment sequences, agent i believes that nature draws i.i.d. sequences from joint densities

$$\pi_t^i(s^t) = \pi^i(s_t)\pi^i(s_{t-1})\cdots\pi^i(s_0)$$

As for attitudes toward bearing risks, agent i has a one-period utility function

$$u(c_t^i) = \ln(c_t^i)$$

with marginal utility of consumption in period t

$$u'(c_t^i) = \frac{1}{c_t^i}$$

Putting its beliefs about its random endowment sequence and its attitudes toward bearing risks together, agent i has intertemporal utility function

$$V^i = \sum_{t=0}^{\infty} \sum_{s^t} \delta^t u(c_t^i(s^t))\pi_t^i(s^t), \tag{23.2}$$

where $\delta \in (0, 1)$ is an intertemporal discount factor, and $u(\cdot)$ is a strictly increasing, concave one-period utility function.

23.6 Social planner's allocation problem

The benevolent dictator has all the information it requires to choose a consumption allocation that maximizes the social welfare criterion

$$W = \lambda V^1 + (1 - \lambda)V^2 \tag{23.3}$$

where $\lambda \in [0, 1]$ is a Pareto weight that tells how much the planner likes agent 1 and $1 - \lambda$ is a Pareto weight that tells how much the social planner likes agent 2.

Setting $\lambda = .5$ expresses "egalitarian" social preferences.

Notice how social welfare criterion (23.3) takes into account both agents' preferences as represented by formula (23.2).

This means that the social planner knows and respects

- each agent's one period utility function $u(\cdot) = \ln(\cdot)$
- each agent i's probability model $\{\pi_t^i(s^t)\}_{t=0}^{\infty}$

Consequently, it is natural to anticipate that these objects will appear in the social planner's rule for allocating the aggregate endowment each period.

First-order necessary conditions for maximizing welfare criterion (23.3) subject to the feasibility constraint (23.1) are

$$\frac{\pi_t^2(s^t)}{\pi_t^1(s^t)} \frac{(1/c_t^2(s^t))}{(1/c_t^1(s^t))} = \frac{\lambda}{1-\lambda}$$

which can be rearranged to become

$$\frac{c_t^1(s^t)}{c_t^2(s^t)} = \frac{\lambda}{1-\lambda} l_t(s^t) \tag{23.4}$$

where

$$l_t(s^t) = \frac{\pi_t^1(s^t)}{\pi_t^2(s^t)}$$

is the likelihood ratio of agent 1's joint density to agent 2's joint density.

Using

$$c_t^1(s^t) + c_t^2(s^t) = 1$$

we can rewrite allocation rule (23.4) as

$$\frac{c_t^1(s^t)}{1 - c_t^1(s^t)} = \frac{\lambda}{1-\lambda} l_t(s^t)$$

or

$$c_t^1(s^t) = \frac{\lambda}{1-\lambda} l_t(s^t)(1 - c_t^1(s^t))$$

which implies that the social planner's allocation rule is

$$c_t^1(s^t) = \frac{\lambda l_t(s^t)}{1 - \lambda + \lambda l_t(s^t)} \tag{23.5}$$

If we define a temporary or **continuation Pareto weight** process as

$$\lambda_t(s^t) = \frac{\lambda l_t(s^t)}{1 - \lambda + \lambda l_t(s^t)},$$

then we can represent the social planner's allocation rule as

$$c_t^1(s^t) = \lambda_t(s^t)$$

and of course

$$c_t^2(s^t) = 1 - \lambda_t(s^t).$$

23.7 If you're so smart, …

Let's compute some values of limiting allocations (23.5) for some interesting possible limiting values of the likelihood ratio process $l_t(s^t)$.

As our first case, let's suppose that

$$l_\infty(s^\infty) = 1; \quad c^1_\infty = \lambda$$

- In this case, both agents are equally smart (or equally not smart) and the consumption allocation stays put at a $\lambda, 1 - \lambda$ split between the two agents.

As our second case, let suppose that

$$l_\infty(s^\infty) = 0; \quad c^1_\infty = 0$$

- In this case, agent 2 is "smarter" than agent 1, and agent 1's share of the aggregate endowment converges to zero.

As our third case, let's suppose that

$$l_\infty(s^\infty) = \infty; \quad c^1_\infty = 1$$

- In this case, agent 1 is smarter than agent 2, and agent 1's share of the aggregate endowment converges to 1.

Note

These three cases are somehow telling us about how relative **wealths** of the agents evolve as time passes.

- when the two agents are equally smart and $\lambda \in (0, 1)$, agent 1's wealth share stays at λ perpetually.
- when agent 1 is smarter and $\lambda \in (0, 1)$, agent 1 eventually "owns" the entire continuation endowment and agent 2 eventually "owns" nothing.
- when agent 2 is smarter and $\lambda \in (0, 1)$, agent 2 eventually "owns" the entire continuation endowment and agent 1 eventually "owns" nothing.

We can define continuation wealths precisely after we construct a competitive equilibrium **price** system below.

Soon we'll do some simulations that will shed further light on possible outcomes.

But before we do that, let's take a detour and study some "shadow prices" for the social planning problem.

We shall soon see that these shadow prices can readily be converted to "equilibrium prices" for a competitive equilibrium.

Doing this will allow us to connect our analysis with claims by [Alchian, 1950] and [Friedman, 1953] that transfers of wealth coming from trades in competitive asset markets eventually make prices of risky assets reflect realistic probability assessments.

23.8 Competitive equilibrium prices

Two fundamental welfare theorems for general equilibrium models lead us to anticipate that there is a connection between the allocation that solves the social planning problem we have been studying and the allocation in a **competitive equilibrium** with complete markets in contingent claims to time- and history-dependent consumption goods.

Note

For the two welfare theorems and their history, see https://en.wikipedia.org/wiki/Fundamental_theorems_of_welfare_economics. Again, for applications to a classic macroeconomic growth model, see *this lecture on a planning problem* and *this lecture on a related competitive equilibrium*

Such a connection prevails for our model.

We'll sketch it now.

In a competitive equilibrium, no social planner dictatorially collects everybody's endowments and then reallocates them.

Instead, there is a comprehensive centralized market that meets at one point in time.

There are **prices** at which price-taking agents can buy or sell whatever goods that they want.

Trade is multilateral in the sense that that there is a "Walrasian auctioneer" who lives outside the model and whose job it is to verify that each agent's budget constraint is satisfied.

That budget constraint requires that the total value of the agent's endowment stream be at least as the total value of its consumption stream.

These values are computed at price vectors that the agents take as given – the agents are "price-takers" who assume that they can buy or sell whatever quantities that they want at those prices.

Suppose that at time -1, before time 0 starts, agent i can purchase one unit $c_t(s^t)$ of consumption at time t after history s^t at price $p_t(s^t)$.

Notice that there is (very long) **vector** of prices.

- there is one price $p_t(s^t)$ for each history s^t at every date $t = 0, 1, \ldots,$.
- so there are as many prices as there are histories and dates.

These prices determined at time -1 before the economy starts.

The market meets once at time -1.

At times $t = 0, 1, 2, \ldots$ trades made at time -1 are simply executed, i.e., the promised deliveries are made.

- in the background, there is an "enforcement" procedure that forces agents to carry out the exchanges or "deliveries" that they agreed to at time -1.

We want to study how agents' probability models influence equilibrium prices.

Agent i faces a **single** intertemporal budget constraint

$$\sum_{t=0}^{\infty}\sum_{s^t} p_t(s^t) c_t^i(s^t) \leq \sum_{t=0}^{\infty}\sum_{s^t} p_t(s^t) y_t^i(s^t) \tag{23.6}$$

According to budget constraint (23.6), trade is **multilateral** in the following sense

- imagine that agent i first sells his random endowment stream $\{y_t^i(s^t)\}$ and then uses the proceeds (i.e., his "wealth") to purchase a random consumption stream $\{c_t^i(s^t)\}$.

Agent i attaches a Lagrange multiplier μ_i to budget constraint (23.6) and once-and-for-all chooses a consumption plan $\{c_t^i(s^t)\}_{t=0}^{\infty}$ that maximizes criterion (23.2) subject to budget constraint (23.6).

This means that the agent i chooses many objects, namely, $c_t^i(s^t)$ for all s^t for $t = 0, 1, 2, \ldots$.

For convenience, let's remind ourselves of criterion V^i defined in (23.2):

$$V^i = \sum_{t=0}^{\infty}\sum_{s^t} \delta^t u(c_t^i(s^t)) \pi_t^i(s^t)$$

First-order necessary conditions for maximizing objective V^i defined in (23.2) with respect to $c_t^i(s^t)$ are

$$\delta^t u'(c_t^i(s^t))\pi_t^i(s^t) = \mu_i p_t(s^t),$$

which we can rearrange to obtain

$$p_t(s^t) = \frac{\delta^t \pi_t^i(s^t)}{\mu_i c_t^i(s^t)} \tag{23.7}$$

for $i = 1, 2$.

If we divide equation (23.7) for agent 1 by the appropriate version of equation (23.7) for agent 2, impose the feasibility condition $c_t^2(s^t) = 1 - c_t^1(s^t)$, and do some algebra, we'll obtain

$$c_t^1(s^t) = \frac{\mu_1 l_t(s^t)}{\mu_2 + \mu_1 l_t(s^t)}. \tag{23.8}$$

We now embark on an extended "guess-and-verify" exercise that involves matching objects in our competitive equilibrium with objects in our social planning problem.

- we'll match consumption allocations in the planning problem with equilibrium consumption allocations in the competitive equilibrium
- we'll match "shadow" prices in the planning problem with competitive equilibrium prices.

Notice that if we set $\mu_1 = 1 - \lambda$ and $\mu_2 = \lambda$, then formula (23.8) agrees with formula (23.5).

- doing this amounts to choosing a **numeraire** or normalization for the price system $\{p_t(s^t)\}_{t=0}^\infty$

> **Note**
>
> For information about how a numeraire must be chosen to pin down the absolute price level in a model like ours that determines only relative prices, see https://en.wikipedia.org/wiki/Numéraire.

If we substitute formula (23.8) for $c_t^1(s^t)$ into formula (23.7) and rearrange, we obtain

$$p_t(s^t) = \frac{\delta^t}{\lambda(1-\lambda)} \pi_t^2(s^t)[1 - \lambda + \lambda l_t(s^t)]$$

or

$$p_t(s^t) = \frac{\delta^t}{\lambda(1-\lambda)} [(1-\lambda)\pi_t^2(s^t) + \lambda \pi_t^1(s^t)] \tag{23.9}$$

According to formula (23.9), we have the following possible limiting cases:

- when $l_\infty = 0$, $c_\infty^1 = 0$ and tails of competitive equilibrium prices reflect agent 2's probability model $\pi_t^2(s^t)$ according to $p_t(s^t) \propto \delta^t \pi_t^2(s^t)$
- when $l_\infty = \infty$, $c_\infty^1 = 1$ and tails of competitive equilibrium prices reflect agent 1's probability model $\pi_t^1(s^t)$ according to $p_t(s^t) \propto \delta^t \pi_t^1(s^t)$
- for small t's, competitive equilibrium prices reflect both agents' probability models.

We leave the verification of the shadow prices to the reader since it follows from the same reasoning.

23.9 Simulations

Now let's implement some simulations when agent 1 believes marginal density

$$\pi^1(s_t) = f(s_t)$$

and agent 2 believes marginal density

$$\pi^2(s_t) = g(s_t)$$

where f and g are Beta distributions like ones that we used in earlier sections of this lecture.

Meanwhile, we'll assume that nature believes a marginal density

$$\pi(s_t) = h(s_t)$$

where $h(s_t)$ is perhaps a mixture of f and g.

First, we write a function to compute the likelihood ratio process

```
def compute_likelihood_ratios(sequences, f, g):
    """Compute likelihood ratios and cumulative products."""
    l_ratios = f(sequences) / g(sequences)
    L_cumulative = np.cumprod(l_ratios, axis=1)
    return l_ratios, L_cumulative
```

Let's compute the Kullback–Leibler discrepancies by quadrature integration.

```
def compute_KL(f, g):
    """
    Compute KL divergence KL(f, g)
    """
    integrand = lambda w: f(w) * np.log(f(w) / g(w))
    val, _ = quad(integrand, 1e-5, 1-1e-5)
    return val
```

We also create a helper function to compute KL divergence with respect to a reference distribution h

```
def compute_KL_h(h, f, g):
    """
    Compute KL divergence with reference distribution h
    """

    Kf = compute_KL(h, f)
    Kg = compute_KL(h, g)

    return Kf, Kg
```

Let's write a Python function that computes agent 1's consumption share

```
def simulate_blume_easley(sequences, f_belief=f, g_belief=g, λ=0.5):
    """Simulate Blume-Easley model consumption shares."""
    l_ratios, l_cumulative = compute_likelihood_ratios(sequences, f_belief, g_belief)
    c1_share = λ * l_cumulative / (1 - λ + λ * l_cumulative)
    return l_cumulative, c1_share
```

Now let's use this function to generate sequences in which

- nature draws from f each period, or

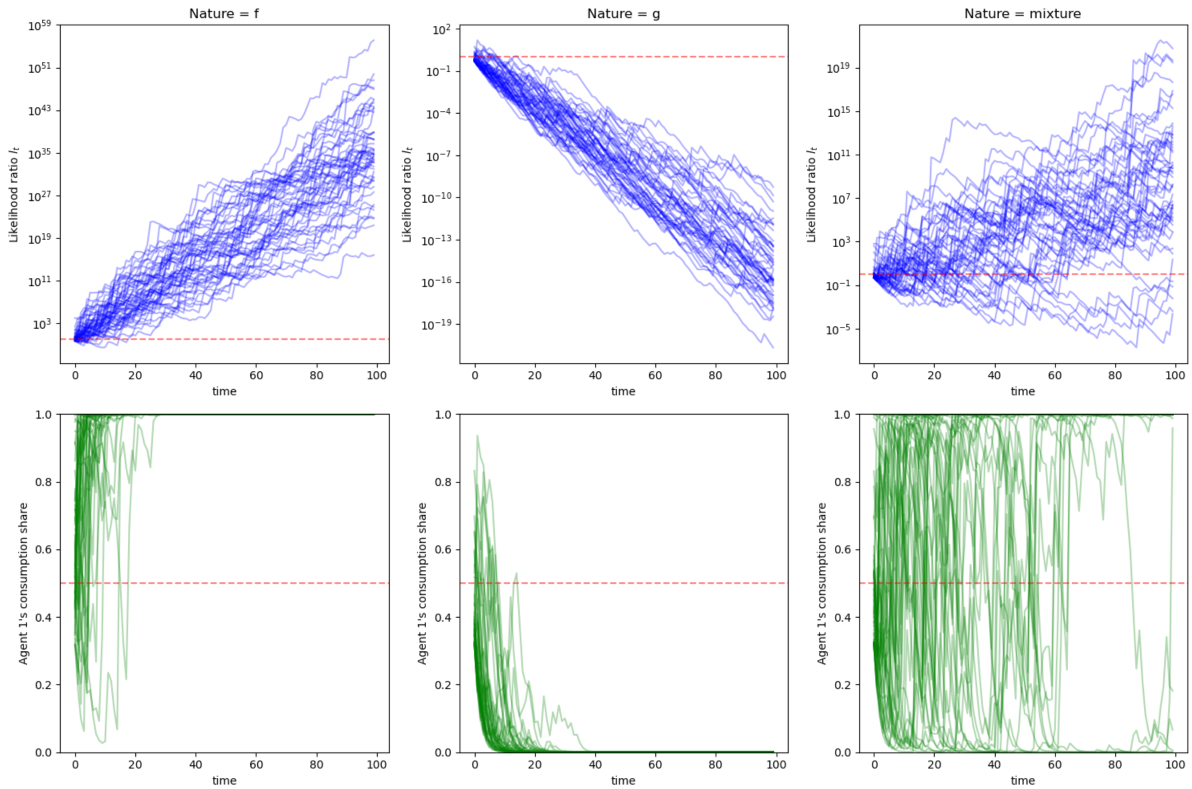
- nature draws from g each period, or
- nature flips a fair coin each period to decide whether to draw from f or g

```
λ = 0.5
T = 100
N = 10000

# Nature follows f, g, or mixture
s_seq_f = np.random.beta(F_a, F_b, (N, T))
s_seq_g = np.random.beta(G_a, G_b, (N, T))

h = jit(lambda x: 0.5 * f(x) + 0.5 * g(x))
model_choices = np.random.rand(N, T) < 0.5
s_seq_h = np.empty((N, T))
s_seq_h[model_choices] = np.random.beta(F_a, F_b, size=model_choices.sum())
s_seq_h[~model_choices] = np.random.beta(G_a, G_b, size=(~model_choices).sum())

l_cum_f, c1_f = simulate_blume_easley(s_seq_f)
l_cum_g, c1_g = simulate_blume_easley(s_seq_g)
l_cum_h, c1_h = simulate_blume_easley(s_seq_h)
```

Before looking at the figure below, have some fun by guessing whether agent 1 or agent 2 will have a larger and larger consumption share as time passes in our three cases.

To make better guesses, let's visualize instances of the likelihood ratio processes in the three cases.

```
fig, axes = plt.subplots(2, 3, figsize=(15, 10))

titles = ["Nature = f", "Nature = g", "Nature = mixture"]
data_pairs = [(l_cum_f, c1_f), (l_cum_g, c1_g), (l_cum_h, c1_h)]

for i, ((l_cum, c1), title) in enumerate(zip(data_pairs, titles)):
    # Likelihood ratios
    ax = axes[0, i]
    for j in range(min(50, l_cum.shape[0])):
        ax.plot(l_cum[j, :], alpha=0.3, color='blue')
    ax.set_yscale('log')
    ax.set_xlabel('time')
    ax.set_ylabel('Likelihood ratio $l_t$')
    ax.set_title(title)
    ax.axhline(y=1, color='red', linestyle='--', alpha=0.5)

    # Consumption shares
    ax = axes[1, i]
    for j in range(min(50, c1.shape[0])):
        ax.plot(c1[j, :], alpha=0.3, color='green')
    ax.set_xlabel('time')
    ax.set_ylabel("Agent 1's consumption share")
    ax.set_ylim([0, 1])
    ax.axhline(y=λ, color='red', linestyle='--', alpha=0.5)

plt.tight_layout()
plt.show()
```



In the left panel, nature chooses f. Agent 1's consumption reaches 1 very quickly.

In the middle panel, nature chooses g. Agent 1's consumption ratio tends to move towards 0 but not as fast as in the first case.

In the right panel, nature flips coins each period. We see a very similar pattern to the processes in the left panel.

The figures in the top panel remind us of the discussion in *this section*.

We invite readers to revisit *that section* and try to infer the relationships among $D_{KL}(f\|g)$, $D_{KL}(g\|f)$, $D_{KL}(h\|f)$, and $D_{KL}(h\|g)$.

Let's compute values of KL divergence

```python
shares = [np.mean(c1_f[:, -1]), np.mean(c1_g[:, -1]), np.mean(c1_h[:, -1])]
Kf_g, Kg_f = compute_KL(f, g), compute_KL(g, f)
Kf_h, Kg_h = compute_KL_h(h, f, g)

print(f"Final shares: f={shares[0]:.3f}, g={shares[1]:.3f}, mix={shares[2]:.3f}")
print(f"KL divergences: \nKL(f,g)={Kf_g:.3f}, KL(g,f)={Kg_f:.3f}")
print(f"KL(h,f)={Kf_h:.3f}, KL(h,g)={Kg_h:.3f}")
```

```
Final shares: f=1.000, g=0.000, mix=0.927
KL divergences:
KL(f,g)=0.759, KL(g,f)=0.344
KL(h,f)=0.073, KL(h,g)=0.281
```

We find that $KL(f,g) > KL(g,f)$ and $KL(h,g) > KL(h,f)$.

The first inequality tells us that the average "surprise" from having belief g when nature chooses f is greater than the "surprise" from having belief f when nature chooses g.

This explains the difference between the first two panels we noted above.

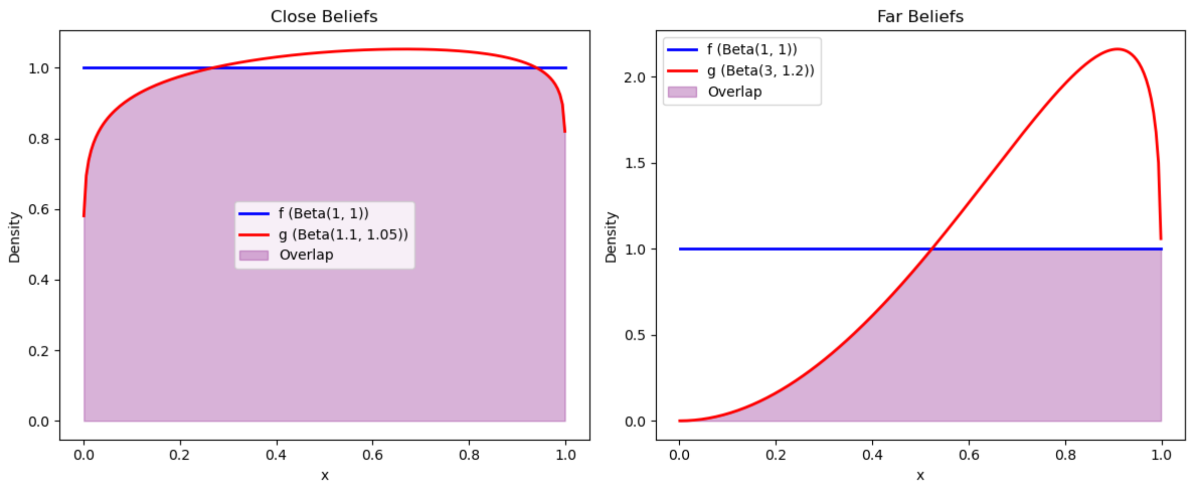
The second inequality tells us that agent 1's belief distribution f is closer to nature's pick than agent 2's belief g.

To make this idea more concrete, let's compare two cases:

- agent 1's belief distribution f is close to agent 2's belief distribution g;
- agent 1's belief distribution f is far from agent 2's belief distribution g.

We use the two distributions visualized below

```
def plot_distribution_overlap(ax, x_range, f_vals, g_vals,
                              f_label='f', g_label='g',
                              f_color='blue', g_color='red'):
    """Plot two distributions with their overlap region."""
    ax.plot(x_range, f_vals, color=f_color, linewidth=2, label=f_label)
    ax.plot(x_range, g_vals, color=g_color, linewidth=2, label=g_label)

    overlap = np.minimum(f_vals, g_vals)
    ax.fill_between(x_range, 0, overlap, alpha=0.3, color='purple', label='Overlap')
    ax.set_xlabel('x')
    ax.set_ylabel('Density')
    ax.legend()

# Define close and far belief distributions
f_close = jit(lambda x: p(x, 1, 1))
g_close = jit(lambda x: p(x, 1.1, 1.05))

f_far = jit(lambda x: p(x, 1, 1))
g_far = jit(lambda x: p(x, 3, 1.2))

# Visualize the belief distributions
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

x_range = np.linspace(0.001, 0.999, 200)

# Close beliefs
f_close_vals = [f_close(x) for x in x_range]
g_close_vals = [g_close(x) for x in x_range]
plot_distribution_overlap(ax1, x_range, f_close_vals, g_close_vals,
                         f_label='f (Beta(1, 1))', g_label='g (Beta(1.1, 1.05))')
ax1.set_title(f'Close Beliefs')

# Far beliefs
f_far_vals = [f_far(x) for x in x_range]
g_far_vals = [g_far(x) for x in x_range]
plot_distribution_overlap(ax2, x_range, f_far_vals, g_far_vals,
                         f_label='f (Beta(1, 1))', g_label='g (Beta(3, 1.2))')
ax2.set_title(f'Far Beliefs')

plt.tight_layout()
plt.show()
```



Let's draw the same consumption ratio plots as above for agent 1.

We replace the simulation paths with median and percentiles to make the figure cleaner.

Staring at the figure below, can we infer the relation between $KL(f, g)$ and $KL(g, f)$?

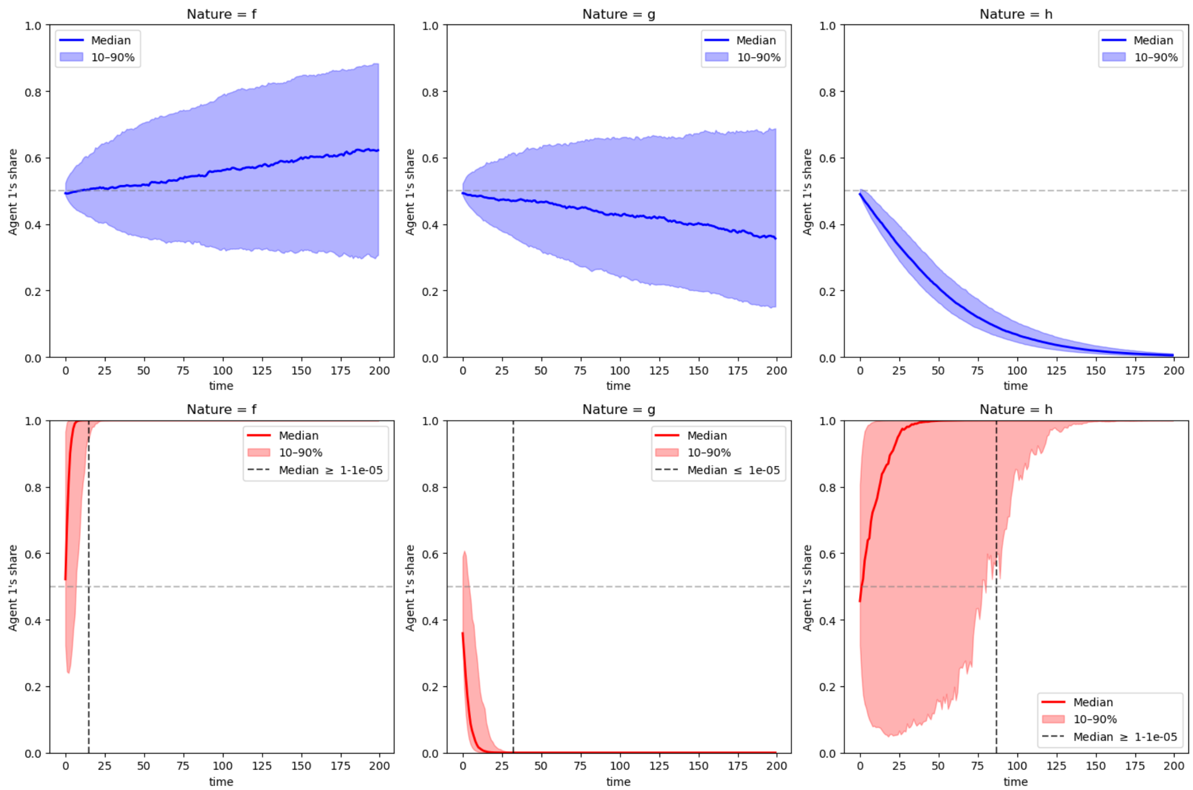
From the right panel, can we infer the relation between $KL(h, g)$ and $KL(h, f)$?

```
fig, axes = plt.subplots(2, 3, figsize=(15, 10))
nature_params = {'close': [(1, 1), (1.1, 1.05), (2, 1.5)],
                 'far':   [(1, 1), (3, 1.2),   (2, 1.5)]}
nature_labels = ["Nature = f", "Nature = g", "Nature = h"]
colors = {'close': 'blue', 'far': 'red'}

threshold = 1e-5  # "close to zero" cutoff

for row, (f_belief, g_belief, label) in enumerate([
                        (f_close, g_close, 'close'),
                        (f_far, g_far, 'far')]):

    for col, nature_label in enumerate(nature_labels):
        params = nature_params[label][col]
        s_seq = np.random.beta(params[0], params[1], (1000, 200))
        _, c1 = simulate_blume_easley(s_seq, f_belief, g_belief, λ)

        median_c1 = np.median(c1, axis=0)
        p10, p90 = np.percentile(c1, [10, 90], axis=0)

        ax = axes[row, col]
        color = colors[label]
        ax.plot(median_c1, color=color, linewidth=2, label='Median')
        ax.fill_between(range(len(median_c1)), p10, p90, alpha=0.3, color=color,␣
↪label='10-90%')
        ax.set_xlabel('time')
        ax.set_ylabel("Agent 1's share")
        ax.set_ylim([0, 1])
        ax.set_title(nature_label)
        ax.axhline(y=λ, color='gray', linestyle='--', alpha=0.5)
        below = np.where(median_c1 < threshold)[0]
        above = np.where(median_c1 > 1-threshold)[0]
        if below.size > 0: first_zero = (below[0], True)
        elif above.size > 0: first_zero = (above[0], False)
```

(continues on next page)

(continued from previous page)

```
        else: first_zero = None
        if first_zero is not None:
            ax.axvline(x=first_zero[0], color='black', linestyle='--',
                       alpha=0.7,
                       label=fr'Median $\leq$ {threshold}' if first_zero[1]
                       else fr'Median $\geq$ 1-{threshold}')
        ax.legend()

plt.tight_layout()
plt.show()
```



Holding to our guesses, let's calculate the four values

```
# Close case
Kf_g, Kg_f = compute_KL(f_close, g_close), compute_KL(g_close, f_close)
Kf_h, Kg_h = compute_KL_h(h, f_close, g_close)

print(f"KL divergences (close): \nKL(f,g)={Kf_g:.3f}, KL(g,f)={Kg_f:.3f}")
print(f"KL(h,f)={Kf_h:.3f}, KL(h,g)={Kg_h:.3f}")

# Far case
Kf_g, Kg_f = compute_KL(f_far, g_far), compute_KL(g_far, f_far)
Kf_h, Kg_h = compute_KL_h(h, f_far, g_far)

print(f"KL divergences (far): \nKL(f,g)={Kf_g:.3f}, KL(g,f)={Kg_f:.3f}")
print(f"KL(h,f)={Kf_h:.3f}, KL(h,g)={Kg_h:.3f}")
```

```
KL divergences (close):
```

(continues on next page)

(continued from previous page)

```
KL(f,g)=0.003, KL(g,f)=0.003
KL(h,f)=0.073, KL(h,g)=0.061
KL divergences (far):
KL(f,g)=0.759, KL(g,f)=0.344
KL(h,f)=0.073, KL(h,g)=0.281
```

We find that in the first case, $KL(f,g) \approx KL(g,f)$ and both are relatively small, so although either agent 1 or agent 2 will eventually consume everything, convergence displayed in the first two panels on the top is pretty slow.

In the first two panels at the bottom, we see convergence occurring faster (as indicated by the black dashed line) because the divergence gaps $KL(f,g)$ and $KL(g,f)$ are larger.

Since $KL(f,g) > KL(g,f)$, we see faster convergence in the first panel at the bottom when nature chooses f than in the second panel where nature chooses g.

This ties in nicely with (22.1).

23.10 Related lectures

Complete markets models with homogeneous beliefs, a kind often used in macroeconomics and finance, are studied in this quantecon lecture *Competitive Equilibria with Arrow Securities*.

[Blume *et al.*, 2018] discuss a paternalistic case against complete markets. They study the consequences of assuming that a social planner disregards individuals preferences in the sense that it ignores the subjective belief components of their preferences and replaces it with the social planner's beliefs about probabilities.

Likelihood processes play an important role in Bayesian learning, as described in *Likelihood Ratio Processes and Bayesian Learning* and as applied in *Job Search VIII: Search with Learning*.

Likelihood ratio processes appear again in Additive and Multiplicative Functionals.

23.11 Exercises

Exercise 23.11.1

Starting from (23.7), show that the competitive equilibrium prices can be expressed as

$$p_t(s^t) = \frac{\delta^t}{\lambda(1-\lambda)} \pi_t^2(s^t) \left[1 - \lambda + \lambda l_t(s^t)\right]$$

Solution

Starting from

$$p_t(s^t) = \frac{\delta^t \pi_t^i(s^t)}{\mu_i c_t^i(s^t)}, \qquad i = 1, 2.$$

Since both expressions equal the same price, we can equate them

$$\frac{\pi_t^1(s^t)}{\mu_1 c_t^1(s^t)} = \frac{\pi_t^2(s^t)}{\mu_2 c_t^2(s^t)}$$

Rearranging gives

$$\frac{c_t^1(s^t)}{c_t^2(s^t)} = \frac{\mu_2}{\mu_1} l_t(s^t)$$

where $l_t(s^t) \equiv \pi_t^1(s^t)/\pi_t^2(s^t)$ is the likelihood ratio process.

Using $c_t^2(s^t) = 1 - c_t^1(s^t)$:

$$\frac{c_t^1(s^t)}{1 - c_t^1(s^t)} = \frac{\mu_2}{\mu_1} l_t(s^t)$$

Solving for $c_t^1(s^t)$

$$c_t^1(s^t) = \frac{\mu_2 l_t(s^t)}{\mu_1 + \mu_2 l_t(s^t)}$$

The planner's solution gives

$$c_t^1(s^t) = \frac{\lambda l_t(s^t)}{1 - \lambda + \lambda l_t(s^t)}$$

To match agent 1's choice in a competitive equilibrium with the planner's choice for agent 1, the following equality must hold

$$\frac{\mu_2}{\mu_1} = \frac{\lambda}{1 - \lambda}$$

Hence we have

$$\mu_1 = 1 - \lambda, \qquad \mu_2 = \lambda$$

With $\mu_1 = 1 - \lambda$ and $c_t^1(s^t) = \frac{\lambda l_t(s^t)}{1-\lambda+\lambda l_t(s^t)}$, we have

$$\begin{aligned}
p_t(s^t) &= \frac{\delta^t \pi_t^1(s^t)}{(1-\lambda) c_t^1(s^t)} \\
&= \frac{\delta^t \pi_t^1(s^t)}{(1-\lambda)} \cdot \frac{1 - \lambda + \lambda l_t(s^t)}{\lambda l_t(s^t)} \\
&= \frac{\delta^t \pi_t^1(s^t)}{(1-\lambda)\lambda l_t(s^t)} [1 - \lambda + \lambda l_t(s^t)].
\end{aligned}$$

Since $\pi_t^1(s^t) = l_t(s^t) \pi_t^2(s^t)$, we have

$$\begin{aligned}
p_t(s^t) &= \frac{\delta^t l_t(s^t) \pi_t^2(s^t)}{(1-\lambda)\lambda l_t(s^t)} [1 - \lambda + \lambda l_t(s^t)] \\
&= \frac{\delta^t \pi_t^2(s^t)}{(1-\lambda)\lambda} [1 - \lambda + \lambda l_t(s^t)] \\
&= \frac{\delta^t}{\lambda(1-\lambda)} \pi_t^2(s^t) [1 - \lambda + \lambda l_t(s^t)].
\end{aligned}$$

Exercise 23.11.2

In this exercise, we'll study two agents, each of whom updates its posterior probability as data arrive.

- each agent applies Bayes' law in the way studied in *Likelihood Ratio Processes and Bayesian Learning*.

The following two models are on the table

$$f(s^t) = f(s_1)f(s_2)\cdots f(s_t)$$

and

$$g(s^t) = g(s_1)g(s_2)\cdots g(s_t)$$

as is an associated likelihood ratio process

$$L(s^t) = \frac{f(s^t)}{g(s^t)}.$$

Let $\pi_0 \in (0,1)$ be a prior probability and

$$\pi_t = \frac{\pi_0 L(s^t)}{\pi_0 L(s^t) + (1-\pi_0)}.$$

Each of our two agents deploys its own version of the mixture model

$$m(s^t) = \pi_t f(s^t) + (1-\pi_t)g(s^t) \tag{23.10}$$

We'll endow each type of consumer with model (23.10).

- The two agents share the same f and g, but
- they have different initial priors, say π_0^1 and π_0^2

Thus, consumer i's probability model is

$$m^i(s^t) = \pi_t^i f(s^t) + (1-\pi_t^i)g(s^t) \tag{23.11}$$

We now hand probability models (23.11) for $i = 1, 2$ to the social planner.

We want to deduce allocation $c^i(s^t), i = 1, 2$, and watch what happens when

- nature's model is f
- nature's model is g

We expect that consumers will eventually learn the "truth", but that one of them will learn faster.

To explore things, please set $f \sim \text{Beta}(1.5, 1)$ and $g \sim \text{Beta}(1, 1.5)$.

Please write Python code that answers the following questions.

- How do consumption shares evolve?
- Which agent learns faster when nature follows f?
- Which agent learns faster when nature follows g?
- How does a difference in initial priors π_0^1 and π_0^2 affect the convergence speed?

Solution

First, let's write helper functions that compute model components including each agent's subjective belief function.

```
def bayesian_update(π_0, L_t):
    """
    Bayesian update of belief probability given likelihood ratio.
    """
    return (π_0 * L_t) / (π_0 * L_t + (1 - π_0))

def mixture_density_belief(s_seq, f_func, g_func, π_seq):
    """
    Compute the mixture density beliefs m^i(s^t) for agent i.
    """
    f_vals = f_func(s_seq)
    g_vals = g_func(s_seq)
    return π_seq * f_vals + (1 - π_seq) * g_vals
```

Now let's write code that simulates the Blume-Easley model with our two agents.

```
def simulate_learning_blume_easley(sequences, f_belief, g_belief,
                                    π_0_1, π_0_2, λ=0.5):
    """
    Simulate Blume-Easley model with learning agents.
    """
    N, T = sequences.shape

    # Initialize arrays to store results
    π_1_seq = np.full((N, T), np.nan)
    π_2_seq = np.full((N, T), np.nan)
    c1_share = np.full((N, T), np.nan)
    l_agents_seq = np.full((N, T), np.nan)

    π_1_seq[:, 0] = π_0_1
    π_2_seq[:, 0] = π_0_2

    for n in range(N):
        # Initialize cumulative likelihood ratio for beliefs
        L_cumul = 1.0

        # Initialize likelihood ratio between agent densities
        l_agents_cumul = 1.0

        for t in range(1, T):
            s_t = sequences[n, t]

            # Compute likelihood ratio for this observation
            l_t = f_belief(s_t) / g_belief(s_t)

            # Update cumulative likelihood ratio
            L_cumul *= l_t

            # Bayesian update of beliefs
            π_1_t = bayesian_update(π_0_1, L_cumul)
            π_2_t = bayesian_update(π_0_2, L_cumul)

            # Store beliefs
            π_1_seq[n, t] = π_1_t
            π_2_seq[n, t] = π_2_t

            # Compute mixture densities for each agent
            m1_t = π_1_t * f_belief(s_t) + (1 - π_1_t) * g_belief(s_t)
```

```
            m2_t = π_2_t * f_belief(s_t) + (1 - π_2_t) * g_belief(s_t)

            # Update cumulative likelihood ratio between agents
            l_agents_cumul *= (m1_t / m2_t)
            l_agents_seq[n, t] = l_agents_cumul

            # c_t^1(s^t) = λ * l_t(s^t) / (1 - λ + λ * l_t(s^t))
            # where l_t(s^t) is the cumulative likelihood ratio between agents
            c1_share[n, t] = λ * l_agents_cumul / (1 - λ + λ * l_agents_cumul)

    return {
        'π_1': π_1_seq,
        'π_2': π_2_seq,
        'c1_share': c1_share,
        'l_agents': l_agents_seq
    }
```

Let's run simulations for different scenarios.

We use $\lambda = 0.5$, $T = 40$, and $N = 1000$.

```
λ = 0.5
T = 40
N = 1000

F_a, F_b = 1.5, 1
G_a, G_b = 1, 1.5

f = jit(lambda x: p(x, F_a, F_b))
g = jit(lambda x: p(x, G_a, G_b))
```

We'll start with different initial priors $\pi_0^i \in (0, 1)$ and widen the gap between them.

```
# Different initial priors
π_0_scenarios = [
    (0.3, 0.7),
    (0.7, 0.3),
    (0.1, 0.9),
]
```

Now we can run simulations for different scenarios

```
# Nature follows f
s_seq_f = np.random.beta(F_a, F_b, (N, T))

# Nature follows g
s_seq_g = np.random.beta(G_a, G_b, (N, T))

results_f = {}
results_g = {}

for i, (π_0_1, π_0_2) in enumerate(π_0_scenarios):
    # When nature follows f
    results_f[i] = simulate_learning_blume_easley(
            s_seq_f, f, g, π_0_1, π_0_2, λ)
    # When nature follows g
    results_g[i] = simulate_learning_blume_easley(
            s_seq_g, f, g, π_0_1, π_0_2, λ)
```

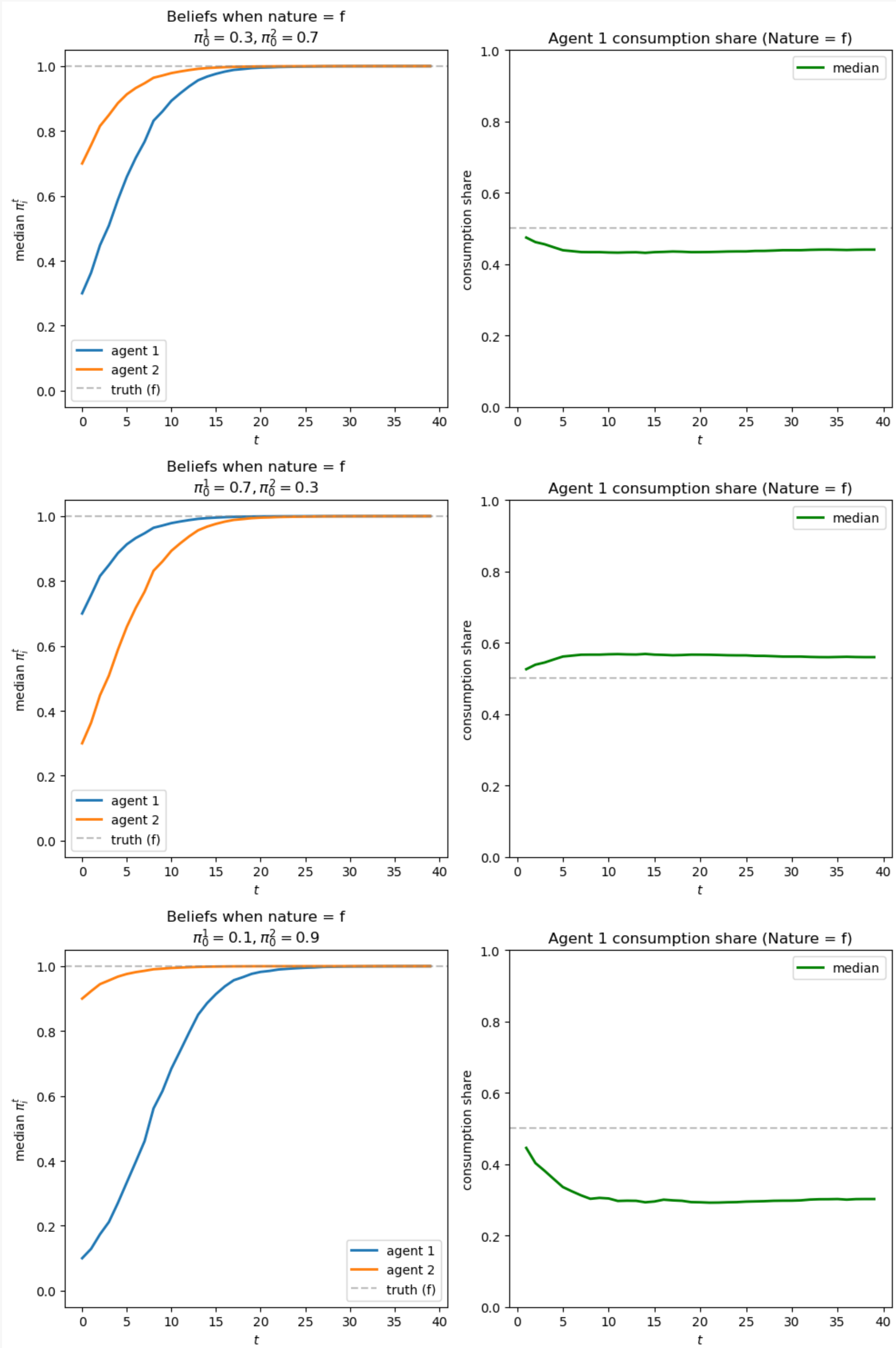
Let's visualize the results

```
def plot_learning_results(results, π_0_scenarios, nature_type, truth_value):
    """
```

```
    Plot beliefs and consumption shares for learning agents.
    """

    fig, axes = plt.subplots(3, 2, figsize=(10, 15))

    scenario_labels = [
        rf'$\pi_0^1 = {π_0_1}, \pi_0^2 = {π_0_2}$'
        for π_0_1, π_0_2 in π_0_scenarios
    ]

    for row, (scenario_idx, scenario_label) in enumerate(
                zip(range(3), scenario_labels)):

        res = results[scenario_idx]

        # Plot beliefs
        ax = axes[row, 0]
        π_1_med = np.median(res['π_1'], axis=0)
        π_2_med = np.median(res['π_2'], axis=0)
        ax.plot(π_1_med, 'C0', label=r'agent 1', linewidth=2)
        ax.plot(π_2_med, 'C1', label=r'agent 2', linewidth=2)
        ax.axhline(y=truth_value, color='gray', linestyle='--',
                   alpha=0.5, label=f'truth ({nature_type})')
        ax.set_title(f'Beliefs when nature = {nature_type}\n{scenario_label}')
        ax.set_ylabel(r'median $\pi_i^t$')
        ax.set_ylim([-0.05, 1.05])
        ax.legend()

        # Plot consumption shares
        ax = axes[row, 1]
        c1_med = np.median(res['c1_share'], axis=0)
        ax.plot(c1_med, 'g-', linewidth=2, label='median')
        ax.axhline(y=0.5, color='gray', linestyle='--',
                   alpha=0.5)
        ax.set_title(f'Agent 1 consumption share (Nature = {nature_type})')
        ax.set_ylabel('consumption share')
        ax.set_ylim([0, 1])
        ax.legend()

        # Add x-labels
        for col in range(2):
            axes[row, col].set_xlabel('$t$')

    plt.tight_layout()
    return fig, axes
```

Now we'll plot outcome when nature follows f:

```
fig_f, axes_f = plot_learning_results(
                results_f, π_0_scenarios, 'f', 1.0)
plt.show()
```



Beliefs when nature = f
$\pi_0^1 = 0.3, \pi_0^2 = 0.7$
median π_i^t
t
agent 1
agent 2
truth (f)
Agent 1 consumption share (Nature = f)
consumption share
median
Beliefs when nature = f
$\pi_0^1 = 0.7, \pi_0^2 = 0.3$
Beliefs when nature = f
$\pi_0^1 = 0.1, \pi_0^2 = 0.9$

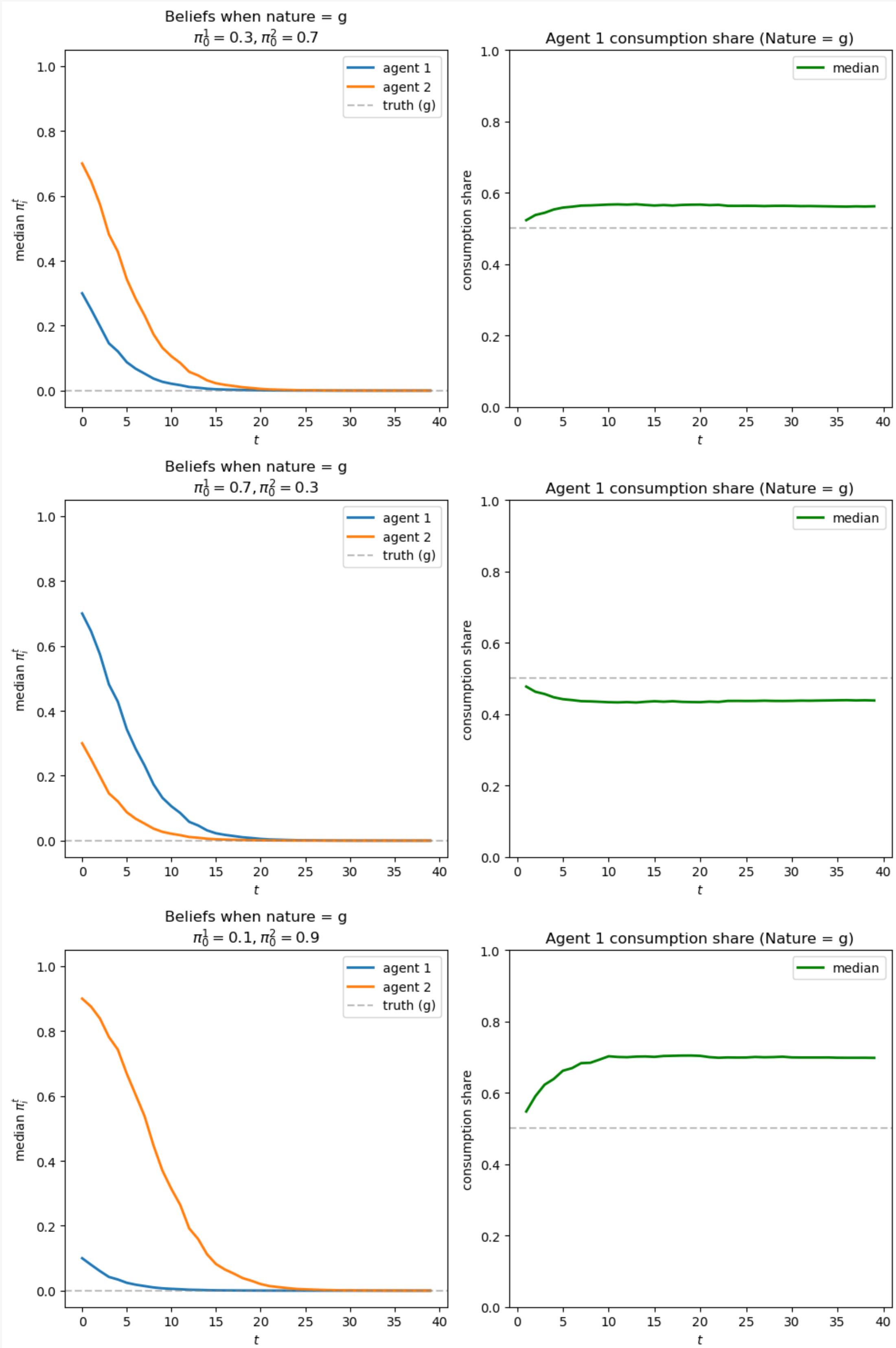
We can see that the agent with the more accurate belief gets higher consumption share.

Moreover, the further apart are initial beliefs, the longer it takes for the consumption ratio to converge.

The longer it takes for the "less accurate" agent to learn, the lower its ultimate consumption share.

Now let's plot outcomes when nature follows g:

```
fig_g, axes_g = plot_learning_results(results_g, π_0_scenarios, 'g', 0.0)
plt.show()
```



Beliefs when nature = g
$\pi_0^1 = 0.3, \pi_0^2 = 0.7$
agent 1
agent 2
truth (g)
median π_i^t
t
Agent 1 consumption share (Nature = g)
median
consumption share
t
Beliefs when nature = g
$\pi_0^1 = 0.7, \pi_0^2 = 0.3$
agent 1
agent 2
truth (g)
median π_i^t
t
Agent 1 consumption share (Nature = g)
median
consumption share
t
Beliefs when nature = g
$\pi_0^1 = 0.1, \pi_0^2 = 0.9$
agent 1
agent 2
truth (g)
median π_i^t
t
Agent 1 consumption share (Nature = g)
median
consumption share
t

We observe symmetrical outcomes.

Exercise 23.11.3

In the previous exercise, we purposefully set the two beta distributions to be relatively close to each other.

That made it challenging to distinguish the distributions.

Now let's study outcomes when the distributions are further apart.

Let's set $f \sim \text{Beta}(2, 5)$ and $g \sim \text{Beta}(5, 2)$.

Please use the Python code you have written to study outcomes.

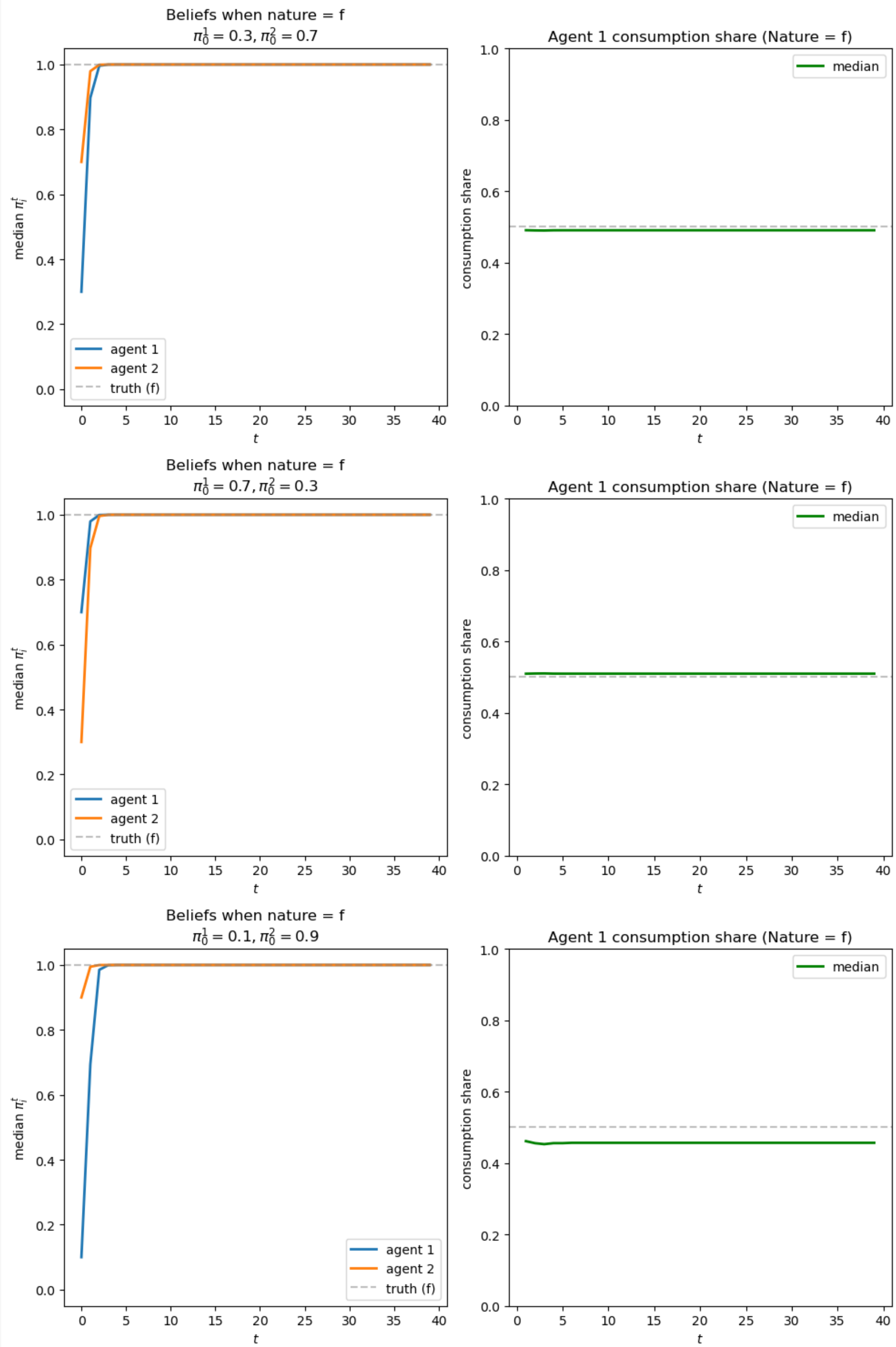
Solution

Here is one solution

```
λ = 0.5
T = 40
N = 1000

F_a, F_b = 2, 5
G_a, G_b = 5, 2

f = jit(lambda x: p(x, F_a, F_b))
g = jit(lambda x: p(x, G_a, G_b))

π_0_scenarios = [
    (0.3, 0.7),
    (0.7, 0.3),
    (0.1, 0.9),
]

s_seq_f = np.random.beta(F_a, F_b, (N, T))
s_seq_g = np.random.beta(G_a, G_b, (N, T))

results_f = {}
results_g = {}

for i, (π_0_1, π_0_2) in enumerate(π_0_scenarios):
    # When nature follows f
    results_f[i] = simulate_learning_blume_easley(
            s_seq_f, f, g, π_0_1, π_0_2, λ)
    # When nature follows g
    results_g[i] = simulate_learning_blume_easley(
            s_seq_g, f, g, π_0_1, π_0_2, λ)
```

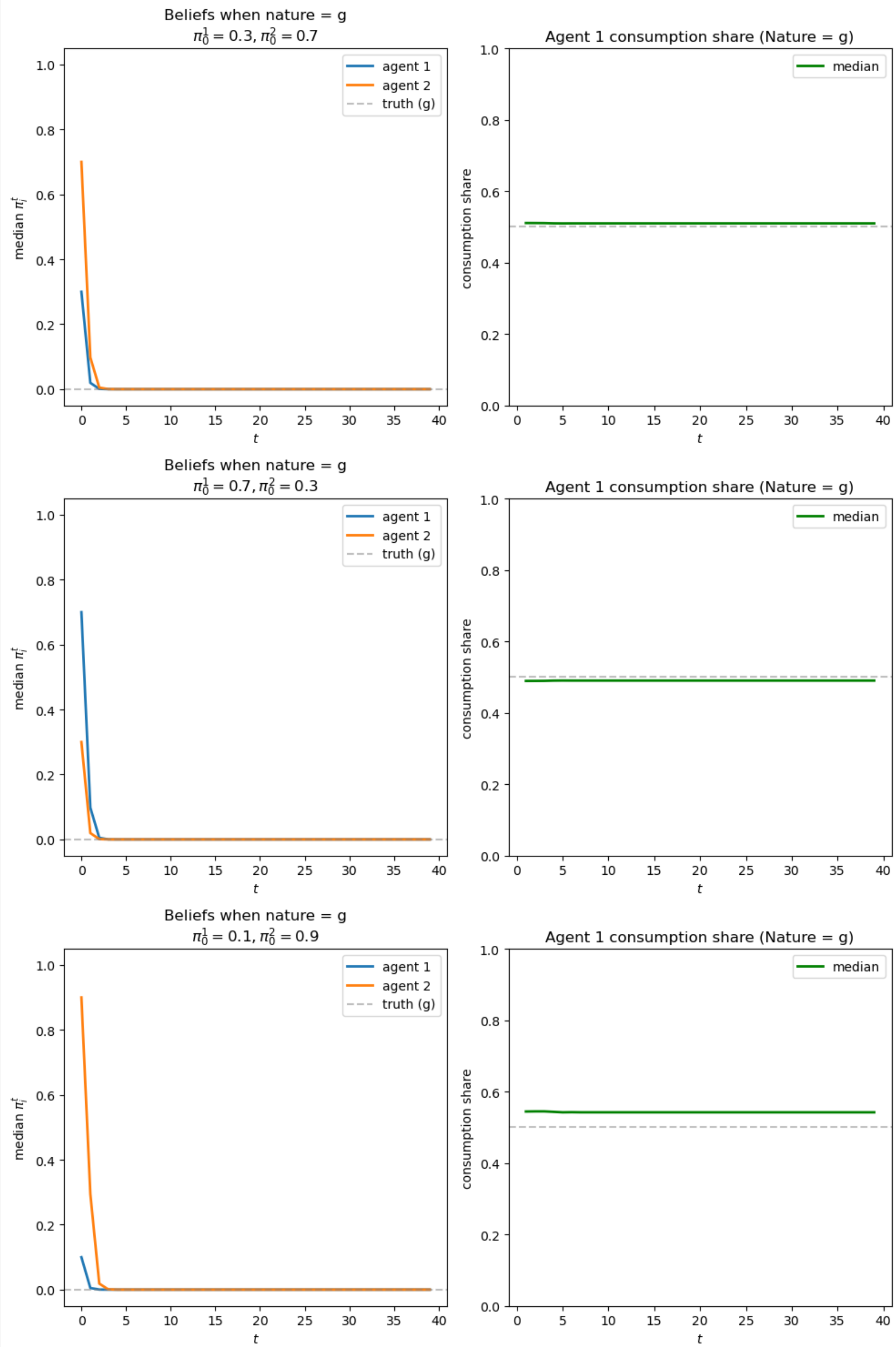
Now let's visualize the results

```
fig_f, axes_f = plot_learning_results(results_f, π_0_scenarios, 'f', 1.0)
plt.show()
```



Beliefs when nature = f
$\pi_0^1 = 0.3, \pi_0^2 = 0.7$
median π_i^t
agent 1
agent 2
truth (f)
t
Agent 1 consumption share (Nature = f)
consumption share
median
t
Beliefs when nature = f
$\pi_0^1 = 0.7, \pi_0^2 = 0.3$
median π_i^t
agent 1
agent 2
truth (f)
t
Agent 1 consumption share (Nature = f)
consumption share
median
t
Beliefs when nature = f
$\pi_0^1 = 0.1, \pi_0^2 = 0.9$
median π_i^t
agent 1
agent 2
truth (f)
t
Agent 1 consumption share (Nature = f)
consumption share
median
t

```
fig_g, axes_g = plot_learning_results(results_g, π_0_scenarios, 'g', 0.0)
plt.show()
```


Beliefs when nature = g
$\pi_0^1 = 0.3, \pi_0^2 = 0.7$
agent 1
agent 2
truth (g)
median π_i^t
t
Agent 1 consumption share (Nature = g)
median
consumption share
t
Beliefs when nature = g
$\pi_0^1 = 0.7, \pi_0^2 = 0.3$
agent 1
agent 2
truth (g)
median π_i^t
t
Agent 1 consumption share (Nature = g)
median
consumption share
t
Beliefs when nature = g
$\pi_0^1 = 0.1, \pi_0^2 = 0.9$
agent 1
agent 2
truth (g)
median π_i^t
t
Agent 1 consumption share (Nature = g)
median
consumption share
t

Evidently, because the two distributions are further apart, it is easier to distinguish them.

So learning occurs more quickly.

So do consumption shares.

Exercise 23.11.4

Two agents have different beliefs about three possible models.

Assume $f(x) \geq 0$, $g(x) \geq 0$, and $h(x) \geq 0$ for $x \in X$ with:

- $\int_X f(x)dx = 1$
- $\int_X g(x)dx = 1$
- $\int_X h(x)dx = 1$

We'll consider two agents:

- Agent 1: $\pi_0^g = 1 - \pi_0^f$, $\pi_0^f \in (0,1)$, $\pi_0^h = 0$ (attaches positive probability only to models f and g)
- Agent 2: $\pi_0^g = \pi_0^f = 1/3$, $\pi_0^h = 1/3$ (attaches equal weights to all three models)

Let f and g be two beta distributions with $f \sim \text{Beta}(3,2)$ and $g \sim \text{Beta}(2,3)$, and set $h = \pi_0^f f + (1 - \pi_0^f)g$ with $\pi_0^f = 0.5$.

Bayes' Law tells us that posterior probabilities on models f and g evolve according to

$$\pi^f(s^t) := \frac{\pi_0^f f(s^t)}{\pi_0^f f(s^t) + \pi_0^g g(s^t) + (1 - \pi_0^f - \pi_0^g)h(s^t)}$$

and

$$\pi^g(s^t) := \frac{\pi_0^g g(s^t)}{\pi_0^f f(s^t) + \pi_0^g g(s^t) + (1 - \pi_0^f - \pi_0^g)h(s^t)}$$

Please simulate and visualize evolutions of posterior probabilities and consumption allocations when:

- Nature permanently draws from f
- Nature permanently draws from g

Solution

Let's implement this three-model case with two agents having different beliefs.

Let's define f and g far apart, with h being a mixture of f and g.

```
F_a, F_b = 3, 2
G_a, G_b = 2, 3
λ = 0.5
π_f_0 = 0.5

f = jit(lambda x: p(x, F_a, F_b))
g = jit(lambda x: p(x, G_a, G_b))
h = jit(lambda x: π_f_0 * f(x) + (1 - π_f_0) * g(x))
```

Now we can define the belief updating for the model

```
@jit(parallel=True)
def compute_posterior_three_models(
    s_seq, f_func, g_func, h_func, π_f_0, π_g_0):
    """
    Compute posterior probabilities for three models.
    """
    N, T = s_seq.shape
    π_h_0 = 1 - π_f_0 - π_g_0

    π_f = np.zeros((N, T))
    π_g = np.zeros((N, T))
    π_h = np.zeros((N, T))

    for n in prange(N):
        # Initialize with priors
        π_f[n, 0] = π_f_0
        π_g[n, 0] = π_g_0
        π_h[n, 0] = π_h_0

        # Compute cumulative likelihoods
        f_cumul = 1.0
        g_cumul = 1.0
        h_cumul = 1.0

        for t in range(1, T):
            s_t = s_seq[n, t]

            # Update cumulative likelihoods
            f_cumul *= f_func(s_t)
            g_cumul *= g_func(s_t)
            h_cumul *= h_func(s_t)

            # Compute posteriors using Bayes' rule
            denominator = π_f_0 * f_cumul + π_g_0 * g_cumul + π_h_0 * h_cumul

            π_f[n, t] = π_f_0 * f_cumul / denominator
            π_g[n, t] = π_g_0 * g_cumul / denominator
            π_h[n, t] = π_h_0 * h_cumul / denominator

    return π_f, π_g, π_h
```

Let’s also write simulation code along the lines of earlier exercises

```
@jit
def bayesian_update_three_models(π_f_0, π_g_0, L_f, L_g, L_h):
    """Bayesian update for three models."""
    π_h_0 = 1 - π_f_0 - π_g_0
    denom = π_f_0 * L_f + π_g_0 * L_g + π_h_0 * L_h
    return π_f_0 * L_f / denom, π_g_0 * L_g / denom, π_h_0 * L_h / denom

@jit
def compute_mixture_density(π_f, π_g, π_h, f_val, g_val, h_val):
    """Compute mixture density for an agent."""
    return π_f * f_val + π_g * g_val + π_h * h_val

@jit(parallel=True)
def simulate_three_model_allocation(sequences, f_func, g_func, h_func,
                                    π_f_0_1, π_g_0_1, π_f_0_2, π_g_0_2, λ=0.5):
```

```
    """
    Simulate Blume-Easley model with learning agents and three models.
    """
    N, T = sequences.shape

    # Initialize arrays to store results
    beliefs_1 = {k: np.full((N, T), np.nan) for k in ['π_f', 'π_g', 'π_h']}
    beliefs_2 = {k: np.full((N, T), np.nan) for k in ['π_f', 'π_g', 'π_h']}
    c1_share = np.full((N, T), np.nan)
    l_agents_seq = np.full((N, T), np.nan)

    # Set initial beliefs
    beliefs_1['π_f'][:, 0] = π_f_0_1
    beliefs_1['π_g'][:, 0] = π_g_0_1
    beliefs_1['π_h'][:, 0] = 1 - π_f_0_1 - π_g_0_1
    beliefs_2['π_f'][:, 0] = π_f_0_2
    beliefs_2['π_g'][:, 0] = π_g_0_2
    beliefs_2['π_h'][:, 0] = 1 - π_f_0_2 - π_g_0_2

    for n in range(N):
        # Initialize cumulative likelihoods
        L_cumul = {'f': 1.0, 'g': 1.0, 'h': 1.0}
        l_agents_cumul = 1.0

        # Calculate initial consumption share at t=0
        l_agents_seq[n, 0] = 1.0
        c1_share[n, 0] = λ * 1.0 / (1 - λ + λ * 1.0)  # This equals λ

        for t in range(1, T):
            s_t = sequences[n, t]

            # Compute densities for current observation
            densities = {
                'f': f_func(s_t),
                'g': g_func(s_t),
                'h': h_func(s_t)
            }

            # Update cumulative likelihoods
            for model in L_cumul:
                L_cumul[model] *= densities[model]

            # Bayesian updates for both agents
            π_f_1, π_g_1, π_h_1 = bayesian_update_three_models(
                π_f_0_1, π_g_0_1, L_cumul['f'], L_cumul['g'], L_cumul['h'])
            π_f_2, π_g_2, π_h_2 = bayesian_update_three_models(
                π_f_0_2, π_g_0_2, L_cumul['f'], L_cumul['g'], L_cumul['h'])

            # Store beliefs
            beliefs_1['π_f'][n, t] = π_f_1
            beliefs_1['π_g'][n, t] = π_g_1
            beliefs_1['π_h'][n, t] = π_h_1
            beliefs_2['π_f'][n, t] = π_f_2
            beliefs_2['π_g'][n, t] = π_g_2
            beliefs_2['π_h'][n, t] = π_h_2

            # Compute mixture densities
```

```
            m1_t = compute_mixture_density(
                π_f_1, π_g_1, π_h_1, densities['f'],
                densities['g'], densities['h'])
            m2_t = compute_mixture_density(
                π_f_2, π_g_2, π_h_2, densities['f'],
                densities['g'], densities['h'])

            # Update cumulative likelihood ratio between agents
            l_agents_cumul *= (m1_t / m2_t)
            l_agents_seq[n, t] = l_agents_cumul

            # Consumption share for agent 1
            c1_share[n, t] = λ * l_agents_cumul / (1 - λ + λ * l_agents_cumul)

    return {
        'π_f_1': beliefs_1['π_f'],
        'π_g_1': beliefs_1['π_g'],
        'π_h_1': beliefs_1['π_h'],
        'π_f_2': beliefs_2['π_f'],
        'π_g_2': beliefs_2['π_g'],
        'π_h_2': beliefs_2['π_h'],
        'c1_share': c1_share,
        'l_agents': l_agents_seq
    }
```

The following code cell defines a plotting function to show evolutions of beliefs and consumption ratios

Now let's run the simulation.

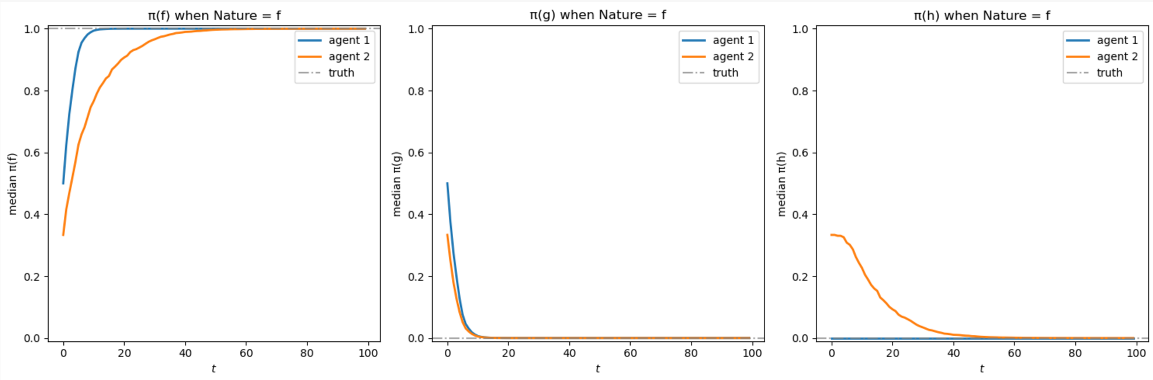
In the simulation below, agent 1 assigns positive probabilities only to f and g, while agent 2 puts equal weights on all three models.

```
T = 100
N = 1000

# Generate sequences for nature f and g
s_seq_f = np.random.beta(F_a, F_b, (N, T))
s_seq_g = np.random.beta(G_a, G_b, (N, T))

# Run simulations
results_f = simulate_three_model_allocation(s_seq_f,
                        f, g, h, π_f_0, 1-π_f_0,
                        1/3, 1/3, λ)
results_g = simulate_three_model_allocation(s_seq_g,
                        f, g, h, π_f_0, 1-π_f_0,
                        1/3, 1/3, λ)
```

Plots below show the evolution of beliefs for each model (f, g, h) separately.

First we show the figure when nature chooses f

```
plot_belief_evolution(results_f, nature='f', figsize=(15, 5))
plt.show()
```



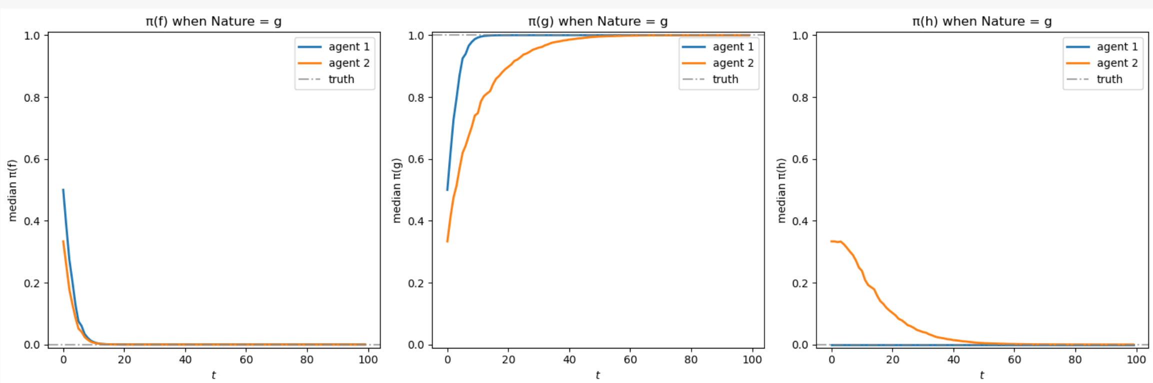
Agent 1's posterior beliefs are depicted in blue and agent 2's posterior beliefs are depicted in orange.

Evidently, when nature draws from f, agent 1 learns faster than agent 2, who, unlike agent 1, attaches a positive prior probability to model h:

- In the leftmost panel, both agents' beliefs for $\pi(f)$ converge toward 1 (the truth)
- Agent 2's belief in model h (rightmost panel) gradually converges to 0 after an initial rise

Now let's plot the belief evolution when nature chooses g:

```
plot_belief_evolution(results_g, nature='g', figsize=(15, 5))
plt.show()
```

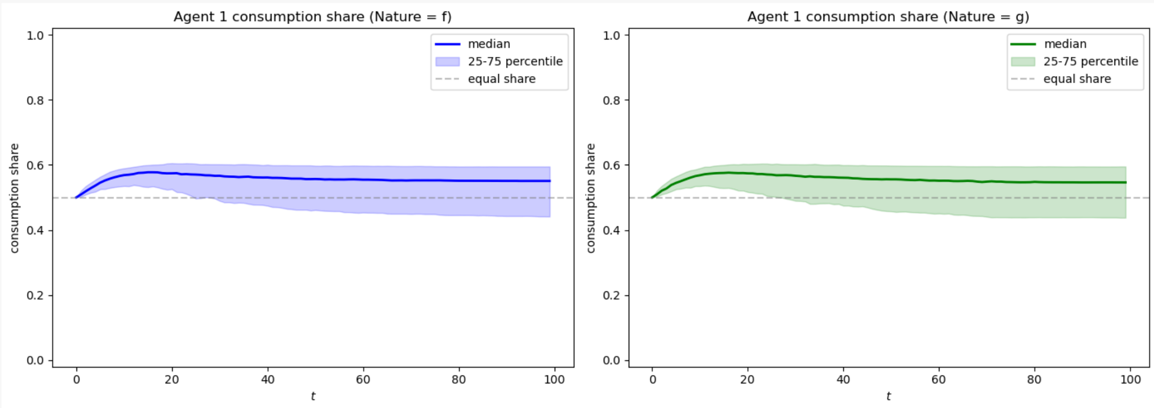


Again, agent 1 learns faster than agent 2.

Before reading the next figure, please guess how consumption shares evolve.

Remember that agent 1 reaches the correct model faster than agent 2

```
plot_consumption_dynamics(results_f, results_g, λ=0.5, figsize=(14, 5))
plt.show()
```



As we expected, agent 1 has a higher consumption share compared to agent 2.

In this exercise, the "truth" is among possible outcomes according to both agents.

Agent 2's model is "more general" because it allows a possibility – that nature is drawing from h – that agent 1's model does not include.

Agent 1 learns more quickly because he uses a simpler model.

Exercise 23.11.5

Now consider two agents with extreme priors about three models.

Consider the same setup as the previous exercise, but now:

- Agent 1: $\pi_0^g = \pi_0^f = \frac{\epsilon}{2} > 0$, where ϵ is close to 0 (e.g., $\epsilon = 0.01$)
- Agent 2: $\pi_0^g = \pi_0^f = 0$ (rigid belief in model h)

Choose h to be close but not equal to either f or g as measured by KL divergence. For example, set $h \sim \text{Beta}(1.2, 1.1)$ and $f \sim \text{Beta}(1, 1)$.

Please simulate and visualize evolutions of posterior probabilities and consumption allocations when:

- Nature permanently draws from f
- Nature permanently draws from g

Solution

To explore this exercise, we increase T to 1000.

Let's specify f, g, and h and verify that h and f are closer than h and g

```
F_a, F_b = 1, 1
G_a, G_b = 3, 1.2
H_a, H_b = 1.2, 1.1

f = jit(lambda x: p(x, F_a, F_b))
g = jit(lambda x: p(x, G_a, G_b))
h = jit(lambda x: p(x, H_a, H_b))

Kh_f = compute_KL(h, f)
```

```
Kh_g = compute_KL(h, g)
Kf_h = compute_KL(f, h)
Kg_h = compute_KL(g, h)

print(f"KL divergences:")
print(f"KL(h,f) = {Kh_f:.4f}, KL(h,g) = {Kh_g:.4f}")
print(f"KL(f,h) = {Kf_h:.4f}, KL(g,h) = {Kg_h:.4f}")
```

```
KL divergences:
KL(h,f) = 0.0092, KL(h,g) = 0.5514
KL(f,h) = 0.0105, KL(g,h) = 0.2919
```

Now we can set the belief models for the two agents

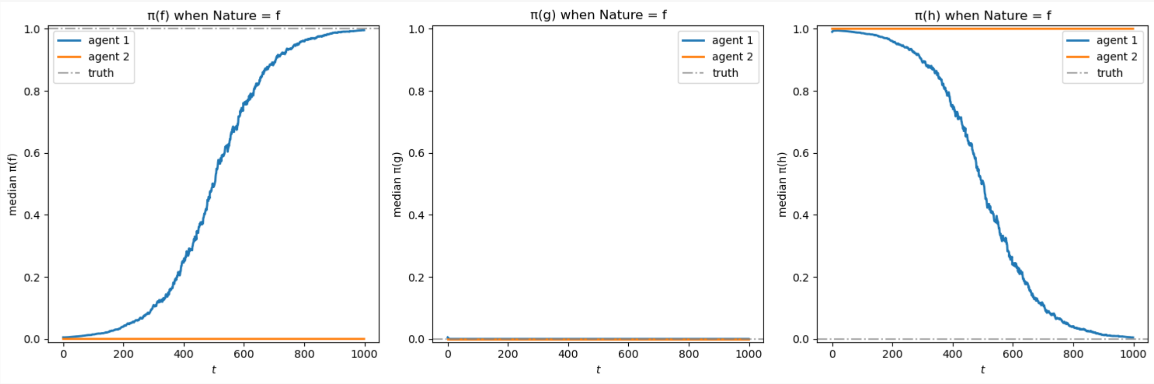
```
ε = 0.01
λ = 0.5

# Agent 1: π_f = ε/2, π_g = ε/2, π_h = 1-ε
# (almost rigid about h)
π_f_1 = ε/2
π_g_1 = ε/2

# Agent 2: π_f = 0, π_g = 0, π_h = 1
# (fully rigid about h)
π_f_2 = 1e-10
π_g_2 = 1e-10
```

Now we can run the simulation

```
T = 1000
N = 1000

# Generate sequences for different nature scenarios
s_seq_f = np.random.beta(F_a, F_b, (N, T))
s_seq_g = np.random.beta(G_a, G_b, (N, T))

# Run simulations for both scenarios
results_f = simulate_three_model_allocation(
                                s_seq_f,
                                f, g, h,
                                π_f_1, π_g_1, π_f_2, π_g_2, λ)
results_g = simulate_three_model_allocation(
                                s_seq_g,
                                f, g, h,
                                π_f_1, π_g_1, π_f_2, π_g_2, λ)
```

Let's plot the belief evolution when nature chooses f

```
plot_belief_evolution(results_f, nature='f', figsize=(15, 5))
plt.show()
```



Observe how slowly agent 1 learns the truth in the leftmost panel showing $\pi(f)$.

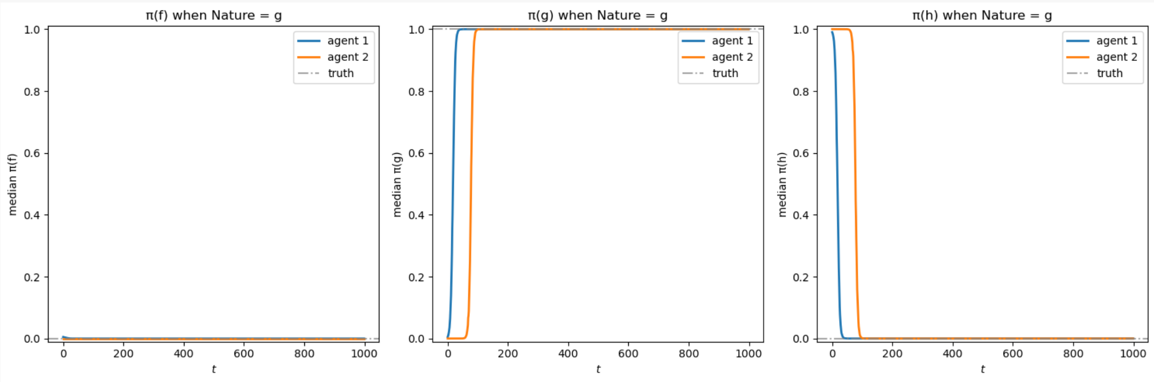
Also note that agent 2 is not updating.

This is because we have specified that f is very difficult to distinguish from h as measured by $KL(f, h)$.

The rigidity regarding h prevents agent 2 from updating its beliefs when observing a very similar model f

Now let's plot the belief evolution when nature chooses g

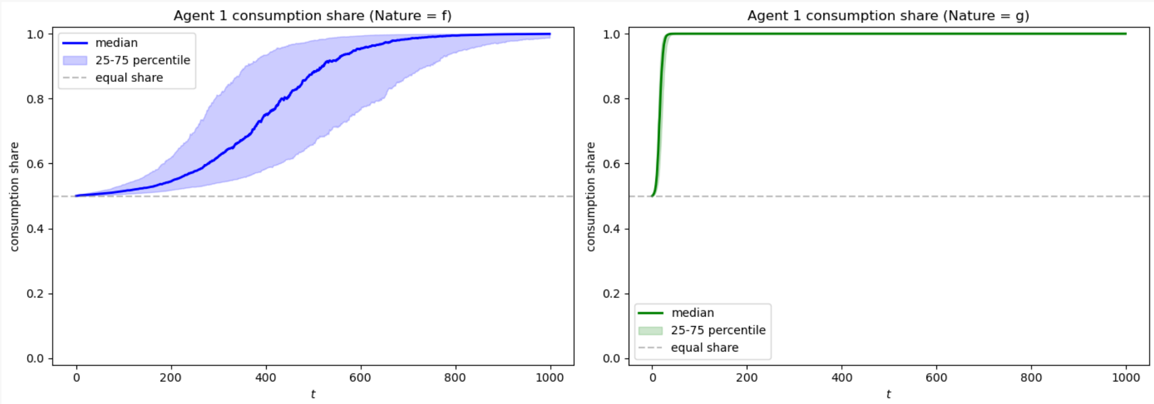
```
plot_belief_evolution(results_g, nature='g', figsize=(15, 5))
plt.show()
```



When nature draws from g, it is further away from h as measured by the KL divergence.

This helps both agents learn the truth more quickly.

```
plot_consumption_dynamics(results_f, results_g,
                              λ=0.5, figsize=(14, 5))
plt.show()
```



In the consumption dynamics plot, notice that agent 1's consumption share converges to 1 both when nature permanently draws from f and when nature permanently draws from g.

CHAPTER

TWENTYFOUR

LIKELIHOOD PROCESSES FOR VAR MODELS

Contents

24.1 Overview

This lecture extends our analysis of likelihood ratio processes to Vector Autoregressions (VARs).

We'll

- Construct likelihood functions for VAR models
- Form likelihood ratio processes for comparing two VAR models
- Visualize the evolution of likelihood ratios over time
- Connect VAR likelihood ratios to the Samuelson multiplier-accelerator model

Our analysis builds on concepts from:

- *Likelihood Ratio Processes*
- *Linear State Space Models*
- *Samuelson Multiplier-Accelerator*

Let's start by importing helpful libraries:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import linalg
from scipy.stats import multivariate_normal as mvn
```

(continues on next page)

(continued from previous page)

```
from quantecon import LinearStateSpace
import quantecon as qe
from numba import jit
from typing import NamedTuple, Optional, Tuple
from collections import namedtuple
```

24.2 VAR model setup

Consider a VAR model of the form:

$$x_{t+1} = Ax_t + Cw_{t+1}$$
$$x_0 \sim \mathcal{N}(\mu_0, \Sigma_0)$$

where:

- x_t is an $n \times 1$ state vector
- $w_{t+1} \sim \mathcal{N}(0, I)$ is an $m \times 1$ vector of shocks
- A is an $n \times n$ transition matrix
- C is an $n \times m$ volatility matrix

Let's define the necessary data structures for the VAR model

```
VARModel = namedtuple('VARModel', ['A', 'C', 'μ_0', 'Σ_0',
                                    'CC', 'CC_inv', 'log_det_CC',
                                    'Σ_0_inv', 'log_det_Σ_0'])
def compute_stationary_var(A, C):
    """
    Compute stationary mean and covariance for VAR model
    """
    n = A.shape[0]

    # Check stability
    eigenvalues = np.linalg.eigvals(A)
    if np.max(np.abs(eigenvalues)) >= 1:
        raise ValueError("VAR is not stationary")

    μ_0 = np.zeros(n)

    # Stationary covariance: solve discrete Lyapunov equation
    # Σ_0 = A @ Σ_0 @ A.T + C @ C.T
    CC = C @ C.T
    Σ_0 = linalg.solve_discrete_lyapunov(A, CC)

    return μ_0, Σ_0

def create_var_model(A, C, μ_0=None, Σ_0=None, stationary=True):
    """
    Create a VAR model with parameters and precomputed matrices
    """
    A = np.asarray(A)
    C = np.asarray(C)
    n = A.shape[0]
    CC = C @ C.T
```

(continues on next page)

(continued from previous page)

```
    if stationary:
        μ_0_comp, Σ_0_comp = compute_stationary_var(A, C)
    else:
        μ_0_comp = μ_0 if μ_0 is not None else np.zeros(n)
        Σ_0_comp = Σ_0 if Σ_0 is not None else np.eye(n)

    # Check if CC is singular
    det_CC = np.linalg.det(CC)
    if np.abs(det_CC) < 1e-10:
        # Use pseudo-inverse for singular case
        CC_inv = np.linalg.pinv(CC)
        CC_reg = CC + 1e-10 * np.eye(CC.shape[0])
        log_det_CC = np.log(np.linalg.det(CC_reg))
    else:
        CC_inv = np.linalg.inv(CC)
        log_det_CC = np.log(det_CC)

    # Same check for Σ_0
    det_Σ_0 = np.linalg.det(Σ_0_comp)
    if np.abs(det_Σ_0) < 1e-10:
        Σ_0_inv = np.linalg.pinv(Σ_0_comp)
        Σ_0_reg = Σ_0_comp + 1e-10 * np.eye(Σ_0_comp.shape[0])
        log_det_Σ_0 = np.log(np.linalg.det(Σ_0_reg))
    else:
        Σ_0_inv = np.linalg.inv(Σ_0_comp)
        log_det_Σ_0 = np.log(det_Σ_0)

    return VARModel(A=A, C=C, μ_0=μ_0_comp, Σ_0=Σ_0_comp,
                    CC=CC, CC_inv=CC_inv, log_det_CC=log_det_CC,
                    Σ_0_inv=Σ_0_inv, log_det_Σ_0=log_det_Σ_0)
```

24.2.1 Joint distribution

The joint probability distribution $f(x_T, x_{T-1}, \ldots, x_0)$ can be factored as:

$$f(x_T, \ldots, x_0) = f(x_T | x_{T-1}) f(x_{T-1} | x_{T-2}) \cdots f(x_1 | x_0) f(x_0)$$

Since the VAR is Markovian, $f(x_{t+1} | x_t, \ldots, x_0) = f(x_{t+1} | x_t)$.

24.2.2 Conditional densities

Given the Gaussian structure, the conditional distribution $f(x_{t+1} | x_t)$ is Gaussian with:

- Mean: Ax_t
- Covariance: CC'

The log conditional density is

$$\log f(x_{t+1} | x_t) = -\frac{n}{2} \log(2\pi) - \frac{1}{2} \log \det(CC') - \frac{1}{2} (x_{t+1} - Ax_t)' (CC')^{-1} (x_{t+1} - Ax_t) \tag{24.1}$$

```python
def log_likelihood_transition(x_next, x_curr, model):
    """
    Compute log likelihood of transition from x_curr to x_next
    """
    x_next = np.atleast_1d(x_next)
    x_curr = np.atleast_1d(x_curr)
    n = len(x_next)
    diff = x_next - model.A @ x_curr
    return -0.5 * (n * np.log(2 * np.pi) + model.log_det_CC +
                   diff @ model.CC_inv @ diff)
```

The log density of the initial state is:

$$\log f(x_0) = -\frac{n}{2}\log(2\pi) - \frac{1}{2}\log\det(\Sigma_0) - \frac{1}{2}(x_0 - \mu_0)'\Sigma_0^{-1}(x_0 - \mu_0)$$

```python
def log_likelihood_initial(x_0, model):
    """
    Compute log likelihood of initial state
    """
    x_0 = np.atleast_1d(x_0)
    n = len(x_0)
    diff = x_0 - model.μ_0
    return -0.5 * (n * np.log(2 * np.pi) + model.log_det_Σ_0 +
                   diff @ model.Σ_0_inv @ diff)
```

Now let's group the likelihood computations into a single function that computes the log likelihood of an entire path

```python
def log_likelihood_path(X, model):
    """
    Compute log likelihood of entire path
    """

    T = X.shape[0] - 1
    log_L = log_likelihood_initial(X[0], model)

    for t in range(T):
        log_L += log_likelihood_transition(X[t+1], X[t], model)

    return log_L

def simulate_var(model, T, N_paths=1):
    """
    Simulate paths from the VAR model
    """
    n = model.A.shape[0]
    m = model.C.shape[1]
    paths = np.zeros((N_paths, T+1, n))

    for i in range(N_paths):
        # Draw initial state
        x = mvn.rvs(mean=model.μ_0, cov=model.Σ_0)
        x = np.atleast_1d(x)
        paths[i, 0] = x

        # Simulate forward
        for t in range(T):
            w = np.random.randn(m)
```

(continues on next page)

(continued from previous page)

```
            x = model.A @ x + model.C @ w
            paths[i, t+1] = x

    return paths if N_paths > 1 else paths[0]
```

24.3 Likelihood ratio process

Now let's compute likelihood ratio processes for comparing two VAR models.

For a VAR model with state vector x_t, the log likelihood ratio at time t is

$$\ell_t = \log \frac{p_f(x_t|x_{t-1})}{p_g(x_t|x_{t-1})}$$

where p_f and p_g are the conditional densities under models f and g respectively.

The cumulative log likelihood ratio process is

$$L_t = \sum_{s=1}^{t} \ell_s = \sum_{s=1}^{t} \log \frac{p_f(x_s|x_{s-1})}{p_g(x_s|x_{s-1})}$$

where $p_f(x_t|x_{t-1})$ and $p_g(x_t|x_{t-1})$ are given by their respective conditional densities defined in (24.1).

Let's write those equations in Python

```
def compute_likelihood_ratio_var(paths, model_f, model_g):
    """
    Compute likelihood ratio process for VAR models
    """
    if paths.ndim == 2:
        paths = paths[np.newaxis, :]

    N_paths, T_plus_1, n = paths.shape
    T = T_plus_1 - 1
    log_L_ratios = np.zeros((N_paths, T+1))

    for i in range(N_paths):
        X = paths[i]

        # Initial log likelihood ratio
        log_L_f_0 = log_likelihood_initial(X[0], model_f)
        log_L_g_0 = log_likelihood_initial(X[0], model_g)
        log_L_ratios[i, 0] = log_L_f_0 - log_L_g_0

        # Recursive computation
        for t in range(1, T+1):
            log_L_f_t = log_likelihood_transition(X[t], X[t-1], model_f)
            log_L_g_t = log_likelihood_transition(X[t], X[t-1], model_g)

            # Update log likelihood ratio
            log_diff = log_L_f_t - log_L_g_t

            log_L_prev = log_L_ratios[i, t-1]
            log_L_new = log_L_prev + log_diff
            log_L_ratios[i, t] = log_L_new
```

(continues on next page)

(continued from previous page)

```
    return log_L_ratios if N_paths > 1 else log_L_ratios[0]
```

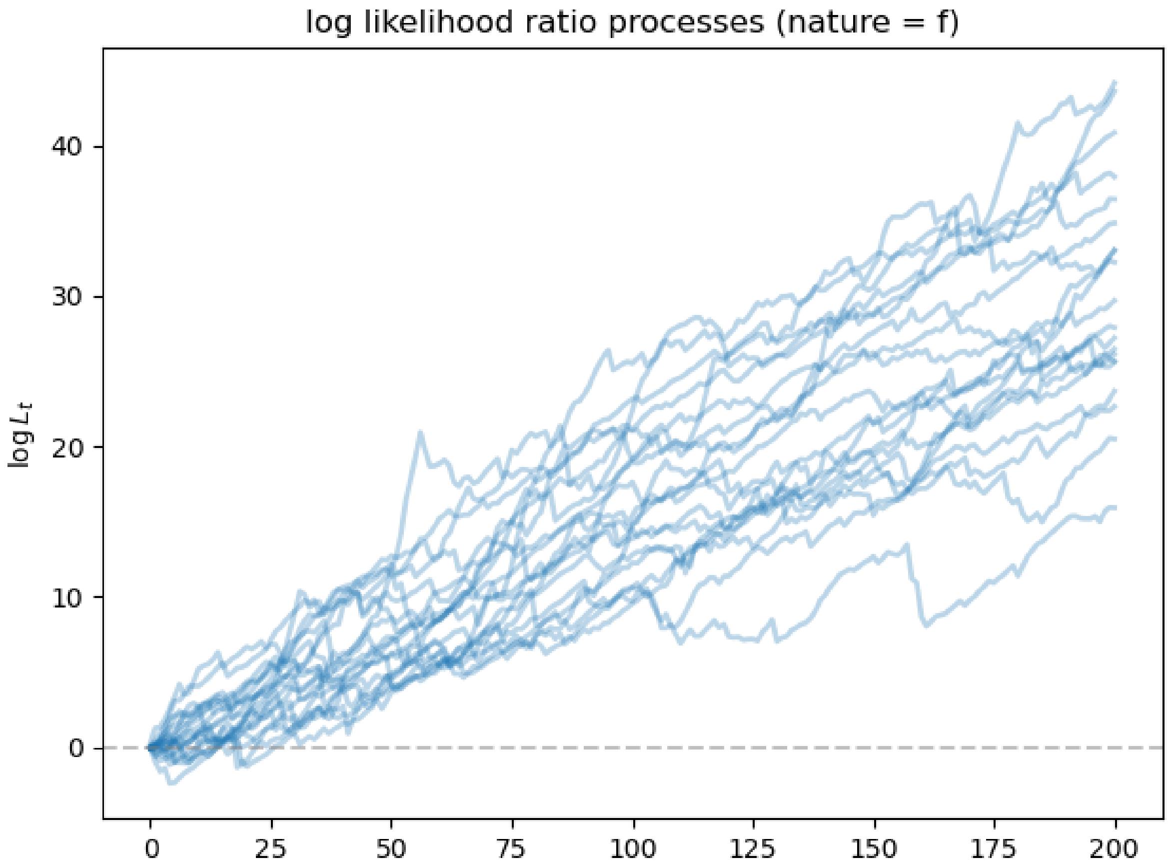
24.4 Example 1: two AR(1) processes

Let's start with a simple example comparing two univariate AR(1) processes with $A_f = 0.8$, $A_g = 0.5$, and $C_f = 0.3$, $C_g = 0.4$

```
# Model f: AR(1) with persistence ρ = 0.8
A_f = np.array([[0.8]])
C_f = np.array([[0.3]])

# Model g: AR(1) with persistence ρ = 0.5
A_g = np.array([[0.5]])
C_g = np.array([[0.4]])

# Create VAR models
model_f = create_var_model(A_f, C_f)
model_g = create_var_model(A_g, C_g)
```

Let's generate 100 paths of length 200 from model f and compute the likelihood ratio processes

```
# Simulate from model f
T = 200
N_paths = 100
paths_from_f = simulate_var(model_f, T, N_paths)

L_ratios_f = compute_likelihood_ratio_var(paths_from_f, model_f, model_g)

fig, ax = plt.subplots()

for i in range(min(20, N_paths)):
    ax.plot(L_ratios_f[i], alpha=0.3, color='C0', lw=2)

ax.axhline(y=0, color='gray', linestyle='--', alpha=0.5)
ax.set_ylabel(r'$\log L_t$')
ax.set_title('log likelihood ratio processes (nature = f)')

plt.tight_layout()
plt.show()
```



As we expected, the likelihood ratio processes goes to $+\infty$ as T increases, indicating that model f is chosen correctly by our algorithm.

24.5 Example 2: bivariate VAR models

Now let's consider an example with bivariate VAR models with

$$A_f = \begin{bmatrix} 0.7 & 0.2 \\ 0.1 & 0.6 \end{bmatrix}, \quad C_f = \begin{bmatrix} 0.3 & 0.1 \\ 0.1 & 0.3 \end{bmatrix}$$

and

$$A_g = \begin{bmatrix} 0.5 & 0.3 \\ 0.2 & 0.5 \end{bmatrix}, \quad C_g = \begin{bmatrix} 0.4 & 0.0 \\ 0.0 & 0.4 \end{bmatrix}$$

```
A_f = np.array([[0.7, 0.2],
                [0.1, 0.6]])

C_f = np.array([[0.3, 0.1],
                [0.1, 0.3]])

A_g = np.array([[0.5, 0.3],
                [0.2, 0.5]])

C_g = np.array([[0.4, 0.0],
```

(continues on next page)

(continued from previous page)

```
                  [0.0, 0.4]])

# Create VAR models
model2_f = create_var_model(A_f, C_f)
model2_g = create_var_model(A_g, C_g)

# Check stationarity
print("model f eigenvalues:", np.linalg.eigvals(A_f))
print("model g eigenvalues:", np.linalg.eigvals(A_g))
```

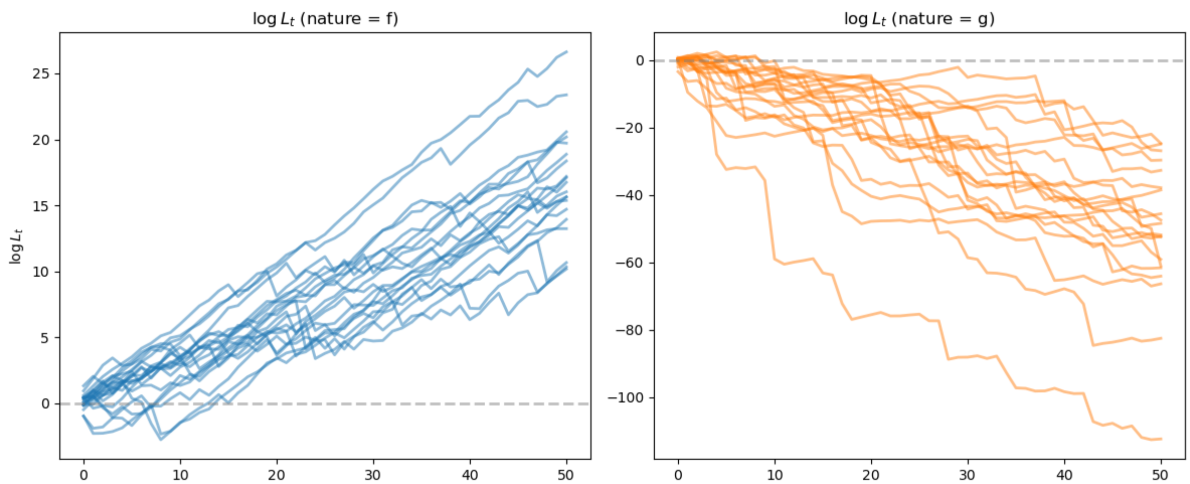
```
model f eigenvalues: [0.8 0.5]
model g eigenvalues: [0.74494897 0.25505103]
```

Let's generate 50 paths of length 50 from both models and compute the likelihood ratio processes

```
# Simulate from both models
T = 50
N_paths = 50

paths_from_f = simulate_var(model2_f, T, N_paths)
paths_from_g = simulate_var(model2_g, T, N_paths)

# Compute likelihood ratios
L_ratios_ff = compute_likelihood_ratio_var(paths_from_f, model2_f, model2_g)
L_ratios_gf = compute_likelihood_ratio_var(paths_from_g, model2_f, model2_g)
```

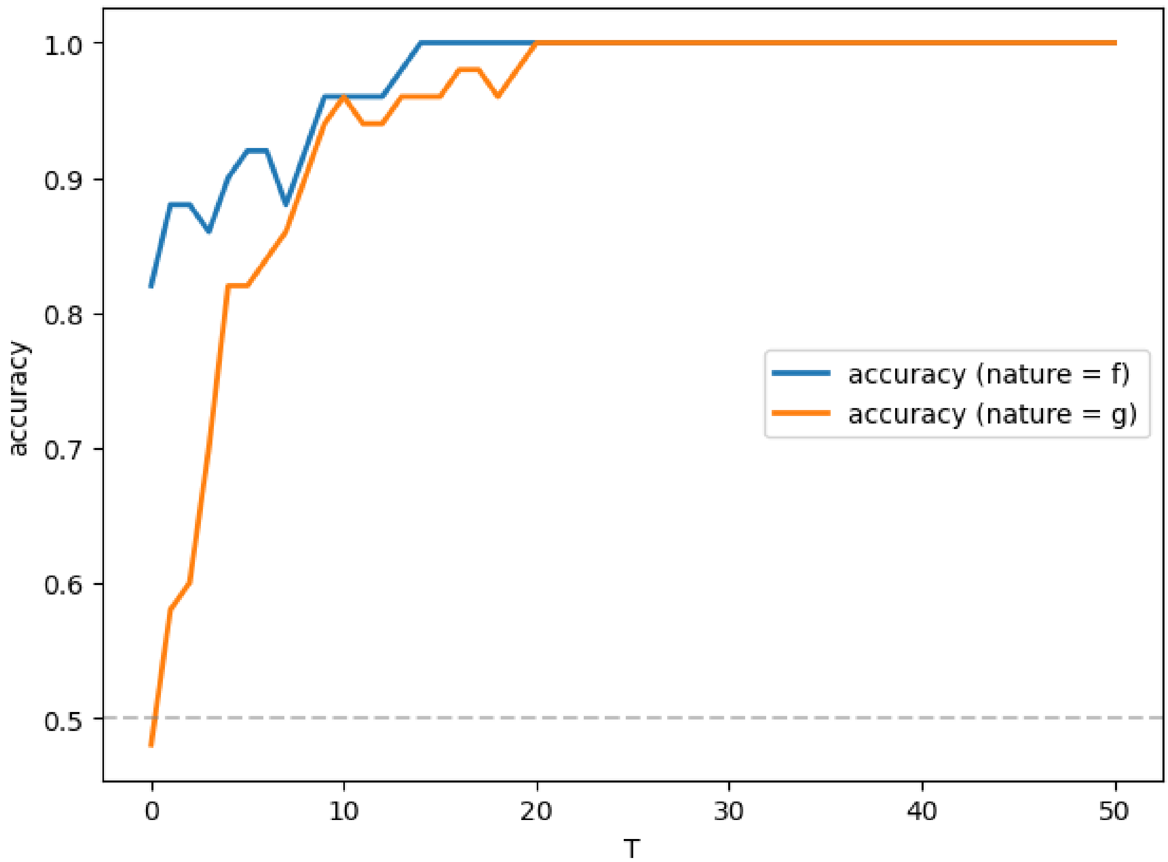
We can see that for paths generated from model f, the likelihood ratio processes tend to go to $+\infty$, while for paths from model g, they tend to go to $-\infty$.

```
# Visualize the results
fig, axes = plt.subplots(1, 2, figsize=(12, 5))

ax = axes[0]
for i in range(min(20, N_paths)):
    ax.plot(L_ratios_ff[i], alpha=0.5, color='C0', lw=2)
ax.axhline(y=0, color='gray', linestyle='--', alpha=0.5, lw=2)
ax.set_title(r'$\log L_t$ (nature = f)')
ax.set_ylabel(r'$\log L_t$')

ax = axes[1]
for i in range(min(20, N_paths)):
    ax.plot(L_ratios_gf[i], alpha=0.5, color='C1', lw=2)
ax.axhline(y=0, color='gray', linestyle='--', alpha=0.5, lw=2)
ax.set_title(r'$\log L_t$ (nature = g)')
plt.tight_layout()
plt.show()
```



Let's apply a Neyman-Pearson frequentist decision rule described in *Likelihood Ratio Processes* that selects model f when $\log L_T \geq 0$ and model g when $\log L_T < 0$

```
fig, ax = plt.subplots()
T_values = np.arange(0, T+1)
accuracy_f = np.zeros(len(T_values))
accuracy_g = np.zeros(len(T_values))

for i, t in enumerate(T_values):
    # Correct selection when data from f
    accuracy_f[i] = np.mean(L_ratios_ff[:, t] > 0)
    # Correct selection when data from g
    accuracy_g[i] = np.mean(L_ratios_gf[:, t] < 0)

ax.plot(T_values, accuracy_f, 'C0', linewidth=2, label='accuracy (nature = f)')
ax.plot(T_values, accuracy_g, 'C1', linewidth=2, label='accuracy (nature = g)')
ax.axhline(y=0.5, color='gray', linestyle='--', alpha=0.5)
ax.set_xlabel('T')
ax.set_ylabel('accuracy')
ax.legend()

plt.tight_layout()
plt.show()
```



Evidently, the accuracy approaches 1 as T increases, and it does so very quickly.

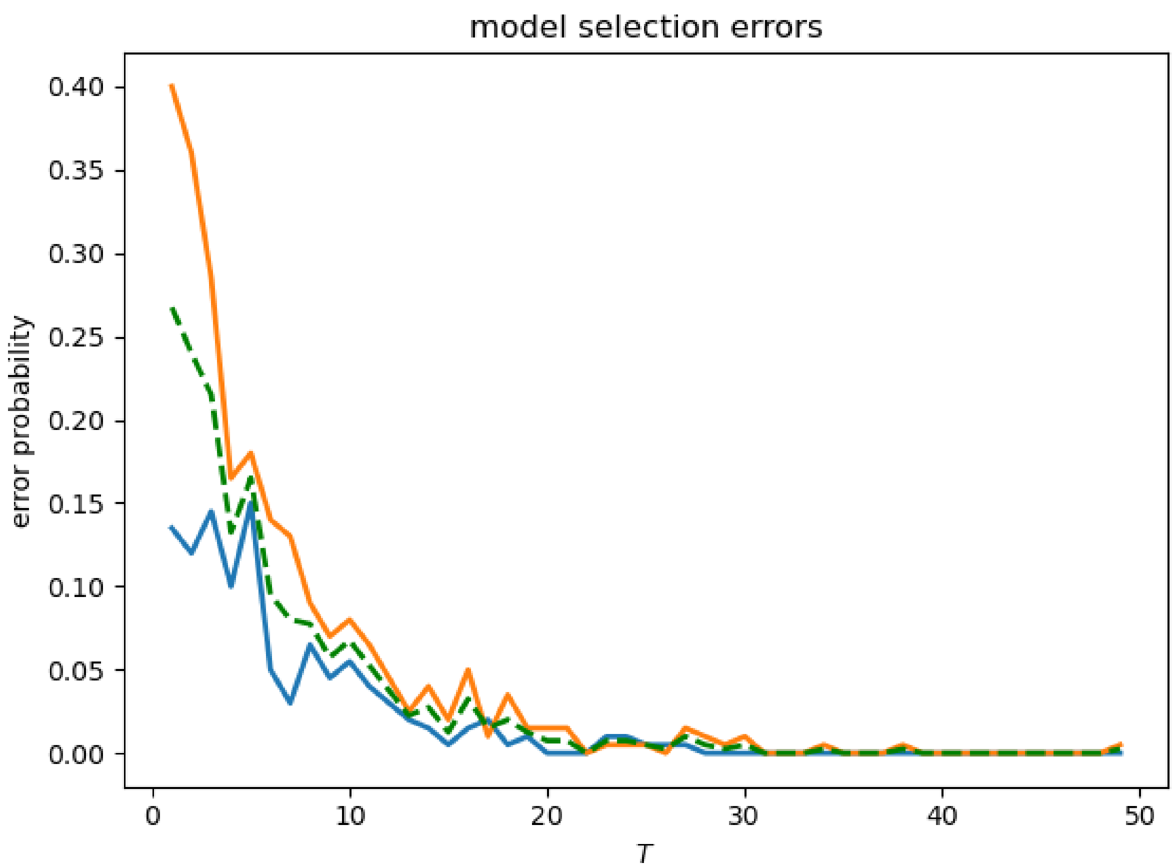
Let's also check the type I and type II errors as functions of T

```
def model_selection_analysis(T_values, model_f, model_g, N_sim=500):
    """
    Analyze model selection performance for different sample sizes
    """
    errors_f = []  # Type I errors
    errors_g = []  # Type II errors

    for T in T_values:
        # Simulate from model f
        paths_f = simulate_var(model_f, T, N_sim//2)
        L_ratios_f = compute_likelihood_ratio_var(paths_f, model_f, model_g)

        # Simulate from model g
        paths_g = simulate_var(model_g, T, N_sim//2)
        L_ratios_g = compute_likelihood_ratio_var(paths_g, model_f, model_g)

        # Decision rule: choose f if log L_T >= 0
        errors_f.append(np.mean(L_ratios_f[:, -1] < 0))
        errors_g.append(np.mean(L_ratios_g[:, -1] >= 0))

    return np.array(errors_f), np.array(errors_g)

T_values = np.arange(1, 50, 1)
```

(continues on next page)

(continued from previous page)

```
errors_f, errors_g = model_selection_analysis(T_values, model2_f, model2_g, N_sim=400)

fig, ax = plt.subplots()

ax.plot(T_values, errors_f, 'C0', linewidth=2, label='type I error')
ax.plot(T_values, errors_g, 'C1', linewidth=2, label='type II error')
ax.plot(T_values, 0.5 * (errors_f + errors_g), 'g--',
linewidth=2, label='average error')
ax.set_xlabel('$T$')
ax.set_ylabel('error probability')
ax.set_title('model selection errors')
plt.tight_layout()
plt.show()
```



24.6 Application: Samuelson multiplier-accelerator

Now let's connect to the Samuelson multiplier-accelerator model.

The model consists of:

- Consumption: $C_t = \gamma + aY_{t-1}$ where $a \in (0, 1)$ is the marginal propensity to consume
- Investment: $I_t = b(Y_{t-1} - Y_{t-2})$ where $b > 0$ is the accelerator coefficient
- Government spending: $G_t = G$ (constant)

We have the national income identity

$$Y_t = C_t + I_t + G_t$$

Equations yields the second-order difference equation:

$$Y_t = (\gamma + G) + (a + b)Y_{t-1} - bY_{t-2} + \sigma\epsilon_t$$

With $\rho_1 = a + b$ and $\rho_2 = -b$, we have:

$$Y_t = (\gamma + G) + \rho_1 Y_{t-1} + \rho_2 Y_{t-2} + \sigma\epsilon_t$$

To fit into our discussion, we write it into state-space representation.

To handle the constant term properly, we use an augmented state vector $\mathbf{x}_t = [1, Y_t, Y_{t-1}]'$:

$$\mathbf{x}_{t+1} = \begin{bmatrix} 1 \\ Y_{t+1} \\ Y_t \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ \gamma + G & \rho_1 & \rho_2 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ Y_t \\ Y_{t-1} \end{bmatrix} + \begin{bmatrix} 0 \\ \sigma \\ 0 \end{bmatrix} \epsilon_{t+1}$$

The observation equation extracts the economic variables:

$$\mathbf{y}_t = \begin{bmatrix} Y_t \\ C_t \\ I_t \end{bmatrix} = \begin{bmatrix} \gamma + G & \rho_1 & \rho_2 \\ \gamma & a & 0 \\ 0 & b & -b \end{bmatrix} \begin{bmatrix} 1 \\ Y_t \\ Y_{t-1} \end{bmatrix}$$

This gives us:

- $Y_t = (\gamma + G) \cdot 1 + \rho_1 Y_{t-1} + \rho_2 Y_{t-2}$ (total output)
- $C_t = \gamma \cdot 1 + aY_{t-1}$ (consumption)
- $I_t = b(Y_{t-1} - Y_{t-2})$ (investment)

```python
def samuelson_to_var(a, b, γ, G, σ):
    """
    Convert Samuelson model parameters to VAR form with augmented state

    Samuelson model:
    - Y_t = C_t + I_t + G
    - C_t = γ + a*Y_{t-1}
    - I_t = b*(Y_{t-1} - Y_{t-2})

    Reduced form: Y_t = (γ+G) + (a+b)*Y_{t-1} - b*Y_{t-2} + σ*ε_t

    State vector is [1, Y_t, Y_{t-1}]'
    """
    ρ_1 = a + b
    ρ_2 = -b

    # State transition matrix for augmented state
    A = np.array([[1,      0,     0],
                  [γ + G,  ρ_1,   ρ_2],
                  [0,      1,     0]])

    # Shock loading matrix
    C = np.array([[0],
                  [σ],
                  [0]])
```

(continues on next page)

(continued from previous page)

```
    # Observation matrix (extracts Y_t, C_t, I_t)
    G_obs = np.array([[γ + G,   ρ_1,   ρ_2],    # Y_t
                      [γ,       a,     0],      # C_t
                      [0,       b,    -b]])     # I_t

    return A, C, G_obs
```

We define functions in the code cell below to get the initial conditions and check stability

Let's implement it and inspect the likelihood ratio processes induced by two Samuelson models with different parameters.

```
def create_samuelson_var_model(a, b, γ, G, σ, stationary_init=False,
                               y_0=None, y_m1=None):
    """
    Create a VAR model from Samuelson parameters
    """
    A, C, G_obs = samuelson_to_var(a, b, γ, G, σ)

    μ_0, Σ_0 = get_samuelson_initial_conditions(
        a, b, γ, G, y_0, y_m1, stationary_init
    )

    # Create VAR model
    model = create_var_model(A, C, μ_0, Σ_0, stationary=False)
    is_stable, roots, max_root, dynamics = check_samuelson_stability(a, b)
    info = {
        'a': a, 'b': b, 'γ': γ, 'G': G, 'σ': σ,
        'ρ_1': a + b, 'ρ_2': -b,
        'steady_state': (γ + G) / (1 - a - b),
        'is_stable': is_stable,
        'roots': roots,
        'max_abs_root': max_root,
        'dynamics': dynamics
    }

    return model, G_obs, info

def simulate_samuelson(model, G_obs, T, N_paths=1):
    """
    Simulate Samuelson model
    """
    # Simulate state paths
    states = simulate_var(model, T, N_paths)

    # Extract observables using G matrix
    if N_paths == 1:
        # Single path: states is (T+1, 3)
        observables = (G_obs @ states.T).T
    else:
        # Multiple paths: states is (N_paths, T+1, 3)
        observables = np.zeros((N_paths, T+1, 3))
        for i in range(N_paths):
            observables[i] = (G_obs @ states[i].T).T

    return states, observables
```

Now let's simulate two Samuelson models with different accelerator coefficients and plot their sample paths

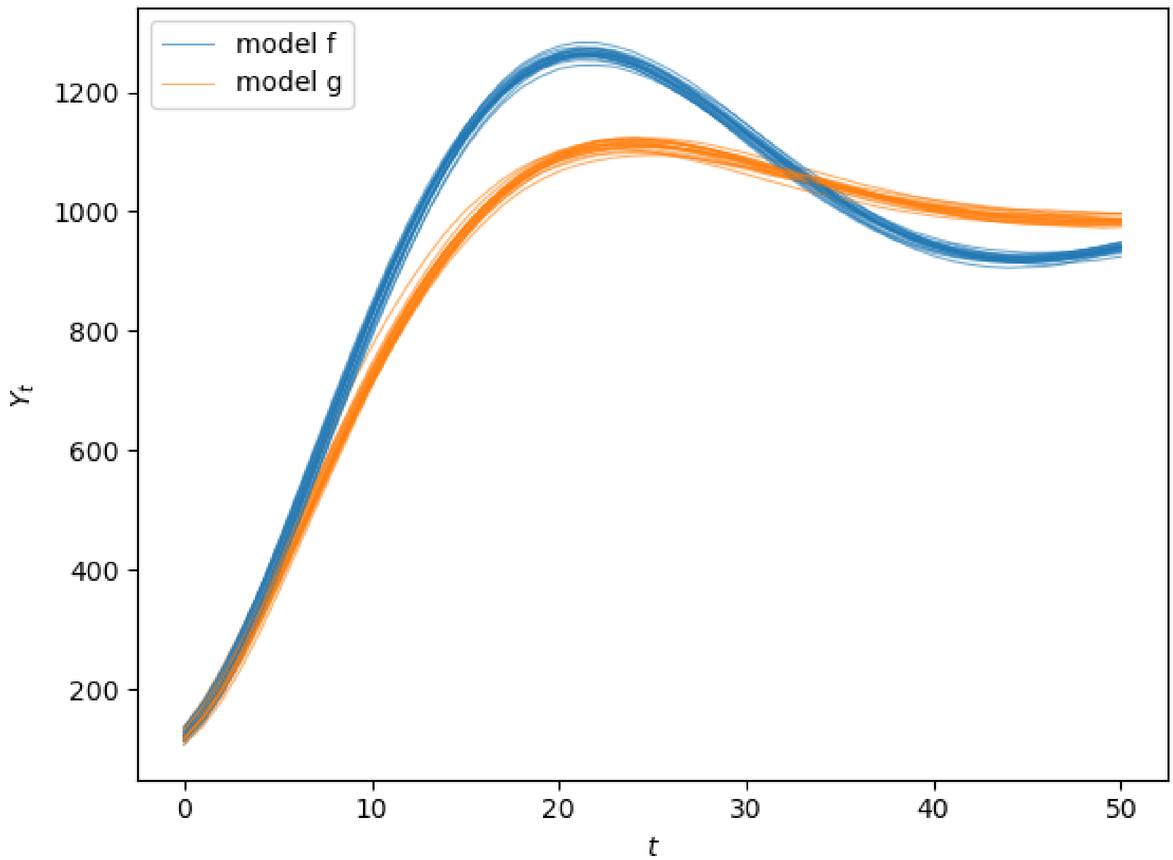
```
# Model f: Higher accelerator coefficient
a_f, b_f = 0.98, 0.9
γ_f, G_f, σ_f = 10, 10, 0.5

# Model g: Lower accelerator coefficient
a_g, b_g = 0.98, 0.85
γ_g, G_g, σ_g = 10, 10, 0.5


model_sam_f, G_obs_f, info_f = create_samuelson_var_model(
    a_f, b_f, γ_f, G_f, σ_f,
    stationary_init=False,
    y_0=100, y_m1=95
)

model_sam_g, G_obs_g, info_g = create_samuelson_var_model(
    a_g, b_g, γ_g, G_g, σ_g,
    stationary_init=False,
    y_0=100, y_m1=95
)

T = 50
N_paths = 50

# Get both states and observables
states_f, obs_f = simulate_samuelson(model_sam_f, G_obs_f, T, N_paths)
states_g, obs_g = simulate_samuelson(model_sam_g, G_obs_g, T, N_paths)

output_paths_f = obs_f[:, :, 0]
output_paths_g = obs_g[:, :, 0]

print("model f:")
print(f"  ρ_1 = a + b = {info_f['ρ_1']:.2f}")
print(f"  ρ_2 = -b = {info_f['ρ_2']:.2f}")
print(f"  roots: {info_f['roots']}")
print(f"  dynamics: {info_f['dynamics']}")

print("\nmodel g:")
print(f"  ρ_1 = a + b = {info_g['ρ_1']:.2f}")
print(f"  ρ_2 = -b = {info_g['ρ_2']:.2f}")
print(f"  roots: {info_g['roots']}")
print(f"  dynamics: {info_g['dynamics']}")


fig, ax = plt.subplots(1, 1)

for i in range(min(20, N_paths)):
    ax.plot(output_paths_f[i], alpha=0.6, color='C0', linewidth=0.8)
    ax.plot(output_paths_g[i], alpha=0.6, color='C1', linewidth=0.8)
ax.set_xlabel('$t$')
ax.set_ylabel('$Y_t$')
ax.legend(['model f', 'model g'], loc='upper left')
plt.tight_layout()
plt.show()
```

```
model f:
  ρ_1 = a + b = 1.88
```

(continues on next page)

(continued from previous page)

```
  ρ_2 = -b = -0.90
  roots: [0.94+0.12806248j 0.94-0.12806248j]
  dynamics: Damped oscillations

model g:
  ρ_1 = a + b = 1.83
  ρ_2 = -b = -0.85
  roots: [0.915+0.11302655j 0.915-0.11302655j]
  dynamics: Damped oscillations
```

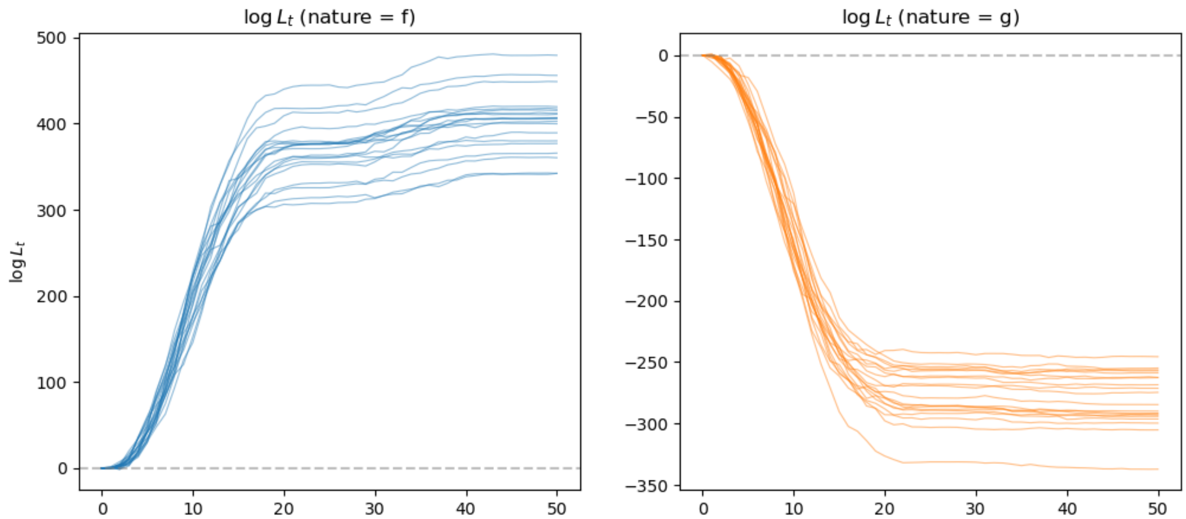



```
# Compute likelihood ratios
L_ratios_ff = compute_likelihood_ratio_var(states_f, model_sam_f, model_sam_g)
L_ratios_gf = compute_likelihood_ratio_var(states_g, model_sam_f, model_sam_g)

fig, axes = plt.subplots(1, 2, figsize=(12, 5))

ax = axes[0]
for i in range(min(20, N_paths)):
    ax.plot(L_ratios_ff[i], alpha=0.5, color='C0', lw=0.8)
ax.axhline(y=0, color='gray', linestyle='--', alpha=0.5)
ax.set_title(r'$\log L_t$ (nature = f)')
ax.set_ylabel(r'$\log L_t$')

ax = axes[1]
for i in range(min(20, N_paths)):
```

(continues on next page)

(continued from previous page)

```
    ax.plot(L_ratios_gf[i], alpha=0.5, color='C1', lw=0.8)
ax.axhline(y=0, color='gray', linestyle='--', alpha=0.5)
ax.set_title(r'$\log L_t$ (nature = g)')
plt.show()
```



In the figure on the left, data are generated by f and the likelihood ratio diverges to plus infinity.

In the figure on the right, data are generated by g and the likelihood ratio diverges to negative infinity.

In both cases, we applied a lower and upper threshold for the log likelihood ratio process for numerical stability since they grow unbounded very quickly.

In both cases, the likelihood ratio processes eventually lead us to select the correct model.

CHAPTER

TWENTYFIVE

MEAN OF A LIKELIHOOD RATIO PROCESS

Contents

25.1 Overview

In *this lecture* we described a peculiar property of a likelihood ratio process, namely, that its mean equals one for all $t \geq 0$ despite it's converging to zero almost surely.

While it is easy to verify that peculiar properly analytically (i.e., in population), it is challenging to use a computer simulation to verify it via an application of a law of large numbers that entails studying sample averages of repeated simulations.

To confront this challenge, this lecture puts **importance sampling** to work to accelerate convergence of sample averages to population means.

We use importance sampling to estimate the mean of a cumulative likelihood ratio $L\left(\omega^t\right) = \prod_{i=1}^t \ell\left(\omega_i\right)$.

We start by importing some Python packages.

```
import numpy as np
from numba import jit, vectorize, prange
import matplotlib.pyplot as plt
from math import gamma
```

25.2 Mathematical expectation of likelihood ratio

In *this lecture*, we studied a likelihood ratio $\ell\left(\omega_t\right)$

$$\ell\left(\omega_t\right)=\frac{f\left(\omega_t\right)}{g\left(\omega_t\right)}$$

where f and g are densities for Beta distributions with parameters F_a, F_b, G_a, G_b.

Assume that an i.i.d. random variable $\omega_t \in \Omega$ is generated by g.

The **cumulative likelihood ratio** $L\left(\omega^t\right)$ is

$$L\left(\omega^t\right)=\prod_{i=1}^t \ell\left(\omega_i\right)$$

Our goal is to approximate the mathematical expectation $E\left[L\left(\omega^t\right)\right]$ well.

In *this lecture*, we showed that $E\left[L\left(\omega^t\right)\right]$ equals 1 for all t.

We want to check out how well this holds if we replace E by with sample averages from simulations.

This turns out to be easier said than done because for Beta distributions assumed above, $L\left(\omega^t\right)$ has a very skewed distribution with a very long tail as $t \rightarrow \infty$.

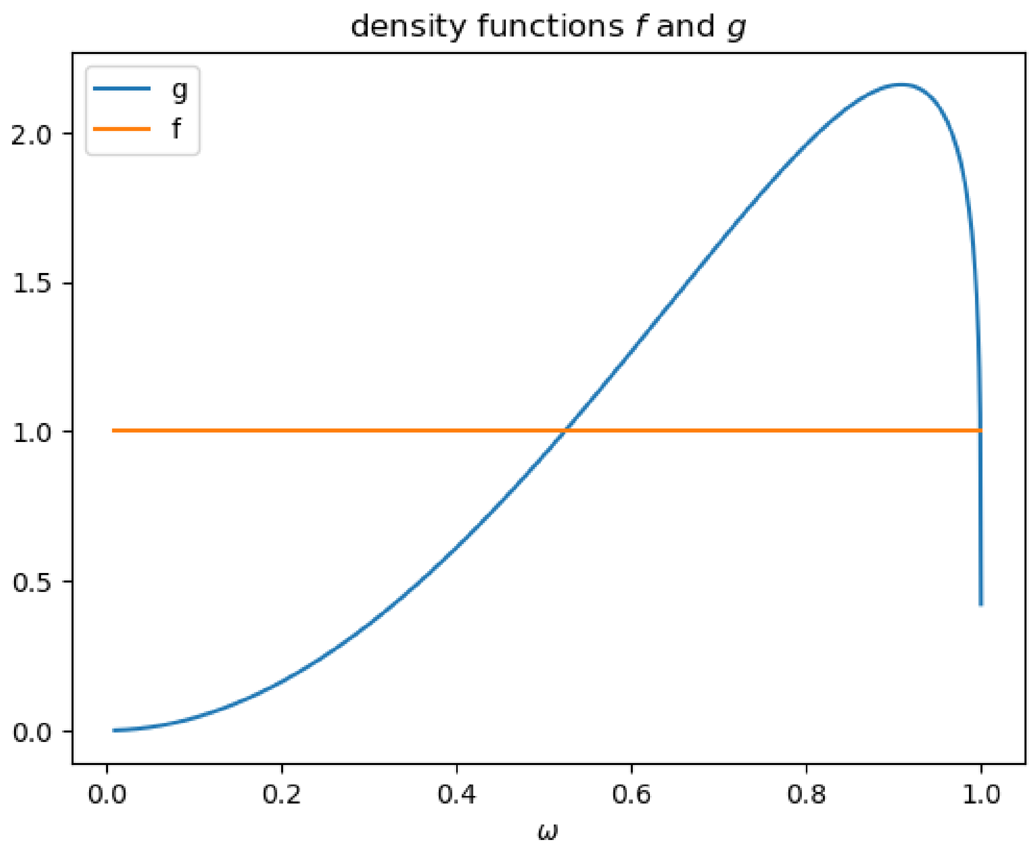
This property makes it difficult efficiently and accurately to estimate the mean by standard Monte Carlo simulation methods.

In this lecture we explore how a standard Monte Carlo method fails and how **importance sampling** provides a more computationally efficient way to approximate the mean of the cumulative likelihood ratio.

We first take a look at the density functions `f` and `g` .

```python
# Parameters in the two beta distributions.
F_a, F_b = 1, 1
G_a, G_b = 3, 1.2

@vectorize
def p(w, a, b):
    r = gamma(a + b) / (gamma(a) * gamma(b))
    return r * w ** (a-1) * (1 - w) ** (b-1)

# The two density functions.
f = jit(lambda w: p(w, F_a, F_b))
g = jit(lambda w: p(w, G_a, G_b))
```

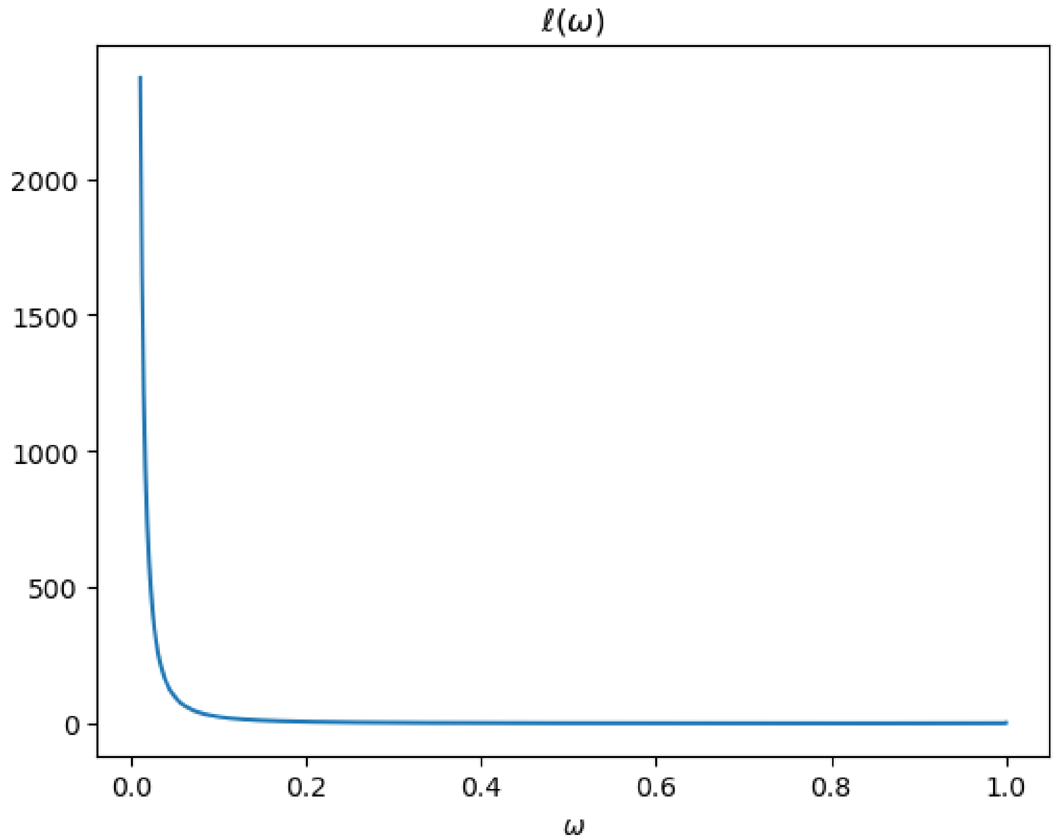
```python
w_range = np.linspace(1e-2, 1-1e-5, 1000)

plt.plot(w_range, g(w_range), label='g')
plt.plot(w_range, f(w_range), label='f')
plt.xlabel(r'$\omega$')
plt.legend()
plt.title('density functions $f$ and $g$')
plt.show()
```



The likelihood ratio is `l(w)=f(w)/g(w)`.

```
l = jit(lambda w: f(w) / g(w))
```

```
plt.plot(w_range, l(w_range))
plt.title(r'$\ell(\omega)$')
plt.xlabel(r'$\omega$')
plt.show()
```



The above plots shows that as $\omega \to 0$, $f(\omega)$ is unchanged and $g(\omega) \to 0$, so the likelihood ratio approaches infinity.

A Monte Carlo approximation of $\hat{E}[L(\omega^t)] = \hat{E}\left[\prod_{i=1}^{t} \ell(\omega_i)\right]$ would repeatedly draw ω from g, calculate the likelihood ratio $\ell(\omega) = \frac{f(\omega)}{g(\omega)}$ for each draw, then average these over all draws.

Because $g(\omega) \to 0$ as $\omega \to 0$, such a simulation procedure undersamples a part of the sample space $[0, 1]$ that it is important to visit often in order to do a good job of approximating the mathematical expectation of the likelihood ratio $\ell(\omega)$.

We illustrate this numerically below.

25.3 Importance sampling

We circumvent the issue by using a *change of distribution* called **importance sampling**.

Instead of drawing from g to generate data during the simulation, we use an alternative distribution h to generate draws of ω.

The idea is to design h so that it oversamples the region of Ω where $\ell(\omega_t)$ has large values but low density under g.

After we construct a sample in this way, we must then weight each realization by the likelihood ratio of g and h when we compute the empirical mean of the likelihood ratio.

By doing this, we properly account for the fact that we are using h and not g to simulate data.

To illustrate, suppose were interested in $E[\ell(\omega)]$.

We could simply compute:

$$\hat{E}^g\left[\ell\left(\omega\right)\right] = \frac{1}{N}\sum_{i=1}^{N}\ell(w_i^g)$$

where ω_i^g indicates that ω_i is drawn from g.

But using our insight from importance sampling, we could instead calculate the object:

$$\hat{E}^h\left[\ell\left(\omega\right)\frac{g(w)}{h(w)}\right] = \frac{1}{N}\sum_{i=1}^{N}\ell(w_i^h)\frac{g(w_i^h)}{h(w_i^h)}$$

where w_i is now drawn from importance distribution h.

Notice that the above two are exactly the same population objects:

$$E^g\left[\ell\left(\omega\right)\right] = \int_\Omega \ell(\omega)g(\omega)d\omega = \int_\Omega \ell(\omega)\frac{g(\omega)}{h(\omega)}h(\omega)d\omega = E^h\left[\ell\left(\omega\right)\frac{g(\omega)}{h(\omega)}\right]$$

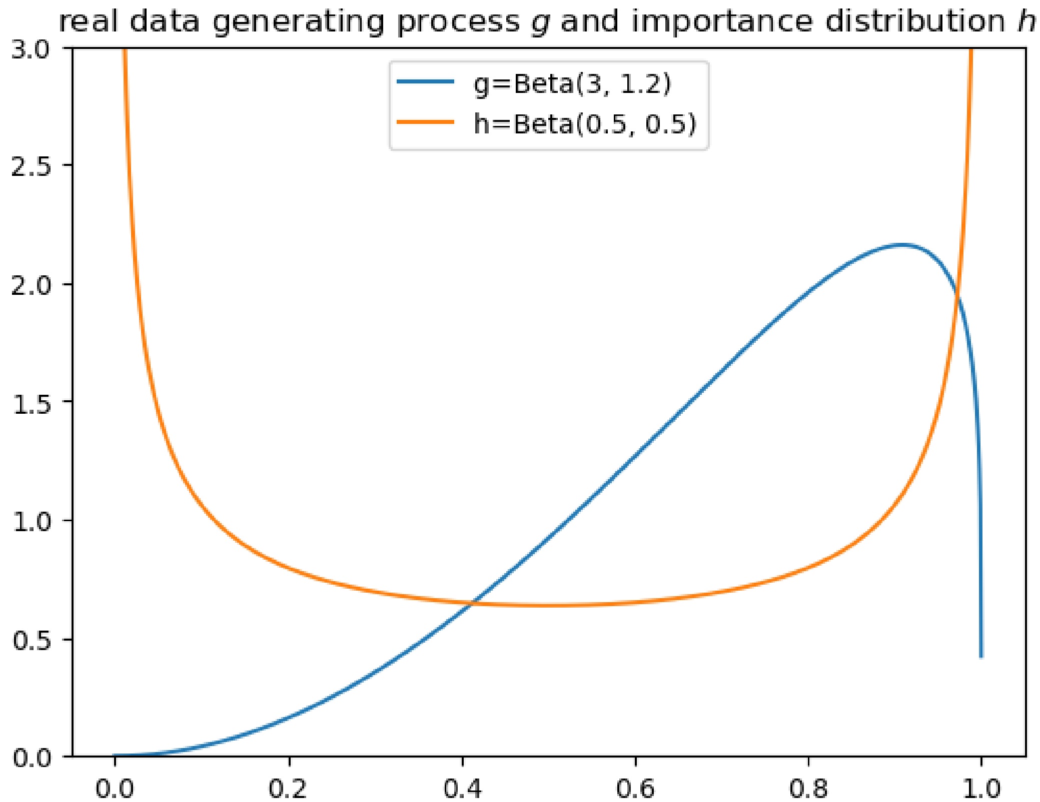
25.4 Selecting a sampling distribution

Since we must use an h that has larger mass in parts of the distribution to which g puts low mass, we use $h = Beta(0.5, 0.5)$ as our importance distribution.

The plots compare g and h.

```
g_a, g_b = G_a, G_b
h_a, h_b = 0.5, 0.5
```

```
w_range = np.linspace(1e-5, 1-1e-5, 1000)

plt.plot(w_range, g(w_range), label=f'g=Beta({g_a}, {g_b})')
plt.plot(w_range, p(w_range, 0.5, 0.5), label=f'h=Beta({h_a}, {h_b})')
plt.title('real data generating process $g$ and importance distribution $h$')
plt.legend()
plt.ylim([0., 3.])
plt.show()
```



25.5 Approximating a cumulative likelihood ratio

We now study how to use importance sampling to approximate $E\left[L(\omega^t)\right] = \left[\prod_{i=1}^{T} \ell\left(\omega_i\right)\right]$.

As above, our plan is to draw sequences ω^t from q and then re-weight the likelihood ratio appropriately:

$$\hat{E}^p\left[L\left(\omega^t\right)\right] = \hat{E}^p\left[\prod_{t=1}^{T} \ell\left(\omega_t\right)\right] = \hat{E}^q\left[\prod_{t=1}^{T} \ell\left(\omega_t\right) \frac{p\left(\omega_t\right)}{q\left(\omega_t\right)}\right] = \frac{1}{N} \sum_{i=1}^{N}\left(\prod_{t=1}^{T} \ell(\omega_{i,t}^h) \frac{p\left(\omega_{i,t}^h\right)}{q\left(\omega_{i,t}^h\right)}\right)$$

where the last equality uses $\omega_{i,t}^h$ drawn from the importance distribution q.

Here $\frac{p(\omega_{i,t}^q)}{q(\omega_{i,t}^q)}$ is the weight we assign to each data point $\omega_{i,t}^q$.

Below we prepare a Python function for computing the importance sampling estimates given any beta distributions p, q.

```
@jit(parallel=True)
def estimate(p_a, p_b, q_a, q_b, T=1, N=10000):

    μ_L = 0
    for i in prange(N):

        L = 1
        weight = 1
        for t in range(T):
            w = np.random.beta(q_a, q_b)
```

(continues on next page)

(continued from previous page)

```
            l = f(w) / g(w)

            L *= l
            weight *= p(w, p_a, p_b) / p(w, q_a, q_b)

        μ_L += L * weight

    μ_L /= N

    return μ_L
```

Consider the case when $T = 1$, which amounts to approximating $E_0\left[\ell\left(\omega\right)\right]$

For the standard Monte Carlo estimate, we can set $p = g$ and $q = g$.

```
estimate(g_a, g_b, g_a, g_b, T=1, N=10000)
```

```
0.9574936684386576
```

For our importance sampling estimate, we set $q = h$.

```
estimate(g_a, g_b, h_a, h_b, T=1, N=10000)
```

```
0.9974345776909089
```

Evidently, even at $T = 1$, our importance sampling estimate is closer to 1 than is the Monte Carlo estimate.

Bigger differences arise when computing expectations over longer sequences, $E_0\left[L\left(\omega^t\right)\right]$.

Setting $T = 10$, we find that the Monte Carlo method severely underestimates the mean while importance sampling still produces an estimate close to its theoretical value of unity.

```
estimate(g_a, g_b, g_a, g_b, T=10, N=10000)
```

```
0.4198482208408195
```

```
estimate(g_a, g_b, h_a, h_b, T=10, N=10000)
```

```
0.9583289538038178
```

The Monte Carlo method underestimates because the likelihood ratio $L(\omega^T) = \prod_{t=1}^{T} \frac{f(\omega_t)}{g(\omega_t)}$ has a highly skewed distribution under g.

Most samples from g produce small likelihood ratios, while the true mean requires occasional very large values that are rarely sampled.

In our case, since $g(\omega) \to 0$ as $\omega \to 0$ while $f(\omega)$ remains constant, the Monte Carlo procedure undersamples precisely where the likelihood ratio $\frac{f(\omega)}{g(\omega)}$ is largest.

As T increases, this problem worsens exponentially, making standard Monte Carlo increasingly unreliable.

Importance sampling with $q = h$ fixes this by sampling more uniformly from regions important to both f and g.

25.6 Distribution of sample mean

We next study the bias and efficiency of the Monte Carlo and importance sampling approaches.

The code below produces distributions of estimates using both Monte Carlo and importance sampling methods.

```
@jit(parallel=True)
def simulate(p_a, p_b, q_a, q_b, N_simu, T=1):

    μ_L_p = np.empty(N_simu)
    μ_L_q = np.empty(N_simu)

    for i in prange(N_simu):
        μ_L_p[i] = estimate(p_a, p_b, p_a, p_b, T=T)
        μ_L_q[i] = estimate(p_a, p_b, q_a, q_b, T=T)

    return μ_L_p, μ_L_q
```

Again, we first consider estimating $E\left[\ell\left(\omega\right)\right]$ by setting T=1.

We simulate 1000 times for each method.

```
N_simu = 1000
μ_L_p, μ_L_q = simulate(g_a, g_b, h_a, h_b, N_simu)
```

```
# standard Monte Carlo (mean and std)
np.nanmean(μ_L_p), np.nanvar(μ_L_p)
```

```
(np.float64(0.9960680602196872), np.float64(0.007697123780502527))
```

```
# importance sampling (mean and std)
np.nanmean(μ_L_q), np.nanvar(μ_L_q)
```

```
(np.float64(0.9999754590354378), np.float64(2.3229415107717145e-05))
```

Although both methods tend to provide a mean estimate of $E\left[\ell\left(\omega\right)\right]$ close to 1, the importance sampling estimates have smaller variance.

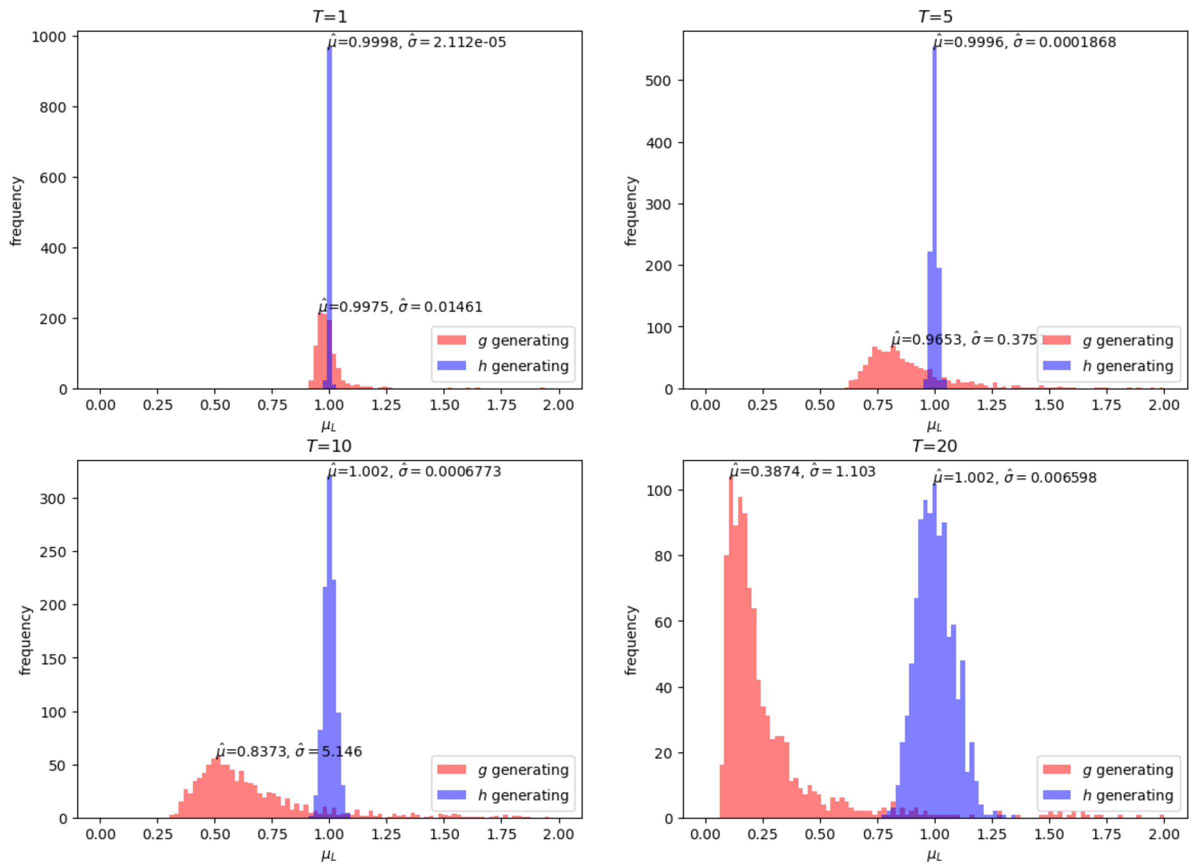
Next, we present distributions of estimates for $\hat{E}\left[L\left(\omega^t\right)\right]$, in cases for $T = 1, 5, 10, 20$.

```
fig, axs = plt.subplots(2, 2, figsize=(14, 10))

μ_range = np.linspace(0, 2, 100)

for i, t in enumerate([1, 5, 10, 20]):
    row = i // 2
    col = i % 2

    μ_L_p, μ_L_q = simulate(g_a, g_b, h_a, h_b, N_simu, T=t)
    μ_hat_p, μ_hat_q = np.nanmean(μ_L_p), np.nanmean(μ_L_q)
    σ_hat_p, σ_hat_q = np.nanvar(μ_L_p), np.nanvar(μ_L_q)

    axs[row, col].set_xlabel('$μ_L$')
    axs[row, col].set_ylabel('frequency')
    axs[row, col].set_title(f'$T$={t}')
    n_p, bins_p, _ = axs[row, col].hist(μ_L_p, bins=μ_range, color='r', alpha=0.5,␣
```

(continues on next page)

(continued from previous page)

```
↪label='$g$ generating')
    n_q, bins_q, _ = axs[row, col].hist(μ_L_q, bins=μ_range, color='b', alpha=0.5,␣
↪label='$h$ generating')
    axs[row, col].legend(loc=4)

    for n, bins, μ_hat, σ_hat in [[n_p, bins_p, μ_hat_p, σ_hat_p],
                                   [n_q, bins_q, μ_hat_q, σ_hat_q]]:
        idx = np.argmax(n)
        axs[row, col].text(bins[idx], n[idx], r'$\hat{μ}$='+f'{μ_hat:.4g}'+r', $\hat
↪{σ}=$'+f'{σ_hat:.4g}')

plt.show()
```



The simulation exercises above show that the importance sampling estimates are unbiased under all T while the standard Monte Carlo estimates are biased downwards.

Evidently, the bias increases with increases in T.

25.7 Choosing a sampling distribution

Above, we arbitraily chose $h = Beta(0.5, 0.5)$ as the importance distribution.

Is there an optimal importance distribution?

In our particular case, since we know in advance that $E_0\left[L\left(\omega^t\right)\right] = 1$, we can use that knowledge to our advantage.

Thus, suppose that we simply use $h = f$.

When estimating the mean of the likelihood ratio (T=1), we get:

$$\hat{E}^f\left[\ell(\omega)\frac{g(\omega)}{f(\omega)}\right] = \hat{E}^f\left[\frac{f(\omega)}{g(\omega)}\frac{g(\omega)}{f(\omega)}\right] = \frac{1}{N}\sum_{i=1}^{N}\ell(w_i^f)\frac{g(w_i^f)}{f(w_i^f)} = 1$$

```
μ_L_p, μ_L_q = simulate(g_a, g_b, F_a, F_b, N_simu)
```

```
# importance sampling (mean and std)
np.nanmean(μ_L_q), np.nanvar(μ_L_q)
```

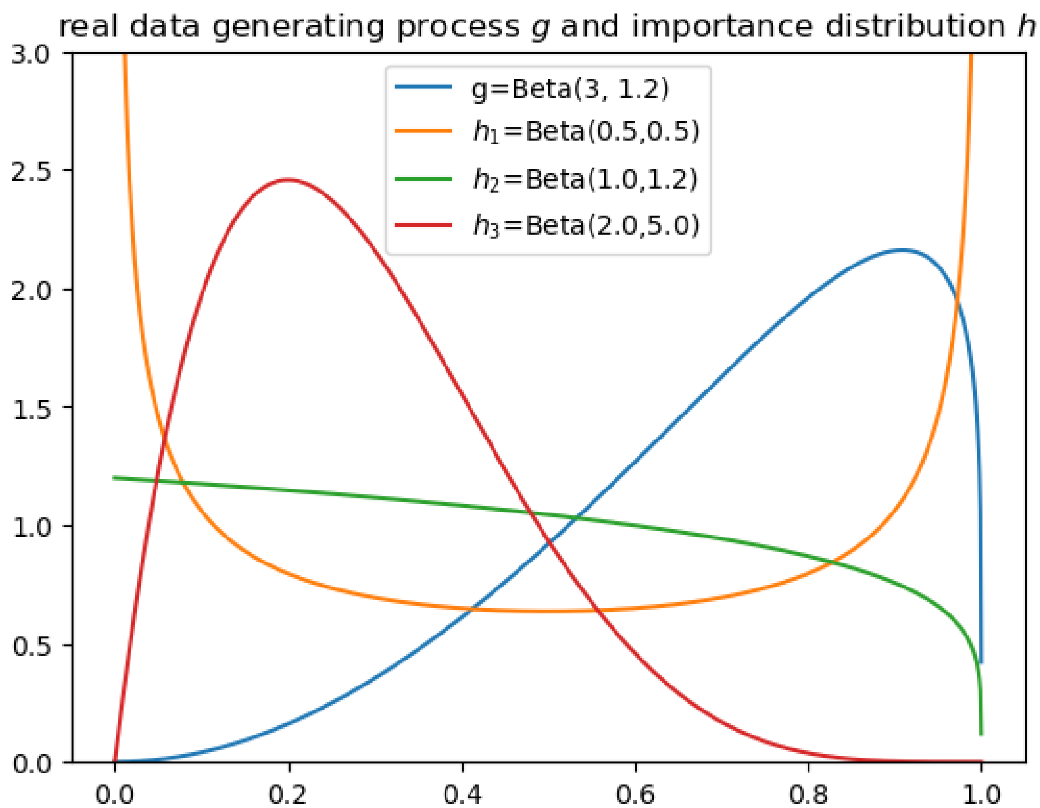
```
(np.float64(1.0), np.float64(0.0))
```

We could also use other distributions as our importance distribution.

Below we choose just a few and compare their sampling properties.

```
a_list = [0.5, 1., 2.]
b_list = [0.5, 1.2, 5.]
```

```
w_range = np.linspace(1e-5, 1-1e-5, 1000)

plt.plot(w_range, g(w_range), label=f'g=Beta({g_a}, {g_b})')
plt.plot(w_range, p(w_range, a_list[0], b_list[0]), label=f'$h_1$=Beta({a_list[0]},{b_
↪list[0]})')
plt.plot(w_range, p(w_range, a_list[1], b_list[1]), label=f'$h_2$=Beta({a_list[1]},{b_
↪list[1]})')
plt.plot(w_range, p(w_range, a_list[2], b_list[2]), label=f'$h_3$=Beta({a_list[2]},{b_
↪list[2]})')
plt.title('real data generating process $g$ and importance distribution $h$')
plt.legend()
plt.ylim([0., 3.])
plt.show()
```



We consider two additional distributions.

As a reminder h_1 is the original $Beta(0.5, 0.5)$ distribution that we used above.

h_2 is the $Beta(1, 1.2)$ distribution.

Note how h_2 has a similar shape to g at higher values of distribution but more mass at lower values.

Our hunch is that h_2 should be a good importance sampling distribution.

h_3 is the $Beta(2, 5)$ distribution.

Note how h_3 has zero mass at values very close to 0 and at values close to 1.

Our hunch is that h_3 will be a poor importance sampling distribution.

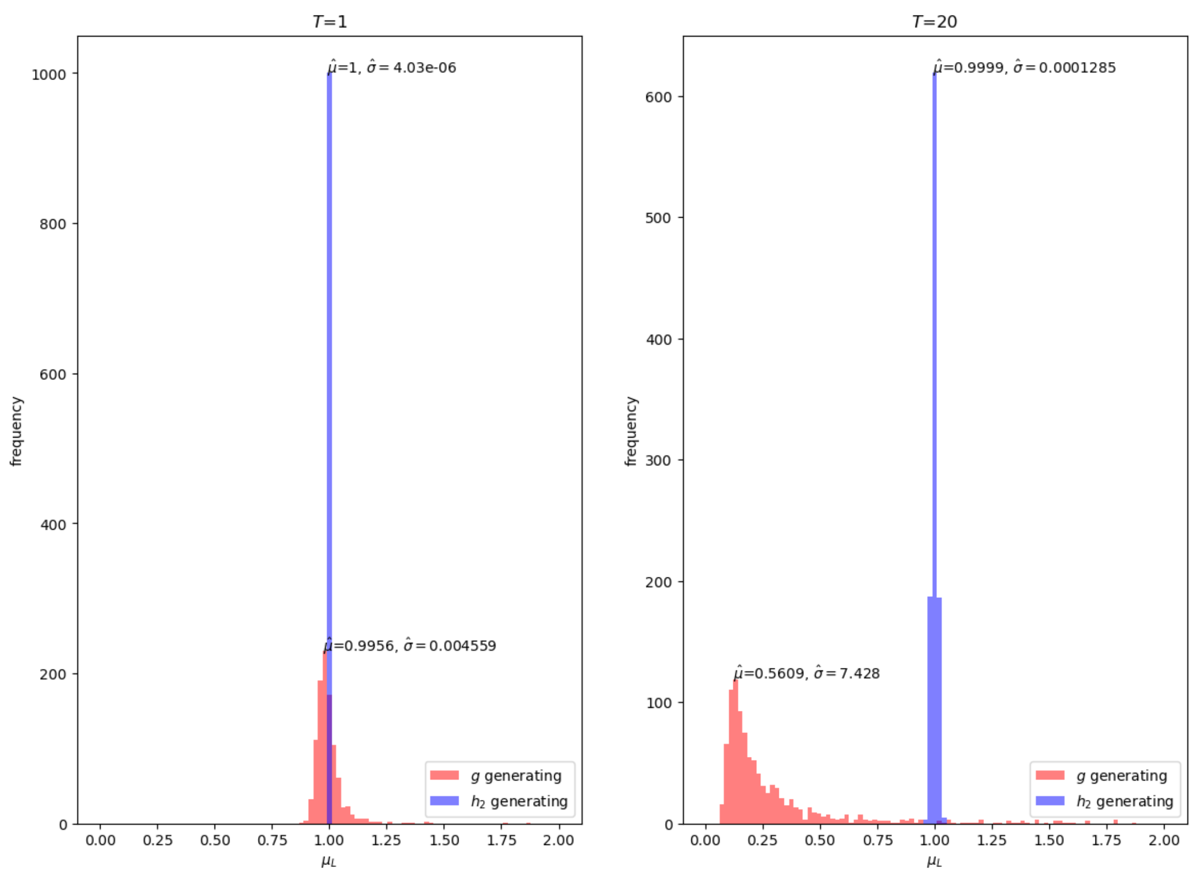
We first simulate a plot the distribution of estimates for $\hat{E}\left[L\left(\omega^t\right)\right]$ using h_2 as the importance sampling distribution.

```
h_a = a_list[1]
h_b = b_list[1]

fig, axs = plt.subplots(1,2, figsize=(14, 10))

μ_range = np.linspace(0, 2, 100)

for i, t in enumerate([1, 20]):


    μ_L_p, μ_L_q = simulate(g_a, g_b, h_a, h_b, N_simu, T=t)
    μ_hat_p, μ_hat_q = np.nanmean(μ_L_p), np.nanmean(μ_L_q)
    σ_hat_p, σ_hat_q = np.nanvar(μ_L_p), np.nanvar(μ_L_q)
```

(continues on next page)

(continued from previous page)

```
    axs[i].set_xlabel('$μ_L$')
    axs[i].set_ylabel('frequency')
    axs[i].set_title(f'$T$={t}')
    n_p, bins_p, _ = axs[i].hist(μ_L_p, bins=μ_range, color='r', alpha=0.5, label='$g
↪$ generating')
    n_q, bins_q, _ = axs[i].hist(μ_L_q, bins=μ_range, color='b', alpha=0.5, label='$h_
↪2$ generating')
    axs[i].legend(loc=4)

    for n, bins, μ_hat, σ_hat in [[n_p, bins_p, μ_hat_p, σ_hat_p],
                                   [n_q, bins_q, μ_hat_q, σ_hat_q]]:
        idx = np.argmax(n)
        axs[i].text(bins[idx], n[idx], r'$\hat{μ}$='+f'{μ_hat:.4g}'+r', $\hat{σ}=$'+f'
↪{σ_hat:.4g}')

plt.show()
```



Our simulations suggest that indeed h_2 is a quite good importance sampling distribution for our problem.

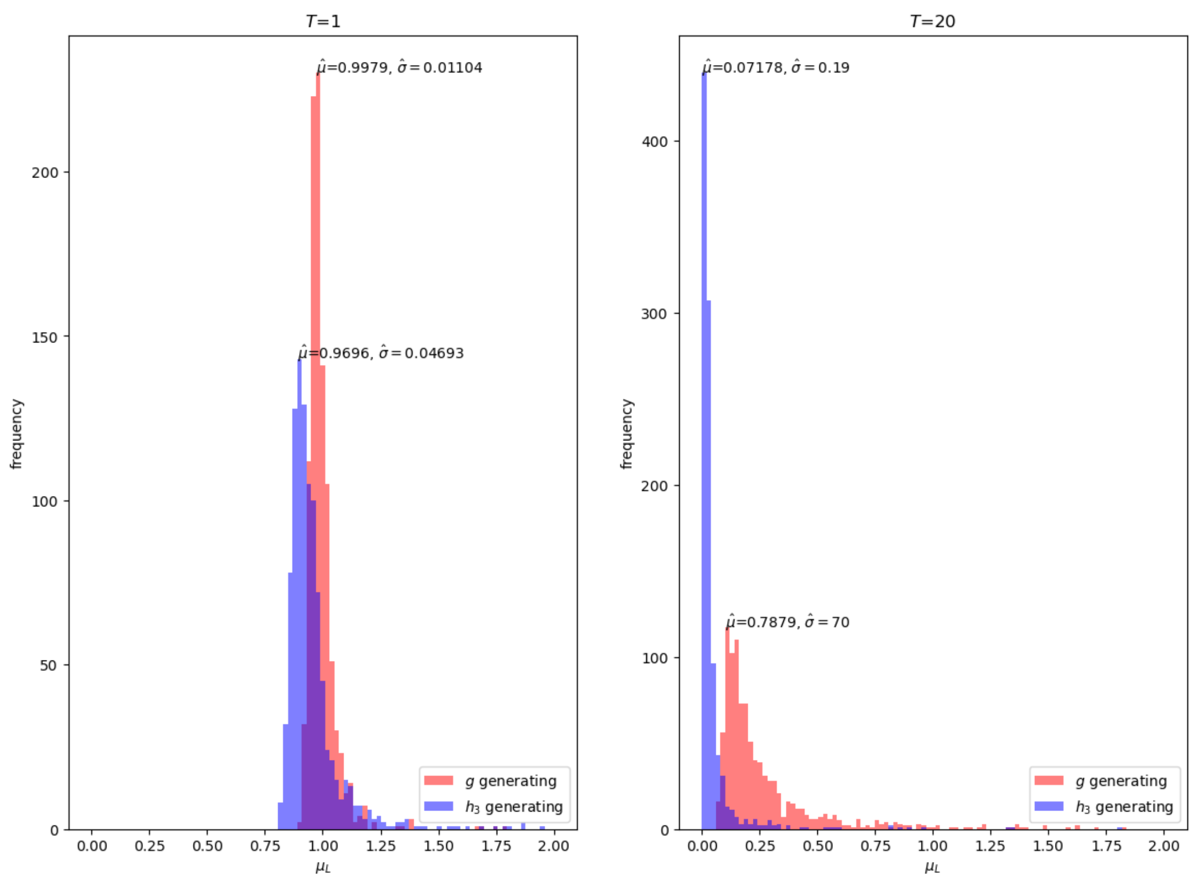
Even at $T = 20$, the mean is very close to 1 and the variance is small.

```
h_a = a_list[2]
h_b = b_list[2]

fig, axs = plt.subplots(1,2, figsize=(14, 10))
```

(continues on next page)

(continued from previous page)

```
μ_range = np.linspace(0, 2, 100)

for i, t in enumerate([1, 20]):


    μ_L_p, μ_L_q = simulate(g_a, g_b, h_a, h_b, N_simu, T=t)
    μ_hat_p, μ_hat_q = np.nanmean(μ_L_p), np.nanmean(μ_L_q)
    σ_hat_p, σ_hat_q = np.nanvar(μ_L_p), np.nanvar(μ_L_q)

    axs[i].set_xlabel('$μ_L$')
    axs[i].set_ylabel('frequency')
    axs[i].set_title(f'$T$={t}')
    n_p, bins_p, _ = axs[i].hist(μ_L_p, bins=μ_range, color='r', alpha=0.5, label='$g
↪$ generating')
    n_q, bins_q, _ = axs[i].hist(μ_L_q, bins=μ_range, color='b', alpha=0.5, label='$h_
↪3$ generating')
    axs[i].legend(loc=4)

    for n, bins, μ_hat, σ_hat in [[n_p, bins_p, μ_hat_p, σ_hat_p],
                                  [n_q, bins_q, μ_hat_q, σ_hat_q]]:
        idx = np.argmax(n)
        axs[i].text(bins[idx], n[idx], r'$\hat{μ}$='+f'{μ_hat:.4g}'+r', $\hat{σ}=$'+f'
↪{σ_hat:.4g}')

plt.show()
```



However, h_3 is evidently a poor importance sampling distribution for our problem, with a mean estimate far away from 1 for $T = 20$.

Notice that even at $T = 1$, the mean estimate with importance sampling is more biased than sampling with just g itself.

Thus, our simulations suggest that for our problem we would be better off simply using Monte Carlo approximations under g than using h_3 as an importance sampling distribution.

CHAPTER

TWENTYSIX

A PROBLEM THAT STUMPED MILTON FRIEDMAN

(and that Abraham Wald solved by inventing sequential analysis)

Contents

26.1 Overview

This is the first of two lectures about a statistical decision problem that a US Navy Captain presented to Milton Friedman and W. Allen Wallis during World War II when they were analysts at the U.S. Government's Statistical Research Group at Columbia University.

This problem led Abraham Wald [Wald, 1947] to formulate **sequential analysis**, an approach to statistical decision problems that is intimately related to dynamic programming.

In the spirit of *this earlier lecture*, the present lecture and its *sequel* approach the problem from two distinct points of view, one frequentist, the other Bayesian.

In this lecture, we describe Wald's formulation of the problem from the perspective of a statistician working within the Neyman-Pearson tradition of a frequentist statistician who thinks about testing hypotheses and consequently use laws of large numbers to investigate limiting properties of particular statistics under a given **hypothesis**, i.e., a vector of **parameters** that pins down a particular member of a manifold of statistical models that interest the statistician.

- From *this lecture on frequentist and bayesian statistics*, please remember that a frequentist statistician routinely calculates functions of sequences of random variables, conditioning on a vector of parameters.

In *this related lecture* we'll discuss another formulation that adopts the perspective of a **Bayesian statistician** who views parameters as random variables that are jointly distributed with observable variables that he is concerned about.

Because we are taking a frequentist perspective that is concerned about relative frequencies conditioned on alternative parameter values, i.e., alternative **hypotheses**, key ideas in this lecture

- Type I and type II statistical errors
  - a type I error occurs when you reject a null hypothesis that is true
  - a type II error occurs when you accept a null hypothesis that is false
- The **power** of a frequentist statistical test
- The **size** of a frequentist statistical test
- The **critical region** of a statistical test
- A **uniformly most powerful test**
- The role of a Law of Large Numbers (LLN) in interpreting **power** and **size** of a frequentist statistical test
- Abraham Wald's **sequential probability ratio test**

We'll begin with some imports:

```
import numpy as np
import matplotlib.pyplot as plt
from numba import njit, prange, vectorize, jit
from numba.experimental import jitclass
from math import gamma
from scipy.integrate import quad
from scipy.stats import beta
from collections import namedtuple
import pandas as pd
import scipy as sp
```

This lecture uses ideas studied in *the lecture on likelihood ratio processes* and *the lecture on Bayesian learning*.

26.2 Source of the Problem

On pages 137-139 of his 1998 book *Two Lucky People* with Rose Friedman [Friedman and Friedman, 1998], Milton Friedman described a problem presented to him and Allen Wallis during World War II, when they worked at the US Government's Statistical Research Group at Columbia University.

Note

See pages 25 and 26 of Allen Wallis's 1980 article [Wallis, 1980] about the Statistical Research Group at Columbia University during World War II for his account of the episode and for important contributions that Harold Hotelling made to formulating the problem. Also see chapter 5 of Jennifer Burns' book about Milton Friedman [Burns, 2023].

Let's listen to Milton Friedman tell us what happened

> In order to understand the story, it is necessary to have an idea of a simple statistical problem, and of the standard procedure for dealing with it. The actual problem out of which sequential analysis grew will serve. The Navy has two alternative designs (say A and B) for a projectile. It wants to determine which is superior. To do so it undertakes a series of paired firings. On each round, it assigns the value 1 or 0 to A accordingly as

> its performance is superior or inferior to that of B and conversely 0 or 1 to B. The Navy asks the statistician how to conduct the test and how to analyze the results.
>
> The standard statistical answer was to specify a number of firings (say 1,000) and a pair of percentages (e.g., 53% and 47%) and tell the client that if A receives a 1 in more than 53% of the firings, it can be regarded as superior; if it receives a 1 in fewer than 47%, B can be regarded as superior; if the percentage is between 47% and 53%, neither can be so regarded.
>
> When Allen Wallis was discussing such a problem with (Navy) Captain Garret L. Schuyler, the captain objected that such a test, to quote from Allen's account, may prove wasteful. If a wise and seasoned ordnance officer like Schuyler were on the premises, he would see after the first few thousand or even few hundred [rounds] that the experiment need not be completed either because the new method is obviously inferior or because it is obviously superior beyond what was hoped for ….

Friedman and Wallis worked on the problem for a while but didn't completely solve it.

Realizing that, they told Abraham Wald about the problem.

That set Wald on a path that led him to create *Sequential Analysis* [Wald, 1947].

26.3 Neyman-Pearson formulation

It is useful to begin by describing the theory underlying the test that the U.S. Navy told Captain G. S. Schuyler to use.

Captain Schuyler's doubts motivated him to tell Milton Friedman and Allen Wallis his conjecture that superior practical procedures existed.

Evidently, the Navy had told Captain Schuyler to use what was then a state-of-the-art Neyman-Pearson hypothesis test.

We'll rely on Abraham Wald's [Wald, 1947] elegant summary of Neyman-Pearson theory.

Watch for these features of the setup:

- the assumption of a *fixed* sample size n
- the application of laws of large numbers, conditioned on alternative probability models, to interpret probabilities α and β of the type I and type II errors defined in the Neyman-Pearson theory

In chapter 1 of **Sequential Analysis** [Wald, 1947] Abraham Wald summarizes the Neyman-Pearson approach to hypothesis testing.

Wald frames the problem as making a decision about a probability distribution that is partially known.

(You have to assume that *something* is already known in order to state a well-posed problem – usually, *something* means *a lot*)

By limiting what is unknown, Wald uses the following simple structure to illustrate the main ideas:

- A decision-maker wants to decide which of two distributions f_0, f_1 govern an IID random variable z.
- The null hypothesis H_0 is the statement that f_0 governs the data.
- The alternative hypothesis H_1 is the statement that f_1 governs the data.
- The problem is to devise and analyze a test of hypothesis H_0 against the alternative hypothesis H_1 on the basis of a sample of a fixed number n independent observations $z_1, z_2, \dots, z_n$ of the random variable z.

To quote Abraham Wald,

> A test procedure leading to the acceptance or rejection of the [null] hypothesis in question is simply a rule specifying, for each possible sample of size n, whether the [null] hypothesis should be accepted or rejected on the basis of the sample. This may also be expressed as follows: A test procedure is simply a subdivision of the totality of all possible samples of size n into two mutually exclusive parts, say part 1 and part 2, together

> with the application of the rule that the [null] hypothesis be accepted if the observed sample is contained in part 2. Part 1 is also called the critical region. Since part 2 is the totality of all samples of size n which are not included in part 1, part 2 is uniquely determined by part 1. Thus, choosing a test procedure is equivalent to determining a critical region.

Let's listen to Wald longer:

> As a basis for choosing among critical regions the following considerations have been advanced by Neyman and Pearson: In accepting or rejecting H_0 we may commit errors of two kinds. We commit an error of the first kind if we reject H_0 when it is true; we commit an error of the second kind if we accept H_0 when H_1 is true. After a particular critical region W has been chosen, the probability of committing an error of the first kind, as well as the probability of committing an error of the second kind is uniquely determined. The probability of committing an error of the first kind is equal to the probability, determined by the assumption that H_0 is true, that the observed sample will be included in the critical region W. The probability of committing an error of the second kind is equal to the probability, determined on the assumption that H_1 is true, that the probability will fall outside the critical region W. For any given critical region W we shall denote the probability of an error of the first kind by α and the probability of an error of the second kind by β.

Let's listen carefully to how Wald applies law of large numbers to interpret α and β:

> The probabilities α and β have the following important practical interpretation: Suppose that we draw a large number of samples of size n. Let M be the number of such samples drawn. Suppose that for each of these M samples we reject H_0 if the sample is included in W and accept H_0 if the sample lies outside W. In this way we make M statements of rejection or acceptance. Some of these statements will in general be wrong. If H_0 is true and if M is large, the probability is nearly 1 (i.e., it is practically certain) that the proportion of wrong statements (i.e., the number of wrong statements divided by M) will be approximately α. If H_1 is true, the probability is nearly 1 that the proportion of wrong statements will be approximately β. Thus, we can say that in the long run [here Wald applies law of large numbers by driving $M \to \infty$ (our comment, not Wald's)] the proportion of wrong statements will be α if H_0 is true and β if H_1 is true.

The quantity α is called the *size* of the critical region, and the quantity $1 - \beta$ is called the *power* of the critical region.

Wald notes that

> one critical region W is more desirable than another if it has smaller values of α and β. Although either α or β can be made arbitrarily small by a proper choice of the critical region W, it is impossible to make both α and β arbitrarily small for a fixed value of n, i.e., a fixed sample size.

Wald summarizes Neyman and Pearson's setup as follows:

> Neyman and Pearson show that a region consisting of all samples $(z_1, z_2, \dots, z_n)$ which satisfy the inequality
>
> $$\frac{f_1(z_1) \cdots f_1(z_n)}{f_0(z_1) \cdots f_0(z_n)} \geq k$$
>
> is a most powerful critical region for testing the hypothesis H_0 against the alternative hypothesis H_1. The term k on the right side is a constant chosen so that the region will have the required size α.

Wald goes on to discuss Neyman and Pearson's concept of *uniformly most powerful* test.

Here is how Wald introduces the notion of a sequential test

> A rule is given for making one of the following three decisions at any stage of the experiment (at the m th trial for each integral value of m): (1) to accept the hypothesis H, (2) to reject the hypothesis H, (3) to continue the experiment by making an additional observation. Thus, such a test procedure is carried out sequentially. On the basis of the first observation, one of the aforementioned decision is made. If the first or second decision is made, the process is terminated. If the third decision is made, a second trial is performed. Again, on the basis of the first two observations, one of the three decision is made. If the third decision is made, a third trial is performed, and so on. The process is continued until either the first or the second

decisions is made. The number n of observations required by such a test procedure is a random variable, since the value of n depends on the outcome of the observations.

26.4 Wald's sequential formulation

By way of contrast to Neyman and Pearson's formulation of the problem, in Wald's formulation

- The sample size n is not fixed but rather a random variable.
- Two parameters A and B that are related to but distinct from Neyman and Pearson's α and β; A and B characterize cut-off rules that Wald uses to determine the random variable n as a function of random outcomes.

Here is how Wald sets up the problem.

A decision-maker can observe a sequence of draws of a random variable z.

He (or she) wants to know which of two probability distributions f_0 or f_1 governs z.

We use beta distributions as examples.

We will also work with Jensen-Shannon divergence introduced in *Statistical Divergence Measures*.

```
@vectorize
def p(x, a, b):
    """Beta distribution density function."""
    r = gamma(a + b) / (gamma(a) * gamma(b))
    return r * x** (a-1) * (1 - x) ** (b-1)

def create_beta_density(a, b):
    """Create a beta density function with specified parameters."""
    return jit(lambda x: p(x, a, b))

def compute_KL(f, g):
    """Compute KL divergence KL(f, g)"""
    integrand = lambda w: f(w) * np.log(f(w) / g(w))
    val, _ = quad(integrand, 1e-5, 1-1e-5)
    return val

def compute_JS(f, g):
    """Compute Jensen-Shannon divergence"""
    def m(w):
        return 0.5 * (f(w) + g(w))

    js_div = 0.5 * compute_KL(f, m) + 0.5 * compute_KL(g, m)
    return js_div
```

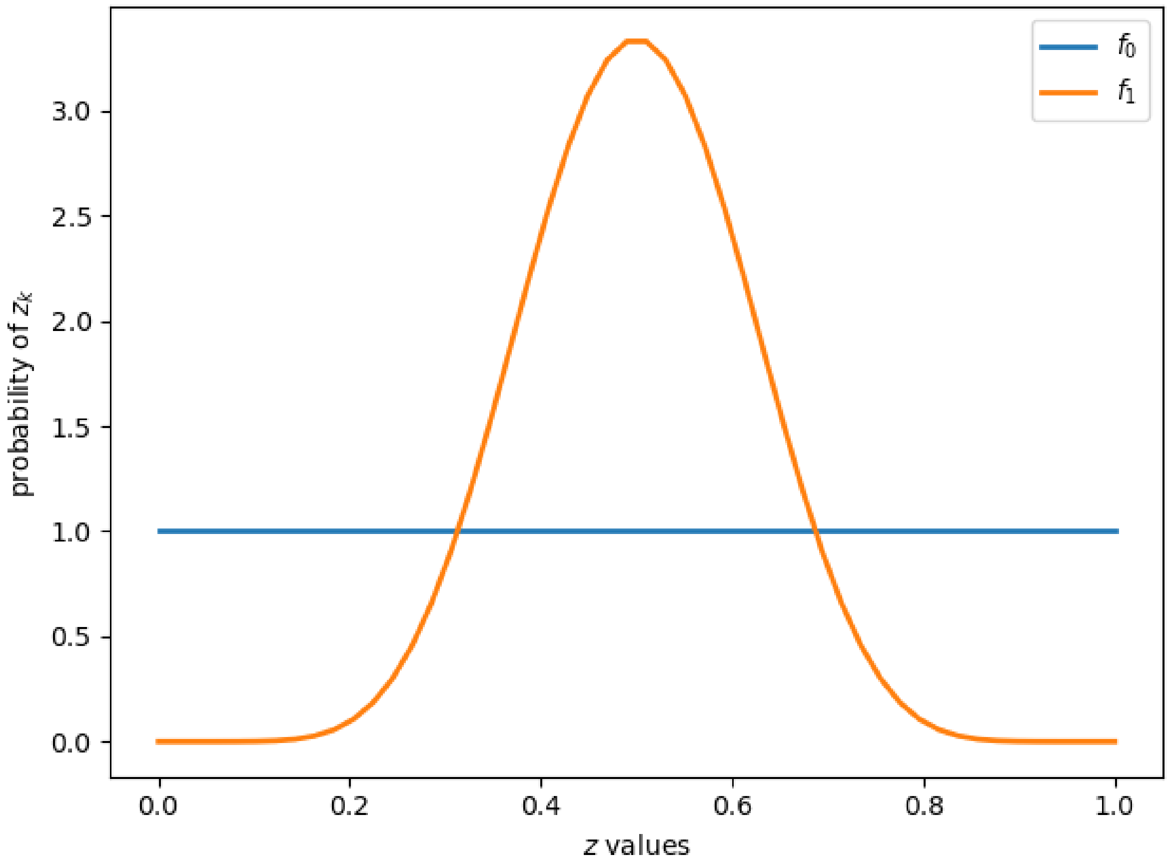
The next figure shows two beta distributions

```
f0 = create_beta_density(1, 1)
f1 = create_beta_density(9, 9)
grid = np.linspace(0, 1, 50)

fig, ax = plt.subplots()
ax.plot(grid, f0(grid), lw=2, label="$f_0$")
ax.plot(grid, f1(grid), lw=2, label="$f_1$")
ax.legend()
ax.set(xlabel="$z$ values", ylabel="probability of $z_k$")
```

(continues on next page)

(continued from previous page)

```
plt.tight_layout()
plt.show()
```



Conditional on knowing that successive observations are drawn from distribution f_0, the sequence of random variables is independently and identically distributed (IID).

Conditional on knowing that successive observations are drawn from distribution f_1, the sequence of random variables is also independently and identically distributed (IID).

But the observer does not know which of the two distributions generated the sequence.

For reasons explained in Exchangeability and Bayesian Updating, this means that the observer thinks that the sequence is not IID.

Consequently, the observer has something to learn, namely, whether the observations are drawn from f_0 or from f_1.

The decision maker wants to decide which of the two distributions is generating outcomes.

26.4.1 Type I and type II errors

If we regard $f = f_0$ as a null hypothesis and $f = f_1$ as an alternative hypothesis, then

- a type I error is an incorrect rejection of a true null hypothesis (a "false positive")
- a type II error is a failure to reject a false null hypothesis (a "false negative")

To repeat ourselves

- α is the probability of a type I error
- β is the probability of a type II error

26.4.2 Choices

After observing $z_k, z_{k-1}, \ldots, z_1$, the decision-maker chooses among three distinct actions:

- He decides that $f = f_0$ and draws no more z's
- He decides that $f = f_1$ and draws no more z's
- He postpones deciding and instead chooses to draw z_{k+1}

Wald defines

- $p_{0m} = f_0(z_1) \cdots f_0(z_m)$
- $p_{1m} = f_1(z_1) \cdots f_1(z_m)$
- $L_m = \frac{p_{1m}}{p_{0m}}$

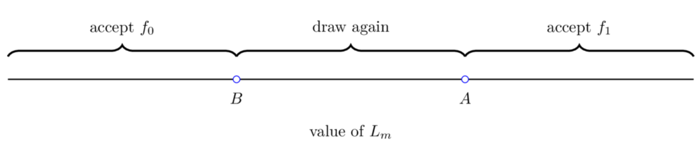
Here $\{L_m\}_{m=0}^{\infty}$ is a **likelihood ratio process**.

Wald's sequential decision rule is parameterized by real numbers $B < A$.

For a given pair A, B, the decision rule is

$$\begin{aligned} \text{accept } f = f_1 \text{ if } L_m \geq A \\ \text{accept } f = f_0 \text{ if } L_m \leq B \\ \text{draw another } z \text{ if } B < L_m < A \end{aligned}$$

The following figure illustrates aspects of Wald's procedure.



26.5 Links between A, B and α, β

In chapter 3 of **Sequential Analysis** [Wald, 1947] Wald establishes the inequalities

$$\begin{aligned} \frac{\alpha}{1-\beta} \leq \frac{1}{A} \\ \frac{\beta}{1-\alpha} \leq B \end{aligned}$$

His analysis of these inequalities leads Wald to recommend the following approximations as rules for setting A and B that come close to attaining a decision maker's target values for probabilities α of a type I and β of a type II error:

$$\begin{aligned} A \approx a(\alpha,\beta) &\equiv \frac{1-\beta}{\alpha} \\ B \approx b(\alpha,\beta) &\equiv \frac{\beta}{1-\alpha} \end{aligned} \tag{26.1}$$

For small values of α and β, Wald shows that approximation (26.1) provides a good way to set A and B.

In particular, Wald constructs a mathematical argument that leads him to conclude that the use of approximation (26.1) rather than the true functions $A(\alpha,\beta), B(\alpha,\beta)$ for setting A and B

> … cannot result in any appreciable increase in the value of either α or β. In other words, for all practical purposes the test corresponding to $A = a(\alpha,\beta)$, $B = b(\alpha,\beta)$ provides as least the same protection against wrong decisions as the test corresponding to $A = A(\alpha,\beta)$ and $B = b(\alpha,\beta)$.
>
> Thus, the only disadvantage that may arise from using $a(\alpha,\beta), b(\alpha,\beta)$ instead of $A(\alpha,\beta), B(\alpha,\beta)$, respectively, is that it may result in an appreciable increase in the number of observations required by the test.

We'll write some Python code to help us illustrate Wald's claims about how α and β are related to the parameters A and B that characterize his sequential probability ratio test.

26.6 Simulations

We experiment with different distributions f_0 and f_1 to examine how Wald's test performs under various conditions.

Our goal in conducting these simulations is to understand trade-offs between decision speed and accuracy associated with Wald's **sequential probability ratio test**.

Specifically, we will watch how:

- The decision thresholds A and B (or equivalently the target error rates α and β) affect the average stopping time
- The discrepancy between distributions f_0 and f_1 affects average stopping times

We will focus on the case where f_0 and f_1 are beta distributions since it is easy to control the overlapping regions of the two densities by adjusting their shape parameters.

First, we define a namedtuple to store all the parameters we need for our simulation studies.

We also compute Wald's recommended thresholds A and B based on the target type I and type II errors α and β

```
SPRTParams = namedtuple('SPRTParams',
                ['α', 'β',  # Target type I and type II errors
                'a0', 'b0', # Shape parameters for f_0
                'a1', 'b1', # Shape parameters for f_1
                'N',        # Number of simulations
                'seed'])

@njit
def compute_wald_thresholds(α, β):
    """Compute Wald's recommended thresholds."""
    A = (1 - β) / α
    B = β / (1 - α)
    return A, B, np.log(A), np.log(B)
```

Now we can run the simulation following Wald's recommendation.

We'll compare the log-likelihood ratio to logarithms of the thresholds $\log(A)$ and $\log(B)$.

The following algorithm underlies our simulations.

1. Compute thresholds $A = \frac{1-\beta}{\alpha}$, $B = \frac{\beta}{1-\alpha}$ and work with $\log A$, $\log B$.
2. Given true distribution (either f_0 or f_1):
   - Initialize log-likelihood ratio $\log L_0 = 0$
   - Repeat:
     - Draw observation z from the true distribution
     - Update: $\log L_{n+1} \leftarrow \log L_n + (\log f_1(z) - \log f_0(z))$
     - If $\log L_{n+1} \geq \log A$: stop, reject H_0
     - If $\log L_{n+1} \leq \log B$: stop, accept H_0
3. Repeat step 2 for N replications with $N/2$ replications for each distribution, compute the empirical type I error $\hat{\alpha}$ and type II error $\hat{\beta}$ with

$$\hat{\alpha} = \frac{\text{\# of times reject } H_0 \text{ when } f_0 \text{ is true}}{\text{\# of replications with } f_0 \text{ true}}$$

$$\hat{\beta} = \frac{\text{\# of times accept } H_0 \text{ when } f_1 \text{ is true}}{\text{\# of replications with } f_1 \text{ true}}$$

```python
@njit
def sprt_single_run(a0, b0, a1, b1, logA, logB, true_f0, seed):
    """Run a single SPRT until a decision is reached."""
    log_L = 0.0
    n = 0
    np.random.seed(seed)

    while True:
        z = np.random.beta(a0, b0) if true_f0 else np.random.beta(a1, b1)
        n += 1

        # Update log-likelihood ratio
        log_L += np.log(p(z, a1, b1)) - np.log(p(z, a0, b0))

        # Check stopping conditions
        if log_L >= logA:
            return n, False  # Reject H0
        elif log_L <= logB:
            return n, True   # Accept H0

@njit(parallel=True)
def run_sprt_simulation(a0, b0, a1, b1, α, β, N, seed):
    """SPRT simulation."""
    A, B, logA, logB = compute_wald_thresholds(α, β)

    stopping_times = np.zeros(N, dtype=np.int64)
    decisions_h0 = np.zeros(N, dtype=np.bool_)
    truth_h0 = np.zeros(N, dtype=np.bool_)

    for i in prange(N):
        true_f0 = (i % 2 == 0)
        truth_h0[i] = true_f0
```

(continues on next page)

(continued from previous page)

```
        n, accept_f0 = sprt_single_run(
                        a0, b0, a1, b1,
                        logA, logB,
                        true_f0, seed + i)
        stopping_times[i] = n
        decisions_h0[i] = accept_f0

    return stopping_times, decisions_h0, truth_h0

def run_sprt(params):
    """Run SPRT simulations with given parameters."""
    stopping_times, decisions_h0, truth_h0 = run_sprt_simulation(
        params.a0, params.b0, params.a1, params.b1,
        params.α, params.β, params.N, params.seed
    )

    # Calculate error rates
    truth_h0_bool = truth_h0.astype(bool)
    decisions_h0_bool = decisions_h0.astype(bool)

    type_I = np.sum(truth_h0_bool & ~decisions_h0_bool) \
            / np.sum(truth_h0_bool)
    type_II = np.sum(~truth_h0_bool & decisions_h0_bool) \
            / np.sum(~truth_h0_bool)

    return {
        'stopping_times': stopping_times,
        'decisions_h0': decisions_h0_bool,
        'truth_h0': truth_h0_bool,
        'type_I': type_I,
        'type_II': type_II
    }

# Run simulation
params = SPRTParams(α=0.05, β=0.10, a0=2, b0=5, a1=5, b1=2, N=20000, seed=1)
results = run_sprt(params)

print(f"Average stopping time: {results['stopping_times'].mean():.2f}")
print(f"Empirical type I  error: {results['type_I']:.3f} (target = {params.α})")
print(f"Empirical type II error: {results['type_II']:.3f} (target = {params.β})")
```

```
Average stopping time: 1.59
Empirical type I  error: 0.012 (target = 0.05)
Empirical type II error: 0.022 (target = 0.1)
```

As anticipated in the passage above in which Wald discussed the quality of $a(\alpha, \beta), b(\alpha, \beta)$ given in approximation (26.1), we find that the algorithm actually gives **lower** type I and type II error rates than the target values.

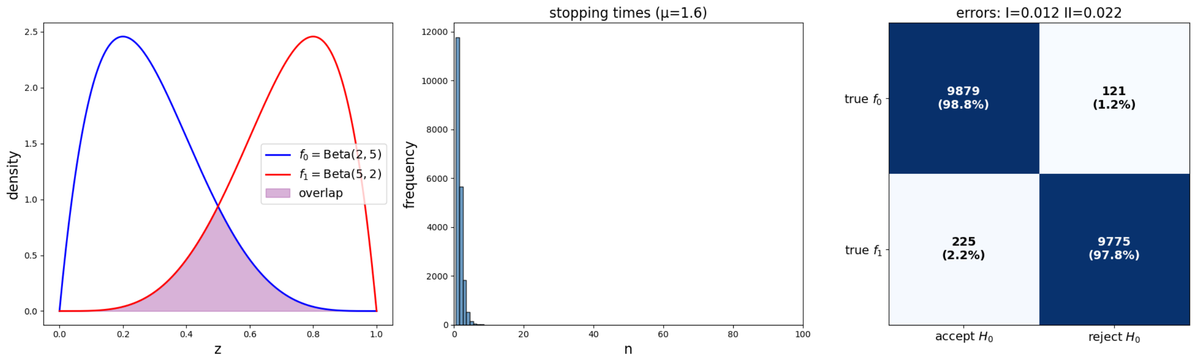
Note

For recent work on the quality of approximation (26.1), see, e.g., [Fischer and Ramdas, 2024].

The following code creates a few graphs that illustrate the results of our simulation.

Let's plot the results of our simulation

```
plot_sprt_results(results, params)
```



In this example, the stopping time stays below 10.

We can construct a 2×2 "confusion matrix" whose diagonal elements count the number of times that Wald's decision rule correctly accepts and rejects the null hypothesis.

```
print("Confusion Matrix data:")
print(f"Type I error: {results['type_I']:.3f}")
print(f"Type II error: {results['type_II']:.3f}")
```

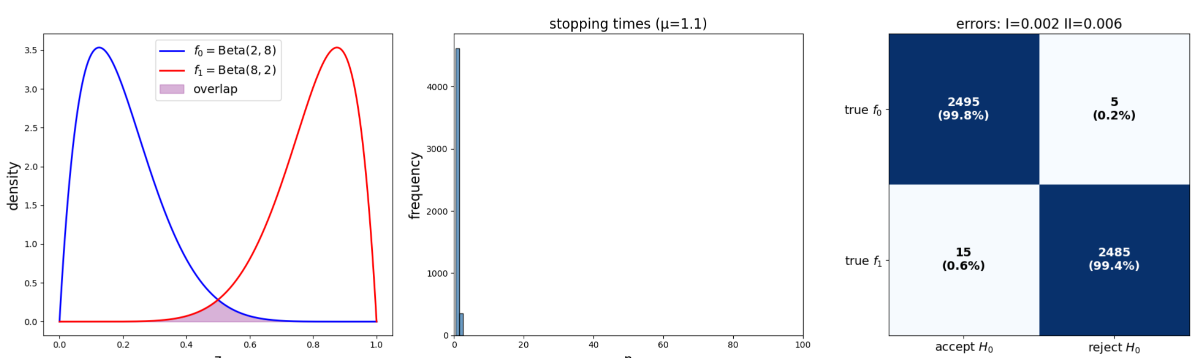
```
Confusion Matrix data:
Type I error: 0.012
Type II error: 0.022
```

Next we use our code to study three different f_0, f_1 pairs having different discrepancies between distributions.

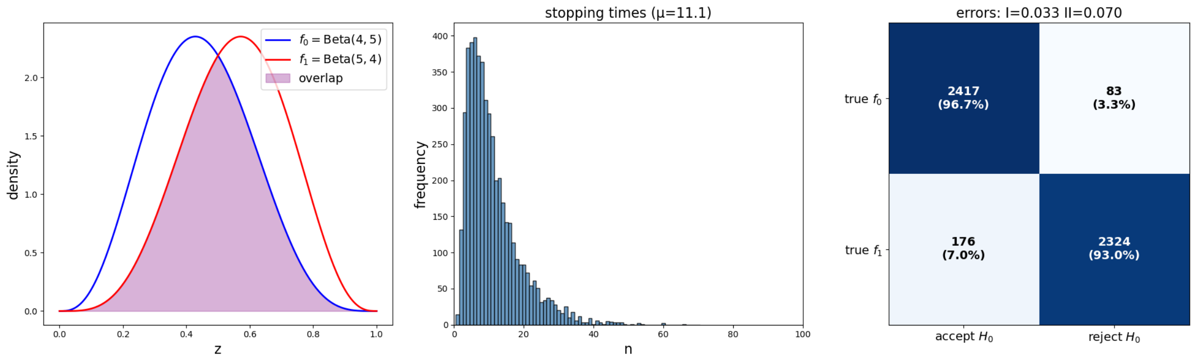
We plot the same three graphs we used above for each pair of distributions

```
params_1 = SPRTParams(α=0.05, β=0.10, a0=2, b0=8, a1=8, b1=2, N=5000, seed=42)
results_1 = run_sprt(params_1)

params_2 = SPRTParams(α=0.05, β=0.10, a0=4, b0=5, a1=5, b1=4, N=5000, seed=42)
results_2 = run_sprt(params_2)

params_3 = SPRTParams(α=0.05, β=0.10, a0=0.5, b0=0.4, a1=0.4,
                      b1=0.5, N=5000, seed=42)
results_3 = run_sprt(params_3)
```

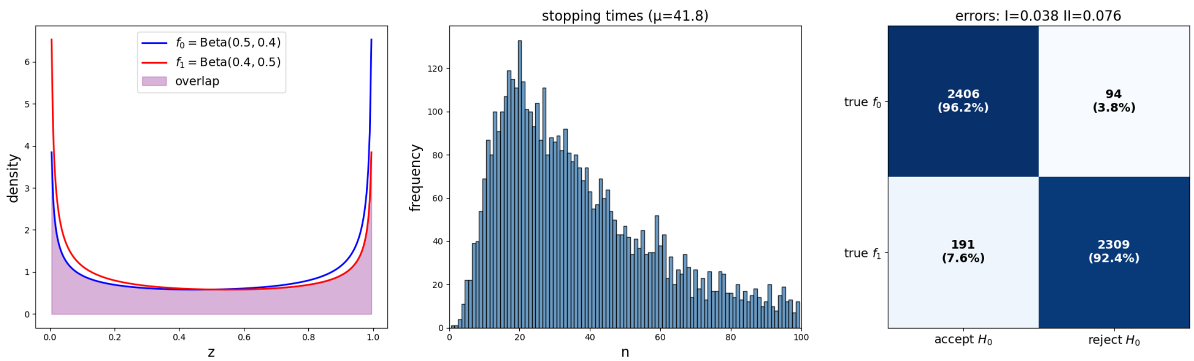
```
plot_sprt_results(results_1, params_1)
```



```
plot_sprt_results(results_2, params_2)
```




```
plot_sprt_results(results_3, params_3)
```



Notice that the stopping times are less when the two distributions are farther apart.

This makes sense.

When two distributions are "far apart", it should not take too long to decide which one is generating the data.

When two distributions are "close", it should takes longer to decide which one is generating the data.

It is tempting to link this pattern to our discussion of *Kullback–Leibler divergence* in *Likelihood Ratio Processes*.

While, KL divergence is larger when two distributions differ more, KL divergence is not symmetric, meaning that the KL divergence of distribution f from distribution g is not necessarily equal to the KL divergence of g from f.

If we want a symmetric measure of divergence that actually a metric, we can instead use Jensen-Shannon distance.

That is what we shall do now.

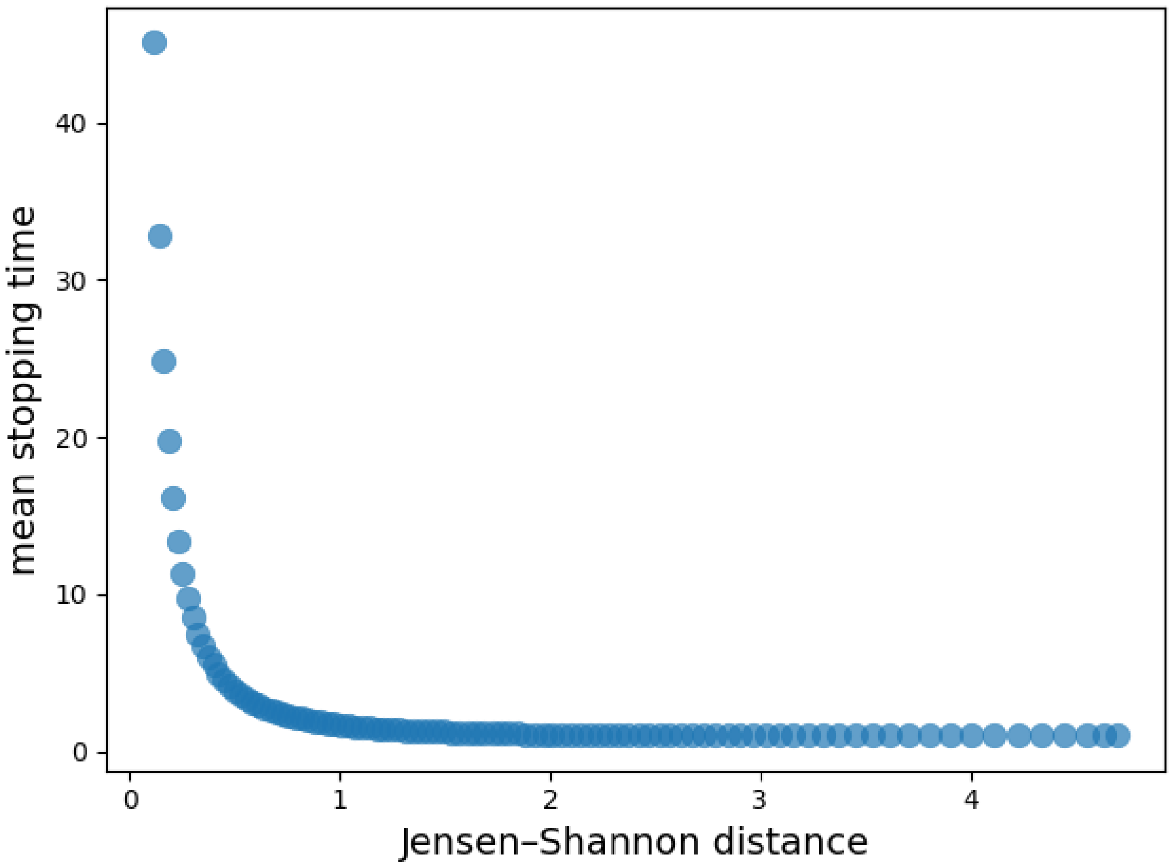
We shall compute Jensen-Shannon distance and plot it against the average stopping times.

```
def js_dist(a0, b0, a1, b1):
    """Jensen—Shannon distance"""
    f0 = create_beta_density(a0, b0)
    f1 = create_beta_density(a1, b1)

    # Mixture
    m = lambda w: 0.5*(f0(w) + f1(w))
    return np.sqrt(0.5*compute_KL(m, f0) + 0.5*compute_KL(m, f1))

def generate_β_pairs(N=100, T=10.0, d_min=0.5, d_max=9.5):
    ds = np.linspace(d_min, d_max, N)
```

(continues on next page)

(continued from previous page)

```
    a0 = (T - ds) / 2
    b0 = (T + ds) / 2
    return list(zip(a0, b0, b0, a0))

param_comb = generate_β_pairs()

# Run simulations for each parameter combination
js_dists = []
mean_stopping_times = []
param_list = []

for a0, b0, a1, b1 in param_comb:
    # Compute KL divergence
    js_div = js_dist(a1, b1, a0, b0)

    # Run SPRT simulation with a fixed set of parameters d d
    params = SPRTParams(α=0.05, β=0.10, a0=a0, b0=b0,
                        a1=a1, b1=b1, N=5000, seed=42)
    results = run_sprt(params)

    js_dists.append(js_div)
    mean_stopping_times.append(results['stopping_times'].mean())
    param_list.append((a0, b0, a1, b1))

# Create the plot
fig, ax = plt.subplots()

scatter = ax.scatter(js_dists, mean_stopping_times,
                    s=80, alpha=0.7, linewidth=0.5)

ax.set_xlabel('Jensen-Shannon distance', fontsize=14)
ax.set_ylabel('mean stopping time', fontsize=14)

plt.tight_layout()
plt.show()
```



The plot demonstrates a clear negative correlation between relative entropy and mean stopping time.

As Jensen-Shannon divergence increases (distributions become more separated), the mean stopping time decreases exponentially.

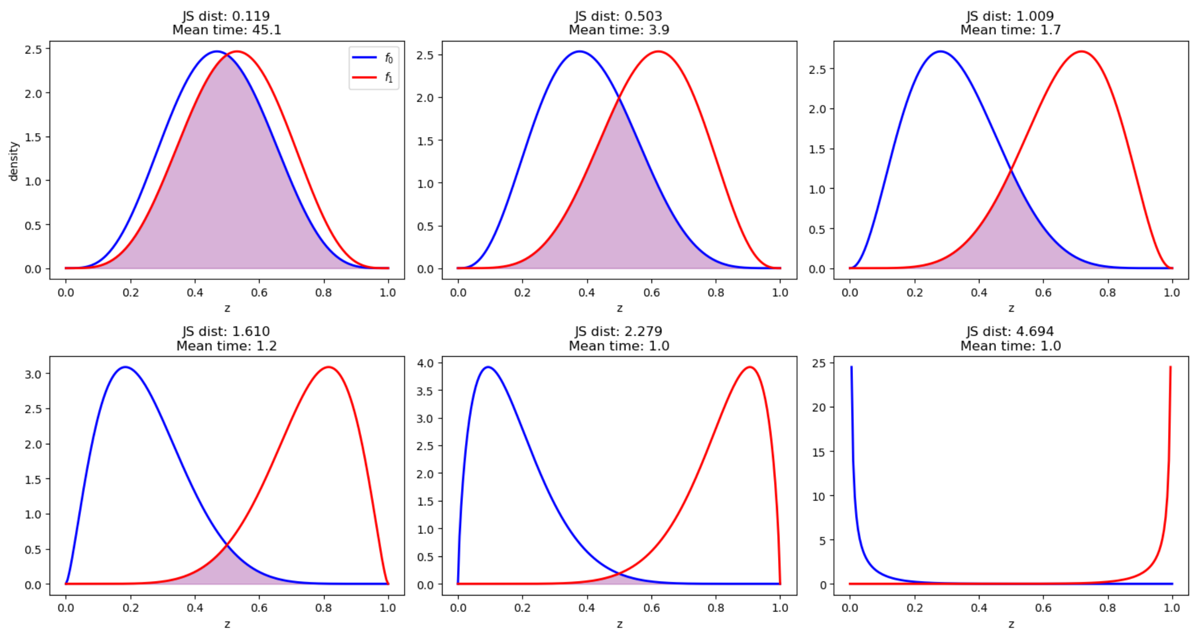
Below are sampled examples from the experiments we have above

```
def plot_beta_distributions_grid(param_list, js_dists, mean_stopping_times,
                                 selected_indices=None):
    """Plot grid of beta distributions with JS distance and stopping times."""
    if selected_indices is None:
        selected_indices = [0, len(param_list)//6, len(param_list)//3,
                            len(param_list)//2, 2*len(param_list)//3, -1]

    fig, axes = plt.subplots(2, 3, figsize=(15, 8))
    z_grid = np.linspace(0, 1, 200)

    for i, idx in enumerate(selected_indices):
        row, col = i // 3, i % 3
        a0, b0, a1, b1 = param_list[idx]

        f0 = create_beta_density(a0, b0)
        f1 = create_beta_density(a1, b1)

        axes[row, col].plot(z_grid, f0(z_grid), 'b-', lw=2, label='$f_0$')
        axes[row, col].plot(z_grid, f1(z_grid), 'r-', lw=2, label='$f_1$')
        axes[row, col].fill_between(z_grid, 0,
```

(continues on next page)

(continued from previous page)

```
                                  np.minimum(f0(z_grid), f1(z_grid)),
                                  alpha=0.3, color='purple')

        axes[row, col].set_title(f'JS dist: {js_dists[idx]:.3f}'
                               f'\nMean time: {mean_stopping_times[idx]:.1f}',
                               fontsize=12)
        axes[row, col].set_xlabel('z', fontsize=10)
        if i == 0:
            axes[row, col].set_ylabel('density', fontsize=10)
            axes[row, col].legend(fontsize=10)

    plt.tight_layout()
    plt.show()

plot_beta_distributions_grid(param_list, js_dists, mean_stopping_times)
```



Again, we find that the stopping time is shorter when the distributions are more separated, as measured by Jensen-Shannon distance.

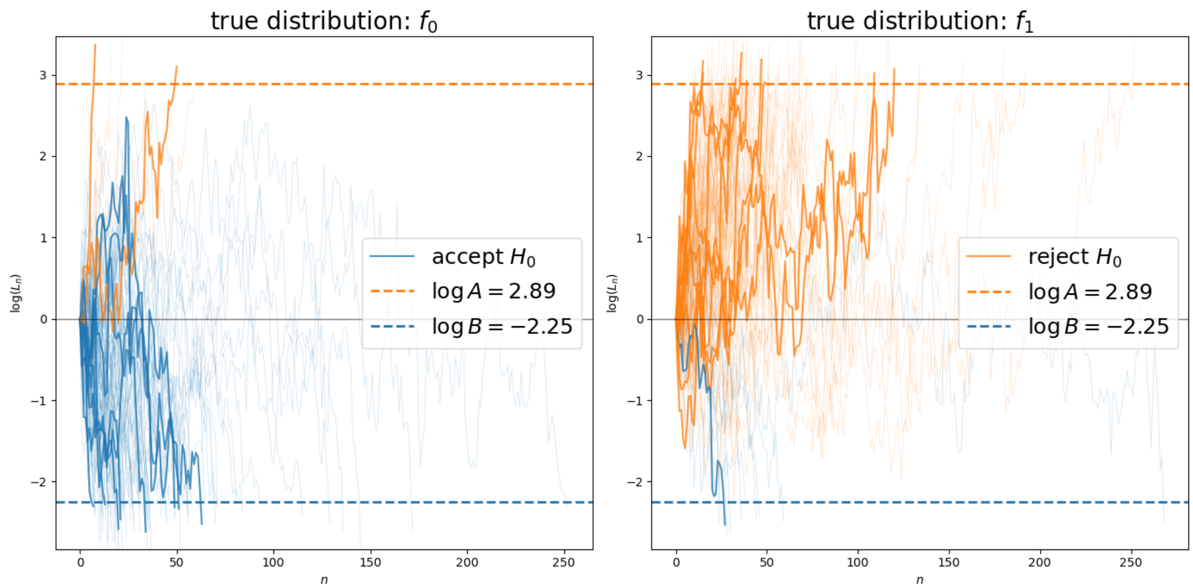
Let's visualize individual likelihood ratio processes to see how they evolve toward the decision boundaries.

```
def plot_likelihood_paths(params, n_highlight=10, n_background=200):
    """Visualize likelihood ratio paths."""
    A, B, logA, logB = compute_wald_thresholds(params.α, params.β)
    f0, f1 = map(lambda ab: create_beta_density(*ab),
             [(params.a0, params.b0),
              (params.a1, params.b1)])

    fig, axes = plt.subplots(1, 2, figsize=(14, 7))

    for dist_idx, (true_f0, ax, title) in enumerate([
        (True, axes[0], 'true distribution: $f_0$'),
        (False, axes[1], 'true distribution: $f_1$')
    ]):
        rng = np.random.default_rng(seed=42 + dist_idx)
```

(continues on next page)

(continued from previous page)

```
        paths_data = []

        # Generate paths
        for path in range(n_background + n_highlight):
            log_L_path, log_L, n = [0.0], 0.0, 0

            while True:
                z = rng.beta(params.a0, params.b0) if true_f0 \
                    else rng.beta(params.a1, params.b1)
                n += 1
                log_L += np.log(f1(z)) - np.log(f0(z))
                log_L_path.append(log_L)

                if log_L >= logA or log_L <= logB:
                    paths_data.append((log_L_path, n, log_L >= logA))
                    break

        # Plot background paths
        for path, _, decision in paths_data[:n_background]:
            ax.plot(range(len(path)), path, color='C1' if decision else 'C0',
                   alpha=0.2, linewidth=0.5)

        # Plot highlighted paths with labels
        for i, (path, _, decision) in enumerate(paths_data[n_background:]):
            ax.plot(range(len(path)), path, color='C1' if decision else 'C0',
                   alpha=0.8, linewidth=1.5,
                   label='reject $H_0$' if decision and i == 0 else (
                         'accept $H_0$' if not decision and i == 0 else ''))

        # Add threshold lines and formatting
        ax.axhline(y=logA, color='C1', linestyle='--', linewidth=2,
                  label=f'$\\log A = {logA:.2f}$')
        ax.axhline(y=logB, color='C0', linestyle='--', linewidth=2,
                  label=f'$\\log B = {logB:.2f}$')
        ax.axhline(y=0, color='black', linestyle='-', alpha=0.5, linewidth=1)

        ax.set_xlabel(r'$n$')
        ax.set_ylabel(r'$\log(L_n)$')
        ax.set_title(title, fontsize=20)
        ax.legend(fontsize=18, loc='center right')

        y_margin = max(abs(logA), abs(logB)) * 0.2
        ax.set_ylim(logB - y_margin, logA + y_margin)

    plt.tight_layout()
    plt.show()

plot_likelihood_paths(params_3, n_highlight=10, n_background=100)
```



Next, let's adjust the decision thresholds A and B and examine how the mean stopping time and the type I and type II error rates change.

In the code below, we adjust Wald's rule by adjusting the thresholds A and B using factors A_f and B_f.

```
@njit(parallel=True)
def run_adjusted_thresholds(a0, b0, a1, b1, α, β, N, seed, A_f, B_f):
    """SPRT simulation with adjusted thresholds."""

    # Calculate original thresholds
    A_original = (1 - β) / α
    B_original = β / (1 - α)

    # Apply adjustment factors
    A_adj = A_original * A_f
    B_adj = B_original * B_f
    logA = np.log(A_adj)
    logB = np.log(B_adj)

    # Pre-allocate arrays
    stopping_times = np.zeros(N, dtype=np.int64)
    decisions_h0 = np.zeros(N, dtype=np.bool_)
    truth_h0 = np.zeros(N, dtype=np.bool_)

    # Run simulations in parallel
    for i in prange(N):
        true_f0 = (i % 2 == 0)
        truth_h0[i] = true_f0

        n, accept_f0 = sprt_single_run(a0, b0, a1, b1,
                                       logA, logB, true_f0, seed + i)
        stopping_times[i] = n
        decisions_h0[i] = accept_f0

    return stopping_times, decisions_h0, truth_h0, A_adj, B_adj

def run_adjusted(params, A_f=1.0, B_f=1.0):
```

(continues on next page)

(continued from previous page)

```
    """Wrapper to run SPRT with adjusted A and B thresholds."""

    stopping_times, decisions_h0, truth_h0, A_adj, B_adj = run_adjusted_thresholds(
        params.a0, params.b0, params.a1, params.b1,
        params.α, params.β, params.N, params.seed, A_f, B_f
    )
    truth_h0_bool = truth_h0.astype(bool)
    decisions_h0_bool = decisions_h0.astype(bool)

    # Calculate error rates
    type_I = np.sum(truth_h0_bool
                    & ~decisions_h0_bool) / np.sum(truth_h0_bool)
    type_II = np.sum(~truth_h0_bool
                    & decisions_h0_bool) / np.sum(~truth_h0_bool)

    return {
        'stopping_times': stopping_times,
        'type_I': type_I,
        'type_II': type_II,
        'A_used': A_adj,
        'B_used': B_adj
    }

adjustments = [
    (5.0, 0.5),
    (1.0, 1.0),
    (0.3, 3.0),
    (0.2, 5.0),
    (0.15, 7.0),
]

results_table = []
for A_f, B_f in adjustments:
    result = run_adjusted(params_2, A_f, B_f)
    results_table.append([
        A_f, B_f,
        f"{result['stopping_times'].mean():.1f}",
        f"{result['type_I']:.3f}",
        f"{result['type_II']:.3f}"
    ])

df = pd.DataFrame(results_table,
                 columns=["A_f", "B_f", "mean stop time",
                          "Type I error", "Type II error"])
df = df.set_index(["A_f", "B_f"])
df
```

```
          mean stop time Type I error Type II error
A_f  B_f
5.00 0.5            16.1        0.006         0.036
1.00 1.0            11.1        0.033         0.070
0.30 3.0             5.5        0.086         0.195
0.20 5.0             3.4        0.120         0.304
0.15 7.0             2.2        0.146         0.410
```

Let's pause and think about the table more carefully by referring back to (26.1).

Recall that $A = \frac{1-\beta}{\alpha}$ and $B = \frac{\beta}{1-\alpha}$.

When we multiply A by a factor less than 1 (making A smaller), we are effectively making it easier to reject the null hypothesis H_0.

This increases the probability of Type I errors.

When we multiply B by a factor greater than 1 (making B larger), we are making it easier to accept the null hypothesis H_0.

This increases the probability of Type II errors.

The table confirms this intuition: as A decreases and B increases from their optimal Wald values, both Type I and Type II error rates increase, while the mean stopping time decreases.

26.7 Related lectures

We'll dig deeper into some of the ideas used here in the following earlier and later lectures:

- In *this sequel*, we reformulate the problem from the perspective of a **Bayesian statistician** who views parameters as vectors of random variables that are jointly distributed with the observables they are concerned about.
- The concept of **exchangeability**, which underlies much of statistical learning, is explored in depth in our *lecture on exchangeable random variables*.
- For a deeper understanding of likelihood ratio processes and their role in frequentist and Bayesian statistical theories, see *Likelihood Ratio Processes*.
- Building on that foundation, *Likelihood Ratio Processes and Bayesian Learning* examines the role of likelihood ratio processes in **Bayesian learning**.
- Finally, *this later lecture* revisits the subject discussed here and examines whether the frequentist decision rule that the Navy ordered the captain to use would perform better or worse than Abraham Wald's sequential decision rule.

26.8 Exercises

In the two exercises below, please try to rewrite the entire SPRT suite in this lecture.

Exercise 26.8.1

In the first exercise, we apply the sequential probability ratio test to distinguish two models generated by 3-state Markov chains

(For a review on likelihood ratio processes for Markov chains, see *this section*.)

Consider distinguishing between two 3-state Markov chain models using Wald's sequential probability ratio test.

You have competing hypotheses about the transition probabilities:

- H_0: The chain follows transition matrix $P^{(0)}$
- H_1: The chain follows transition matrix $P^{(1)}$

Given transition matrices:

$$P^{(0)} = \begin{bmatrix} 0.7 & 0.2 & 0.1 \\ 0.3 & 0.5 & 0.2 \\ 0.1 & 0.3 & 0.6 \end{bmatrix}, \quad P^{(1)} = \begin{bmatrix} 0.5 & 0.3 & 0.2 \\ 0.2 & 0.6 & 0.2 \\ 0.2 & 0.2 & 0.6 \end{bmatrix}$$

For a sequence of observations $(x_0, x_1, \ldots, x_t)$, the likelihood ratio is:

$$\Lambda_t = \frac{\pi^{(1)}_{x_0}}{\pi^{(0)}_{x_0}} \prod_{s=1}^{t} \frac{P^{(1)}_{x_{s-1},x_s}}{P^{(0)}_{x_{s-1},x_s}}$$

where $\pi^{(i)}$ is the stationary distribution under hypothesis i.

Tasks:

1. Implement the likelihood ratio computation for Markov chains
2. Implement Wald's sequential test with Type I error $\alpha = 0.05$ and Type II error $\beta = 0.10$
3. Run 1000 simulations under each hypothesis and compute empirical error rates
4. Analyze the distribution of stopping times

The test stops when:

- $\Lambda_t \geq A = \frac{1-\beta}{\alpha} = 18$: Reject H_0
- $\Lambda_t \leq B = \frac{\beta}{1-\alpha} = 0.105$: Accept H_0

Solution

Below is one solution to the exercise.

In the lecture, we write the code more verbosely to illustrate the concepts clearly.

In the code below, we simplified some of the code structure for a shorter presentation.

First we define the parameters for the Markov chain SPRT

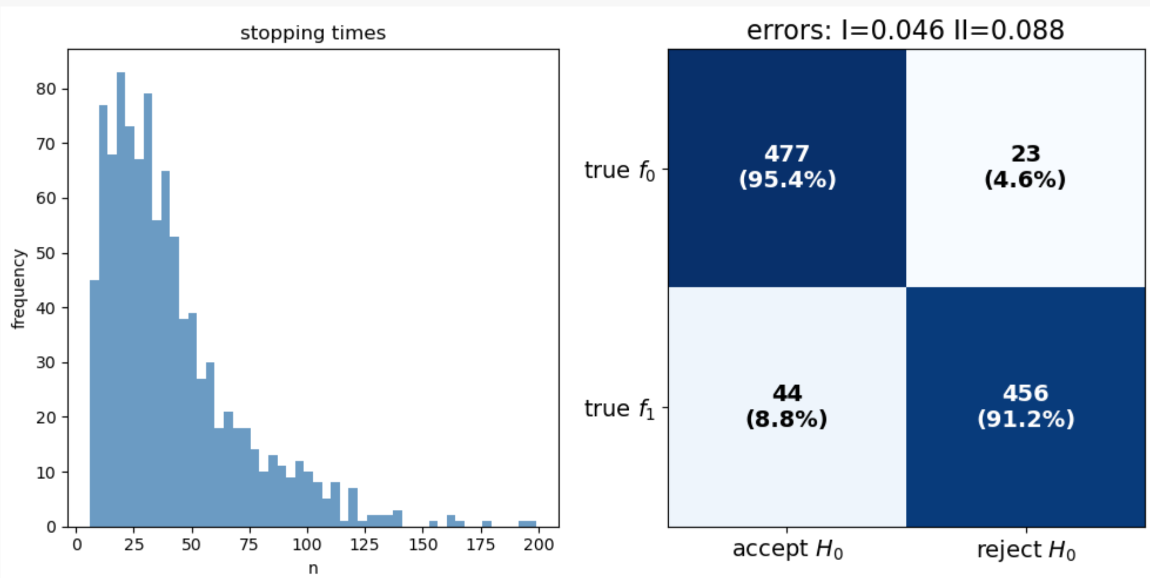
```
MarkovSPRTParams = namedtuple('MarkovSPRTParams',
            ['α', 'β', 'P_0', 'P_1', 'N', 'seed'])

def compute_stationary_distribution(P):
    """Compute stationary distribution of transition matrix P."""
    eigenvalues, eigenvectors = np.linalg.eig(P.T)
    idx = np.argmin(np.abs(eigenvalues - 1))
    pi = np.real(eigenvectors[:, idx])
    return pi / pi.sum()

@njit
def simulate_markov_chain(P, pi_0, T, seed):
    """Simulate a Markov chain path."""
    np.random.seed(seed)
    path = np.zeros(T, dtype=np.int32)

    cumsum_pi = np.cumsum(pi_0)
    path[0] = np.searchsorted(cumsum_pi, np.random.uniform())

    for t in range(1, T):
        cumsum_row = np.cumsum(P[path[t-1]])
        path[t] = np.searchsorted(cumsum_row, np.random.uniform())

    return path
```

Here we define the function that runs SPRT for Markov chains

```
@njit
def markov_sprt_single_run(P_0, P_1, π_0, π_1,
                           logA, logB, true_P, true_π, seed):
    """Run single SPRT for Markov chains."""
    max_n = 10000
    path = simulate_markov_chain(true_P, true_π, max_n, seed)

    log_L = np.log(π_1[path[0]] / π_0[path[0]])
    if log_L >= logA: return 1, False
    if log_L <= logB: return 1, True

    for t in range(1, max_n):
        prev_state, curr_state = path[t-1], path[t]
        p_1, p_0 = P_1[prev_state, curr_state], P_0[prev_state, curr_state]

        if p_0 > 0:
            log_L += np.log(p_1 / p_0)
        elif p_1 > 0:
            log_L = np.inf

        if log_L >= logA: return t+1, False
        if log_L <= logB: return t+1, True

    return max_n, log_L < 0

def run_markov_sprt(params):
    """Run SPRT for Markov chains."""
    π_0 = compute_stationary_distribution(params.P_0)
    π_1 = compute_stationary_distribution(params.P_1)
    A, B, logA, logB = compute_wald_thresholds(params.α, params.β)

    stopping_times = np.zeros(params.N, dtype=np.int64)
    decisions_h0 = np.zeros(params.N, dtype=bool)
    truth_h0 = np.zeros(params.N, dtype=bool)

    for i in range(params.N):
        true_P, true_π = (params.P_0, π_0) if i % 2 == 0 else (params.P_1, π_1)
        truth_h0[i] = i % 2 == 0

        n, accept_h0 = markov_sprt_single_run(
            params.P_0, params.P_1, π_0, π_1, logA, logB,
            true_P, true_π, params.seed + i)

        stopping_times[i] = n
        decisions_h0[i] = accept_h0

    type_I = np.sum(truth_h0 & ~decisions_h0) / np.sum(truth_h0)
    type_II = np.sum(~truth_h0 & decisions_h0) / np.sum(~truth_h0)

    return {
        'stopping_times': stopping_times, 'decisions_h0': decisions_h0,
        'truth_h0': truth_h0, 'type_I': type_I, 'type_II': type_II
    }
```

Now we can run the SPRT for the Markov chain models and visualize the results

```python
# Run Markov chain SPRT
P_0 = np.array([[0.7, 0.2, 0.1],
                [0.3, 0.5, 0.2],
                [0.1, 0.3, 0.6]])

P_1 = np.array([[0.5, 0.3, 0.2],
                [0.2, 0.6, 0.2],
                [0.2, 0.2, 0.6]])

params_markov = MarkovSPRTParams(α=0.05, β=0.10,
                        P_0=P_0, P_1=P_1, N=1000, seed=42)
results_markov = run_markov_sprt(params_markov)


fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 5))

ax1.hist(results_markov['stopping_times'],
            bins=50, color="steelblue", alpha=0.8)
ax1.set_title("stopping times")
ax1.set_xlabel("n")
ax1.set_ylabel("frequency")

plot_confusion_matrix(results_markov, ax2)

plt.tight_layout()
plt.show()
```



Exercise 26.8.2

In this exercise, apply Wald's sequential test to distinguish between two VAR(1) models with different dynamics and noise structures.

For a review of the likelihood ratio process with VAR models, see *Likelihood Processes For VAR Models*.

Given VAR models under each hypothesis:

- H_0: $x_{t+1} = A^{(0)} x_t + C^{(0)} w_{t+1}$
- H_1: $x_{t+1} = A^{(1)} x_t + C^{(1)} w_{t+1}$

where $w_t \sim \mathcal{N}(0, I)$ and:

$$A^{(0)} = \begin{bmatrix} 0.8 & 0.1 \\ 0.2 & 0.7 \end{bmatrix}, \quad C^{(0)} = \begin{bmatrix} 0.3 & 0.1 \\ 0.1 & 0.3 \end{bmatrix}$$

$$A^{(1)} = \begin{bmatrix} 0.6 & 0.2 \\ 0.3 & 0.5 \end{bmatrix}, \quad C^{(1)} = \begin{bmatrix} 0.4 & 0 \\ 0 & 0.4 \end{bmatrix}$$

Tasks:

1. Implement the VAR likelihood ratio using the functions from the VAR lecture
2. Implement Wald's sequential test with $\alpha = 0.05$ and $\beta = 0.10$
3. Analyze performance under both hypotheses and with model misspecification
4. Compare with the Markov chain case in terms of stopping times and accuracy

Solution

Below is one solution to the exercise.

First we define the parameters for the VAR models and simulator

```
VARSPRTParams = namedtuple('VARSPRTParams',
            ['α', 'β', 'A_0', 'C_0', 'A_1', 'C_1', 'N', 'seed'])

def create_var_model(A, C):
    """Create VAR model."""
    μ_0 = np.zeros(A.shape[0])
    CC = C @ C.T
    Σ_0 = sp.linalg.solve_discrete_lyapunov(A, CC)

    CC_inv = np.linalg.inv(CC + 1e-10 * np.eye(CC.shape[0]))
    Σ_0_inv = np.linalg.inv(Σ_0 + 1e-10 * np.eye(Σ_0.shape[0]))

    return {
        'A': A, 'C': C, 'μ_0': μ_0, 'Σ_0': Σ_0,
        'CC_inv': CC_inv, 'Σ_0_inv': Σ_0_inv,
        'log_det_CC': np.log(
            np.linalg.det(CC + 1e-10 * np.eye(CC.shape[0]))),
        'log_det_Σ_0': np.log(
            np.linalg.det(Σ_0 + 1e-10 * np.eye(Σ_0.shape[0])))
    }
```

Now we define the likelihood ratio for the VAR models and the SPRT function similar to the Markov chain case

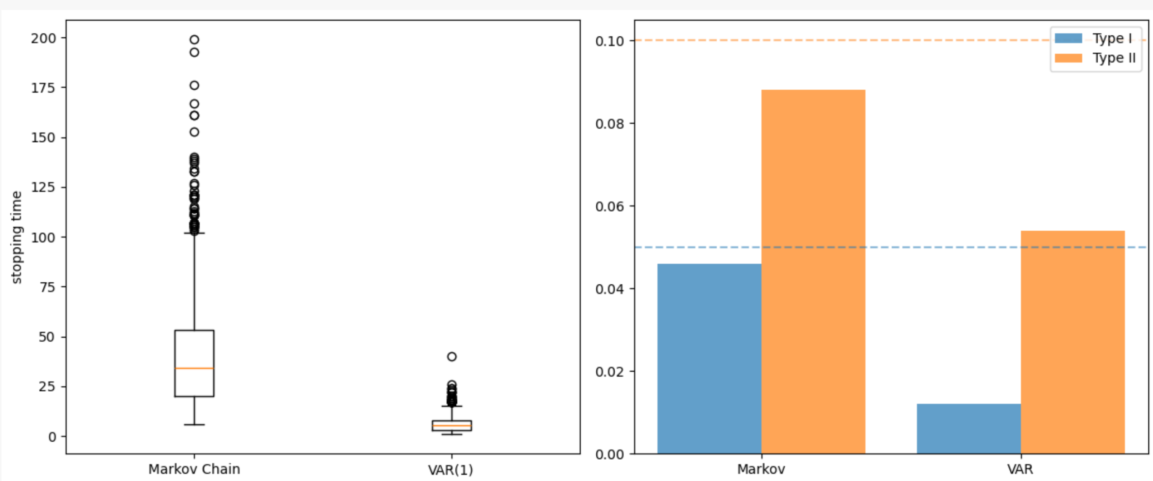
```
def var_log_likelihood(x_curr, x_prev, model, initial=False):
    """Compute VAR log-likelihood."""
    n = len(x_curr)
    if initial:
        diff = x_curr - model['μ_0']
        return -0.5 * (n * np.log(2 * np.pi) + model['log_det_Σ_0'] +
                      diff @ model['Σ_0_inv'] @ diff)
    else:
        diff = x_curr - model['A'] @ x_prev
```

```
        return -0.5 * (n * np.log(2 * np.pi) + model['log_det_CC'] +
                       diff @ model['CC_inv'] @ diff)

def var_sprt_single_run(model_0, model_1, model_true,
                        logA, logB, seed):
    """Single VAR SPRT run."""
    np.random.seed(seed)
    max_T = 500

    # Generate VAR path
    Σ_chol = np.linalg.cholesky(model_true['Σ_0'])
    x = model_true['μ_0'] + Σ_chol @ np.random.randn(
            len(model_true['μ_0']))

    # Initial likelihood ratio
    log_L = (var_log_likelihood(x, None, model_1, True) -
             var_log_likelihood(x, None, model_0, True))

    if log_L >= logA: return 1, False
    if log_L <= logB: return 1, True

    # Sequential updates
    for t in range(1, max_T):
        x_prev = x.copy()
        w = np.random.randn(model_true['C'].shape[1])
        x = model_true['A'] @ x + model_true['C'] @ w

        log_L += (var_log_likelihood(x, x_prev, model_1) -
                  var_log_likelihood(x, x_prev, model_0))

        if log_L >= logA: return t+1, False
        if log_L <= logB: return t+1, True

    return max_T, log_L < 0

def run_var_sprt(params):
    """Run VAR SPRT."""

    model_0 = create_var_model(params.A_0, params.C_0)
    model_1 = create_var_model(params.A_1, params.C_1)
    A, B, logA, logB = compute_wald_thresholds(params.α, params.β)

    stopping_times = np.zeros(params.N)
    decisions_h0 = np.zeros(params.N, dtype=bool)
    truth_h0 = np.zeros(params.N, dtype=bool)

    for i in range(params.N):
        model_true = model_0 if i % 2 == 0 else model_1
        truth_h0[i] = i % 2 == 0

        n, accept_h0 = var_sprt_single_run(model_0, model_1, model_true,
                                           logA, logB, params.seed + i)
        stopping_times[i] = n
        decisions_h0[i] = accept_h0

    type_I = np.sum(truth_h0 & ~decisions_h0) / np.sum(truth_h0)
    type_II = np.sum(~truth_h0 & decisions_h0) / np.sum(~truth_h0)
```

```
    return {'stopping_times': stopping_times,
            'decisions_h0': decisions_h0,
            'truth_h0': truth_h0,
            'type_I': type_I, 'type_II': type_II}
```

Let's run SPRT and visualize the results

```
# Run VAR SPRT
A_0 = np.array([[0.8, 0.1],
                [0.2, 0.7]])
C_0 = np.array([[0.3, 0.1],
                [0.1, 0.3]])
A_1 = np.array([[0.6, 0.2],
                [0.3, 0.5]])
C_1 = np.array([[0.4, 0.0],
                [0.0, 0.4]])

params_var = VARSPRTParams(α=0.05, β=0.10,
                A_0=A_0, C_0=C_0, A_1=A_1, C_1=C_1,
                N=1000, seed=42)
results_var = run_var_sprt(params_var)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

ax1.boxplot([results_markov['stopping_times'],
             results_var['stopping_times']],
           tick_labels=['Markov Chain', 'VAR(1)'])
ax1.set_ylabel('stopping time')

x = np.arange(2)
ax2.bar(x - 0.2, [results_markov['type_I'], results_var['type_I']],
        0.4, label='Type I', alpha=0.7)
ax2.bar(x + 0.2, [results_markov['type_II'], results_var['type_II']],
        0.4, label='Type II', alpha=0.7)
ax2.axhline(y=0.05, linestyle='--', alpha=0.5, color='C0')
ax2.axhline(y=0.10, linestyle='--', alpha=0.5, color='C1')
ax2.set_xticks(x), ax2.set_xticklabels(['Markov', 'VAR'])
ax2.legend()
plt.tight_layout()
plt.show()
```



CHAPTER

TWENTYSEVEN

A BAYESIAN FORMULATION OF FRIEDMAN AND WALD'S PROBLEM

Contents

27.1 Overview

This lecture revisits the statistical decision problem presented to Milton Friedman and W. Allen Wallis during World War II when they were analysts at the U.S. Government's Statistical Research Group at Columbia University.

In *an earlier lecture*, we described how Abraham Wald [Wald, 1947] solved the problem by extending frequentist hypothesis testing techniques and formulating the problem sequentially.

Note

Wald's idea of formulating the problem sequentially created links to the **dynamic programming** that Richard Bellman developed in the 1950s.

As we learned in *Elementary Probability with Matrices* and *Two Meanings of Probability*, a frequentist statistician views a probability distribution as measuring relative frequencies of a statistic that he anticipates constructing from a very long sequence of i.i.d. draws from a known probability distribution.

That known probability distribution is his 'hypothesis'.

A frequentist statistician studies the distribution of that statistic under that known probability distribution

- when the distribution is a member of a set of parameterized probability distributions, his hypothesis takes the form of a particular parameter vector.
- this is what we mean when we say that the frequentist statistician 'conditions on the parameters'
- he regards the parameters as fixed numbers that are known to nature, but not to him.

- the statistician copes with his ignorance of those parameters by constructing type I and type II errors associated with frequentist hypothesis testing.

In this lecture, we reformulate Friedman and Wald's problem by transforming our point of view from the 'objective' frequentist perspective of *the lecture on Wald's sequential analysis* to an explicitly 'subjective' perspective taken by a Bayesian decision maker who regards parameters not as fixed numbers but as (hidden) random variables that are jointly distributed with the random variables that he can observe by sampling from that joint distribution.

To form that joint distribution, the Bayesian statistician supplements the conditional distributions used by the frequentist statistician with a prior probability distribution over the parameters that represents his personal, subjective opinion about them.

That lets the Bayesian statistician calculate the joint distribution that he requires to calculate the conditional distributions that he wants.

To proceed in this way, we endow our decision maker with

- an initial prior subjective probability $\pi_{-1} \in (0, 1)$ that nature uses to generate $\{z_k\}$ as a sequence of i.i.d. draws from f_1 rather than f_0.
- faith in Bayes' law as a way to revise his subjective beliefs as observations on $\{z_k\}$ sequence arrive.
- a loss function that tells how the decision maker values type I and type II errors.

In our *previous frequentist version*, key ideas in play were:

- Type I and type II statistical errors
  - a type I error occurs when you reject a null hypothesis that is true
  - a type II error occurs when you accept a null hypothesis that is false
- Abraham Wald's **sequential probability ratio test**
- The **power** of a statistical test
- The **critical region** of a statistical test
- A **uniformly most powerful test**

In this lecture about a Bayesian reformulation of the problem, additional ideas at work are

- an initial prior probability π_{-1} that model f_1 generates the data
- Bayes' Law
- a sequence of posterior probabilities that model f_1 is generating the data
- dynamic programming

This lecture uses ideas studied in the lectures on *likelihood ratio processes*, *their roles in Bayesian learning*, and *this lecture on exchangeability*.

We'll begin with some imports:

```python
import numpy as np
import matplotlib.pyplot as plt
from numba import jit, prange, float64, int64
from numba.experimental import jitclass
from math import gamma
```

27.2 A Dynamic Programming Approach

The following presentation of the problem closely follows Dmitri Bertsekas's treatment in **Dynamic Programming and Stochastic Control** [Bertsekas, 1975].

A decision-maker can observe a sequence of draws of a random variable z.

He (or she) wants to know which of two probability distributions f_0 or f_1 governs z.

Conditional on knowing that successive observations are drawn from distribution f_0, the sequence of random variables is independently and identically distributed (IID).

Conditional on knowing that successive observations are drawn from distribution f_1, the sequence of random variables is also independently and identically distributed (IID).

But the observer does not know which of the two distributions generated the sequence.

For reasons explained in Exchangeability and Bayesian Updating, this means that the sequence is not IID.

The observer has something to learn, namely, whether the observations are drawn from f_0 or from f_1.

The decision maker wants to decide which of the two distributions is generating outcomes.

We adopt a Bayesian formulation.

The decision maker begins with a prior probability

$$\pi_{-1} = \mathbb{P}\{f = f_1 \mid \text{ no observations}\} \in (0, 1)$$

> **Note**
>
> In Bertsekas [1975], the belief is associated with the distribution f_0, but here we associate the belief with the distribution f_1 to match the discussions in *the lecture on Wald's sequential analysis*.

After observing $k+1$ observations $z_k, z_{k-1}, \dots, z_0$, he updates his personal probability that the observations are described by distribution f_1 to

$$\pi_k = \mathbb{P}\{f = f_1 \mid z_k, z_{k-1}, \dots, z_0\}$$

which is calculated recursively by applying Bayes' law:

$$\pi_{k+1} = \frac{\pi_k f_1(z_{k+1})}{(1 - \pi_k) f_0(z_{k+1}) + \pi_k f_1(z_{k+1})}, \quad k = -1, 0, 1, \dots$$

After observing $z_k, z_{k-1}, \dots, z_0$, the decision-maker believes that z_{k+1} has probability distribution

$$f_{\pi_k}(v) = (1 - \pi_k) f_0(v) + \pi_k f_1(v),$$

which is a mixture of distributions f_0 and f_1, with the weight on f_1 being the posterior probability that $f = f_1$[1].

To illustrate such a distribution, let's inspect some mixtures of beta distributions.

The density of a beta probability distribution with parameters a and b is

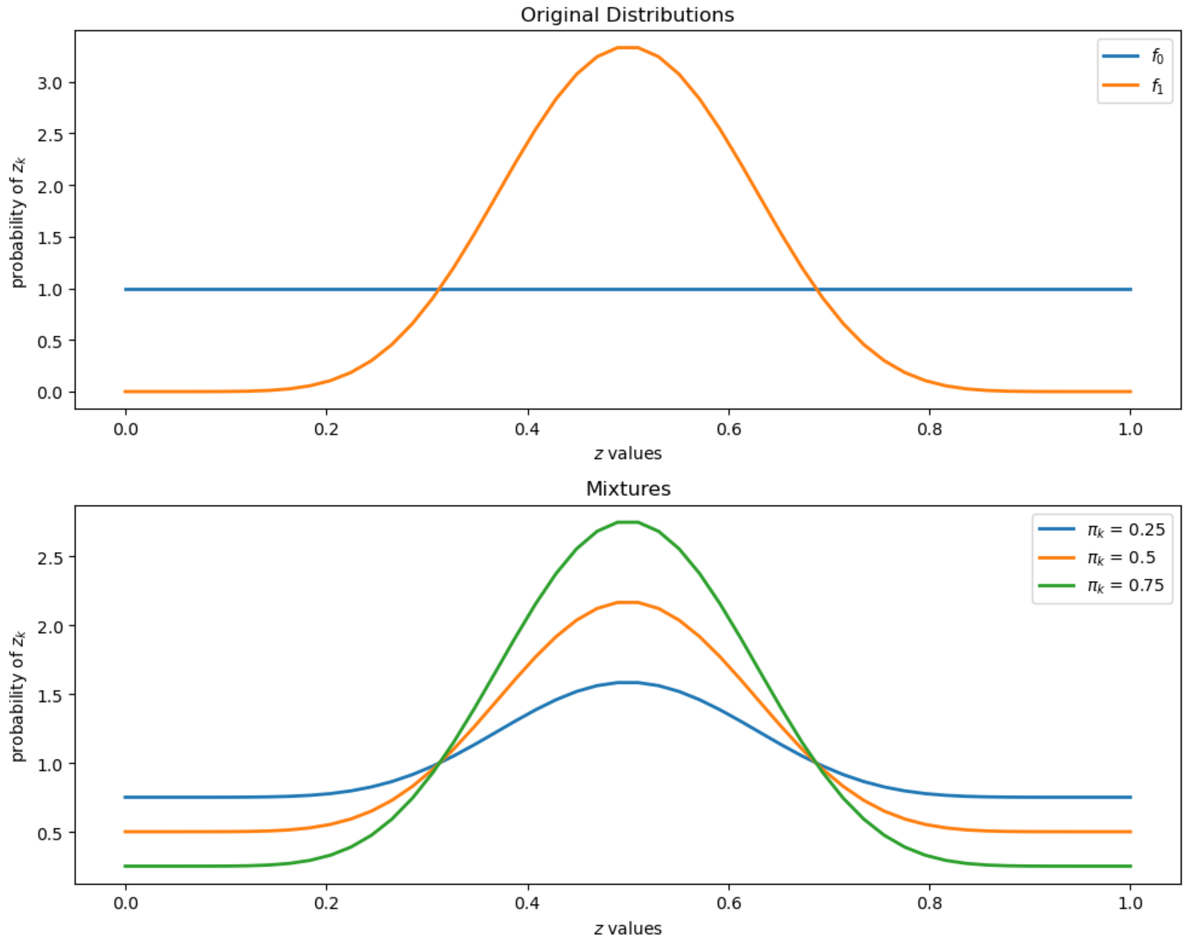
$$f(z; a, b) = \frac{\Gamma(a + b) z^{a-1} (1 - z)^{b-1}}{\Gamma(a) \Gamma(b)} \quad \text{where} \quad \Gamma(t) := \int_0^\infty x^{t-1} e^{-x} dx$$

The next figure shows two beta distributions in the top panel.

The bottom panel presents mixtures of these distributions, with various mixing probabilities π_k

[1] The decision maker acts as if he believes that the sequence of random variables $[z_0, z_1, \dots]$ is *exchangeable*. See Exchangeability and Bayesian Updating and [Kreps, 1988] chapter 11, for discussions of exchangeability.

```python
@jit
def p(x, a, b):
    r = gamma(a + b) / (gamma(a) * gamma(b))
    return r * x**(a-1) * (1 - x)**(b-1)

f0 = lambda x: p(x, 1, 1)
f1 = lambda x: p(x, 9, 9)
grid = np.linspace(0, 1, 50)

fig, axes = plt.subplots(2, figsize=(10, 8))

axes[0].set_title("Original Distributions")
axes[0].plot(grid, f0(grid), lw=2, label="$f_0$")
axes[0].plot(grid, f1(grid), lw=2, label="$f_1$")

axes[1].set_title("Mixtures")
for π in 0.25, 0.5, 0.75:
    y = (1 - π) * f0(grid) + π * f1(grid)
    axes[1].plot(grid, y, lw=2, label=fr"$\pi_k$ = {π}")

for ax in axes:
    ax.legend()
    ax.set(xlabel="$z$ values", ylabel="probability of $z_k$")

plt.tight_layout()
plt.show()
```



27.2.1 Losses and Costs

After observing $z_k, z_{k-1}, \ldots, z_0$, the decision-maker chooses among three distinct actions:

- He decides that $f = f_0$ and draws no more z's
- He decides that $f = f_1$ and draws no more z's
- He postpones deciding now and instead chooses to draw a z_{k+1}

Associated with these three actions, the decision-maker can suffer three kinds of losses:

- A loss L_0 if he decides $f = f_0$ when actually $f = f_1$
- A loss L_1 if he decides $f = f_1$ when actually $f = f_0$
- A cost c if he postpones deciding and chooses instead to draw another z

27.2.2 Digression on Type I and Type II Errors

If we regard $f = f_0$ as a null hypothesis and $f = f_1$ as an alternative hypothesis, then L_1 and L_0 are losses associated with two types of statistical errors

- a type I error is an incorrect rejection of a true null hypothesis (a "false positive")
- a type II error is a failure to reject a false null hypothesis (a "false negative")

So when we treat $f = f_0$ as the null hypothesis

- We can think of L_1 as the loss associated with a type I error.
- We can think of L_0 as the loss associated with a type II error.

27.2.3 Intuition

Before proceeding, let's try to guess what an optimal decision rule might look like.

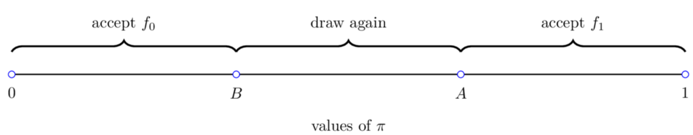
Suppose at some given point in time that π is close to 1.

Then our prior beliefs and the evidence so far point strongly to $f = f_1$.

If, on the other hand, π is close to 0, then $f = f_0$ is strongly favored.

Finally, if π is in the middle of the interval $[0, 1]$, then we are confronted with more uncertainty.

This reasoning suggests a sequential decision rule that we illustrate in the following figure:



As we'll see, this is indeed the correct form of the decision rule.

Our problem is to determine threshold values A, B that somehow depend on the parameters described above.

You might like to pause at this point and try to predict the impact of a parameter such as c or L_0 on A or B.

27.2.4 A Bellman Equation

Let $J(\pi)$ be the total loss for a decision-maker with current belief π who chooses optimally.

Principles of **dynamic programming** teach us that an optimal loss function J satisfies the following the Bellman functional equation

$$J(\pi) = \min \left\{ \underbrace{\pi L_0}_{\text{accept } f_0}, \underbrace{(1-\pi) L_1}_{\text{accept } f_1}, \underbrace{c + \mathbb{E}[J(\pi')]}_{\text{draw again}} \right\} \tag{27.1}$$

where π' is the random variable defined by Bayes' Law

$$\pi' = \kappa(z', \pi) = \frac{\pi f_1(z')}{(1-\pi) f_0(z') + \pi f_1(z')}$$

when π is fixed and z' is drawn from the current best guess, which is the distribution f defined by

$$f_\pi(v) = (1-\pi) f_0(v) + \pi f_1(v)$$

In the Bellman equation, minimization is over three actions:

1. Accept the hypothesis that $f = f_0$
2. Accept the hypothesis that $f = f_1$
3. Postpone deciding and draw again

We can represent the Bellman equation as

$$J(\pi) = \min\left\{\pi L_0,\ (1-\pi)L_1,\ h(\pi)\right\} \tag{27.2}$$

where $\pi \in [0, 1]$ and

- πL_0 is the expected loss associated with accepting f_0 (i.e., the cost of making a type II error).
- $(1-\pi)L_1$ is the expected loss associated with accepting f_1 (i.e., the cost of making a type I error).
- $h(\pi) := c + \mathbb{E}[J(\pi')]$; this is the continuation value; i.e., the expected cost associated with drawing one more z.

The optimal decision rule is characterized by two numbers $A, B \in (0,1) \times (0,1)$ that satisfy

$$\pi L_0 < \min\{(1-\pi)L_1, c + \mathbb{E}[J(\pi')]\} \text{ if } \pi \leq B$$

and

$$(1-\pi)L_1 < \min\{\pi L_0, c + \mathbb{E}[J(\pi')]\} \text{ if } \pi \geq A$$

The optimal decision rule is then

$$\begin{aligned} \text{accept } f = f_1 \text{ if } \pi \geq A \\ \text{accept } f = f_0 \text{ if } \pi \leq B \\ \text{draw another } z \text{ if } B < \pi < A \end{aligned}$$

Our aim is to compute the cost function J as well as the associated cutoffs A and B.

To help make our computations more manageable, we can use (27.2) to write the continuation cost $h(\pi)$ as

$$\begin{aligned} h(\pi) &= c + \mathbb{E}[J(\pi')] \\ &= c + \mathbb{E}_{\pi'} \min\{\pi' L_0, (1-\pi')L_1, h(\pi')\} \\ &= c + \int \min\{\kappa(z', \pi)L_0, (1-\kappa(z', \pi))L_1, h(\kappa(z', \pi))\} f_\pi(z')dz' \end{aligned} \tag{27.3}$$

The equality

$$h(\pi) = c + \int \min\{\kappa(z', \pi)L_0, (1-\kappa(z', \pi))L_1, h(\kappa(z', \pi))\} f_\pi(z')dz' \tag{27.4}$$

is an equation in an unknown function h.

> **Note**
>
> Such an equation is called a **functional equation**.

Using the functional equation, (27.4), for the continuation cost, we can back out optimal choices using the right side of (27.2).

This functional equation can be solved by taking an initial guess and iterating to find a fixed point.

Thus, we iterate with an operator Q, where

$$Qh(\pi) = c + \int \min\{\kappa(z', \pi)L_0, (1-\kappa(z', \pi))L_1, h(\kappa(z', \pi))\} f_\pi(z')dz'$$

27.3 Implementation

First, we will construct a `jitclass` to store the parameters of the model

```
wf_data = [('a0', float64),          # Parameters of beta distributions
           ('b0', float64),
           ('a1', float64),
           ('b1', float64),
           ('c', float64),           # Cost of another draw
           ('π_grid_size', int64),
           ('L0', float64),          # Cost of selecting f0 when f1 is true
           ('L1', float64),          # Cost of selecting f1 when f0 is true
           ('π_grid', float64[:]),
           ('mc_size', int64),
           ('z0', float64[:]),
           ('z1', float64[:])]
```

```
@jitclass(wf_data)
class WaldFriedman:

    def __init__(self,
                 c=1.25,
                 a0=1,
                 b0=1,
                 a1=3,
                 b1=1.2,
                 L0=25,
                 L1=25,
                 π_grid_size=200,
                 mc_size=1000):

        self.a0, self.b0 = a0, b0
        self.a1, self.b1 = a1, b1
        self.c, self.π_grid_size = c, π_grid_size
        self.L0, self.L1 = L0, L1
        self.π_grid = np.linspace(0, 1, π_grid_size)
        self.mc_size = mc_size

        self.z0 = np.random.beta(a0, b0, mc_size)
        self.z1 = np.random.beta(a1, b1, mc_size)

    def f0(self, x):

        return p(x, self.a0, self.b0)

    def f1(self, x):

        return p(x, self.a1, self.b1)

    def f0_rvs(self):
        return np.random.beta(self.a0, self.b0)

    def f1_rvs(self):
        return np.random.beta(self.a1, self.b1)

    def κ(self, z, π):
        """
```

(continues on next page)

(continued from previous page)

```
        Updates π using Bayes' rule and the current observation z
        """

        f0, f1 = self.f0, self.f1

        π_f0, π_f1 = (1 - π) * f0(z), π * f1(z)
        π_new = π_f1 / (π_f0 + π_f1)

        return π_new
```

As in *Optimal Savings III: Stochastic Returns*, to approximate a continuous value function

- We iterate at a finite grid of possible values of π.
- When we evaluate $\mathbb{E}[J(\pi')]$ between grid points, we use linear interpolation.

We define the operator function `Q` below.

```
@jit(nopython=True, parallel=True)
def Q(h, wf):

    c, π_grid = wf.c, wf.π_grid
    L0, L1 = wf.L0, wf.L1
    z0, z1 = wf.z0, wf.z1
    mc_size = wf.mc_size

    κ = wf.κ

    h_new = np.empty_like(π_grid)
    h_func = lambda p: np.interp(p, π_grid, h)

    for i in prange(len(π_grid)):
        π = π_grid[i]

        # Find the expected value of J by integrating over z
        integral_f0, integral_f1 = 0, 0
        for m in range(mc_size):
            π_0 = κ(z0[m], π)  # Draw z from f0 and update π
            integral_f0 += min(π_0 * L0, (1 - π_0) * L1, h_func(π_0))

            π_1 = κ(z1[m], π)  # Draw z from f1 and update π
            integral_f1 += min(π_1 * L0, (1 - π_1) * L1, h_func(π_1))

        integral = ((1 - π) * integral_f0 + π * integral_f1) / mc_size

        h_new[i] = c + integral

    return h_new
```

To solve the key functional equation, we will iterate using `Q` to find the fixed point

```
@jit
def solve_model(wf, tol=1e-4, max_iter=1000):
    """
    Compute the continuation cost function

    * wf is an instance of WaldFriedman
    """
```

(continues on next page)

(continued from previous page)

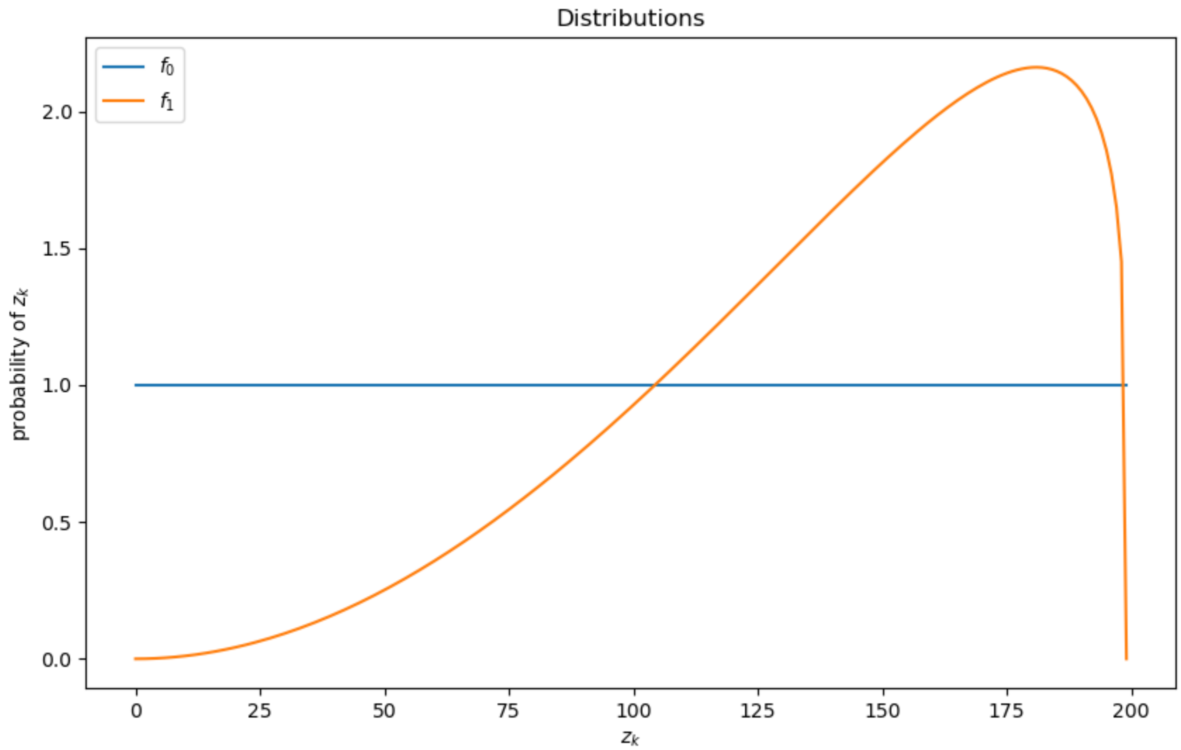
```
    # Set up loop
    h = np.zeros(len(wf.π_grid))
    i = 0
    error = tol + 1

    while i < max_iter and error > tol:
        h_new = Q(h, wf)
        error = np.max(np.abs(h - h_new))
        i += 1
        h = h_new

    if error > tol:
        print("Failed to converge!")

    return h_new
```

27.4 Analysis

Let's inspect outcomes.

We will be using the default parameterization with distributions like so

```
wf = WaldFriedman()

fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(wf.f0(wf.π_grid), label="$f_0$")
ax.plot(wf.f1(wf.π_grid), label="$f_1$")
ax.set(ylabel="probability of $z_k$", xlabel="$z_k$", title="Distributions")
ax.legend()

plt.show()
```



27.4.1 Cost Function

To solve the model, we will call our `solve_model` function

```
h_star = solve_model(wf)    # Solve the model
```

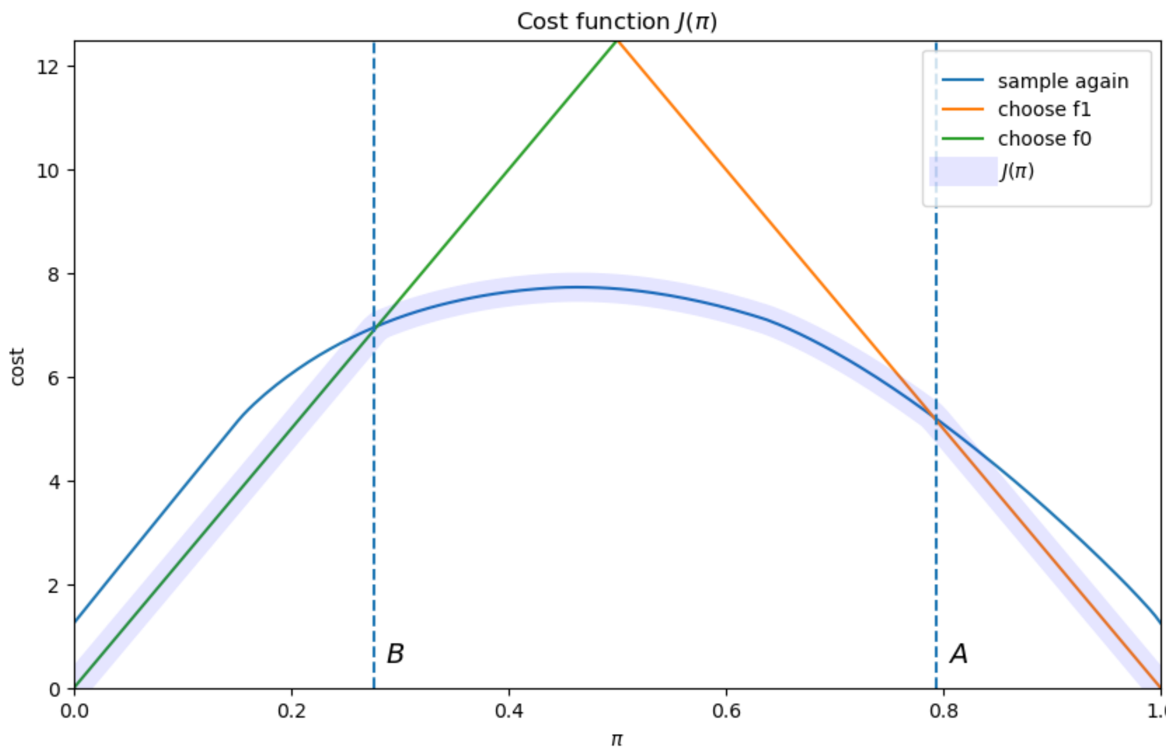
We will also set up a function to compute the cutoffs A and B and plot these on our cost function plot

```
@jit
def find_cutoff_rule(wf, h):

    """
    This function takes a continuation cost function and returns the
    corresponding cutoffs of where you transition between continuing and
    choosing a specific model
    """

    π_grid = wf.π_grid
    L0, L1 = wf.L0, wf.L1

    # Evaluate cost at all points on grid for choosing a model
    cost_f0 = π_grid * L0
    cost_f1 = (1 - π_grid) * L1

    # Find B: largest π where cost_f0 <= min(cost_f1, h)
    optimal_cost = np.minimum(np.minimum(cost_f0, cost_f1), h)
    choose_f0 = (cost_f0 <= cost_f1) & (cost_f0 <= h)

    if np.any(choose_f0):
```

(continues on next page)

(continued from previous page)

```
        B = π_grid[choose_f0][-1]  # Last point where we choose f0
    else:
        assert False, "No point where we choose f0"

    # Find A: smallest π where cost_f1 <= min(cost_f0, h)
    choose_f1 = (cost_f1 <= cost_f0) & (cost_f1 <= h)

    if np.any(choose_f1):
        A = π_grid[choose_f1][0]  # First point where we choose f1
    else:
        assert False, "No point where we choose f1"

    return (B, A)

B, A = find_cutoff_rule(wf, h_star)
cost_L0 = wf.π_grid * wf.L0
cost_L1 = (1 - wf.π_grid) * wf.L1

fig, ax = plt.subplots(figsize=(10, 6))

ax.plot(wf.π_grid, h_star, label='sample again')
ax.plot(wf.π_grid, cost_L1, label='choose f1')
ax.plot(wf.π_grid, cost_L0, label='choose f0')
ax.plot(wf.π_grid,
        np.amin(np.column_stack([h_star, cost_L0, cost_L1]),axis=1),
        lw=15, alpha=0.1, color='b', label=r'$J(\pi)$')

ax.annotate(r"$B$", xy=(B + 0.01, 0.5), fontsize=14)
ax.annotate(r"$A$", xy=(A + 0.01, 0.5), fontsize=14)

plt.vlines(B, 0, (1 - B) * wf.L1, linestyle="--")
plt.vlines(A, 0, A * wf.L0, linestyle="--")

ax.set(xlim=(0, 1), ylim=(0, 0.5 * max(wf.L0, wf.L1)), ylabel="cost",
       xlabel=r"$\pi$", title=r"Cost function $J(\pi)$")

plt.legend(borderpad=1.1)
plt.show()
```



The cost function J equals πL_0 for $\pi \leq B$, and $(1-\pi)L_1$ for $\pi \geq A$.

The slopes of the two linear pieces of the cost function $J(\pi)$ are determined by L_0 and $-L_1$.

The cost function J is smooth in the interior region, where the posterior probability assigned to f_1 is in the indecisive region $\pi \in (B, A)$.

The decision-maker continues to sample until the probability that he attaches to model f_1 falls below B or above A.

27.4.2 Simulations

The next figure shows the outcomes of 500 simulations of the decision process.

On the left is a histogram of **stopping times**, i.e., the number of draws of z_k required to make a decision.

The average number of draws is around 6.6.

On the right is the fraction of correct decisions at the stopping time.

In this case, the decision-maker is correct 80% of the time

```
def simulate(wf, true_dist, h_star, π_0=0.5):

    """
    This function takes an initial condition and simulates until it
    stops (when a decision is made)
    """

    f0, f1 = wf.f0, wf.f1
    f0_rvs, f1_rvs = wf.f0_rvs, wf.f1_rvs
    π_grid = wf.π_grid
    κ = wf.κ
```

(continues on next page)

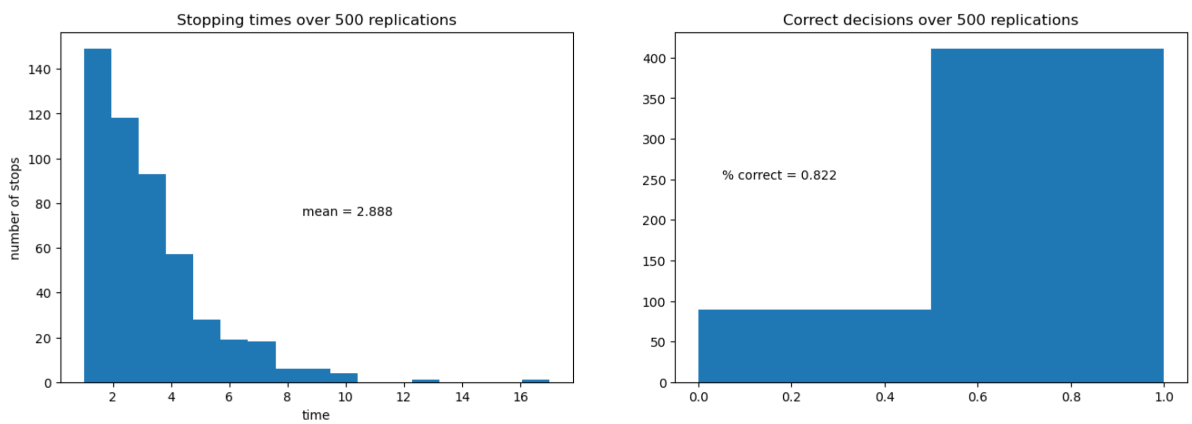
(continued from previous page)

```
    if true_dist == "f0":
        f, f_rvs = wf.f0, wf.f0_rvs
    elif true_dist == "f1":
        f, f_rvs = wf.f1, wf.f1_rvs

    # Find cutoffs
    B, A = find_cutoff_rule(wf, h_star)

    # Initialize a couple of useful variables
    decision_made = False
    π = π_0
    t = 0

    while decision_made is False:
        z = f_rvs()
        t = t + 1
        π = κ(z, π)
        if π < B:
            decision_made = True
            decision = 0
        elif π > A:
            decision_made = True
            decision = 1

    if true_dist == "f0":
        if decision == 0:
            correct = True
        else:
            correct = False

    elif true_dist == "f1":
        if decision == 1:
            correct = True
        else:
            correct = False

    return correct, π, t

def stopping_dist(wf, h_star, ndraws=250, true_dist="f0"):

    """
    Simulates repeatedly to get distributions of time needed to make a
    decision and how often they are correct
    """

    tdist = np.empty(ndraws, int)
    cdist = np.empty(ndraws, bool)

    for i in range(ndraws):
        correct, π, t = simulate(wf, true_dist, h_star)
        tdist[i] = t
        cdist[i] = correct

    return cdist, tdist

def simulation_plot(wf):
```

(continues on next page)

(continued from previous page)

```
    h_star = solve_model(wf)
    ndraws = 500
    cdist, tdist = stopping_dist(wf, h_star, ndraws)

    fig, ax = plt.subplots(1, 2, figsize=(16, 5))

    ax[0].hist(tdist, bins=np.max(tdist))
    ax[0].set_title(f"Stopping times over {ndraws} replications")
    ax[0].set(xlabel="time", ylabel="number of stops")
    ax[0].annotate(f"mean = {np.mean(tdist)}", xy=(max(tdist) / 2,
                   max(np.histogram(tdist, bins=max(tdist))[0]) / 2))

    ax[1].hist(cdist.astype(int), bins=2)
    ax[1].set_title(f"Correct decisions over {ndraws} replications")
    ax[1].annotate(f"% correct = {np.mean(cdist)}",
                   xy=(0.05, ndraws / 2))

    plt.show()

simulation_plot(wf)
```



27.4.3 Comparative Statics

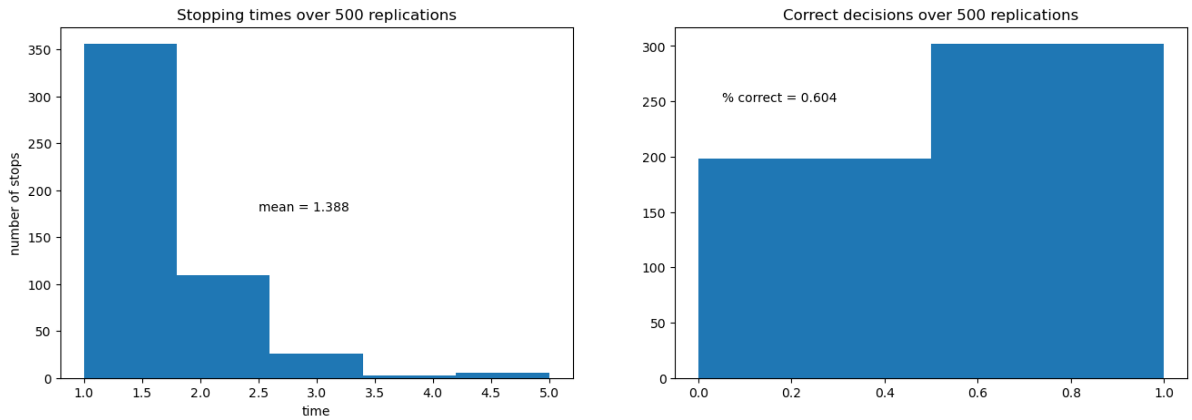
Now let's consider the following exercise.

We double the cost of drawing an additional observation.

Before you look, think about what will happen:

- Will the decision-maker be correct more or less often?
- Will he make decisions sooner or later?

```
wf = WaldFriedman(c=2.5)
simulation_plot(wf)
```



Increased cost per draw has induced the decision-maker to take fewer draws before deciding.

Because he decides with fewer draws, the percentage of time he is correct drops.

This leads to him having a higher expected loss when he puts equal weight on both models.

To facilitate comparative statics, we invite you to adjust the parameters of the model and investigate

- effects on the smoothness of the value function in the indecisive middle range as we increase the number of grid points in the piecewise linear approximation.
- effects of different settings for the cost parameters L_0, L_1, c, the parameters of two beta distributions f_0 and f_1, and the number of points and linear functions m to use in the piecewise continuous approximation to the value function.
- various simulations from f_0 and associated distributions of waiting times to making a decision.
- associated histograms of correct and incorrect decisions.

CHAPTER

TWENTYEIGHT

EXCHANGEABILITY AND BAYESIAN UPDATING

Contents

28.1 Overview

This lecture studies learning via Bayes' Law.

We touch foundations of Bayesian statistical inference invented by Bruno DeFinetti [de Finetti, 1937].

The relevance of DeFinetti's work for economists is presented forcefully by David Kreps in chapter 11 of [Kreps, 1988].

An example that we study in this lecture is a key component of *this lecture* that augments the *classic* job search model of McCall [McCall, 1970] by presenting an unemployed worker with a statistical inference problem.

Here we create graphs that illustrate the role that a likelihood ratio plays in Bayes' Law.

We'll use such graphs to provide insights into mechanics driving outcomes in *this lecture* about learning in an augmented McCall job search model.

Among other things, this lecture discusses connections between the statistical concepts of sequences of random variables that are

- independently and identically distributed
- exchangeable (also known as *conditionally* independently and identically distributed)

Understanding these concepts is essential for appreciating how Bayesian updating works.

You can read about exchangeability here.

Because another term for **exchangeable** is **conditionally independent**, we want to convey an answer to the question *conditional on what?*

We also tell why an assumption of independence precludes learning while an assumption of conditional independence makes learning possible.

Below, we'll often use

- W to denote a random variable
- w to denote a particular realization of a random variable W

Let's start with some imports:

```
import matplotlib.pyplot as plt
from numba import jit, vectorize
from math import gamma
import scipy.optimize as op
from scipy.integrate import quad
import numpy as np
```

28.2 Independently and Identically Distributed

We begin by looking at the notion of an **independently and identically distributed sequence** of random variables.

An independently and identically distributed sequence is often abbreviated as IID.

Two notions are involved

- **independence**
- **identically distributed**

A sequence $W_0, W_1, \ldots$ is **independently distributed** if the joint probability density of the sequence is the **product** of the densities of the components of the sequence.

The sequence $W_0, W_1, \ldots$ is **independently and identically distributed** (IID) if in addition the marginal density of W_t is the same for all $t = 0, 1, \ldots$.

For example, let $p(W_0, W_1, \ldots)$ be the **joint density** of the sequence and let $p(W_t)$ be the **marginal density** for a particular W_t for all $t = 0, 1, \ldots$.

Then the joint density of the sequence $W_0, W_1, \ldots$ is IID if

$$p(W_0, W_1, \ldots) = p(W_0)p(W_1) \cdots$$

so that the joint density is the product of a sequence of identical marginal densities.

28.2.1 IID Means Past Observations Don't Tell Us Anything About Future Observations

If a sequence is random variables is IID, past information provides no information about future realizations.

Therefore, there is **nothing to learn** from the past about the future.

To understand these statements, let the joint distribution of a sequence of random variables $\{W_t\}_{t=0}^T$ that is not necessarily IID be

$$p(W_T, W_{T-1}, \ldots, W_1, W_0)$$

Using the laws of probability, we can always factor such a joint density into a product of conditional densities:

$$\begin{aligned} p(W_T, W_{T-1}, \ldots, W_1, W_0) = & p(W_T | W_{T-1}, \ldots, W_0) p(W_{T-1} | W_{T-2}, \ldots, W_0) \cdots \\ & \cdots p(W_1 | W_0) p(W_0) \end{aligned}$$

In general,

$$p(W_t | W_{t-1}, \ldots, W_0) \neq p(W_t)$$

which states that the **conditional density** on the left side does not equal the **marginal density** on the right side.

But in the special IID case,

$$p(W_t | W_{t-1}, \ldots, W_0) = p(W_t),$$

so that the partial history $W_{t-1}, \ldots, W_0$ contains no information about the probability of W_t.

So in the IID case, there is **nothing to learn** about the densities of future random variables from past random variables.

But when the sequence is not IID, there is something to learn about the future from observations of past random variables.

We turn next to an instance of the general case in which the sequence is not IID.

Please watch for what can be learned from the past and when.

28.3 A Setting in Which Past Observations Are Informative

Let $\{W_t\}_{t=0}^\infty$ be a sequence of nonnegative scalar random variables with a joint probability distribution constructed as follows.

There are two distinct cumulative distribution functions F and G that have densities f and g, respectively, for a nonnegative scalar random variable W.

Before the start of time, say at time $t = -1$, "nature" once and for all selects **either** f **or** g.

Thereafter at each time $t \geq 0$, nature draws a random variable W_t from the selected distribution.

So the data are permanently generated as independently and identically distributed (IID) draws from **either** F **or** G.

We could say that *objectively*, meaning *after* nature has chosen either F or G, the probability that the data are generated as draws from F is either 0 or 1.

We now drop into this setting a partially informed decision maker who

- knows both F and G, but
- does not know whether at $t = -1$ nature had drawn F or whether nature had drawn G once-and-for-all

Thus, although our decision maker knows F and knows G, he does not know which of these two known distributions nature had selected to draw from.

The decision maker describes his ignorance with a **subjective probability** $\tilde{\pi}$ and reasons as if nature had selected F with probability $\tilde{\pi} \in (0, 1)$ and G with probability $1 - \tilde{\pi}$.

Thus, we assume that the decision maker

- **knows** both F and G
- **doesn't know** which of these two distributions that nature has drawn
- expresses his ignorance by **acting as if** or **thinking that** nature chose distribution F with probability $\tilde{\pi} \in (0, 1)$ and distribution G with probability $1 - \tilde{\pi}$
- at date $t \geq 0$ knows the partial history $w_t, w_{t-1}, \ldots, w_0$

To proceed, we want to know the decision maker's belief about the joint distribution of the partial history.

We'll discuss that next and in the process describe the concept of **exchangeability**.

28.4 Relationship Between IID and Exchangeable

Conditional on nature selecting F, the joint density of the sequence $W_0, W_1, \ldots$ is

$$f(W_0)f(W_1)\cdots$$

Conditional on nature selecting G, the joint density of the sequence $W_0, W_1, \ldots$ is

$$g(W_0)g(W_1)\cdots$$

Thus, **conditional on nature having selected** F, the sequence $W_0, W_1, \ldots$ is independently and identically distributed.

Furthermore, **conditional on nature having selected** G, the sequence $W_0, W_1, \ldots$ is also independently and identically distributed.

But what about the **unconditional distribution** of a partial history?

The unconditional distribution of $W_0, W_1, \ldots$ is evidently

$$h(W_0, W_1, \ldots) \equiv \tilde{\pi}[f(W_0)f(W_1)\cdots] + (1 - \tilde{\pi})[g(W_0)g(W_1)\cdots] \tag{28.1}$$

Under the unconditional distribution $h(W_0, W_1, \ldots)$, the sequence $W_0, W_1, \ldots$ is **not** independently and identically distributed.

To verify this claim, it is sufficient to notice, for example, that

$$h(W_0, W_1) = \tilde{\pi}f(W_0)f(W_1) + (1 - \tilde{\pi})g(W_0)g(W_1) \neq (\tilde{\pi}f(W_0) + (1 - \tilde{\pi})g(W_0))(\tilde{\pi}f(W_1) + (1 - \tilde{\pi})g(W_1))$$

Thus, the conditional distribution

$$h(W_1|W_0) \equiv \frac{h(W_0, W_1)}{(\tilde{\pi}f(W_0) + (1 - \tilde{\pi})g(W_0))} \neq (\tilde{\pi}f(W_1) + (1 - \tilde{\pi})g(W_1))$$

This means that random variable W_0 contains information about random variable W_1.

So there is something to learn from the past about the future.

28.5 Exchangeability

While the sequence $W_0, W_1, \ldots$ is not IID, it can be verified that it is **exchangeable**, which means that the joint distributions $h(W_0, W_1)$ and $h(W_1, W_0)$ of the "re-ordered" sequences satisfy

$$h(W_0, W_1) = h(W_1, W_0)$$

and so on.

More generally, a sequence of random variables is said to be **exchangeable** if the joint probability distribution for a sequence does not change when the positions in the sequence in which finitely many of random variables appear are altered.

Equation (28.1) represents our instance of an exchangeable joint density over a sequence of random variables as a **mixture** of two IID joint densities over a sequence of random variables.

A Bayesian statistician interprets the mixing parameter $\tilde{\pi} \in (0, 1)$ as a decision maker's subjective belief – the decision maker's **prior probability** – that nature had selected probability distribution F.

Note

DeFinetti [de Finetti, 1937] established a related representation of an exchangeable process created by mixing sequences of IID Bernoulli random variables with parameter $\theta \in (0, 1)$ and mixing probability density $\pi(\theta)$ that a Bayesian statistician would interpret as a prior over the unknown Bernoulli parameter θ.

28.6 Bayes' Law

We noted above that in our example model there is something to learn about about the future from past data drawn from our particular instance of a process that is exchangeable but not IID.

But how can we learn?

And about what?

The answer to the *about what* question is $\tilde{\pi}$.

The answer to the *how* question is to use Bayes' Law.

Another way to say *use Bayes' Law* is to say *from a (subjective) joint distribution, compute an appropriate conditional distribution*.

Let's dive into Bayes' Law in this context.

Let q represent the distribution that nature actually draws w from and let

$$\pi = \mathbb{P}\{q = f\}$$

where we regard π as a decision maker's **subjective probability** (also called a **personal probability**).

Suppose that at $t \geq 0$, the decision maker has observed a history $w^t \equiv [w_t, w_{t-1}, \ldots, w_0]$.

We let

$$\pi_t = \mathbb{P}\{q = f | w^t\}$$

where we adopt the convention

$$\pi_{-1} = \tilde{\pi}$$

The distribution of w_{t+1} conditional on w^t is then

$$\pi_t f + (1 - \pi_t) g.$$

Bayes' rule for updating π_{t+1} is

$$\pi_{t+1} = \frac{\pi_t f(w_{t+1})}{\pi_t f(w_{t+1}) + (1 - \pi_t) g(w_{t+1})} \tag{28.2}$$

Equation (28.2) follows from Bayes' rule, which tells us that

$$\mathbb{P}\{q = f \,|\, W = w\} = \frac{\mathbb{P}\{W = w \,|\, q = f\}\mathbb{P}\{q = f\}}{\mathbb{P}\{W = w\}}$$

where

$$\mathbb{P}\{W = w\} = \sum_{a \in \{f,g\}} \mathbb{P}\{W = w \,|\, q = a\}\mathbb{P}\{q = a\}$$

28.7 More Details about Bayesian Updating

Let's stare at and rearrange Bayes' Law as represented in equation (28.2) with the aim of understanding how the **posterior** probability π_{t+1} is influenced by the **prior** probability π_t and the **likelihood ratio**

$$l(w) = \frac{f(w)}{g(w)}$$

It is convenient for us to rewrite the updating rule (28.2) as

$$\pi_{t+1} = \frac{\pi_t f\left(w_{t+1}\right)}{\pi_t f\left(w_{t+1}\right) + \left(1 - \pi_t\right) g\left(w_{t+1}\right)} = \frac{\pi_t \frac{f(w_{t+1})}{g(w_{t+1})}}{\pi_t \frac{f(w_{t+1})}{g(w_{t+1})} + \left(1 - \pi_t\right)} = \frac{\pi_t l\left(w_{t+1}\right)}{\pi_t l\left(w_{t+1}\right) + \left(1 - \pi_t\right)}$$

This implies that

$$\frac{\pi_{t+1}}{\pi_t} = \frac{l\left(w_{t+1}\right)}{\pi_t l\left(w_{t+1}\right) + \left(1 - \pi_t\right)} \begin{cases} > 1 & \text{if } l\left(w_{t+1}\right) > 1 \\ \leq 1 & \text{if } l\left(w_{t+1}\right) \leq 1 \end{cases} \tag{28.3}$$

Notice how the likelihood ratio and the prior interact to determine whether an observation w_{t+1} leads the decision maker to increase or decrease the subjective probability he/she attaches to distribution F.

When the likelihood ratio $l(w_{t+1})$ exceeds one, the observation w_{t+1} nudges the probability π put on distribution F upward, and when the likelihood ratio $l(w_{t+1})$ is less that one, the observation w_{t+1} nudges π downward.

Representation (28.3) is the foundation of some graphs that we'll use to display the dynamics of $\{\pi_t\}_{t=0}^{\infty}$ that are induced by Bayes' Law.

We'll plot $l\left(w\right)$ as a way to enlighten us about how learning – i.e., Bayesian updating of the probability π that nature has chosen distribution f – works.

To create the Python infrastructure to do our work for us, we construct a wrapper function that displays informative graphs given parameters of f and g.

```
@vectorize
def p(x, a, b):
    "The general beta distribution function."
    r = gamma(a + b) / (gamma(a) * gamma(b))
```

(continues on next page)

(continued from previous page)

```
    return r * x ** (a-1) * (1 - x) ** (b-1)

def learning_example(F_a=1, F_b=1, G_a=3, G_b=1.2):
    """
    A wrapper function that displays the updating rule of belief π,
    given the parameters which specify F and G distributions.
    """

    f = jit(lambda x: p(x, F_a, F_b))
    g = jit(lambda x: p(x, G_a, G_b))

    # l(w) = f(w) / g(w)
    l = lambda w: f(w) / g(w)
    # objective function for solving l(w) = 1
    obj = lambda w: l(w) - 1

    x_grid = np.linspace(0, 1, 100)
    π_grid = np.linspace(1e-3, 1-1e-3, 100)

    w_max = 1
    w_grid = np.linspace(1e-12, w_max-1e-12, 100)

    # the mode of beta distribution
    # use this to divide w into two intervals for root finding
    G_mode = (G_a - 1) / (G_a + G_b - 2)
    roots = np.empty(2)
    roots[0] = op.root_scalar(obj, bracket=[1e-10, G_mode]).root
    roots[1] = op.root_scalar(obj, bracket=[G_mode, 1-1e-10]).root

    fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(18, 5))

    ax1.plot(l(w_grid), w_grid, label='$l$', lw=2)
    ax1.vlines(1., 0., 1., linestyle="--")
    ax1.hlines(roots, 0., 2., linestyle="--")
    ax1.set_xlim([0., 2.])
    ax1.legend(loc=4)
    ax1.set(xlabel='$l(w)=f(w)/g(w)$', ylabel='$w$')

    ax2.plot(f(x_grid), x_grid, label='$f$', lw=2)
    ax2.plot(g(x_grid), x_grid, label='$g$', lw=2)
    ax2.vlines(1., 0., 1., linestyle="--")
    ax2.hlines(roots, 0., 2., linestyle="--")
    ax2.legend(loc=4)
    ax2.set(xlabel='$f(w), g(w)$', ylabel='$w$')

    area1 = quad(f, 0, roots[0])[0]
    area2 = quad(g, roots[0], roots[1])[0]
    area3 = quad(f, roots[1], 1)[0]

    ax2.text((f(0) + f(roots[0])) / 4, roots[0] / 2, f"{area1: .3g}")
    ax2.fill_between([0, 1], 0, roots[0], color='blue', alpha=0.15)
    ax2.text(np.mean(g(roots)) / 2, np.mean(roots), f"{area2: .3g}")
    w_roots = np.linspace(roots[0], roots[1], 20)
    ax2.fill_betweenx(w_roots, 0, g(w_roots), color='orange', alpha=0.15)
    ax2.text((f(roots[1]) + f(1)) / 4, (roots[1] + 1) / 2, f"{area3: .3g}")
    ax2.fill_between([0, 1], roots[1], 1, color='blue', alpha=0.15)
```

(continues on next page)

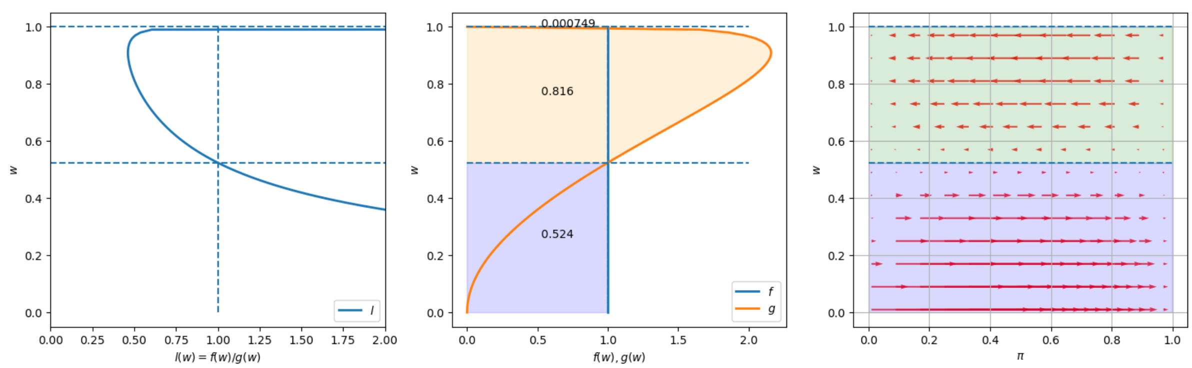
(continued from previous page)

```
    W = np.arange(0.01, 0.99, 0.08)
    Π = np.arange(0.01, 0.99, 0.08)

    ΔW = np.zeros((len(W), len(Π)))
    ΔΠ = np.empty((len(W), len(Π)))
    for i, w in enumerate(W):
        for j, π in enumerate(Π):
            lw = l(w)
            ΔΠ[i, j] = π * (lw / (π * lw + 1 - π) - 1)

    q = ax3.quiver(Π, W, ΔΠ, ΔW, scale=2, color='r', alpha=0.8)

    ax3.fill_between(π_grid, 0, roots[0], color='blue', alpha=0.15)
    ax3.fill_between(π_grid, roots[0], roots[1], color='green', alpha=0.15)
    ax3.fill_between(π_grid, roots[1], w_max, color='blue', alpha=0.15)
    ax3.hlines(roots, 0., 1., linestyle="--")
    ax3.set(xlabel=r'$\pi$', ylabel='$w$')
    ax3.grid()

    plt.show()
```

Now we'll create a group of graphs that illustrate dynamics induced by Bayes' Law.

We'll begin with Python function default values of various objects, then change them in a subsequent example.

```
learning_example()
```



Please look at the three graphs above created for an instance in which f is a uniform distribution on $[0, 1]$ (i.e., a Beta distribution with parameters $F_a = 1, F_b = 1$), while g is a Beta distribution with the default parameter values $G_a = 3, G_b = 1.2$.

The graph on the left plots the likelihood ratio $l(w)$ as the absciassa axis against w as the ordinate.

The middle graph plots both $f(w)$ and $g(w)$ against w, with the horizontal dotted lines showing values of w at which the likelihood ratio equals 1.

The graph on the right plots arrows to the right that show when Bayes' Law makes π increase and arrows to the left that show when Bayes' Law make π decrease.

Lengths of the arrows show magnitudes of the force from Bayes' Law impelling π to change.

These lengths depend on both the prior probability π on the abscissa axis and the evidence in the form of the current draw of w on the ordinate axis.

The fractions in the colored areas of the middle graphs are probabilities under F and G, respectively, that realizations of w fall into the interval that updates the belief π in a correct direction (i.e., toward 0 when G is the true distribution, and toward 1 when F is the true distribution).

For example, in the above example, under true distribution F, π will be updated toward 0 if w falls into the interval $[0.524, 0.999]$, which occurs with probability $1 - .524 = .476$ under F.

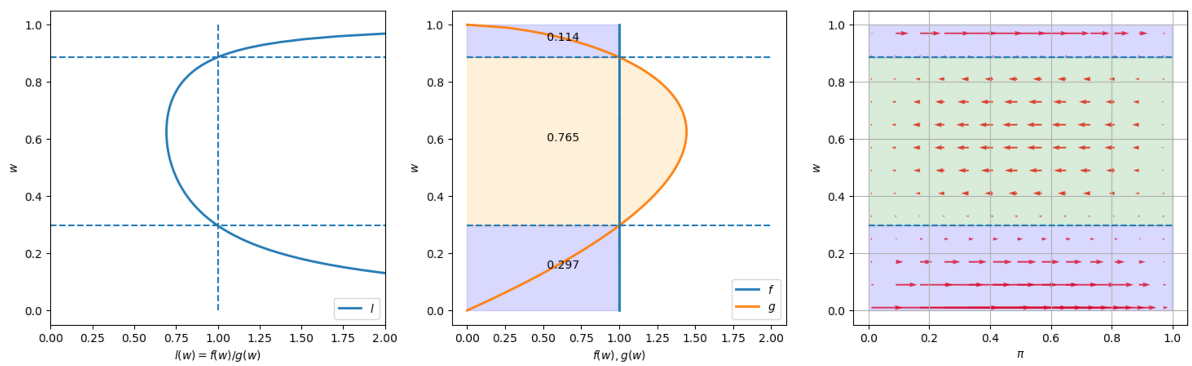
But this would occur with probability 0.816 if G were the true distribution.

The fraction 0.816 in the orange region is the integral of $g(w)$ over this interval.

Next we use our code to create graphs for another instance of our model.

We keep F the same as in the preceding instance, namely a uniform distribution, but now assume that G is a Beta distribution with parameters $G_a = 2, G_b = 1.6$.

```
learning_example(G_a=2, G_b=1.6)
```



Notice how the likelihood ratio, the middle graph, and the arrows compare with the previous instance of our example.

28.8 Appendix

28.8.1 Sample Paths of π_t

Now we'll have some fun by plotting multiple realizations of sample paths of π_t under two possible assumptions about nature's choice of distribution, namely

- that nature permanently draws from F
- that nature permanently draws from G

Outcomes depend on a peculiar property of likelihood ratio processes discussed in this lecture.

To proceed, we create some Python code.

```
def function_factory(F_a=1, F_b=1, G_a=3, G_b=1.2):

    # define f and g
    f = jit(lambda x: p(x, F_a, F_b))
    g = jit(lambda x: p(x, G_a, G_b))

    @jit
    def update(a, b, π):
        "Update π by drawing from beta distribution with parameters a and b"

        # Draw
        w = np.random.beta(a, b)
```

(continues on next page)

(continued from previous page)

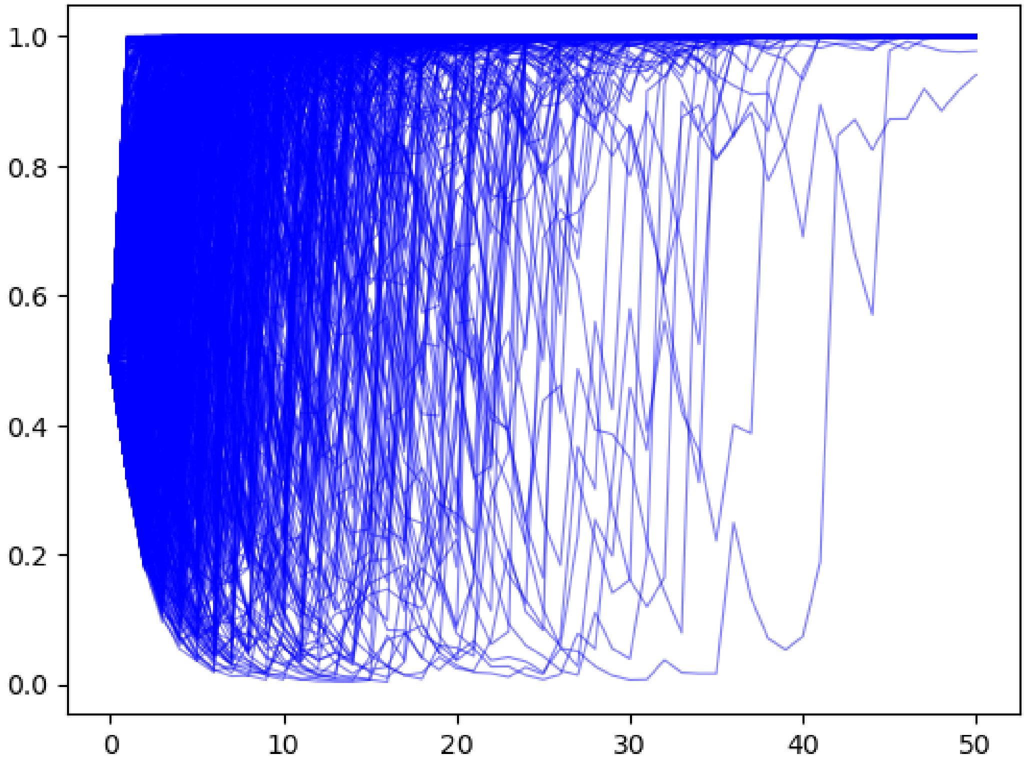
```
        # Update belief
        π = 1 / (1 + ((1 - π) * g(w)) / (π * f(w)))

        return π

    @jit
    def simulate_path(a, b, T=50):
        "Simulates a path of beliefs π with length T"

        π = np.empty(T+1)

        # initial condition
        π[0] = 0.5

        for t in range(1, T+1):
            π[t] = update(a, b, π[t-1])

        return π

    def simulate(a=1, b=1, T=50, N=200, display=True):
        "Simulates N paths of beliefs π with length T"

        π_paths = np.empty((N, T+1))
        if display:
            fig = plt.figure()

        for i in range(N):
            π_paths[i] = simulate_path(a=a, b=b, T=T)
            if display:
                plt.plot(range(T+1), π_paths[i], color='b', lw=0.8, alpha=0.5)

        if display:
            plt.show()

        return π_paths

    return simulate
```

```
simulate = function_factory()
```

We begin by generating N simulated $\{\pi_t\}$ paths with T periods when the sequence is truly IID draws from F. We set an initial prior $\pi_{-1} = .5$.

```
T = 50
```

```
# when nature selects F
π_paths_F = simulate(a=1, b=1, T=T, N=1000)
```

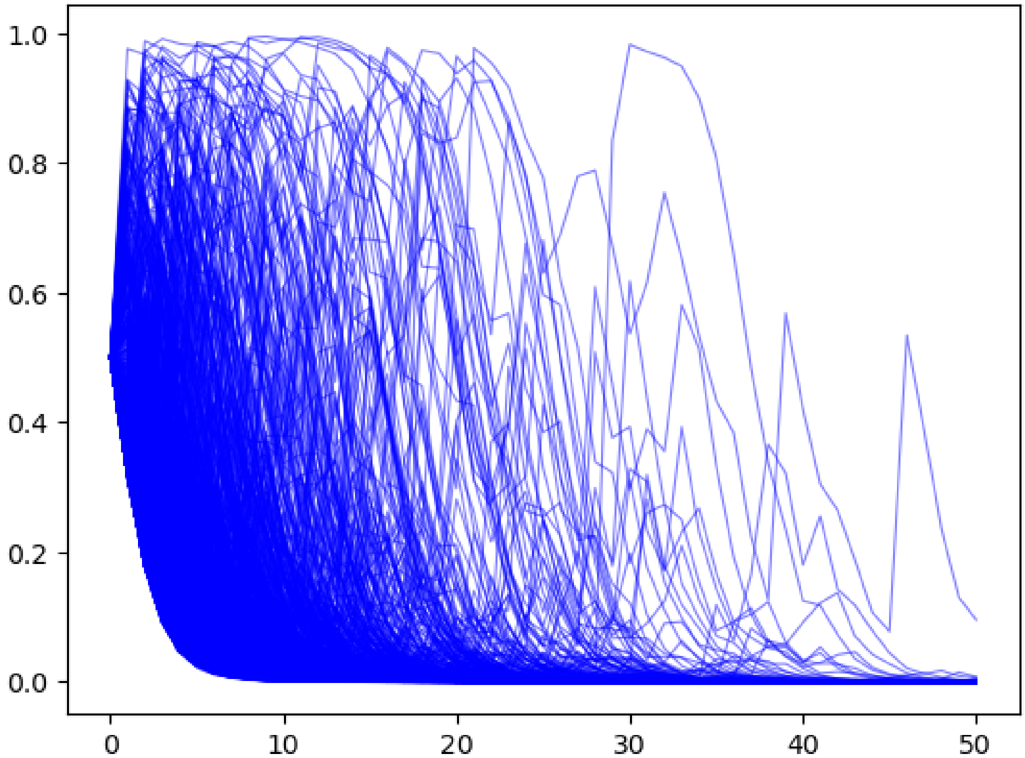


In the above example, for most paths $\pi_t \to 1$.

So Bayes' Law evidently eventually discovers the truth for most of our paths.

Next, we generate paths with T periods when the sequence is truly IID draws from G. Again, we set the initial prior $\pi_{-1} = .5$.

```
# when nature selects G
π_paths_G = simulate(a=3, b=1.2, T=T, N=1000)
```



In the above graph we observe that now most paths $\pi_t \to 0$.

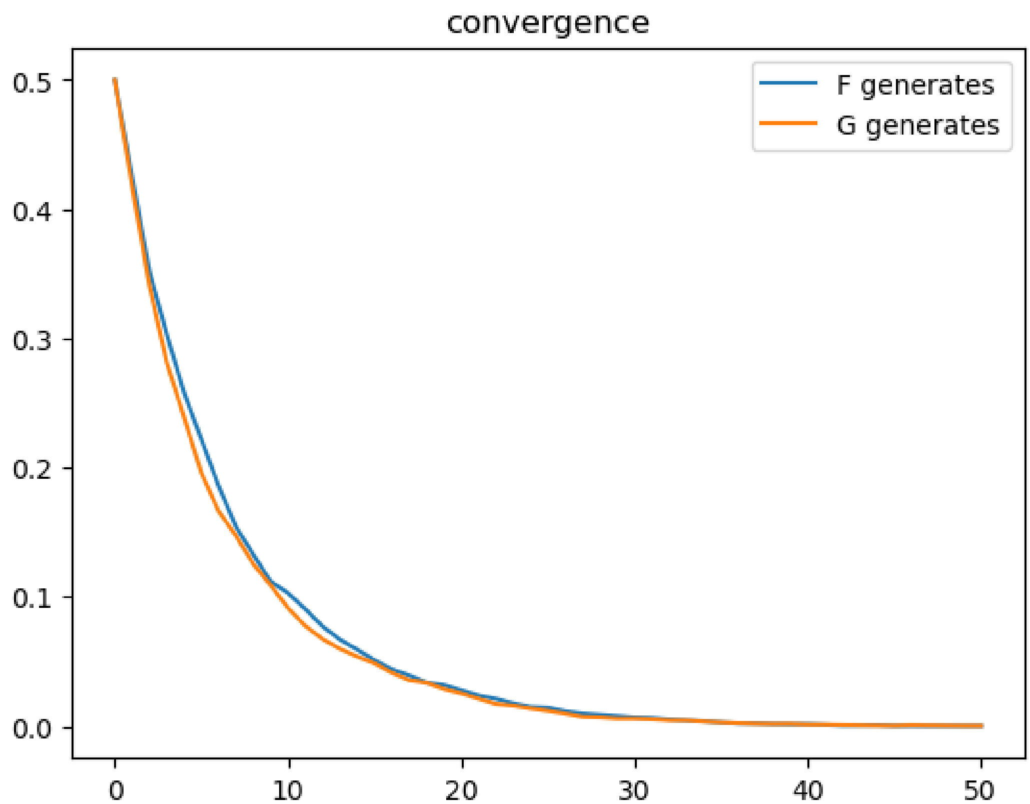
28.8.2 Rates of convergence

We study rates of convergence of π_t to 1 when nature generates the data as IID draws from F and of convergence of π_t to 0 when nature generates IID draws from G.

We do this by averaging across simulated paths of $\{\pi_t\}_{t=0}^T$.

Using N simulated π_t paths, we compute $1 - \sum_{i=1}^N \pi_{i,t}$ at each t when the data are generated as draws from F and compute $\sum_{i=1}^N \pi_{i,t}$ when the data are generated as draws from G.

```
plt.plot(range(T+1), 1 - np.mean(π_paths_F, 0), label='F generates')
plt.plot(range(T+1), np.mean(π_paths_G, 0), label='G generates')
plt.legend()
plt.title("convergence");
```



From the above graph, rates of convergence appear not to depend on whether F or G generates the data.

28.8.3 Graph of Ensemble Dynamics of π_t

More insights about the dynamics of $\{\pi_t\}$ can be gleaned by computing conditional expectations of $\frac{\pi_{t+1}}{\pi_t}$ as functions of π_t via integration with respect to the pertinent probability distribution:

$$
\begin{aligned}
E\left[\frac{\pi_{t+1}}{\pi_t}\bigg| q=a,\pi_t\right] &= E\left[\frac{l\left(w_{t+1}\right)}{\pi_t l\left(w_{t+1}\right)+\left(1-\pi_t\right)}\bigg| q=a,\pi_t\right], \\
&= \int_0^1 \frac{l\left(w_{t+1}\right)}{\pi_t l\left(w_{t+1}\right)+\left(1-\pi_t\right)} a\left(w_{t+1}\right) dw_{t+1}
\end{aligned}
$$

where $a = f, g$.

The following code approximates the integral above:

```
def expected_ratio(F_a=1, F_b=1, G_a=3, G_b=1.2):

    # define f and g
    f = jit(lambda x: p(x, F_a, F_b))
    g = jit(lambda x: p(x, G_a, G_b))

    l = lambda w: f(w) / g(w)
    integrand_f = lambda w, π: f(w) * l(w) / (π * l(w) + 1 - π)
    integrand_g = lambda w, π: g(w) * l(w) / (π * l(w) + 1 - π)

    π_grid = np.linspace(0.02, 0.98, 100)
```

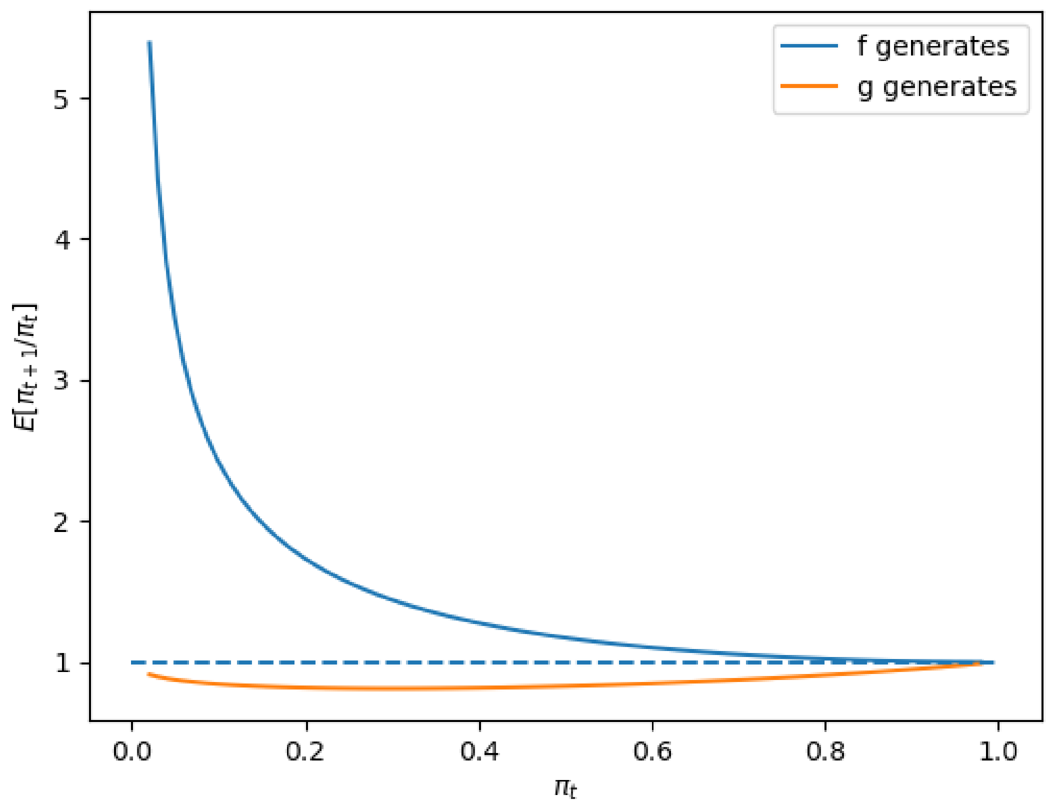
(continues on next page)

(continued from previous page)

```
    expected_rario = np.empty(len(π_grid))
    for q, inte in zip(["f", "g"], [integrand_f, integrand_g]):
        for i, π in enumerate(π_grid):
            expected_rario[i]= quad(inte, 0, 1, args=(π,))[0]
        plt.plot(π_grid, expected_rario, label=f"{q} generates")

    plt.hlines(1, 0, 1, linestyle="--")
    plt.xlabel(r"$\pi_t$")
    plt.ylabel(r"$E[\pi_{t+1}/\pi_t]$")
    plt.legend()

    plt.show()
```

First, consider the case where $F_a = F_b = 1$ and $G_a = 3, G_b = 1.2$.

```
expected_ratio()
```

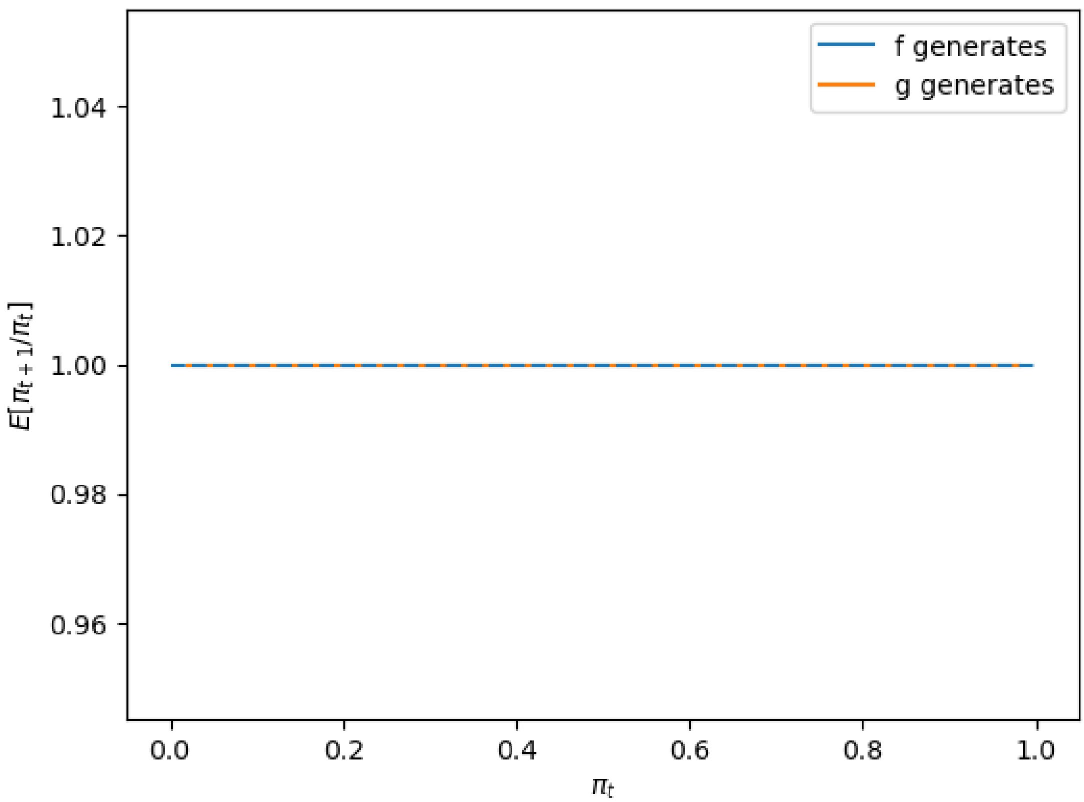


The above graphs shows that when F generates the data, π_t on average always heads north, while when G generates the data, π_t heads south.

Next, we'll look at a degenerate case in whcih f and g are identical beta distributions, and $F_a = G_a = 3, F_b = G_b = 1.2$.

In a sense, here there is nothing to learn.

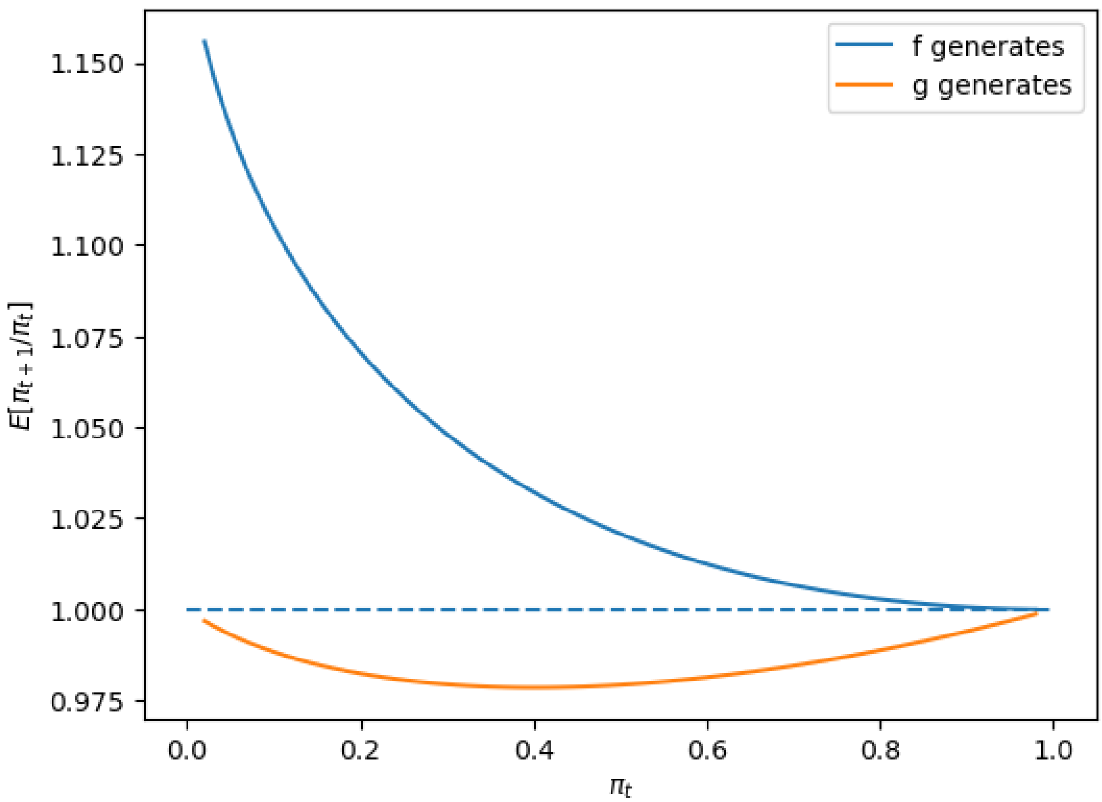
```
expected_ratio(F_a=3, F_b=1.2)
```



The above graph says that π_t is inert and remains at its initial value.

Finally, let's look at a case in which f and g are neither very different nor identical, in particular one in which $F_a = 2, F_b = 1$ and $G_a = 3, G_b = 1.2$.

```
expected_ratio(F_a=2, F_b=1, G_a=3, G_b=1.2)
```



28.9 Sequels

We'll apply and dig deeper into some of the ideas presented in this lecture:

- *this lecture* describes **likelihood ratio processes** and their role in frequentist and Bayesian statistical theories
- *this lecture* studies whether a World War II US Navy Captain's hunch that a (frequentist) decision rule that the Navy had told him to use was inferior to a sequential rule that Abraham Wald had not yet designed.

CHAPTER

TWENTYNINE

LIKELIHOOD RATIO PROCESSES AND BAYESIAN LEARNING

29.1 Overview

This lecture describes the role that **likelihood ratio processes** play in **Bayesian learning**.

As in *this lecture*, we'll use a simple statistical setting from *this lecture*.

We'll focus on how a likelihood ratio process and a **prior** probability determine a **posterior** probability.

We'll derive a convenient recursion for today's posterior as a function of yesterday's posterior and today's multiplicative increment to a likelihood process.

We'll also present a useful generalization of that formula that represents today's posterior in terms of an initial prior and today's realization of the likelihood ratio process.

We'll study how, at least in our setting, a Bayesian eventually learns the probability distribution that generates the data, an outcome that rests on the asymptotic behavior of likelihood ratio processes studied in *this lecture*.

We'll also drill down into the psychology of our Bayesian learner and study dynamics under his subjective beliefs.

This lecture provides technical results that underly outcomes to be studied in *this lecture* and *this lecture* and *this lecture*.

We'll begin by loading some Python modules.

```
import matplotlib.pyplot as plt
import numpy as np
from numba import vectorize, jit, prange
from math import gamma
import pandas as pd
from scipy.integrate import quad


import seaborn as sns
colors = sns.color_palette()

@jit
def set_seed():
    np.random.seed(142857)
set_seed()
```

29.2 The Setting

We begin by reviewing the setting in *this lecture*, which we adopt here too.

A nonnegative random variable W has one of two probability density functions, either f or g.

Before the beginning of time, nature once and for all decides whether she will draw a sequence of IID draws from f or from g.

We will sometimes let q be the density that nature chose once and for all, so that q is either f or g, permanently.

Nature knows which density it permanently draws from, but we the observers do not.

We do know both f and g, but we don't know which density nature chose.

But we want to know.

To do that, we use observations.

We observe a sequence $\{w_t\}_{t=1}^T$ of T IID draws from either f or g.

We want to use these observations to infer whether nature chose f or g.

A **likelihood ratio process** is a useful tool for this task.

To begin, we define the key component of a likelihood ratio process, namely, the time t likelihood ratio as the random variable

$$\ell(w_t) = \frac{f(w_t)}{g(w_t)}, \quad t \geq 1.$$

We assume that f and g both put positive probabilities on the same intervals of possible realizations of the random variable W.

That means that under the g density, $\ell(w_t) = \frac{f(w_t)}{g(w_t)}$ is evidently a nonnegative random variable with mean 1.

A **likelihood ratio process** for sequence $\{w_t\}_{t=1}^{\infty}$ is defined as

$$L(w^t) = \prod_{i=1}^{t} \ell(w_i),$$

where $w^t = \{w_1, \ldots, w_t\}$ is a history of observations up to and including time t.

Sometimes for shorthand we'll write

$$L_t = L(w^t) = \frac{f(w^t)}{g(w^t)}$$

where we use the conventions that $f(w^t) = f(w_1)f(w_2)\ldots f(w_t)$ and $g(w^t) = g(w_1)g(w_2)\ldots g(w_t)$.

Notice that the likelihood process satisfies the **recursion** or **multiplicative decomposition**

$$L(w^t) = \ell(w_t)L(w^{t-1}).$$

The likelihood ratio and its logarithm are key tools for making inferences using a classic frequentist approach due to Neyman and Pearson [Neyman and Pearson, 1933].

We'll again deploy the following Python code from *this lecture* that evaluates f and g as two different beta distributions, then computes and simulates an associated likelihood ratio process by generating a sequence w^t from *some* probability distribution, for example, a sequence of IID draws from g.

```
# Parameters in the two beta distributions.
F_a, F_b = 1, 1
G_a, G_b = 3, 1.2

@vectorize
def p(x, a, b):
    r = gamma(a + b) / (gamma(a) * gamma(b))
    return r * x** (a-1) * (1 - x) ** (b-1)

# The two density functions.
f = jit(lambda x: p(x, F_a, F_b))
g = jit(lambda x: p(x, G_a, G_b))
```

```
@jit
def simulate(a, b, T=50, N=500):
    '''
    Generate N sets of T observations of the likelihood ratio,
    return as N x T matrix.

    '''

    l_arr = np.empty((N, T))

    for i in range(N):

        for j in range(T):
            w = np.random.beta(a, b)
            l_arr[i, j] = f(w) / g(w)

    return l_arr
```

We'll also use the following Python code to prepare some informative simulations

```
l_arr_g = simulate(G_a, G_b, N=50000)
l_seq_g = np.cumprod(l_arr_g, axis=1)
```

```
l_arr_f = simulate(F_a, F_b, N=50000)
l_seq_f = np.cumprod(l_arr_f, axis=1)
```

29.3 Likelihood Ratio Processes and Bayes' Law

Let $\pi_0 \in [0, 1]$ be a Bayesian statistician's prior probability that nature generates w^t as a sequence of i.i.d. draws from distribution f.

- here "probability" is to be interpreted as a way to summarize or express a subjective opinion
- it does **not** mean an anticipated relative frequency as sample size grows without limit

Let π_{t+1} be a Bayesian posterior probability defined as

$$\pi_{t+1} = \text{Prob}(q = f | w^{t+1}) \tag{29.1}$$

The likelihood ratio process is a principal actor in the formula that governs the evolution of the posterior probability π_t, an instance of **Bayes' Law**.

Let's derive a couple of formulas for π_{t+1}, one in terms of likelihood ratio $l(w_t)$, the other in terms of $L(w^t)$.

To begin, we use the notational conventions

- $f(w^{t+1}) \equiv f(w_1)f(w_2)\cdots f(w_{t+1})$
- $g(w^{t+1}) \equiv g(w_1)g(w_2)\cdots g(w_{t+1})$
- $\pi_0 = \text{Prob}(q = f|\emptyset)$
- $\pi_t = \text{Prob}(q = f|w^t)$

Here the symbol $\emptyset$ means "empty set" or "no data".

With no data in hand, our Bayesian statistician thinks that the probability density of the sequence w^{t+1} is

$$\text{Prob}(w^{t+1}|\emptyset) = \pi_0 f(w^{t+1}) + (1-\pi_0)g(w^{t+1})$$

Laws of probability say that the joint distribution $\text{Prob}(AB)$ of events A and B are connected to the conditional distributions $\text{Prob}(A|B)$ and $\text{Prob}(B|A)$ by

$$\text{Prob}(AB) = \text{Prob}(A|B)\text{Prob}(B) = \text{Prob}(B|A)\text{Prob}(A). \tag{29.2}$$

We are interested in events

$$A = \{q = f\}, \quad B = \{w^{t+1}\},$$

where braces $\{\cdot\}$ are our shorthand for "event".

So in our setting, probability laws (29.2) imply that

$$\text{Prob}(q = f|w^{t+1})\text{Prob}(w^{t+1}|\emptyset) = \text{Prob}(w^{t+1}|q = f)\text{Prob}(q = f|\emptyset)$$

or

$$\pi_{t+1}\left[\pi_0 f(w^{t+1}) + (1-\pi_0)g(w^{t+1})\right] = f(w^{t+1})\pi_0$$

or

$$\pi_{t+1} = \frac{f(w^{t+1})\pi_0}{\pi_0 f(w^{t+1}) + (1-\pi_0)g(w^{t+1})}$$

Dividing both the numerator and the denominator on the right side of the above equation by $g(w^{t+1})$ implies

$$\pi_{t+1} = \frac{\pi_0 L\left(w^{t+1}\right)}{\pi_0 L\left(w^{t+1}\right) + 1 - \pi_0}. \tag{29.3}$$

Formula (29.3) can be regarded as a one step revision of prior probability π_0 after seeing the batch of data $\{w_i\}_{i=1}^{t+1}$.

Formula (29.3) shows the key role that the likelihood ratio process $L\left(w^{t+1}\right)$ plays in determining the posterior probability π_{t+1}.

Formula (29.3) is the foundation for the insight that, because of how the likelihood ratio process behaves as $t \to +\infty$, the likelihood ratio process dominates the initial prior π_0 in determining the limiting behavior of π_t.

29.3.1 A recursive formula

We can use a similar line of argument to get a recursive version of formula (29.3).

The laws of probability imply

$$\text{Prob}(q = f|w^{t+1}) = \frac{\text{Prob}(q = f|w^t)f(w_{t+1})}{\text{Prob}(q = f|w^t)f(w_{t+1}) + (1 - \text{Prob}(q = f|w^t))g(w_{t+1})}$$

or

$$\pi_{t+1} = \frac{\pi_t f(w_{t+1})}{\pi_t f(w_{t+1}) + (1-\pi_t) g(w_{t+1})} \tag{29.4}$$

Evidently, the above equation asserts that

$$\text{Prob}(q = f | w^{t+1}) = \frac{\text{Prob}(q = f | w^t) f(w_{t+1})}{\text{Prob}(w_{t+1})}$$

Dividing both the numerator and the denominator on the right side of the equation (29.4) by $g(w_{t+1})$ implies the recursion

$$\pi_{t+1} = \frac{\pi_t l_t(w_{t+1})}{\pi_t l_t(w_{t+1}) + 1 - \pi_t} \tag{29.5}$$

with π_0 being a Bayesian prior probability that $q = f$, i.e., a personal or subjective belief about q based on our having seen no data.

Formula (29.3) can be deduced by iterating on equation (29.5).

Below we define a Python function that updates belief π using likelihood ratio ℓ according to recursion (29.5)

```
@jit
def update(π, l):
    "Update π using likelihood l"

    # Update belief
    π = π * l / (π * l + 1 - π)

    return π
```

As mentioned above, formula (29.3) shows the key role that the likelihood ratio process $L\left(w^{t+1}\right)$ plays in determining the posterior probability π_{t+1}.

As $t \to +\infty$, the likelihood ratio process dominates the initial prior π_0 in determining the limiting behavior of π_t.

To illustrate this insight, below we will plot graphs showing **one** simulated path of the likelihood ratio process L_t along with two paths of π_t that are associated with the *same* realization of the likelihood ratio process but *different* initial prior probabilities π_0.

First, we tell Python two values of π_0.

```
π1, π2 = 0.2, 0.8
```

Next we generate paths of the likelihood ratio process L_t and the posterior π_t for a history of IID draws from density f.

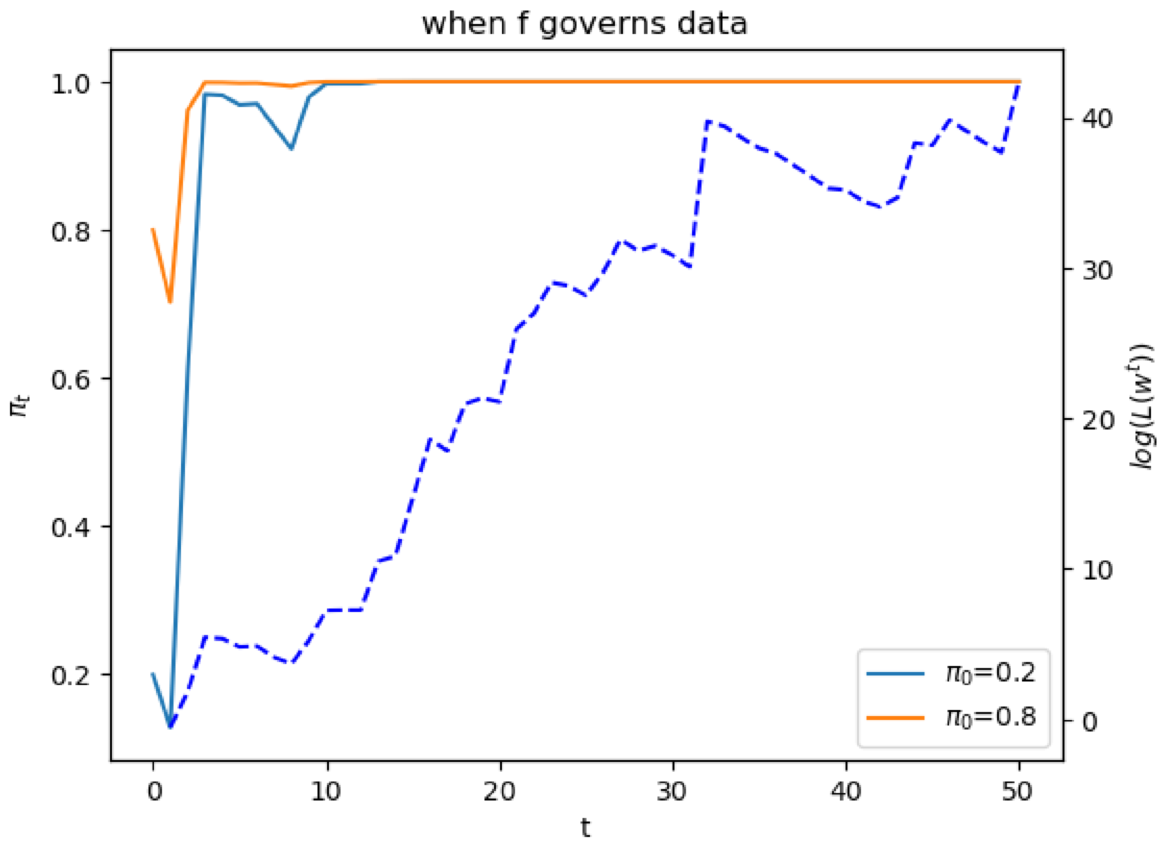
```
T = l_arr_f.shape[1]
π_seq_f = np.empty((2, T+1))
π_seq_f[:, 0] = π1, π2

for t in range(T):
    for i in range(2):
        π_seq_f[i, t+1] = update(π_seq_f[i, t], l_arr_f[0, t])
```

```
fig, ax1 = plt.subplots()

for i in range(2):
    ax1.plot(range(T+1), π_seq_f[i, :], label=fr"$\pi_0$={π_seq_f[i, 0]}")
```

(continues on next page)

(continued from previous page)

```
ax1.set_ylabel(r"$\pi_t$")
ax1.set_xlabel("t")
ax1.legend()
ax1.set_title("when f governs data")

ax2 = ax1.twinx()
ax2.plot(range(1, T+1), np.log(l_seq_f[0, :]), '--', color='b')
ax2.set_ylabel("$log(L(w^{t}))$")

plt.show()
```



The dotted line in the graph above records the logarithm of the likelihood ratio process log $L(w^t)$.

Please note that there are two different scales on the y axis.

Now let's study what happens when the history consists of IID draws from density g

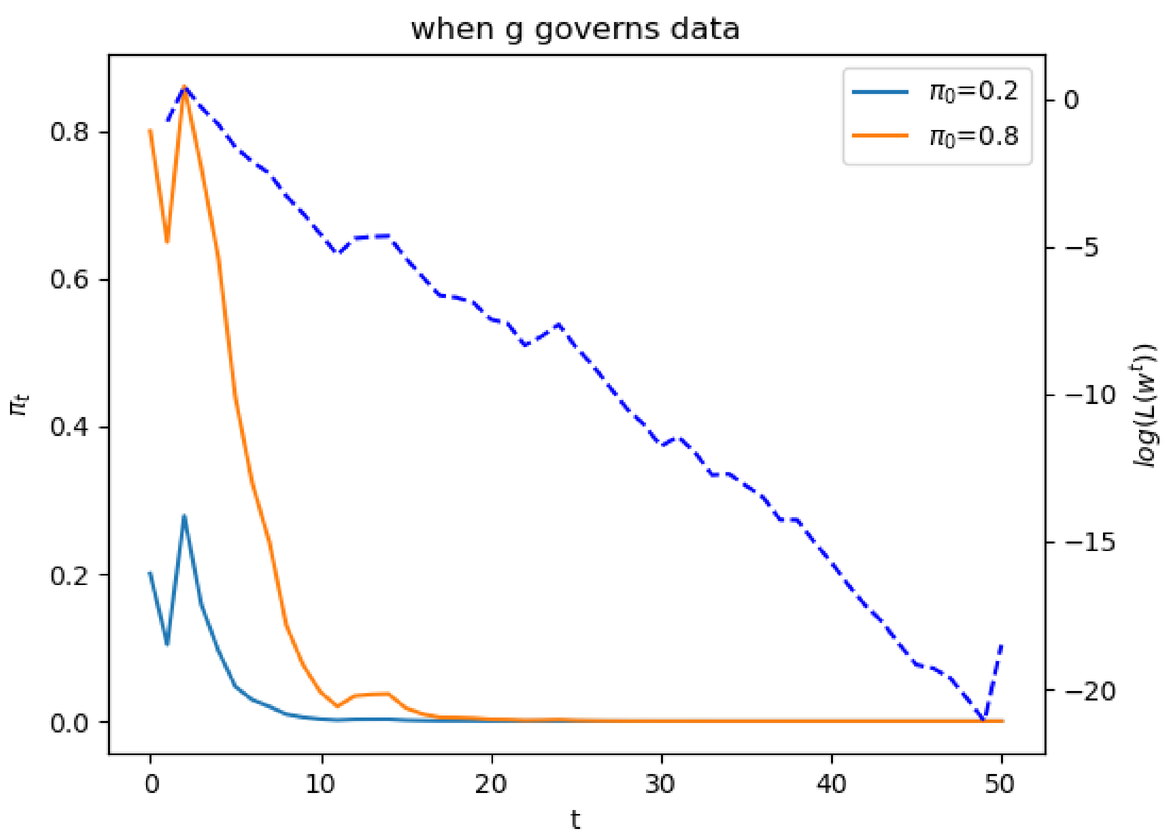
```
T = l_arr_g.shape[1]
π_seq_g = np.empty((2, T+1))
π_seq_g[:, 0] = π1, π2

for t in range(T):
    for i in range(2):
        π_seq_g[i, t+1] = update(π_seq_g[i, t], l_arr_g[0, t])
```

```
fig, ax1 = plt.subplots()
```

(continues on next page)

(continued from previous page)

```
for i in range(2):
    ax1.plot(range(T+1), π_seq_g[i, :], label=fr"$\pi_0$={π_seq_g[i, 0]}")

ax1.set_ylabel(r"$\pi_t$")
ax1.set_xlabel("t")
ax1.legend()
ax1.set_title("when g governs data")

ax2 = ax1.twinx()
ax2.plot(range(1, T+1), np.log(l_seq_g[0, :]), '--', color='b')
ax2.set_ylabel("$log(L(w^{t}))$")

plt.show()
```



Below we offer Python code that verifies that nature chose permanently to draw from density f.

```
π_seq = np.empty((2, T+1))
π_seq[:, 0] = π1, π2

for i in range(2):
    πL = π_seq[i, 0] * l_seq_f[0, :]
    π_seq[i, 1:] = πL / (πL + 1 - π_seq[i, 0])
```

```
np.abs(π_seq - π_seq_f).max() < 1e-10
```

```
np.True_
```

We thus conclude that the likelihood ratio process is a key ingredient of the formula (29.3) for a Bayesian's posterior probability that nature has drawn history w^t as repeated draws from density f.

29.4 Another timing protocol

Let's study how the posterior probability $\pi_t = \text{Prob}(q = f|w^t)$ behaves when nature generates the history $w^t = \{w_1, w_2, \dots, w_t\}$ under a different timing protocol.

Until now we assumed that before time 1 nature somehow chose to draw w^t as an iid sequence from **either** f **or** g.

Nature's decision about whether to draw from f or g was thus **permanent**.

We now assume a different timing protocol in which before **each period** $t = 1, 2, \dots$ nature

- flips an x-weighted coin, then
- draws from f if it has drawn a "head"
- draws from g if it has drawn a "tail".

Under this timing protocol, nature draws permanently from **neither** f **nor** g, so a statistician who thinks that nature is drawing i.i.d. draws **permanently** from one of them is mistaken.

- in truth, nature actually draws **permanently** from an x-mixture of f and g – a distribution that is neither f nor g when $x \in (0, 1)$

Thus, the Bayesian prior π_0 and the sequence of posterior probabilities described by equation (29.3) should **not** be interpreted as the statistician's opinion about the mixing parameter x under the alternative timing protocol in which nature draws from an x-mixture of f and g.

This is clear when we remember the definition of π_t in equation (29.1), which for convenience we repeat here:

$$\pi_{t+1} = \text{Prob}(q = f|w^{t+1})$$

Let's write some Python code to study how π_t behaves when nature actually generates data as i.i.d. draws from neither f nor from g but instead as i.i.d. draws from an x-mixture of two beta distributions.

> **Note**
>
> This is a situation in which the statistician's model is misspecified, so we should anticipate that a Kullback-Liebler divergence with respect to an x-mixture distribution will shape outcomes.

We can study how π_t would behave for various values of nature's mixing probability x.

First, let's create a function to simulate data under the mixture timing protocol:

```python
@jit
def simulate_mixture_path(x_true, T):
    """
    Simulate T observations under mixture timing protocol.
    """
    w = np.empty(T)
    for t in range(T):
        if np.random.rand() < x_true:
            w[t] = np.random.beta(F_a, F_b)
        else:
            w[t] = np.random.beta(G_a, G_b)
    return w
```

Let's generate a sequence of observations from this mixture model with a true mixing probability of $x = 0.5$.

We will first use this sequence to study how π_t behaves.

Note

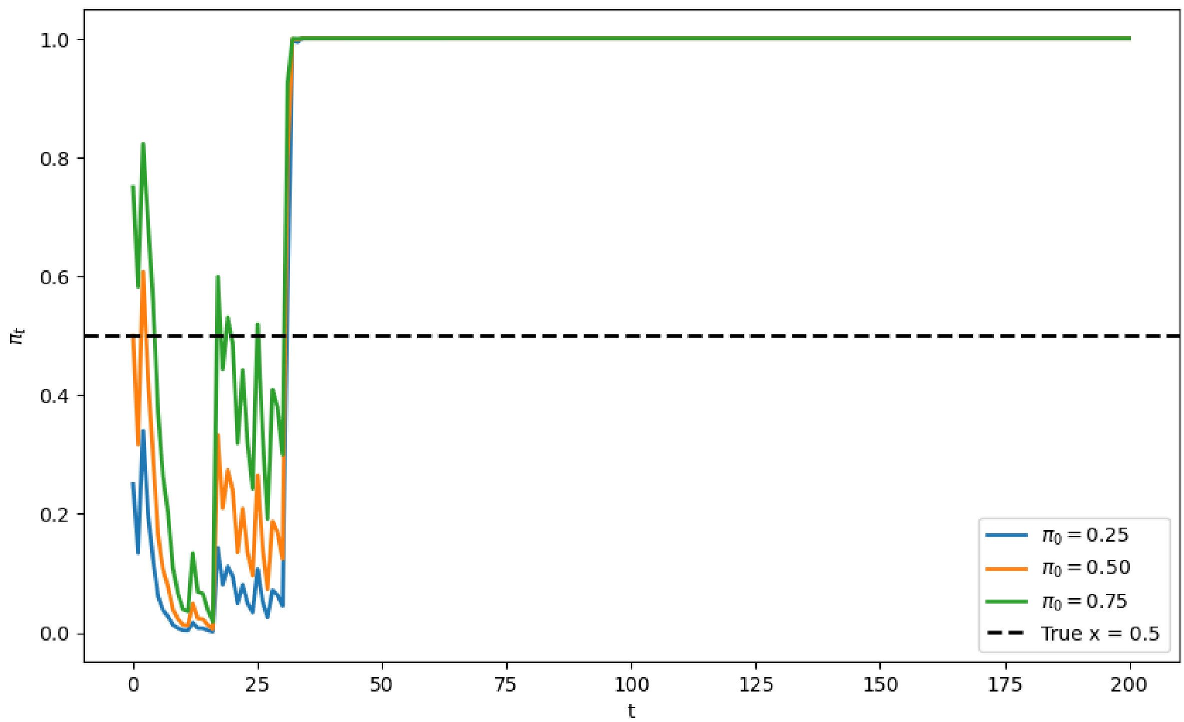
Later, we can use it to study how a statistician who knows that nature generates data from an x-mixture of f and g could construct maximum likelihood or Bayesian estimators of x along with the free parameters of f and g.

```
x_true = 0.5
T_mix = 200

# Three different priors with means 0.25, 0.5, 0.75
prior_params = [(1, 3), (1, 1), (3, 1)]
prior_means = [a/(a+b) for a, b in prior_params]

# Generate one path of observations from the mixture
set_seed()
w_mix = simulate_mixture_path(x_true, T_mix)
```

29.4.1 Behavior of π_t under wrong model

Let's study how the posterior probability π_t that nature permanently draws from f behaves when data are actually generated by an x-mixture of f and g.

```
fig, ax = plt.subplots(figsize=(10, 6))
T_plot = 200

for i, mean0 in enumerate(prior_means):
    π_wrong = np.empty(T_plot + 1)
    π_wrong[0] = mean0

    # Compute likelihood ratios for the mixture data
    for t in range(T_plot):
        l_t = f(w_mix[t]) / g(w_mix[t])
        π_wrong[t + 1] = update(π_wrong[t], l_t)

    ax.plot(range(T_plot + 1), π_wrong,
            label=fr'$\pi_0 = ${mean0:.2f}',
            color=colors[i], linewidth=2)

ax.axhline(y=x_true, color='black', linestyle='--',
           label=f'True x = {x_true}', linewidth=2)
ax.set_xlabel('t')
ax.set_ylabel(r'$\pi_t$')
ax.legend()
plt.show()
```



Evidently, π_t converges to 1.

This indicates that the model concludes that the data is generated by f.

Why does this happen?

Given $x = 0.5$, the data generating process is a mixture of f and g: $m(w) = \frac{1}{2}f(w) + \frac{1}{2}g(w)$.

Let's check the *KL divergence* of the mixture distribution m from both f and g.

```
def compute_KL(f, g):
    """
    Compute KL divergence KL(f, g)
    """
    integrand = lambda w: f(w) * np.log(f(w) / g(w))
    val, _ = quad(integrand, 1e-5, 1-1e-5)
    return val


def compute_div_m(f, g):
    """
    Compute Jensen-Shannon divergence
    """
    def m(w):
        return 0.5 * (f(w) + g(w))

    return compute_KL(m, f), compute_KL(m, g)


KL_f, KL_g = compute_div_m(f, g)

print(f'KL(m, f) = {KL_f:.3f}\nKL(m, g) = {KL_g:.3f}')
```

```
KL(m, f) = 0.073
KL(m, g) = 0.281
```

Since $KL(m, f) < KL(m, g)$, f is "closer" to the mixture distribution m.

Hence by our discussion on KL divergence and likelihood ratio process in *Likelihood Ratio Processes*, $\log(L_t) \to \infty$ as $t \to \infty$.

Now looking back to the key equation (29.3).

Consider the function

$$h(z) = \frac{\pi_0 z}{\pi_0 z + 1 - \pi_0}.$$

The limit $\lim_{z\to\infty} h(z)$ is 1.

Hence $\pi_t \to 1$ as $t \to \infty$ for any $\pi_0 \in (0, 1)$.

This explains what we observed in the plot above.

But how can we learn the true mixing parameter x?

This topic is taken up in *Incorrect Models*.

We explore how to learn the true mixing parameter x in the exercise of *Incorrect Models*.

29.5 Behavior of Posterior Probability $\{\pi_t\}$ Under Subjective Probability Distribution

We'll end this lecture by briefly studying what our Bayesian learner expects to learn under the subjective beliefs π_t cranked out by Bayes' law.

This will provide us with some perspective on our application of Bayes's law as a theory of learning.

As we shall see, at each time t, the Bayesian learner knows that he will be surprised.

But he expects that new information will not lead him to change his beliefs.

And it won't on average under his subjective beliefs.

We'll continue with our setting in which a McCall worker knows that successive draws of his wage are drawn from either F or G, but does not know which of these two distributions nature has drawn once-and-for-all before time 0.

We'll review and reiterate and rearrange some formulas that we have encountered above and in associated lectures.

The worker's initial beliefs induce a joint probability distribution over a potentially infinite sequence of draws $w_0, w_1, \ldots$.

Bayes' law is simply an application of laws of probability to compute the conditional distribution of the tth draw w_t conditional on $[w_0, \ldots, w_{t-1}]$.

After our worker puts a subjective probability π_{-1} on nature having selected distribution F, we have in effect assumed from the start that the decision maker **knows** the joint distribution for the process $\{w_t\}_{t=0}$.

We assume that the worker also knows the laws of probability theory.

A respectable view is that Bayes' law is less a theory of learning than a statement about the consequences of information inflows for a decision maker who thinks he knows the truth (i.e., a joint probability distribution) from the beginning.

29.5.1 Mechanical details again

At time 0 **before** drawing a wage offer, the worker attaches probability $\pi_{-1} \in (0, 1)$ to the distribution being F.

Before drawing a wage at time 0, the worker thus believes that the density of w_0 is

$$h(w_0; \pi_{-1}) = \pi_{-1} f(w_0) + (1 - \pi_{-1}) g(w_0).$$

Let $a \in \{f, g\}$ be an index that indicates whether nature chose permanently to draw from distribution f or from distribution g.

After drawing w_0, the worker uses Bayes' law to deduce that the posterior probability $\pi_0 = \text{Prob}(a = f | w_0)$ that the density is $f(w)$ is

$$\pi_0 = \frac{\pi_{-1} f(w_0)}{\pi_{-1} f(w_0) + (1 - \pi_{-1}) g(w_0)}.$$

More generally, after making the tth draw and having observed $w_t, w_{t-1}, \dots, w_0$, the worker believes that the probability that w_{t+1} is being drawn from distribution F is

$$\pi_t = \pi_t(w_t | \pi_{t-1}) \equiv \frac{\pi_{t-1} f(w_t)/g(w_t)}{\pi_{t-1} f(w_t)/g(w_t) + (1 - \pi_{t-1})} \tag{29.6}$$

or

$$\pi_t = \frac{\pi_{t-1} l_t(w_t)}{\pi_{t-1} l_t(w_t) + 1 - \pi_{t-1}}$$

and that the density of w_{t+1} conditional on $w_t, w_{t-1}, \dots, w_0$ is

$$h(w_{t+1}; \pi_t) = \pi_t f(w_{t+1}) + (1 - \pi_t) g(w_{t+1}).$$

Notice that

$$\begin{aligned} E(\pi_t | \pi_{t-1}) &= \int \Big[\frac{\pi_{t-1} f(w)}{\pi_{t-1} f(w) + (1 - \pi_{t-1}) g(w)} \Big] \Big[\pi_{t-1} f(w) + (1 - \pi_{t-1}) g(w) \Big] dw \\ &= \pi_{t-1} \int f(w) dw \\ &= \pi_{t-1}, \end{aligned}$$

so that the process π_t is a **martingale**.

Indeed, it is a **bounded martingale** because each π_t, being a probability, is between 0 and 1.

In the first line in the above string of equalities, the term in the first set of brackets is just π_t as a function of w_t, while the term in the second set of brackets is the density of w_t conditional on $w_{t-1}, \dots, w_0$ or equivalently conditional on the *sufficient statistic* π_{t-1} for $w_{t-1}, \dots, w_0$.

Notice that here we are computing $E(\pi_t | \pi_{t-1})$ under the **subjective** density described in the second term in brackets.

Because $\{\pi_t\}$ is a bounded martingale sequence, it follows from the **martingale convergence theorem** that π_t converges almost surely to a random variable in $[0, 1]$.

Practically, this means that probability one is attached to sample paths $\{\pi_t\}_{t=0}^{\infty}$ that converge.

According to the theorem, different sample paths can converge to different limiting values.

Thus, let $\{\pi_t(\omega)\}_{t=0}^{\infty}$ denote a particular sample path indexed by a particular $\omega \in \Omega$.

We can think of nature as drawing an $\omega \in \Omega$ from a probability distribution $\text{Prob}\Omega$ and then generating a single realization (or *simulation*) $\{\pi_t(\omega)\}_{t=0}^{\infty}$ of the process.

The limit points of $\{\pi_t(\omega)\}_{t=0}^{\infty}$ as $t \to +\infty$ are realizations of a random variable that is swept out as we sample ω from Ω and construct repeated draws of $\{\pi_t(\omega)\}_{t=0}^{\infty}$.

By staring at law of motion (29.5) or (29.6) , we can figure out some things about the probability distribution of the limit points

$$\pi_\infty(\omega) = \lim_{t\to+\infty} \pi_t(\omega).$$

Evidently, since the likelihood ratio $\ell(w_t)$ differs from 1 when $f \neq g$, as we have assumed, the only possible fixed points of (29.6) are

$$\pi_\infty(\omega) = 1$$

and

$$\pi_\infty(\omega) = 0$$

Thus, for some realizations, $\lim_{\to+\infty} \pi_t(\omega) = 1$ while for other realizations, $\lim_{\to+\infty} \pi_t(\omega) = 0$.

Now let's remember that $\{\pi_t\}_{t=0}^{\infty}$ is a martingale and apply the law of iterated expectations.

The law of iterated expectations implies

$$E_t \pi_{t+j} = \pi_t$$

and in particular

$$E_{-1} \pi_{t+j} = \pi_{-1}.$$

Applying the above formula to π_∞, we obtain

$$E_{-1} \pi_\infty(\omega) = \pi_{-1}$$

where the mathematical expectation E_{-1} here is taken with respect to the probability measure $\text{Prob}(\Omega)$.

Since the only two values that $\pi_\infty(\omega)$ can take are 1 and 0, we know that for some $\lambda \in [0, 1]$

$$\text{Prob}\Big(\pi_\infty(\omega) = 1\Big) = \lambda, \quad \text{Prob}\Big(\pi_\infty(\omega) = 0\Big) = 1 - \lambda$$

and consequently that

$$E_{-1} \pi_\infty(\omega) = \lambda \cdot 1 + (1-\lambda) \cdot 0 = \lambda$$

Combining this equation with equation (20), we deduce that the probability that $\text{Prob}(\Omega)$ attaches to $\pi_\infty(\omega)$ being 1 must be π_{-1}.

Thus, under the worker's subjective distribution, π_{-1} of the sample paths of $\{\pi_t\}$ will converge pointwise to 1 and $1 - \pi_{-1}$ of the sample paths will converge pointwise to 0.

29.5.2 Some simulations

Let's watch the martingale convergence theorem at work in some simulations of our learning model under the worker's subjective distribution.

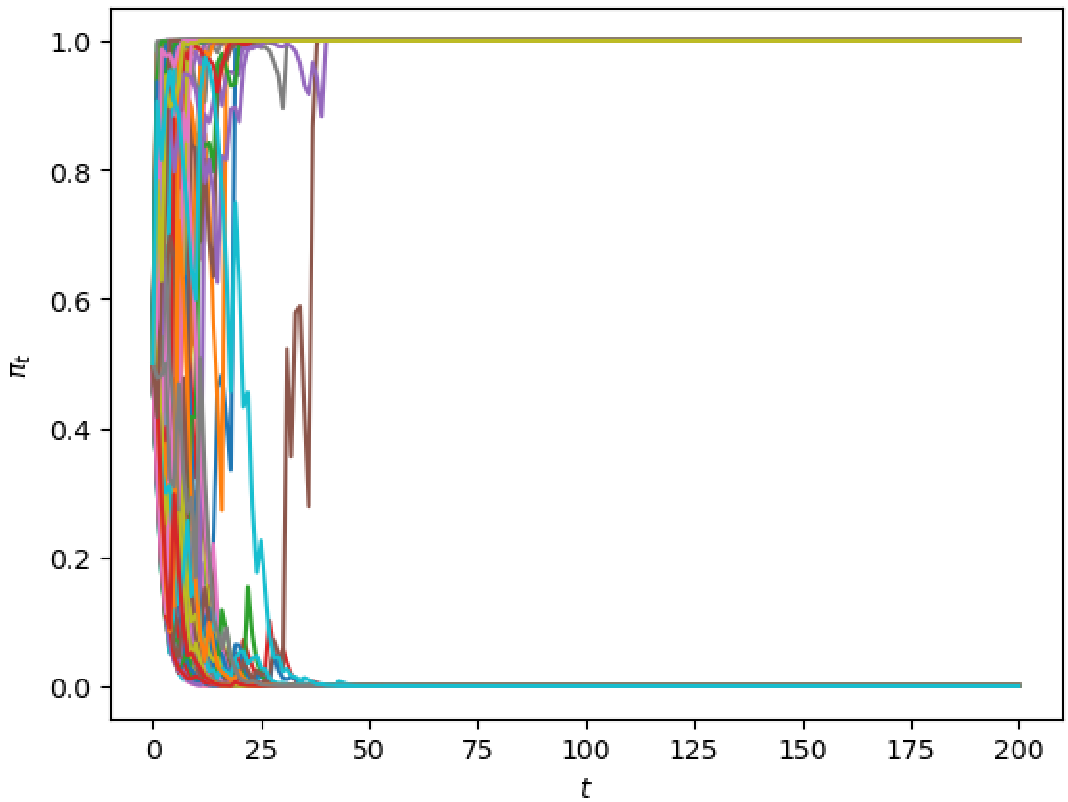
Let us simulate $\{\pi_t\}_{t=0}^{T}$, $\{w_t\}_{t=0}^{T}$ paths where for each $t \geq 0$, w_t is drawn from the subjective distribution

$$\pi_{t-1} f(w_t) + (1 - \pi_{t-1}) g(w_t)$$

We'll plot a large sample of paths.

```
@jit
def martingale_simulate(π0, N=5000, T=200):

    π_path = np.empty((N,T+1))
    w_path = np.empty((N,T))
    π_path[:,0] = π0

    for n in range(N):
        π = π0
        for t in range(T):
            # draw w
            if np.random.rand() <= π:
                w = np.random.beta(F_a, F_b)
            else:
                w = np.random.beta(G_a, G_b)
            π = π*f(w)/g(w)/(π*f(w)/g(w) + 1 - π)
            π_path[n,t+1] = π
            w_path[n,t] = w

    return π_path, w_path

def fraction_0_1(π0, N, T, decimals):

    π_path, w_path = martingale_simulate(π0, N=N, T=T)
    values, counts = np.unique(np.round(π_path[:,-1], decimals=decimals), return_
↪counts=True)
    return values, counts

def create_table(π0s, N=10000, T=500, decimals=2):

    outcomes = []
    for π0 in π0s:
        values, counts = fraction_0_1(π0, N=N, T=T, decimals=decimals)
        freq = counts/N
        outcomes.append(dict(zip(values, freq)))
    table = pd.DataFrame(outcomes).sort_index(axis=1).fillna(0)
    table.index = π0s
    return table

# simulate
T = 200
π0 = .5

π_path, w_path = martingale_simulate(π0=π0, T=T, N=10000)
```

```
fig, ax = plt.subplots()
for i in range(100):
    ax.plot(range(T+1), π_path[i, :])

ax.set_xlabel('$t$')
ax.set_ylabel(r'$\pi_t$')
plt.show()
```

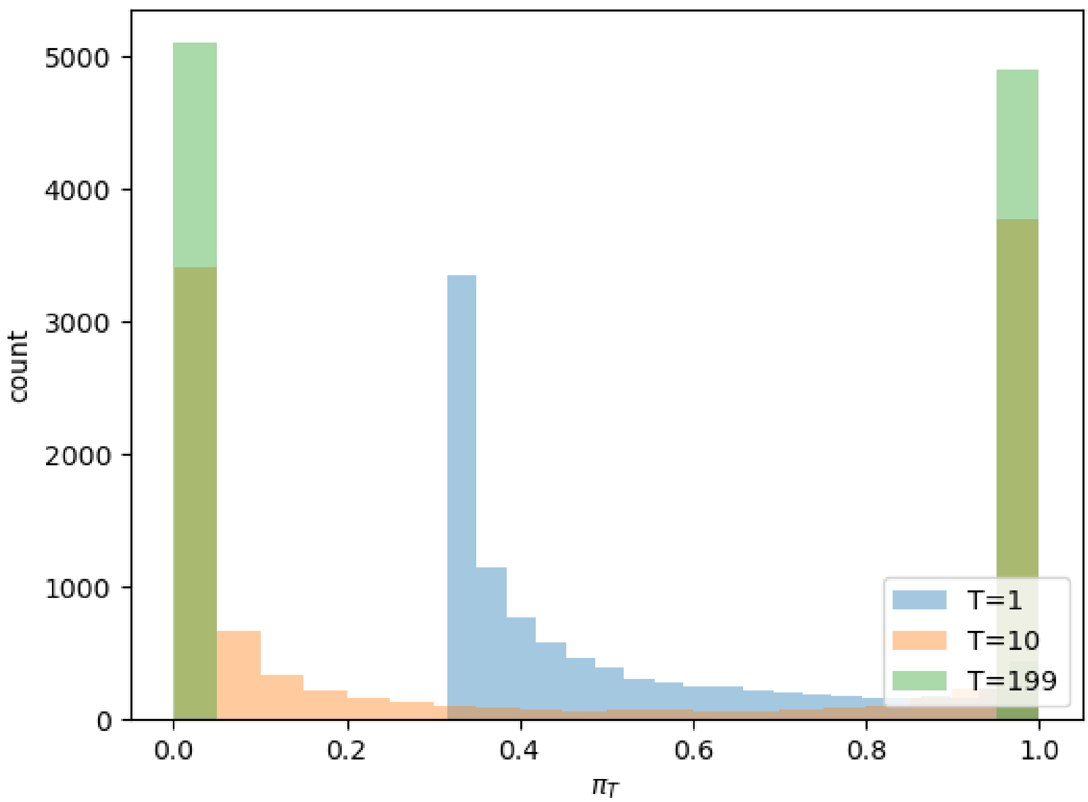


The above graph indicates that

- each of paths converges
- some of the paths converge to 1
- some of the paths converge to 0
- none of the paths converge to a limit point not equal to 0 or 1

Convergence actually occurs pretty fast, as the following graph of the cross-ensemble distribution of π_t for various small t's indicates.

```
fig, ax = plt.subplots()
for t in [1, 10, T-1]:
    ax.hist(π_path[:,t], bins=20, alpha=0.4, label=f'T={t}')

ax.set_ylabel('count')
ax.set_xlabel(r'$\pi_T$')
ax.legend(loc='lower right')
plt.show()
```



Evidently, by $t = 199$, π_t has converged to either 0 or 1.

The fraction of paths that have converged to 1 is .5

The fractions of paths that have converged to 0 is also .5.

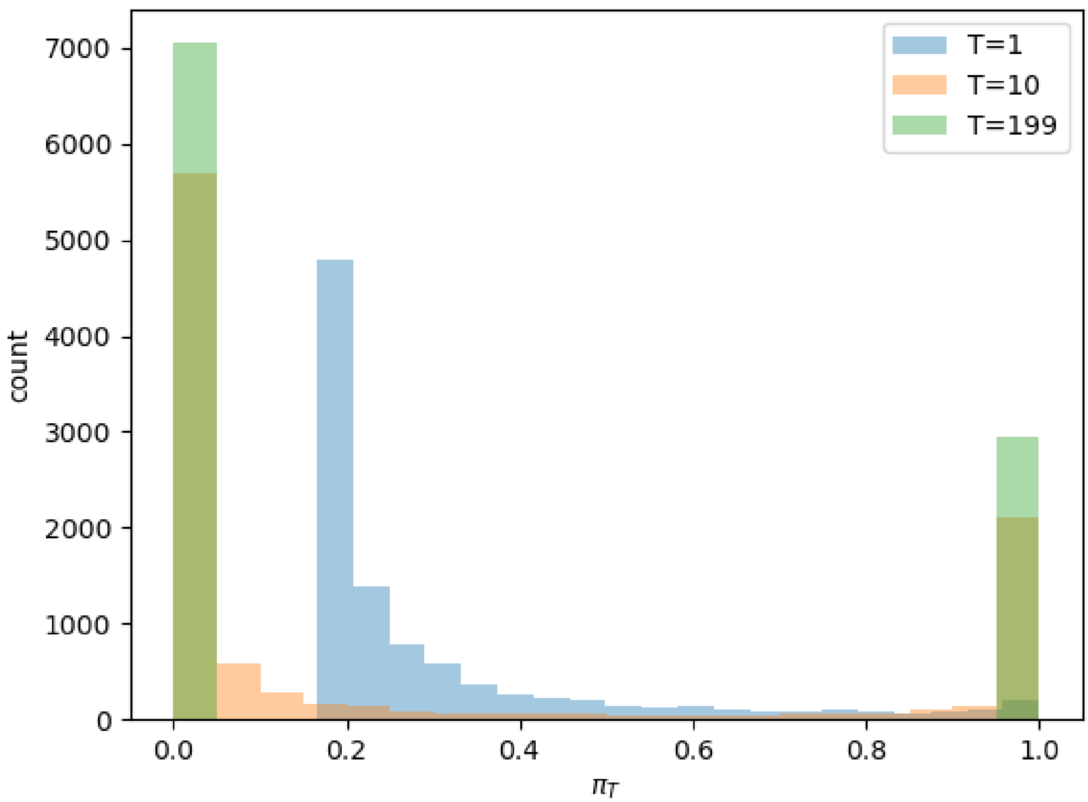
Does the fraction .5 ring a bell?

Yes, it does: it equals the value of $\pi_0 = .5$ that we used to generate each sequence in the ensemble.

So let's change π_0 to .3 and watch what happens to the distribution of the ensemble of π_t's for various t's.

```
# simulate
T = 200
π0 = .3

π_path3, w_path3 = martingale_simulate(π0=π0, T=T, N=10000)
```

```
fig, ax = plt.subplots()
for t in [1, 10, T-1]:
    ax.hist(π_path3[:,t], bins=20, alpha=0.4, label=f'T={t}')

ax.set_ylabel('count')
ax.set_xlabel(r'$\pi_T$')
ax.legend(loc='upper right')
plt.show()
```

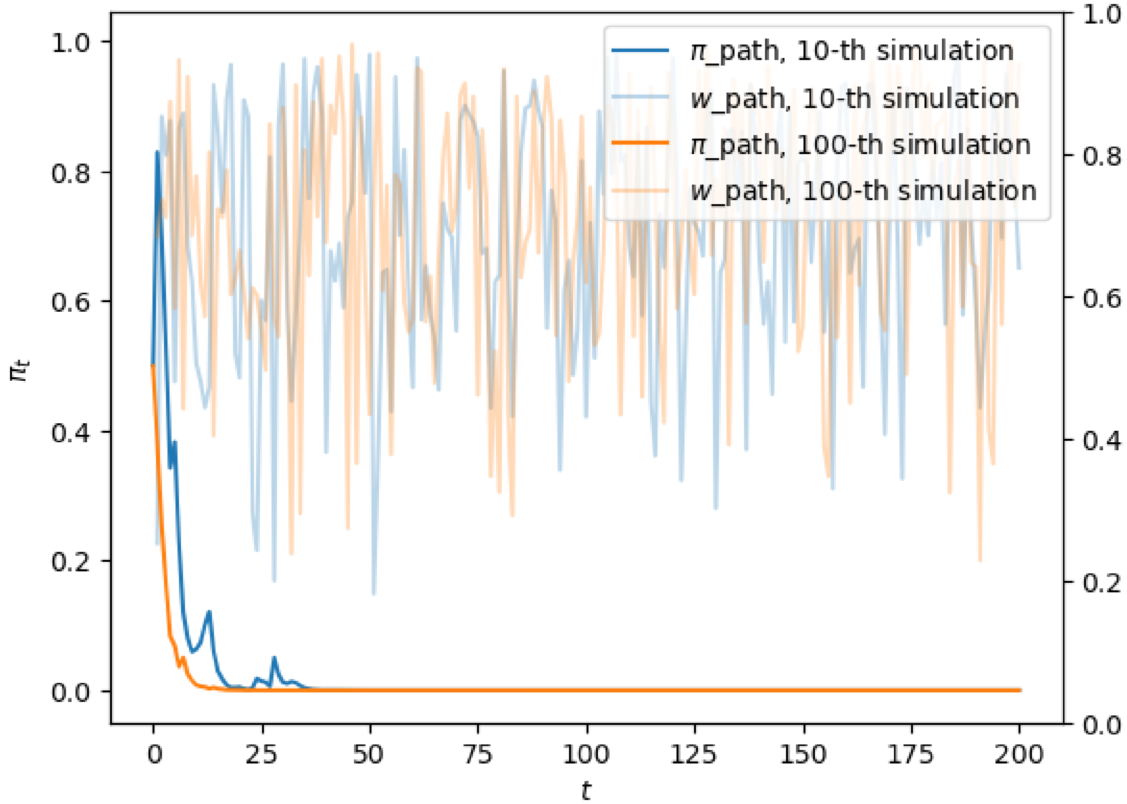


For the preceding ensemble that assumed $\pi_0 = .5$, the following graph shows two paths of w_t's and the π_t sequences that gave rise to them.

Notice that one of the paths involves systematically higher w_t's, outcomes that push π_t upward.

The luck of the draw early in a simulation pushes the subjective distribution to draw from F more frequently along a sample path, and this pushes π_t toward 0.

```
fig, ax = plt.subplots()
for i, j in enumerate([10, 100]):
    ax.plot(range(T+1), π_path[j,:], color=colors[i], label=fr'$\pi$_path, {j}-th␣
↪simulation')
    ax.plot(range(1,T+1), w_path[j,:], color=colors[i], label=fr'$w$_path, {j}-th␣
↪simulation', alpha=0.3)

ax.legend(loc='upper right')
ax.set_xlabel('$t$')
ax.set_ylabel(r'$\pi_t$')
ax2 = ax.twinx()
ax2.set_ylabel("$w_t$")
plt.show()
```



29.6 Initial Prior is Verified by Paths Drawn from Subjective Conditional Densities

Now let's use our Python code to generate a table that checks out our earlier claims about the probability distribution of the pointwise limits $\pi_\infty(\omega)$.

We'll use our simulations to generate a histogram of this distribution.

In the following table, the left column in bold face reports an assumed value of π_{-1}.

The second column reports the fraction of $N = 10000$ simulations for which π_t had converged to 0 at the terminal date $T = 500$ for each simulation.

The third column reports the fraction of $N = 10000$ simulations for which π_t had converged to 1 at the terminal date $T = 500$ for each simulation.

```
# create table
table = create_table(list(np.linspace(0,1,11)), N=10000, T=500)
table
```

	0.0	1.0
0.0	1.0000	0.0000
0.1	0.8984	0.1016
0.2	0.8000	0.2000
0.3	0.6981	0.3019
0.4	0.6004	0.3996

(continues on next page)

(continued from previous page)

```
0.5  0.4968  0.5032
0.6  0.3995  0.6005
0.7  0.3007  0.6993
0.8  0.2074  0.7926
0.9  0.0964  0.9036
1.0  0.0000  1.0000
```

The fraction of simulations for which π_t had converged to 1 is indeed always close to π_{-1}, as anticipated.

29.7 Drilling Down a Little Bit

To understand how the local dynamics of π_t behaves, it is enlightening to consult the variance of π_t conditional on π_{t-1}.

Under the subjective distribution this conditional variance is defined as

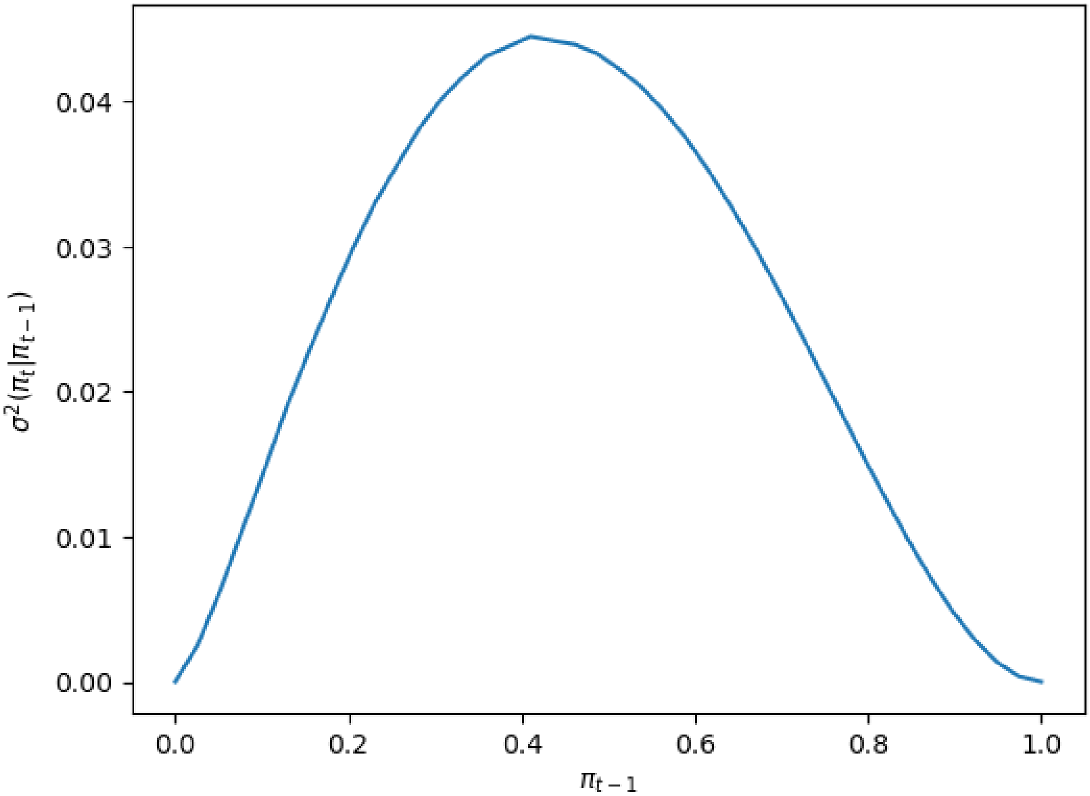
$$\sigma^2(\pi_t|\pi_{t-1}) = \int \Big[\frac{\pi_{t-1}f(w)}{\pi_{t-1}f(w) + (1-\pi_{t-1})g(w)} - \pi_{t-1}\Big]^2 \Big[\pi_{t-1}f(w) + (1-\pi_{t-1})g(w)\Big] dw$$

We can use a Monte Carlo simulation to approximate this conditional variance.

We approximate it for a grid of points $\pi_{t-1} \in [0, 1]$.

Then we'll plot it.

```python
@jit
def compute_cond_var(pi, mc_size=int(1e6)):
    # create monte carlo draws
    mc_draws = np.zeros(mc_size)

    for i in prange(mc_size):
        if np.random.rand() <= pi:
            mc_draws[i] = np.random.beta(F_a, F_b)
        else:
            mc_draws[i] = np.random.beta(G_a, G_b)

    dev = pi*f(mc_draws)/(pi*f(mc_draws) + (1-pi)*g(mc_draws)) - pi
    return np.mean(dev**2)

pi_array = np.linspace(0, 1, 40)
cond_var_array = []

for pi in pi_array:
    cond_var_array.append(compute_cond_var(pi))

fig, ax = plt.subplots()
ax.plot(pi_array, cond_var_array)
ax.set_xlabel(r'$\pi_{t-1}$')
ax.set_ylabel(r'$\sigma^{2}(\pi_{t}\vert \pi_{t-1})$')
plt.show()
```



The shape of the conditional variance as a function of π_{t-1} is informative about the behavior of sample paths of $\{\pi_t\}$.

Notice how the conditional variance approaches 0 for π_{t-1} near either 0 or 1.

The conditional variance is nearly zero only when the agent is almost sure that w_t is drawn from F, or is almost sure it is drawn from G.

29.8 Related Lectures

This lecture has been devoted to building some useful infrastructure that will help us understand inferences that are the foundations of results described in *this lecture* and *this lecture* and *this lecture*.

CHAPTER

THIRTY

INCORRECT MODELS

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

```
!pip install numpyro jax
```

30.1 Overview

This is a sequel to *this quantecon lecture*.

We discuss two ways to create a compound lottery and their consequences.

A compound lottery can be said to create a *mixture distribution*.

Our two ways of constructing a compound lottery will differ in their **timing**.

- in one, mixing between two possible probability distributions will occur once and all at the beginning of time
- in the other, mixing between the same two possible probability distributions will occur each period

The statistical setting is close but not identical to the problem studied in that quantecon lecture.

In that lecture, there were two i.i.d. processes that could possibly govern successive draws of a non-negative random variable W.

Nature decided once and for all whether to make a sequence of IID draws from either f or from g.

That lecture studied an agent who knew both f and g but did not know which distribution nature chose at time -1.

The agent represented that ignorance by assuming that nature had chosen f or g by flipping an unfair coin that put probability π_{-1} on probability distribution f.

That assumption allowed the agent to construct a subjective joint probability distribution over the random sequence $\{W_t\}_{t=0}^{\infty}$.

We studied how the agent would then use the laws of conditional probability and an observed history $w^t = \{w_s\}_{s=0}^t$ to form

$$\pi_t = E[\text{nature chose distribution} f | w^t], \quad t = 0, 1, 2, \ldots$$

However, in the setting of this lecture, that rule imputes to the agent an incorrect model.

The reason is that now the wage sequence is actually described by a different statistical model.

Thus, we change the *quantecon lecture* specification in the following way.

Now, **each period** $t \geq 0$, nature flips a possibly unfair coin that comes up f with probability α and g with probability $1 - \alpha$.

Thus, nature perpetually draws from the **mixture distribution** with c.d.f.

$$H(w) = \alpha F(w) + (1 - \alpha) G(w), \quad \alpha \in (0, 1)$$

We'll study two agents who try to learn about the wage process, but who use different statistical models.

Both types of agent know f and g but neither knows α.

Our first type of agent erroneously thinks that at time -1 nature once and for all chose f or g and thereafter permanently draws from that distribution.

Our second type of agent knows, correctly, that nature mixes f and g with mixing probability $\alpha \in (0, 1)$ each period, though the agent doesn't know the mixing parameter.

Our first type of agent applies the learning algorithm described in *this quantecon lecture*.

In the context of the statistical model that prevailed in that lecture, that was a good learning algorithm and it enabled the Bayesian learner eventually to learn the distribution that nature had drawn at time -1.

This is because the agent's statistical model was *correct* in the sense of being aligned with the data generating process.

But in the present context, our type 1 decision maker's model is incorrect because the model h that actually generates the data is neither f nor g and so is beyond the support of the models that the agent thinks are possible.

Nevertheless, we'll see that our first type of agent muddles through and eventually learns something interesting and useful, even though it is not *true*.

Instead, it turns out that our type 1 agent who is armed with a wrong statistical model ends up learning whichever probability distribution, f or g, is in a special sense *closest* to the h that actually generates the data.

We'll tell the sense in which it is closest.

Our second type of agent understands that nature mixes between f and g each period with a fixed mixing probability α.

But the agent doesn't know α.

The agent sets out to learn α using Bayes' law applied to his model.

His model is correct in the sense that it includes the actual data generating process h as a possible distribution.

In this lecture, we'll learn about

- how nature can *mix* between two distributions f and g to create a new distribution h.
- The Kullback-Leibler statistical divergence https://en.wikipedia.org/wiki/Kullback–Leibler_divergence that governs statistical learning under an incorrect statistical model
- A useful Python function `numpy.searchsorted` that, in conjunction with a uniform random number generator, can be used to sample from an arbitrary distribution

As usual, we'll start by importing some Python tools.

```python
import matplotlib.pyplot as plt
import numpy as np
from numba import vectorize, jit
from math import gamma
import pandas as pd
import scipy.stats as sp
from scipy.integrate import quad

import seaborn as sns
colors = sns.color_palette()

import numpyro
import numpyro.distributions as dist
from numpyro.infer import MCMC, NUTS

import jax.numpy as jnp
from jax import random

np.random.seed(142857)

@jit
def set_seed():
    np.random.seed(142857)
set_seed()
```

Let's use Python to generate two beta distributions

```python
# Parameters in the two beta distributions.
F_a, F_b = 1, 1
G_a, G_b = 3, 1.2

@vectorize
def p(x, a, b):
    r = gamma(a + b) / (gamma(a) * gamma(b))
    return r * x** (a-1) * (1 - x) ** (b-1)

# The two density functions.
f = jit(lambda x: p(x, F_a, F_b))
g = jit(lambda x: p(x, G_a, G_b))
```

```python
@jit
def simulate(a, b, T=50, N=500):
    '''
    Generate N sets of T observations of the likelihood ratio,
    return as N x T matrix.

    '''

    l_arr = np.empty((N, T))

    for i in range(N):

        for j in range(T):
            w = np.random.beta(a, b)
            l_arr[i, j] = f(w) / g(w)

    return l_arr
```

We'll also use the following Python code to prepare some informative simulations

```
l_arr_g = simulate(G_a, G_b, N=50000)
l_seq_g = np.cumprod(l_arr_g, axis=1)
```

```
l_arr_f = simulate(F_a, F_b, N=50000)
l_seq_f = np.cumprod(l_arr_f, axis=1)
```

30.2 Sampling from Compound Lottery H

We implement two methods to draw samples from our mixture model $\alpha F + (1 - \alpha)G$.

We'll generate samples using each of them and verify that they match well.

Here is pseudo code for a direct "method 1" for drawing from our compound lottery:

- Step one:
  - use the numpy.random.choice function to flip an unfair coin that selects distribution F with prob α and G with prob $1 - \alpha$
- Step two:
  - draw from either F or G, as determined by the coin flip.
- Step three:
  - put the first two steps in a big loop and do them for each realization of w

Our second method uses a uniform distribution and the following fact that we also described and used in the quantecon lecture https://python.quantecon.org/prob_matrix.html:

- If a random variable X has c.d.f. F, then a random variable $F^{-1}(U)$ also has c.d.f. F, where U is a uniform random variable on $[0, 1]$.

In other words, if $X \sim F(x)$ we can generate a random sample from F by drawing a random sample from a uniform distribution on $[0, 1]$ and computing $F^{-1}(U)$.

We'll use this fact in conjunction with the `numpy.searchsorted` command to sample from H directly.

See https://numpy.org/doc/stable/reference/generated/numpy.searchsorted.html for the `searchsorted` function.

See the Mr. P Solver video on Monte Carlo simulation to see other applications of this powerful trick.

In the Python code below, we'll use both of our methods and confirm that each of them does a good job of sampling from our target mixture distribution.

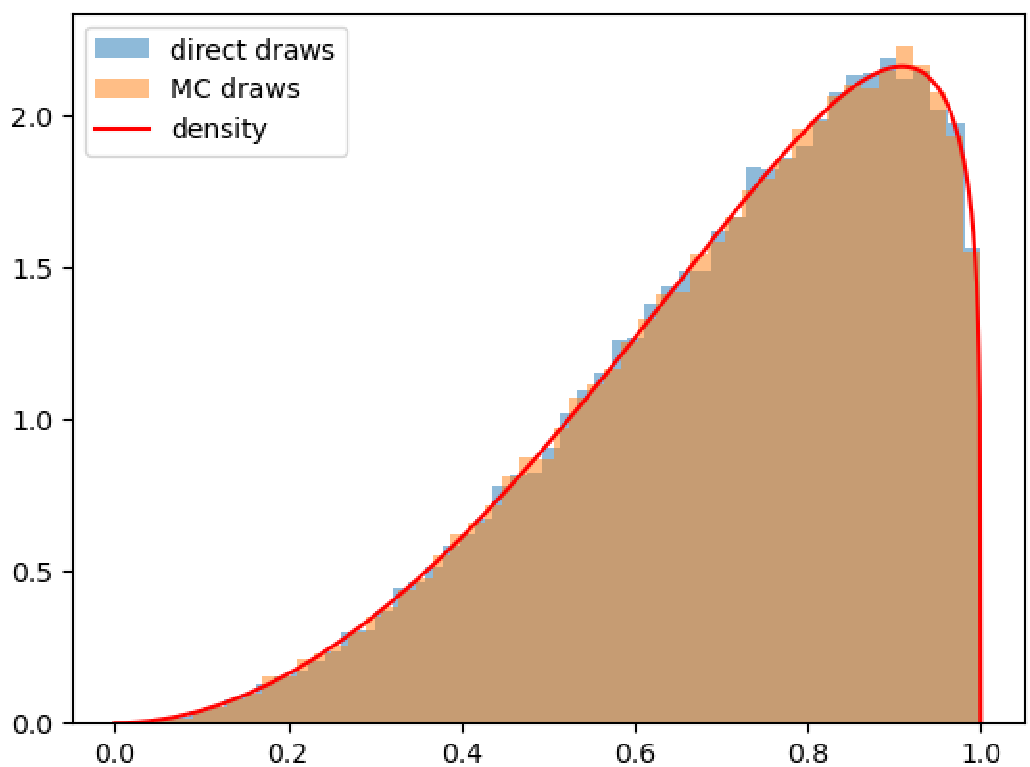
```
@jit
def draw_lottery(p, N):
    "Draw from the compound lottery directly."

    draws = []
    for i in range(0, N):
        if np.random.rand()<=p:
            draws.append(np.random.beta(F_a, F_b))
        else:
            draws.append(np.random.beta(G_a, G_b))
    return np.array(draws)

def draw_lottery_MC(p, N):
```

(continues on next page)

(continued from previous page)

```
    "Draw from the compound lottery using the Monte Carlo trick."

    xs = np.linspace(1e-8,1-(1e-8),10000)
    CDF = p*sp.beta.cdf(xs, F_a, F_b) + (1-p)*sp.beta.cdf(xs, G_a, G_b)

    Us = np.random.rand(N)
    draws = xs[np.searchsorted(CDF[:-1], Us)]
    return draws
```

```
# verify
N = 100000
α = 0.0

sample1 = draw_lottery(α, N)
sample2 = draw_lottery_MC(α, N)

# plot draws and density function
plt.hist(sample1, 50, density=True, alpha=0.5, label='direct draws')
plt.hist(sample2, 50, density=True, alpha=0.5, label='MC draws')

xs = np.linspace(0,1,1000)
plt.plot(xs, α*f(xs)+(1-α)*g(xs), color='red', label='density')

plt.legend()
plt.show()
```



30.3 Type 1 Agent

We'll now study what our type 1 agent learns

Remember that our type 1 agent uses the wrong statistical model, thinking that nature mixed between f and g once and for all at time -1.

The type 1 agent thus uses the learning algorithm studied in *this quantecon lecture*.

We'll briefly review that learning algorithm now.

Let π_t be a Bayesian posterior defined as

$$\pi_t = \text{Prob}(q = f|w^t)$$

The likelihood ratio process plays a principal role in the formula that governs the evolution of the posterior probability π_t, an instance of **Bayes' Law**.

Bayes' law implies that $\{\pi_t\}$ obeys the recursion

$$\pi_t = \frac{\pi_{t-1} l_t(w_t)}{\pi_{t-1} l_t(w_t) + 1 - \pi_{t-1}} \tag{30.1}$$

with π_0 being a Bayesian prior probability that $q = f$, i.e., a personal or subjective belief about q based on our having seen no data.

Below we define a Python function that updates belief π using likelihood ratio ℓ according to recursion (30.1)

```python
@jit
def update(π, l):
    "Update π using likelihood l"

    # Update belief
    π = π * l / (π * l + 1 - π)

    return π
```

Formula (30.1) can be generalized by iterating on it and thereby deriving an expression for the time t posterior π_{t+1} as a function of the time 0 prior π_0 and the likelihood ratio process $L(w^{t+1})$ at time t.

To begin, notice that the updating rule

$$\pi_{t+1} = \frac{\pi_t \ell(w_{t+1})}{\pi_t \ell(w_{t+1}) + (1 - \pi_t)}$$

implies

$$\begin{aligned}
\frac{1}{\pi_{t+1}} &= \frac{\pi_t \ell(w_{t+1}) + (1 - \pi_t)}{\pi_t \ell(w_{t+1})} \\
&= 1 - \frac{1}{\ell(w_{t+1})} + \frac{1}{\ell(w_{t+1})} \frac{1}{\pi_t}. \\
\Rightarrow \frac{1}{\pi_{t+1}} - 1 &= \frac{1}{\ell(w_{t+1})} \left(\frac{1}{\pi_t} - 1 \right).
\end{aligned}$$

Therefore

$$\frac{1}{\pi_{t+1}} - 1 = \frac{1}{\prod_{i=1}^{t+1} \ell(w_i)} \left(\frac{1}{\pi_0} - 1 \right) = \frac{1}{L(w^{t+1})} \left(\frac{1}{\pi_0} - 1 \right).$$

Since $\pi_0 \in (0,1)$ and $L\left(w^{t+1}\right) > 0$, we can verify that $\pi_{t+1} \in (0,1)$.

After rearranging the preceding equation, we can express π_{t+1} as a function of $L\left(w^{t+1}\right)$, the likelihood ratio process at $t+1$, and the initial prior π_0

$$\pi_{t+1} = \frac{\pi_0 L\left(w^{t+1}\right)}{\pi_0 L\left(w^{t+1}\right) + 1 - \pi_0}. \tag{30.2}$$

Formula (30.2) generalizes formula (30.1).

Formula (30.2) can be regarded as a one step revision of prior probability π_0 after seeing the batch of data $\{w_i\}_{i=1}^{t+1}$.

30.4 What a type 1 Agent Learns when Mixture H Generates Data

We now study what happens when the mixture distribution $h;\alpha$ truly generated the data each period.

The sequence π_t continues to converge, despite the agent's misspecified model, and the limit is either 0 or 1.

This is true even though in truth nature always mixes between f and g.

After verifying that claim about possible limit points of π_t sequences, we'll drill down and study what fundamental force determines the limiting value of π_t.

Let's set a value of α and then watch how π_t evolves.

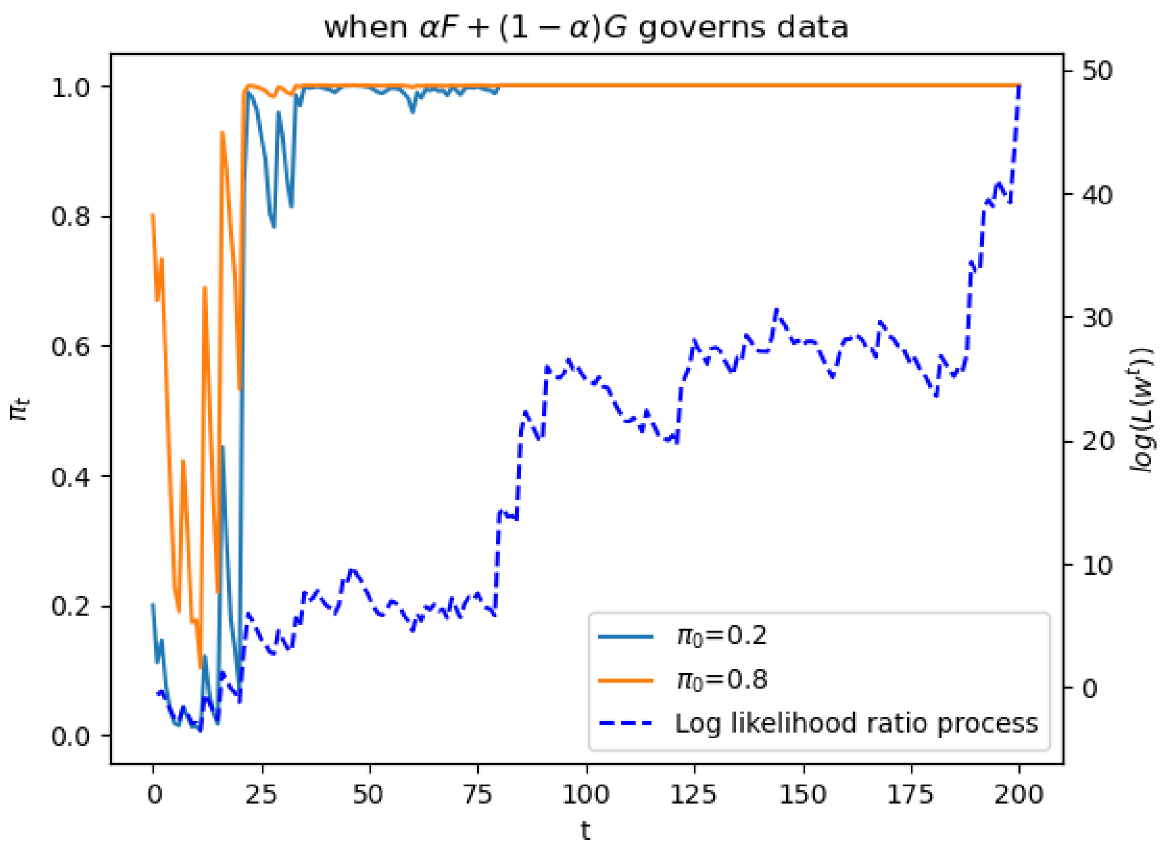
```
def simulate_mixed(α, T=50, N=500):
    """
    Generate N sets of T observations of the likelihood ratio,
    return as N x T matrix, when the true density is mixed h;α
    """

    w_s = draw_lottery(α, N*T).reshape(N, T)
    l_arr = f(w_s) / g(w_s)

    return l_arr

def plot_π_seq(α, π1=0.2, π2=0.8, T=200):
    """
    Compute and plot π_seq and the log likelihood ratio process
    when the mixed distribution governs the data.
    """

    l_arr_mixed = simulate_mixed(α, T=T, N=50)
    l_seq_mixed = np.cumprod(l_arr_mixed, axis=1)

    T = l_arr_mixed.shape[1]
    π_seq_mixed = np.empty((2, T+1))
    π_seq_mixed[:, 0] = π1, π2

    for t in range(T):
        for i in range(2):
            π_seq_mixed[i, t+1] = update(π_seq_mixed[i, t], l_arr_mixed[0, t])

    # plot
    fig, ax1 = plt.subplots()
    for i in range(2):
        ax1.plot(range(T+1), π_seq_mixed[i, :], label=rf"$\pi_0$={π_seq_mixed[i, 0]}")
```

(continues on next page)

(continued from previous page)

```
    ax1.plot(np.nan, np.nan,  '--', color='b', label='Log likelihood ratio process')
    ax1.set_ylabel(r"$\pi_t$")
    ax1.set_xlabel("t")
    ax1.legend()
    ax1.set_title("when $\\alpha F + (1-\\alpha)G$ governs data")

    ax2 = ax1.twinx()
    ax2.plot(range(1, T+1), np.log(l_seq_mixed[0, :]), '--', color='b')
    ax2.set_ylabel("$log(L(w^{t}))$")

    plt.show()
```

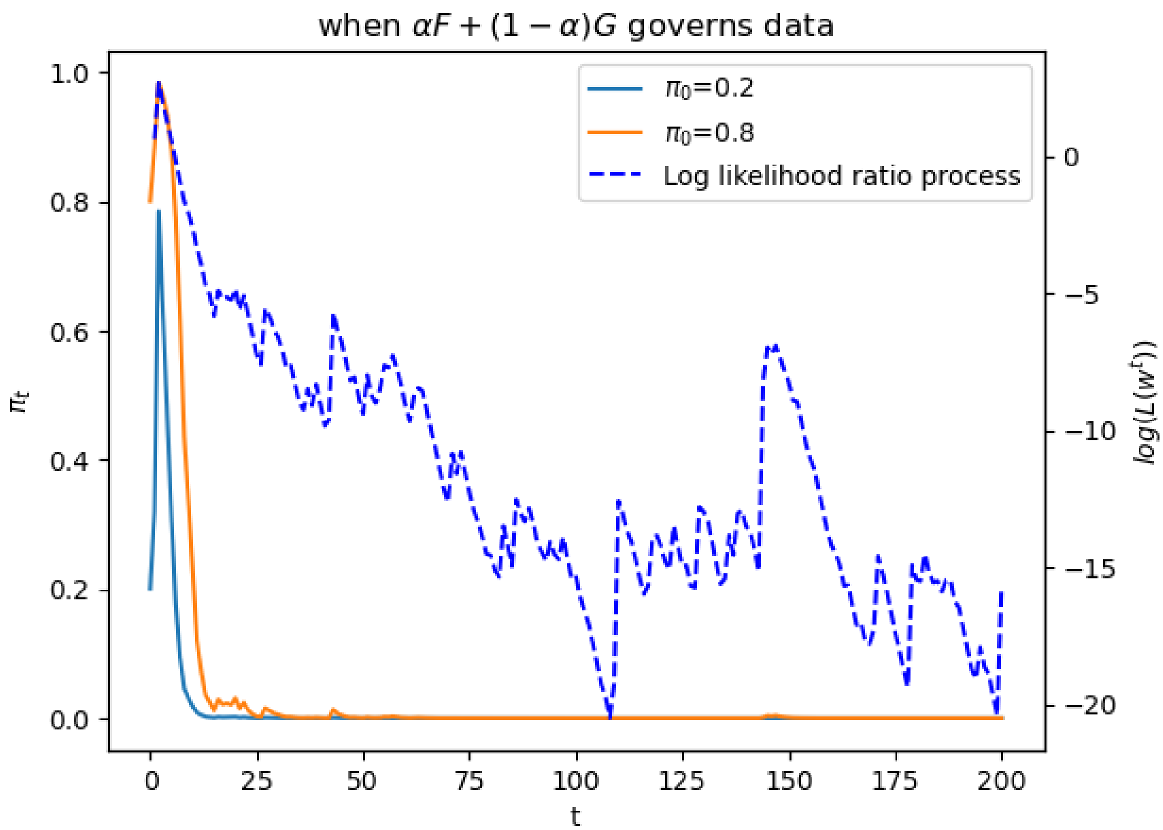
```
plot_π_seq(α = 0.6)
```



The above graph shows a sample path of the log likelihood ratio process as the blue dotted line, together with sample paths of π_t that start from two distinct initial conditions.

Let's see what happens when we change α

```
plot_π_seq(α = 0.2)
```



Evidently, α is having a big effect on the destination of π_t as $t \to +\infty$

30.5 Kullback-Leibler Divergence Governs Limit of π_t

To understand what determines whether the limit point of π_t is 0 or 1 and how the answer depends on the true value of the mixing probability $\alpha \in (0,1)$ that generates

$$h(w) \equiv h(w|\alpha) = \alpha f(w) + (1-\alpha) g(w)$$

we shall compute the following two Kullback-Leibler divergences

$$KL_g(\alpha) = \int \log \left(\frac{h(w)}{g(w)} \right) h(w) dw$$

and

$$KL_f(\alpha) = \int \log \left(\frac{h(w)}{f(w)} \right) h(w) dw$$

We shall plot both of these functions against α as we use α to vary $h(w) = h(w|\alpha)$.

The limit of π_t is determined by

$$\min_{f,g} \{KL_g, KL_f\}$$

The only possible limits are 0 and 1.

As $t \to +\infty$, π_t goes to one if and only if $KL_f < KL_g$

```
@vectorize
def KL_g(α):
    "Compute the KL divergence KL(h, g)."
    err = 1e-8                              # to avoid 0 at end points
    ws = np.linspace(err, 1-err, 10000)
    gs, fs = g(ws), f(ws)
    hs = α*fs + (1-α)*gs
    return np.sum(np.log(hs/gs)*hs)/10000

@vectorize
def KL_f(α):
    "Compute the KL divergence KL(h, f)."
    err = 1e-8                              # to avoid 0 at end points
    ws = np.linspace(err, 1-err, 10000)
    gs, fs = g(ws), f(ws)
    hs = α*fs + (1-α)*gs
    return np.sum(np.log(hs/fs)*hs)/10000


# compute KL using quad in Scipy
def KL_g_quad(α):
    "Compute the KL divergence KL(h, g) using scipy.integrate."
    h = lambda x: α*f(x) + (1-α)*g(x)
    return quad(lambda x: h(x) * np.log(h(x)/g(x)), 0, 1)[0]

def KL_f_quad(α):
    "Compute the KL divergence KL(h, f) using scipy.integrate."
    h = lambda x: α*f(x) + (1-α)*g(x)
    return quad(lambda x: h(x) * np.log(h(x)/f(x)), 0, 1)[0]

# vectorize
KL_g_quad_v = np.vectorize(KL_g_quad)
KL_f_quad_v = np.vectorize(KL_f_quad)


# Let us find the limit point
def π_lim(α, T=5000, π_0=0.4):
    "Find limit of π sequence."
    π_seq = np.zeros(T+1)
    π_seq[0] = π_0
    l_arr = simulate_mixed(α, T, N=1)[0]

    for t in range(T):
        π_seq[t+1] = update(π_seq[t], l_arr[t])
    return π_seq[-1]

π_lim_v = np.vectorize(π_lim)
```

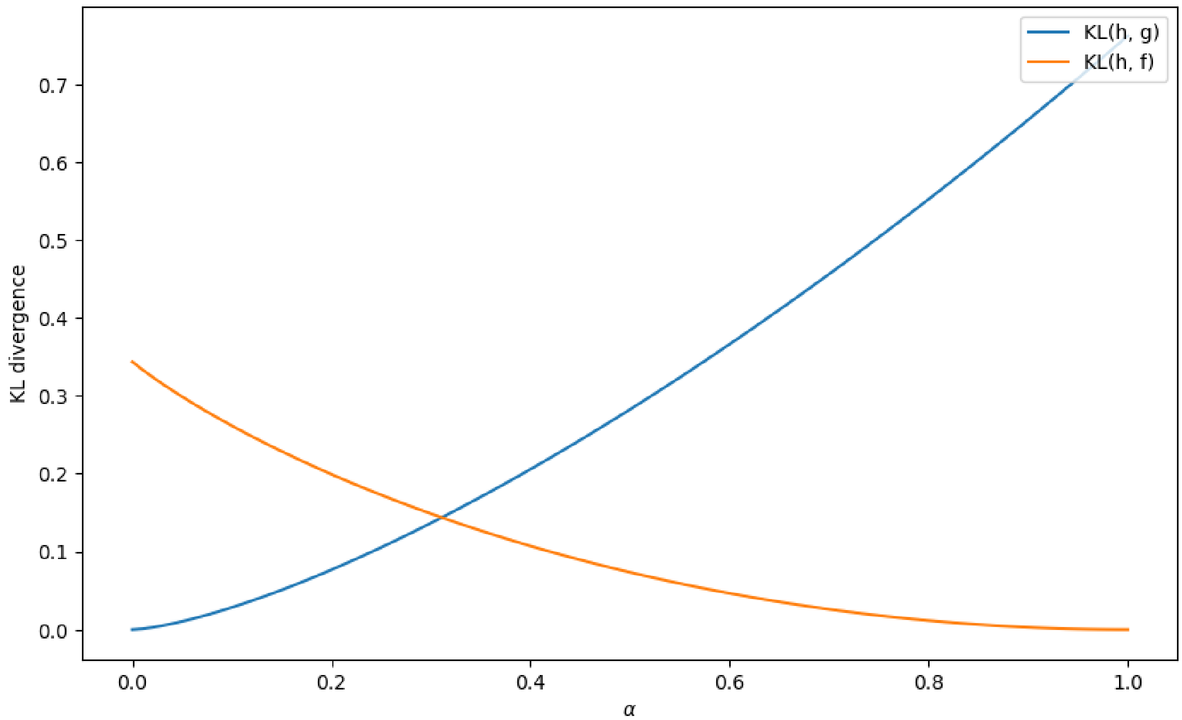
Let us first plot the KL divergences $KL_g(\alpha), KL_f(\alpha)$ for each α.

```
α_arr = np.linspace(0, 1, 100)
KL_g_arr = KL_g(α_arr)
KL_f_arr = KL_f(α_arr)

fig, ax = plt.subplots(1, figsize=[10, 6])

ax.plot(α_arr, KL_g_arr, label='KL(h, g)')
```

(continues on next page)

(continued from previous page)

```
ax.plot(α_arr, KL_f_arr, label='KL(h, f)')
ax.set_ylabel('KL divergence')
ax.set_xlabel(r'$\alpha$')

ax.legend(loc='upper right')
plt.show()
```



Let's compute an α for which the KL divergence between h and g is the same as that between h and f.

```
# where KL_f = KL_g
discretion = α_arr[np.argmin(np.abs(KL_g_arr-KL_f_arr))]
```

We can compute and plot the convergence point π_∞ for each α to verify that the convergence is indeed governed by the KL divergence.

The blue circles show the limiting values of π_t that simulations discover for different values of α recorded on the x axis.

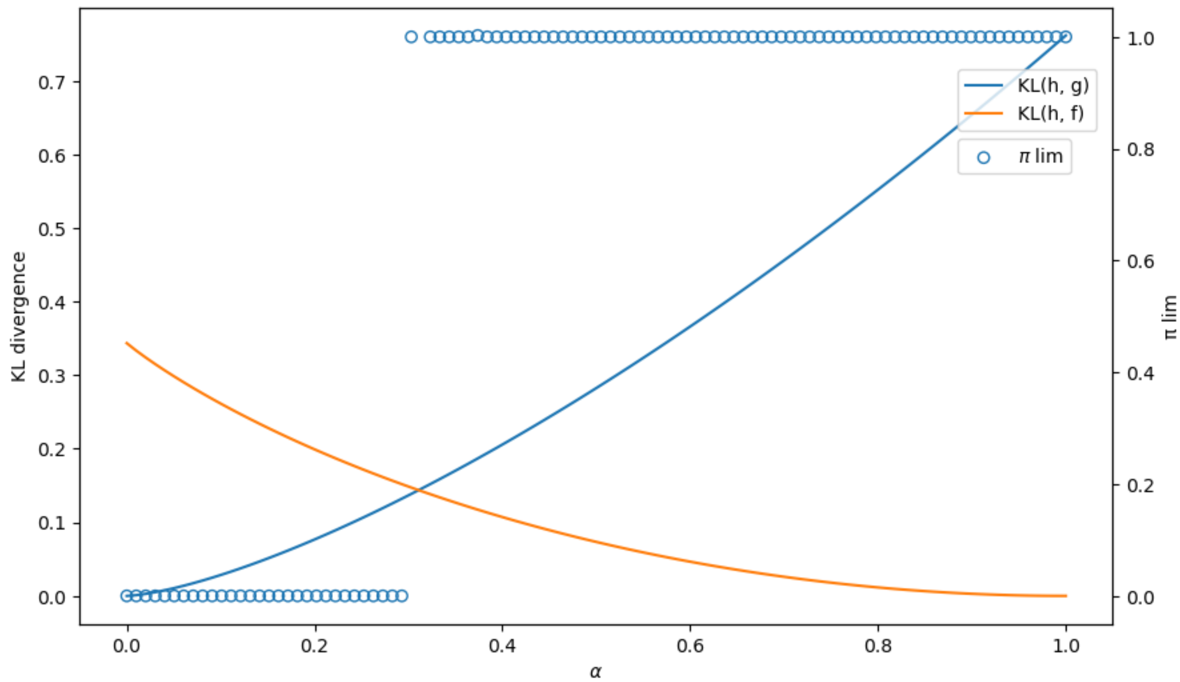
Thus, the graph below confirms how a minimum KL divergence governs what our type 1 agent eventually learns.

```
α_arr_x = α_arr[(α_arr<discretion)|(α_arr>discretion)]
π_lim_arr = π_lim_v(α_arr_x)

# plot
fig, ax = plt.subplots(1, figsize=[10, 6])

ax.plot(α_arr, KL_g_arr, label='KL(h, g)')
ax.plot(α_arr, KL_f_arr, label='KL(h, f)')
ax.set_ylabel('KL divergence')
ax.set_xlabel(r'$\alpha$')

# plot KL
```

(continues on next page)

(continued from previous page)

```
ax2 = ax.twinx()
# plot limit point
ax2.scatter(α_arr_x, π_lim_arr,
            facecolors='none',
            edgecolors='tab:blue',
            label=r'$\pi$ lim')
ax2.set_ylabel('π lim')

ax.legend(loc=[0.85, 0.8])
ax2.legend(loc=[0.85, 0.73])
plt.show()
```



Evidently, our type 1 learner who applies Bayes' law to his misspecified set of statistical models eventually learns an approximating model that is as close as possible to the true model, as measured by its Kullback-Leibler divergence:

- When α is small, $KL_g < KL_f$ meaning the divergence of g from h is smaller than that of f and so the limit point of π_t is close to 0.
- When α is large, $KL_f < KL_g$ meaning the divergence of f from h is smaller than that of g and so the limit point of π_t is close to 1.

30.6 Type 2 Agent

We now describe how our type 2 agent formulates his learning problem and what he eventually learns.

Our type 2 agent understands the correct statistical model but does not know α.

We apply Bayes law to deduce an algorithm for learning α under the assumption that the agent knows that

$$h(w) = h(w|\alpha)$$

but does not know α.

We'll assume that the person starts out with a prior probability $\pi_0(\alpha)$ on $\alpha \in (0,1)$ where the prior has one of the forms that we deployed in *this quantecon lecture*.

We'll fire up `numpyro` and apply it to the present situation.

Bayes' law now takes the form

$$\pi_{t+1}(\alpha) = \frac{h(w_{t+1}|\alpha)\pi_t(\alpha)}{\int h(w_{t+1}|\hat{\alpha})\pi_t(\hat{\alpha})d\hat{\alpha}}$$

We'll use numpyro to approximate this equation.

We'll create graphs of the posterior $\pi_t(\alpha)$ as $t \to +\infty$ corresponding to ones presented in the quantecon lecture https://python.quantecon.org/bayes_nonconj.html.

We anticipate that a posterior distribution will collapse around the true α as $t \to +\infty$.

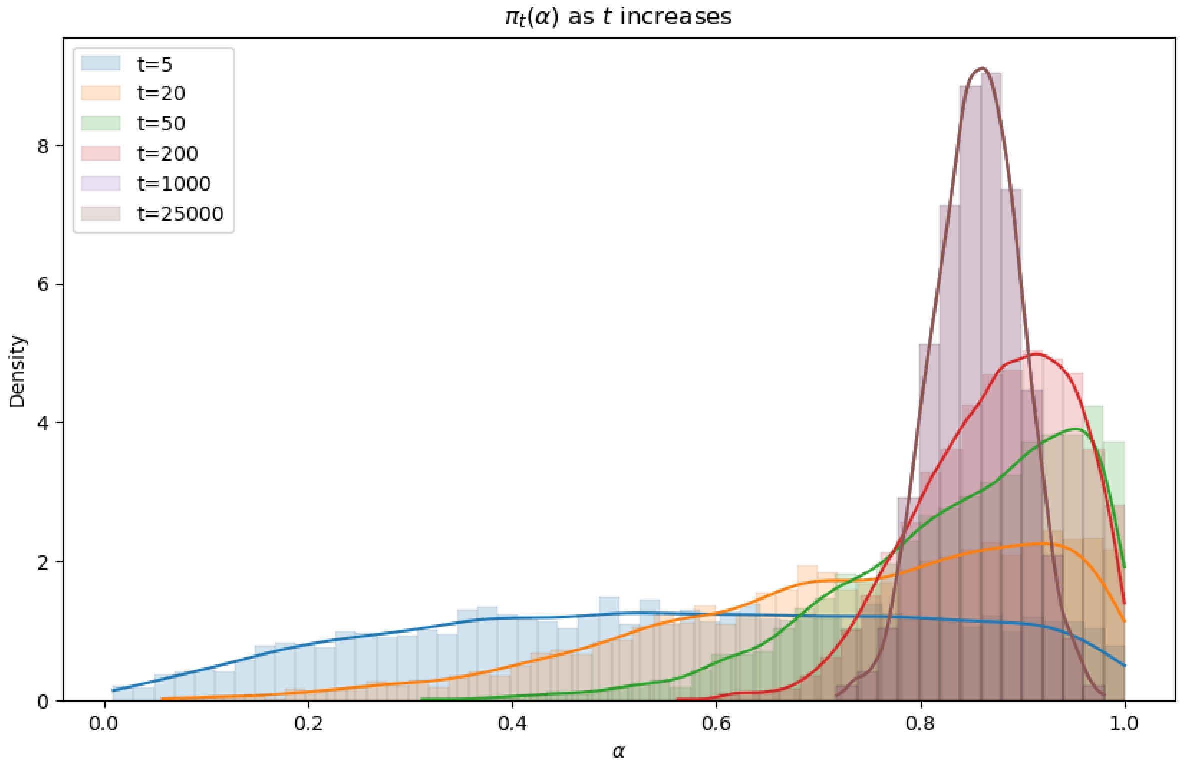
Let us try a uniform prior first.

We use the `Mixture` class in numpyro to construct the likelihood function.

```
α = 0.8

# simulate data with true α
data = draw_lottery(α, 1000)
sizes = [5, 20, 50, 200, 1000, 25000]

def model(w):
    α = numpyro.sample('α', dist.Uniform(low=0.0, high=1.0))

    y_samp = numpyro.sample('w',
        dist.Mixture(dist.Categorical(jnp.array([α, 1-α])), [dist.Beta(F_a, F_b),␣
↪dist.Beta(G_a, G_b)]), obs=w)

def MCMC_run(ws):
    "Compute posterior using MCMC with observed ws"

    kernel = NUTS(model)
    mcmc = MCMC(kernel, num_samples=5000, num_warmup=1000, progress_bar=False)

    mcmc.run(rng_key=random.PRNGKey(142857), w=jnp.array(ws))
    sample = mcmc.get_samples()
    return sample['α']
```

The following code generates the graph below that displays Bayesian posteriors for α at various history lengths.

```
fig, ax = plt.subplots(figsize=(10, 6))

for i in range(len(sizes)):
    sample = MCMC_run(data[:sizes[i]])
    sns.histplot(
        data=sample, kde=True, stat='density', alpha=0.2, ax=ax,
        color=colors[i], binwidth=0.02, linewidth=0.05, label=f't={sizes[i]}'
    )
ax.set_title(r'$\pi_t(\alpha)$ as $t$ increases')
ax.legend()
ax.set_xlabel(r'$\alpha$')
plt.show()
```



Evidently, the Bayesian posterior narrows in on the true value $\alpha = .8$ of the mixing parameter as the length of a history of observations grows.

30.7 Concluding Remarks

Our type 1 person deploys an incorrect statistical model.

He believes that either f or g generated the w process, but just doesn't know which one.

That is wrong because nature is actually mixing each period with mixing probability α.

Our type 1 agent eventually believes that either f or g generated the w sequence, the outcome being determined by the model, either f or g, whose KL divergence relative to h is smaller.

Our type 2 agent has a different statistical model, one that is correctly specified.

He knows the parametric form of the statistical model but not the mixing parameter α.

He knows that he does not know it.

But by using Bayes' law in conjunction with his statistical model and a history of data, he eventually acquires a more and more accurate inference about α.

This little laboratory exhibits some important general principles that govern outcomes of Bayesian learning of misspecified models.

Thus, the following situation prevails quite generally in empirical work.

A scientist approaches the data with a manifold S of statistical models $s(X|\theta)$, where s is a probability distribution over a random vector X, $\theta \in \Theta$ is a vector of parameters, and Θ indexes the manifold of models.

The scientist with observations that he interprets as realizations x of the random vector X wants to solve an **inverse problem** of somehow *inverting* $s(x|\theta)$ to infer θ from x.

But the scientist's model is misspecified, being only an approximation to an unknown model h that nature uses to generate X.

If the scientist uses Bayes' law or a related likelihood-based method to infer θ, it occurs quite generally that for large sample sizes the inverse problem infers a θ that minimizes the KL divergence of the scientist's model s relative to nature's model h.

30.8 Exercises

Exercise 30.8.1

In *Likelihood Ratio Processes and Bayesian Learning*, we studied the consequence of applying likelihood ratio and Bayes' law to a misspecified statistical model.

In that lecture, we used a model selection algorithm to study the case where the true data generating process is a mixture.

In this lecture, we studied how to correctly "learn" a model generated by a mixing process using a Bayesian approach.

To fix the algorithm we used in *Likelihood Ratio Processes and Bayesian Learning*, a correct Bayesian approach should directly model the uncertainty about x and update beliefs about it as new data arrives.

Here is the algorithm:

First we specify a prior distribution for x given by $x \sim \text{Beta}(\alpha_0, \beta_0)$ with expectation $\mathbb{E}[x] = \frac{\alpha_0}{\alpha_0+\beta_0}$.

The likelihood for a single observation w_t is $p(w_t|x) = xf(w_t) + (1-x)g(w_t)$.

For a sequence $w^t = (w_1, \dots, w_t)$, the likelihood is $p(w^t|x) = \prod_{i=1}^t p(w_i|x)$.

The posterior distribution is updated using $p(x|w^t) \propto p(w^t|x)p(x)$.

Recursively, the posterior after w_t is $p(x|w^t) \propto p(w_t|x)p(x|w^{t-1})$.

Without a conjugate prior, we can approximate the posterior by discretizing x into a grid.

Your task is to implement this algorithm in Python.

You can verify your implementation by checking that the posterior mean converges to the true value of x as t increases in *Likelihood Ratio Processes and Bayesian Learning*.

Solution

Here is one solution:

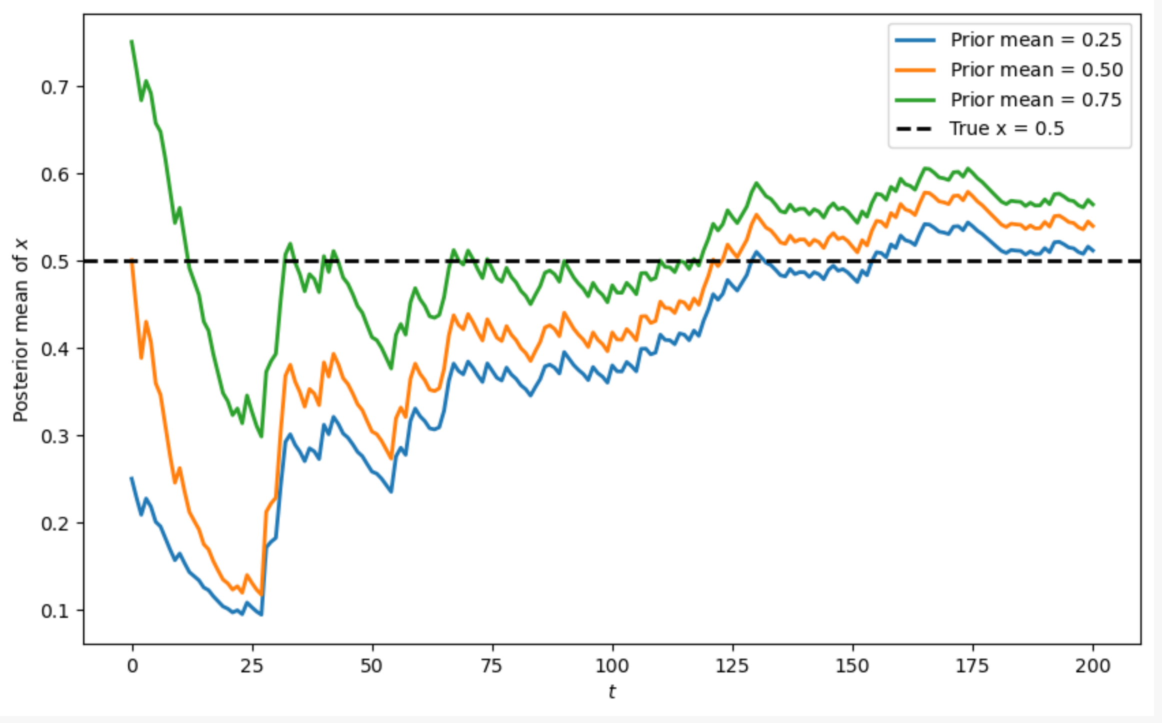
First we define the mixture probability and parameters of prior distributions

```
x_true = 0.5
T_mix = 200

# Three different priors with means 0.25, 0.5, 0.75
prior_params = [(1, 3), (1, 1), (3, 1)]
prior_means = [a/(a+b) for a, b in prior_params]

w_mix = draw_lottery(x_true, T_mix)
```

```python
@jit
def learn_x_bayesian(observations, α0, β0, grid_size=2000):
    """
    Sequential Bayesian learning of the mixing probability x
    using a grid approximation.
    """
    w = np.asarray(observations)
    T = w.size

    x_grid = np.linspace(1e-3, 1 - 1e-3, grid_size)

    # Log prior
    log_prior = (α0 - 1) * np.log(x_grid) + (β0 - 1) * np.log1p(-x_grid)

    μ_path = np.empty(T + 1)
    μ_path[0] = α0 / (α0 + β0)

    log_post = log_prior.copy()

    for t in range(T):
        wt = w[t]
        # P(w_t | x) = x f(w_t) + (1 - x) g(w_t)
        like = x_grid * f(wt) + (1 - x_grid) * g(wt)
        log_post += np.log(like)

        # normalize
        log_post -= log_post.max()
        post = np.exp(log_post)
        post /= post.sum()

        μ_path[t + 1] = x_grid @ post

    return μ_path

x_posterior_means = [learn_x_bayesian(w_mix, α0, β0) for α0, β0 in prior_params]
```

Let's visualize how the posterior mean of x evolves over time, starting from three different prior beliefs.

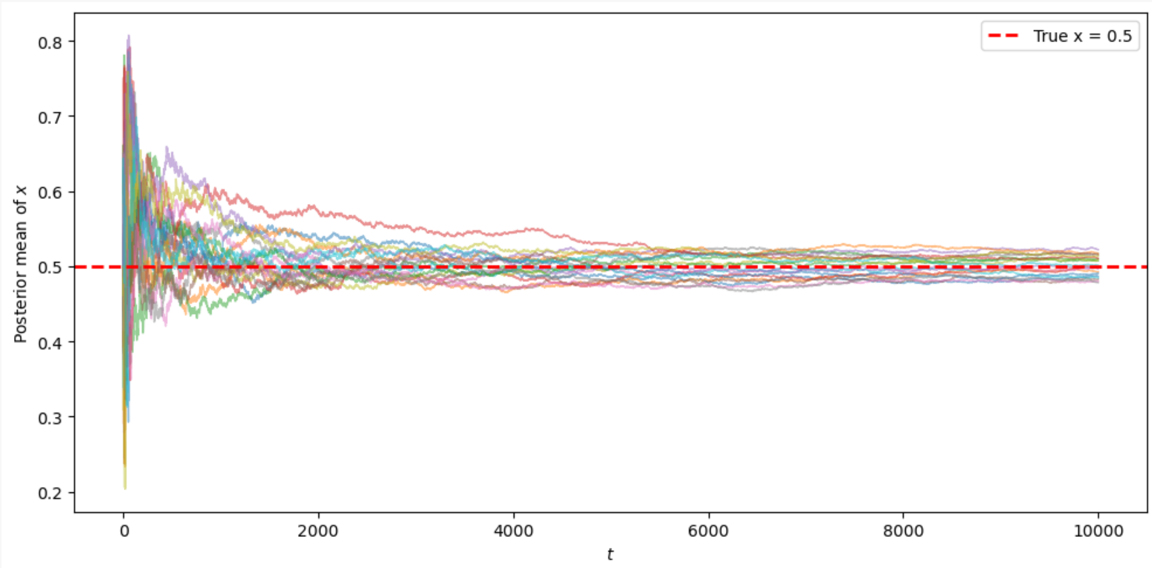
```python
fig, ax = plt.subplots(figsize=(10, 6))

for i, (x_means, mean0) in enumerate(zip(x_posterior_means, prior_means)):
    ax.plot(range(T_mix + 1), x_means,
            label=fr'Prior mean = ${mean0:.2f}$',
            color=colors[i], linewidth=2)

ax.axhline(y=x_true, color='black', linestyle='--',
           label=f'True x = {x_true}', linewidth=2)
ax.set_xlabel('$t$')
ax.set_ylabel('Posterior mean of $x$')
ax.legend()
plt.show()
```



The plot shows that regardless of the initial prior belief, all three posterior means eventually converge towards the true value of $x = 0.5$.

Next, let's look at multiple simulations with a longer time horizon, all starting from a uniform prior.

```
set_seed()
n_paths = 20
T_long = 10_000

fig, ax = plt.subplots(figsize=(10, 5))

for j in range(n_paths):
    w_path = draw_lottery(x_true, T_long)
    x_means = learn_x_bayesian(w_path, 1, 1)  # Uniform prior
    ax.plot(range(T_long + 1), x_means, alpha=0.5, linewidth=1)

ax.axhline(y=x_true, color='red', linestyle='--',
           label=f'True x = {x_true}', linewidth=2)
ax.set_ylabel('Posterior mean of $x$')
ax.set_xlabel('$t$')
ax.legend()
plt.tight_layout()
plt.show()
```



We can see that the posterior mean of x converges to the true value $x = 0.5$.

CHAPTER

THIRTYONE

BAYESIAN VERSUS FREQUENTIST DECISION RULES

Contents


```
import matplotlib.pyplot as plt
import numpy as np
from numba import jit, prange, float64, int64
from numba.experimental import jitclass
from math import gamma
from scipy.optimize import minimize
```

31.1 Overview

This lecture follows up on ideas presented in the following lectures:

- *A Problem that Stumped Milton Friedman*
- *A Bayesian Formulation of Friedman and Wald's Problem*
- *Exchangeability and Bayesian Updating*
- *Likelihood Ratio Processes*

A Problem that Stumped Milton Friedman described a problem that a Navy Captain presented to Milton Friedman during World War II.

The Navy had told the Captain to use a decision rule for quality control.

In particular, the Navy had ordered the Captain to use an instance of a **frequentist decision rule**.

The Captain doubted that that rule was a good one.

Milton Friedman recognized the Captain's conjecture as posing a challenging statistical problem that he and other members of the US Government's Statistical Research Group at Columbia University proceeded to try to solve.

A member of the group, the great mathematician and economist Abraham Wald, soon solved the problem.

A good way to formulate the problem is to use some ideas from Bayesian statistics that we describe in this lecture *Exchangeability and Bayesian Updating* and in this lecture *Likelihood Ratio Processes*, which describes the link between Bayesian updating and likelihood ratio processes.

The present lecture uses Python to generate simulations that evaluate expected losses under the Neyman-Pearson **frequentist** procedure that the Navy captain questioned and the **Bayesian** decision rule described in *A Bayesian Formulation of Friedman and Wald's Problem*.

The simulations confirm the Navy Captain's hunch that there is a better rule than the Neyman-Pearson likelihood ratio test that the Navy had told him to use.

31.2 Setup

To formalize the problem that had confronted the Navy Captain, we consider a setting with the following parts.

- Each period a decision maker draws a non-negative random variable Z. He knows that two probability distributions are possible, f_0 and f_1, and that which ever distribution it is remains fixed over time. The decision maker believes that before the beginning of time, nature once and for all had selected either f_0 or f_1 and that the probability that it selected f_0 is probability π^*.
- The decision maker observes a sample $\{z_i\}_{i=0}^{t}$ from the distribution chosen by nature.

The decision maker wants to decide which distribution actually governs Z.

He is worried about two types of errors and the losses that they will impose on him.

- a loss $\bar{L}_1$ from a **type I error** that occurs if he decides that $f = f_1$ when actually $f = f_0$
- a loss $\bar{L}_0$ from a **type II error** that occurs if he decides that $f = f_0$ when actually $f = f_1$

The decision maker pays a cost c for drawing another z.

We mainly borrow parameters from the quantecon lecture *A Bayesian Formulation of Friedman and Wald's Problem* except that we increase both $\bar{L}_0$ and $\bar{L}_1$ from 25 to 100 to encourage the Bayesian decision rule to take more draws before deciding.

We set the cost c of taking one more draw at 1.25.

We set the probability distributions f_0 and f_1 to be beta distributions with $a_0 = b_0 = 1$, $a_1 = 3$, and $b_1 = 1.2$, respectively.

Below is some Python code that sets up these objects.

```
@jit
def p(x, a, b):
    "Beta distribution."

    r = gamma(a + b) / (gamma(a) * gamma(b))

    return r * x**(a-1) * (1 - x)**(b-1)
```

We start with defining a `jitclass` that stores parameters and functions we need to solve problems for both the Bayesian and frequentist Navy Captains.

```
wf_data = [
    ('c', float64),           # unemployment compensation
    ('a0', float64),          # parameters of beta distribution
    ('b0', float64),
    ('a1', float64),
    ('b1', float64),
    ('L0', float64),          # cost of selecting f0 when f1 is true
    ('L1', float64),          # cost of selecting f1 when f0 is true
    ('π_grid', float64[:]),   # grid of beliefs π
    ('π_grid_size', int64),
    ('mc_size', int64),       # size of Monto Carlo simulation
    ('z0', float64[:]),       # sequence of random values
    ('z1', float64[:])        # sequence of random values
]
```

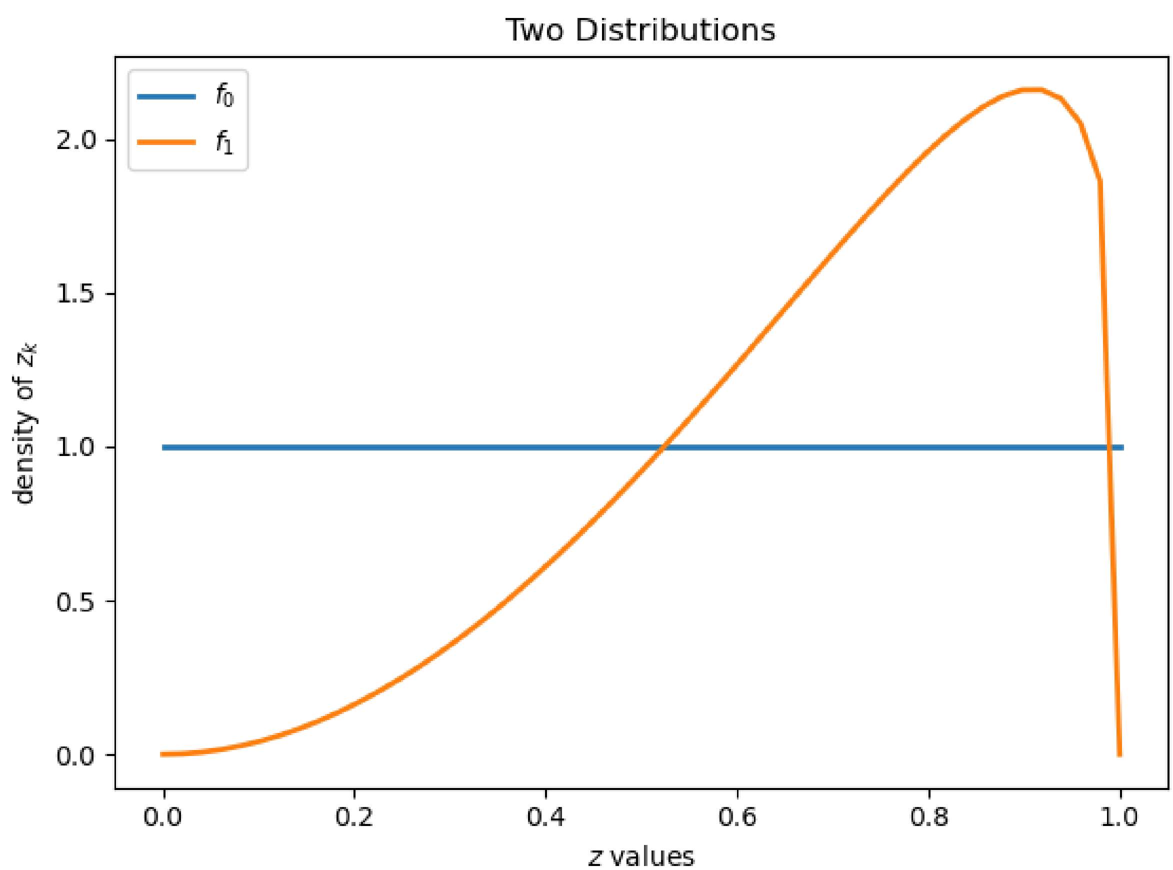
```
@jitclass(wf_data)
class WaldFriedman:

    def __init__(self,
                 c=1.25,
                 a0=1,
                 b0=1,
                 a1=3,
                 b1=1.2,
                 L0=100,
                 L1=100,
                 π_grid_size=200,
                 mc_size=1000):

        self.c, self.π_grid_size = c, π_grid_size
        self.a0, self.b0, self.a1, self.b1 = a0, b0, a1, b1
        self.L0, self.L1 = L0, L1
        self.π_grid = np.linspace(0, 1, π_grid_size)
        self.mc_size = mc_size

        self.z0 = np.random.beta(a0, b0, mc_size)
        self.z1 = np.random.beta(a1, b1, mc_size)

    def f0(self, x):

        return p(x, self.a0, self.b0)

    def f1(self, x):

        return p(x, self.a1, self.b1)

    def κ(self, z, π):
        """
        Updates π using Bayes' rule and the current observation z
        """

        a0, b0, a1, b1 = self.a0, self.b0, self.a1, self.b1

        π_f0, π_f1 = π * p(z, a0, b0), (1 - π) * p(z, a1, b1)
        π_new = π_f0 / (π_f0 + π_f1)
```

(continues on next page)

(continued from previous page)

```
        return π_new
```

```
wf = WaldFriedman()

grid = np.linspace(0, 1, 50)

plt.figure()

plt.title("Two Distributions")
plt.plot(grid, wf.f0(grid), lw=2, label="$f_0$")
plt.plot(grid, wf.f1(grid), lw=2, label="$f_1$")

plt.legend()
plt.xlabel("$z$ values")
plt.ylabel("density of $z_k$")

plt.tight_layout()
plt.show()
```



Above, we plot the two possible probability densities f_0 and f_1

31.3 Frequentist Decision Rule

The Navy told the Captain to use a Neyman-Pearson likelihood ratio decision rule.

That decision rule is characterized by

- a sample size t, and
- a cutoff value d of a likelihood ratio

Let $L(z^t) = \prod_{i=0}^{t} \frac{f_0(z_i)}{f_1(z_i)}$ be the likelihood ratio associated with observing the sequence $\{z_i\}_{i=0}^{t}$.

The decision rule associated with a sample size t is:

- decide that f_0 is the distribution if the likelihood ratio is greater than d
- decide that f_1 is the distribution if the likelihood ratio is less than d

For our purposes here, we want to compute an expected loss from using this rule, where we borrow loss parameters $\bar{L}_1$ and $\bar{L}_2$ from *A Bayesian Formulation of Friedman and Wald's Problem*.

Let null and alternative hypotheses be

- null: H_0: $f = f_0$,
- alternative H_1: $f = f_1$.

Given sample size t and cutoff d, under the model described above, the mathematical expectation of total loss is

$$\bar{V}_{fre}(t, d) = ct + \pi^* PFA \times \bar{L}_1 + (1 - \pi^*)(1 - PD) \times \bar{L}_0 \tag{31.1}$$

$$\begin{aligned} \text{where} \quad PFA &= \Pr\{L(z^t) < d \mid q = f_0\} \\ PD &= \Pr\{L(z^t) < d \mid q = f_1\} \end{aligned}$$

Here

- PFA denotes the probability of a **false alarm**, i.e., rejecting H_0 when it is true
- PD denotes the probability of a **detection error**, i.e., not rejecting H_0 when H_1 is true

For a given sample size t, the pairs (PFA, PD) lie on a **receiver operating characteristic curve**.

- by choosing d, we select a particular pair (PFA, PD) along the curve for a given t

To see some receiver operating characteristic curves, please see this lecture *Likelihood Ratio Processes*.

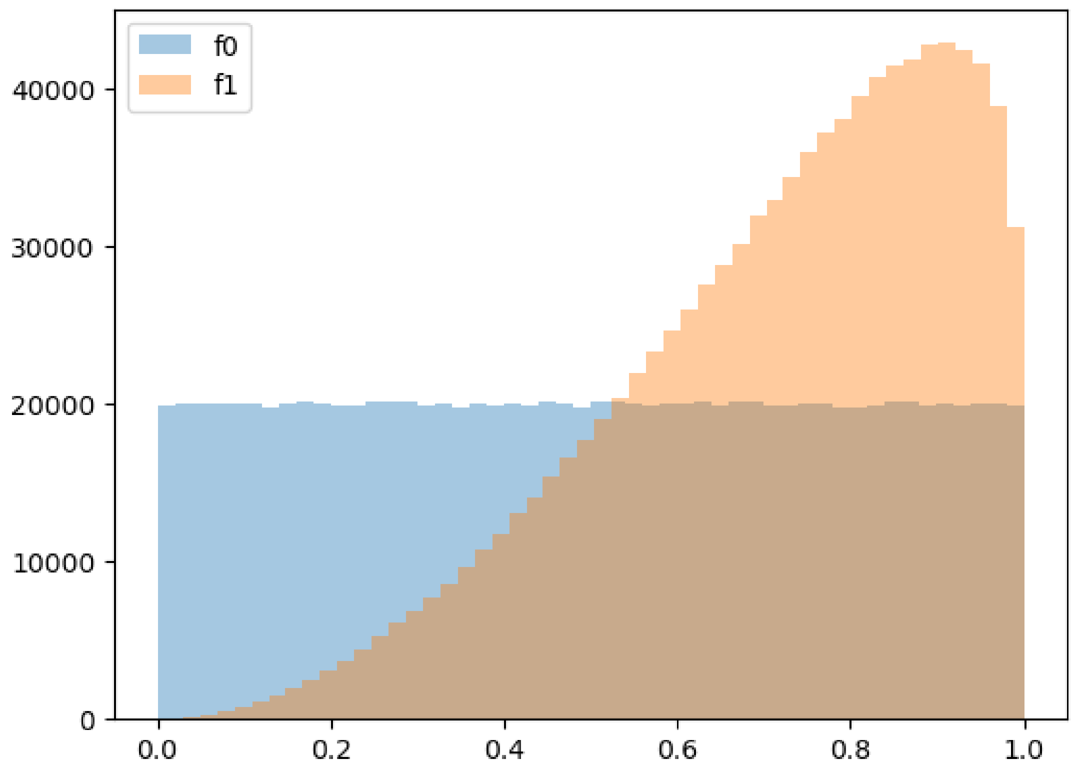
To solve for $\bar{V}_{fre}(t, d)$ numerically, we first simulate sequences of z when either f_0 or f_1 generates data.

Let's plot empirical distributions, i.e., histograms, associated with f_0 and f_1.

```
N = 10000
T = 100
```

```
z0_arr = np.random.beta(wf.a0, wf.b0, (N, T))
z1_arr = np.random.beta(wf.a1, wf.b1, (N, T))
```

```
plt.hist(z0_arr.flatten(), bins=50, alpha=0.4, label='f0')
plt.hist(z1_arr.flatten(), bins=50, alpha=0.4, label='f1')
plt.legend()
plt.show()
```



We can compute sequences of likelihood ratios using simulated samples.

```
l = lambda z: wf.f0(z) / wf.f1(z)
```

```
l0_arr = l(z0_arr)
l1_arr = l(z1_arr)

L0_arr = np.cumprod(l0_arr, 1)
L1_arr = np.cumprod(l1_arr, 1)
```

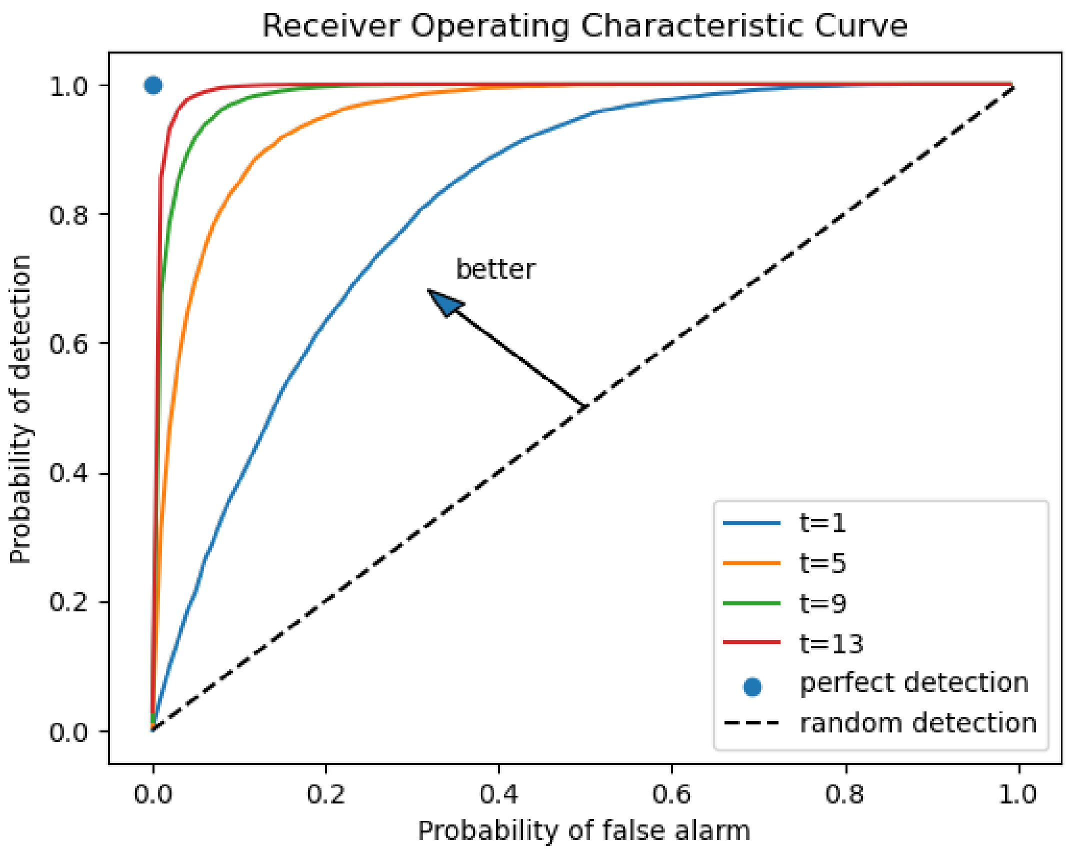
With an empirical distribution of likelihood ratios in hand, we can draw **receiver operating characteristic curves** by enumerating (PFA, PD) pairs given each sample size t.

```
PFA = np.arange(0, 100, 1)

for t in range(1, 15, 4):
    percentile = np.percentile(L0_arr[:, t], PFA)
    PD = [np.sum(L1_arr[:, t] < p) / N for p in percentile]

    plt.plot(PFA / 100, PD, label=f"t={t}")

plt.scatter(0, 1, label="perfect detection")
plt.plot([0, 1], [0, 1], color='k', ls='--', label="random detection")

plt.arrow(0.5, 0.5, -0.15, 0.15, head_width=0.03)
plt.text(0.35, 0.7, "better")
plt.xlabel("Probability of false alarm")
plt.ylabel("Probability of detection")
plt.legend()
plt.title("Receiver Operating Characteristic Curve")
```

(continues on next page)

(continued from previous page)

```
plt.show()
```



We can minimize the expected total loss presented in equation (31.1) by choosing (t, d).

Doing that delivers an expected loss

$$\bar{V}_{fre} = \min_{t,d} \bar{V}_{fre}(t, d).$$

We first consider the case in which $\pi^* = \Pr\{\text{nature selects } f_0\} = 0.5$.

We can solve the minimization problem in two steps.

First, we fix t and find the optimal cutoff d and consequently the minimal $\bar{V}_{fre}(t)$.

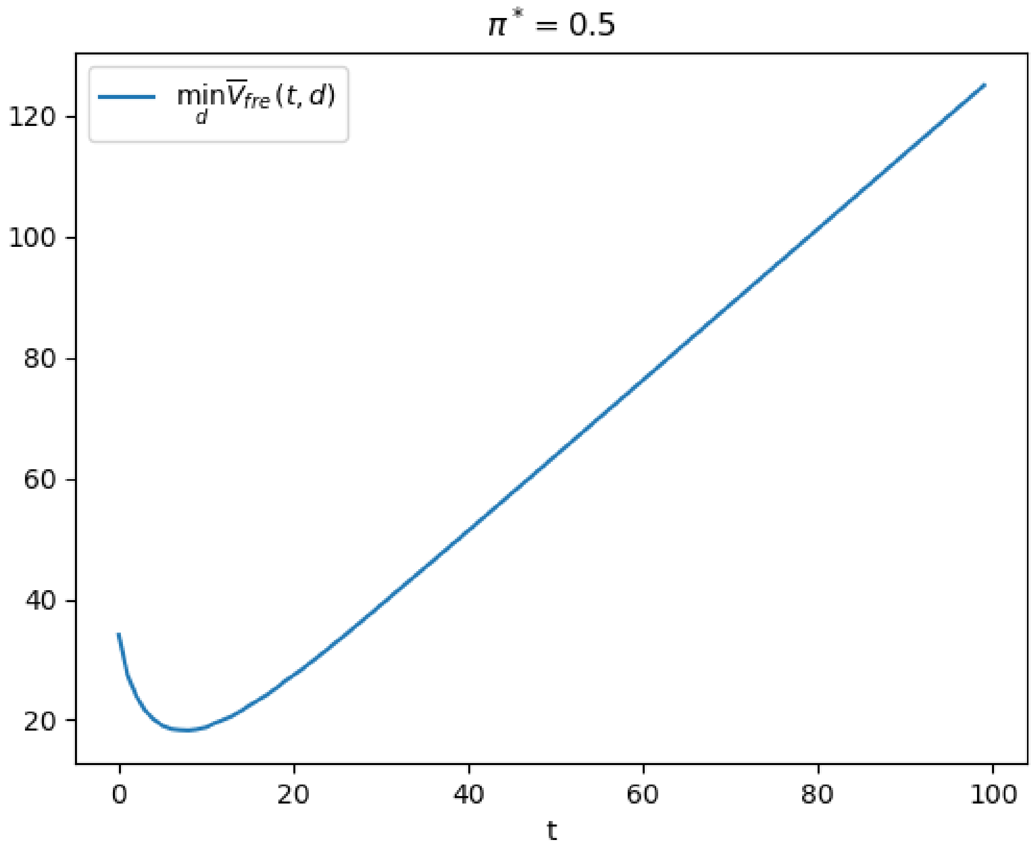
Here is Python code that does that and then plots a useful graph.

```
@jit
def V_fre_d_t(d, t, L0_arr, L1_arr, π_star, wf):

    N = L0_arr.shape[0]

    PFA = np.sum(L0_arr[:, t-1] < d) / N
    PD = np.sum(L1_arr[:, t-1] < d) / N

    V = π_star * PFA * wf.L1 + (1 - π_star) * (1 - PD) * wf.L0

    return V
```

```
def V_fre_t(t, L0_arr, L1_arr, π_star, wf):

    res = minimize(V_fre_d_t, 1, args=(t, L0_arr, L1_arr, π_star, wf), method='Nelder-
↪Mead')
    V = res.fun
    d = res.x

    PFA = np.sum(L0_arr[:, t-1] < d) / N
    PD = np.sum(L1_arr[:, t-1] < d) / N

    return V, PFA, PD
```

```
def compute_V_fre(L0_arr, L1_arr, π_star, wf):

    T = L0_arr.shape[1]

    V_fre_arr = np.empty(T)
    PFA_arr = np.empty(T)
    PD_arr = np.empty(T)

    for t in range(1, T+1):
        V, PFA, PD = V_fre_t(t, L0_arr, L1_arr, π_star, wf)
        V_fre_arr[t-1] = wf.c * t + V
        PFA_arr[t-1] = PFA
        PD_arr[t-1] = PD

    return V_fre_arr, PFA_arr, PD_arr
```

```
π_star = 0.5
V_fre_arr, PFA_arr, PD_arr = compute_V_fre(L0_arr, L1_arr, π_star, wf)

plt.plot(range(T), V_fre_arr, label=r'$\min_{d} \overline{V}_{fre}(t,d)$')
plt.xlabel('t')
plt.title(r'$\pi^*=0.5$')
plt.legend()
plt.show()
```




```
t_optimal = np.argmin(V_fre_arr) + 1
```

The above graph illustrates how minimizing over t tells the frequentist to draw t_{optimal} observations and then decide.

Let's now change the value of π^* and watch how the decision rule changes.

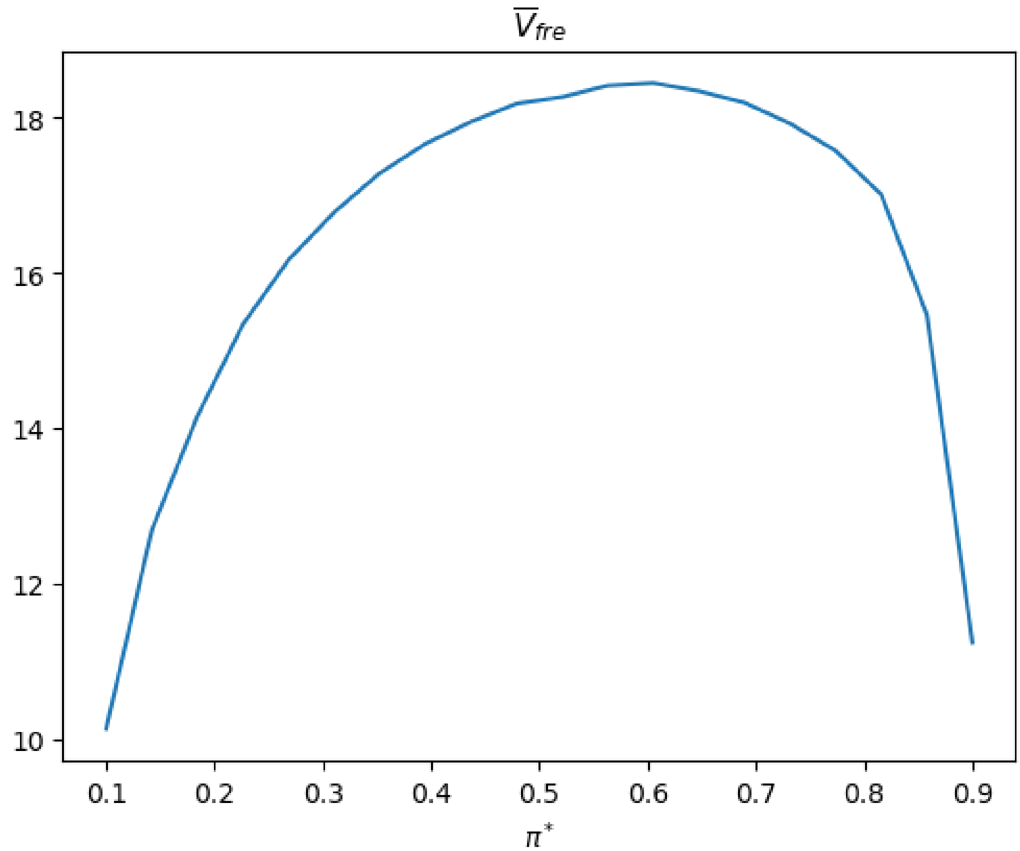
```
n_π = 20
π_star_arr = np.linspace(0.1, 0.9, n_π)

V_fre_bar_arr = np.empty(n_π)
t_optimal_arr = np.empty(n_π)
PFA_optimal_arr = np.empty(n_π)
PD_optimal_arr = np.empty(n_π)

for i, π_star in enumerate(π_star_arr):
    V_fre_arr, PFA_arr, PD_arr = compute_V_fre(L0_arr, L1_arr, π_star, wf)
    t_idx = np.argmin(V_fre_arr)

    V_fre_bar_arr[i] = V_fre_arr[t_idx]
    t_optimal_arr[i] = t_idx + 1
    PFA_optimal_arr[i] = PFA_arr[t_idx]
    PD_optimal_arr[i] = PD_arr[t_idx]
```

```
plt.plot(π_star_arr, V_fre_bar_arr)
plt.xlabel(r'$\pi^*$')
plt.title(r'$\overline{V}_{fre}$')
```

(continues on next page)

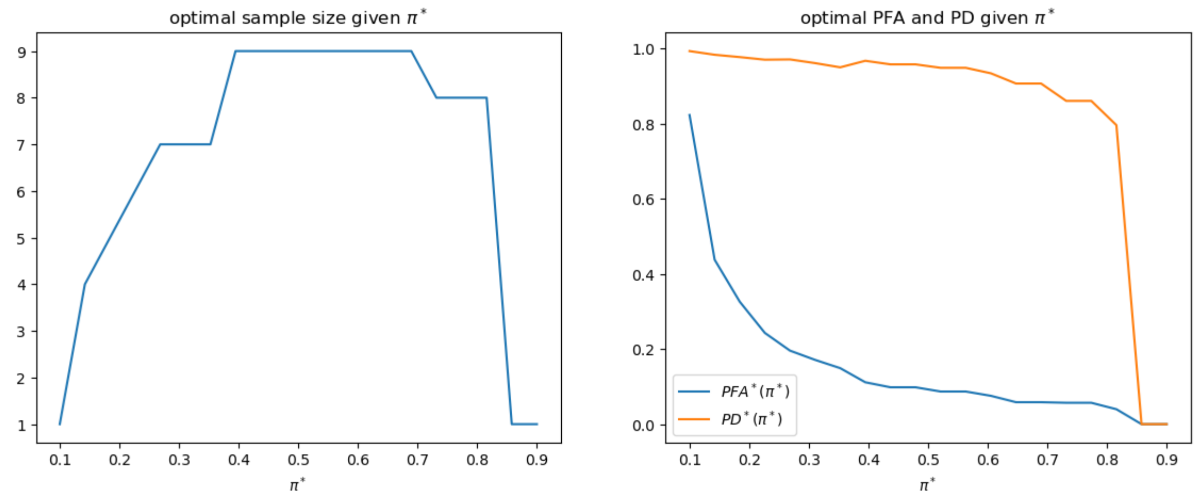
(continued from previous page)

```
plt.show()
```



The following shows how optimal sample size t and targeted (PFA, PD) change as π^* varies.

```
fig, axs = plt.subplots(1, 2, figsize=(14, 5))

axs[0].plot(π_star_arr, t_optimal_arr)
axs[0].set_xlabel(r'$\pi^*$')
axs[0].set_title(r'optimal sample size given $\pi^*$')

axs[1].plot(π_star_arr, PFA_optimal_arr, label=r'$PFA^*(\pi^*)$')
axs[1].plot(π_star_arr, PD_optimal_arr, label=r'$PD^*(\pi^*)$')
axs[1].set_xlabel(r'$\pi^*$')
axs[1].legend()
axs[1].set_title(r'optimal PFA and PD given $\pi^*$')

plt.show()
```



31.4 Bayesian Decision Rule

In *A Problem that Stumped Milton Friedman*, we learned how Abraham Wald confirmed the Navy Captain's hunch that there is a better decision rule.

In *A Bayesian Formulation of Friedman and Wald's Problem* we presented a Bayesian procedure that makes decisions by comparing a Bayesian posterior probability π with cutoff probabilities called A and B.

To proceed, we borrow some Python code from the quantecon lecture *A Bayesian Formulation of Friedman and Wald's Problem* that computes optimal values of A and B.

```
@jit(parallel=True)
def Q(h, wf):

    c, π_grid = wf.c, wf.π_grid
    L0, L1 = wf.L0, wf.L1
    z0, z1 = wf.z0, wf.z1
    mc_size = wf.mc_size

    κ = wf.κ

    h_new = np.empty_like(π_grid)
    h_func = lambda p: np.interp(p, π_grid, h)

    for i in prange(len(π_grid)):
        π = π_grid[i]

        # Find the expected value of J by integrating over z
        integral_f0, integral_f1 = 0, 0
        for m in range(mc_size):
            π_0 = κ(z0[m], π)  # Draw z from f0 and update π
            integral_f0 += min((1 - π_0) * L0, π_0 * L1, h_func(π_0))

            π_1 = κ(z1[m], π)  # Draw z from f1 and update π
            integral_f1 += min((1 - π_1) * L0, π_1 * L1, h_func(π_1))

        integral = (π * integral_f0 + (1 - π) * integral_f1) / mc_size
```

(continues on next page)

(continued from previous page)

```
        h_new[i] = c + integral

    return h_new
```

```
@jit
def solve_model(wf, tol=1e-4, max_iter=1000):
    """
    Compute the continuation value function

    * wf is an instance of WaldFriedman
    """

    # Set up loop
    h = np.zeros(len(wf.π_grid))
    i = 0
    error = tol + 1

    while i < max_iter and error > tol:
        h_new = Q(h, wf)
        error = np.max(np.abs(h - h_new))
        i += 1
        h = h_new

    if error > tol:
        print("Failed to converge!")

    return h_new
```

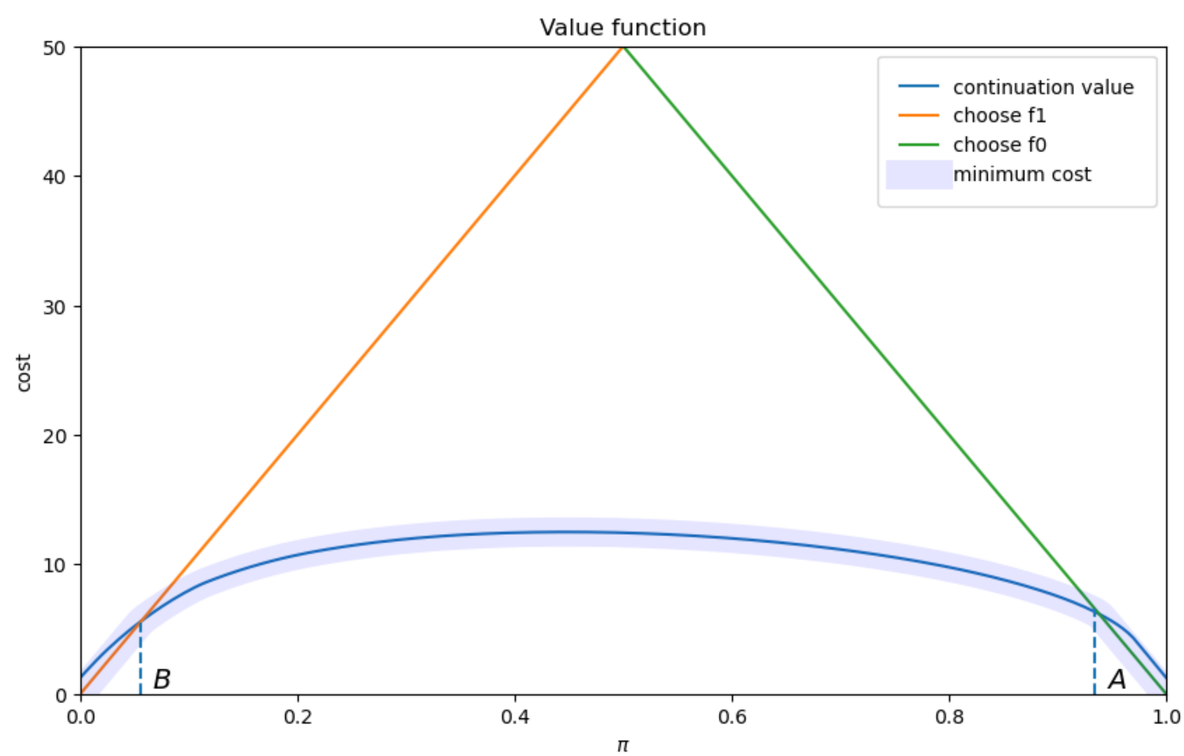
```
h_star = solve_model(wf)
```

```
@jit
def find_cutoff_rule(wf, h):

    """
    This function takes a continuation value function and returns the
    corresponding cutoffs of where you transition between continuing and
    choosing a specific model
    """

    π_grid = wf.π_grid
    L0, L1 = wf.L0, wf.L1

    # Evaluate cost at all points on grid for choosing a model
    payoff_f0 = (1 - π_grid) * L0
    payoff_f1 = π_grid * L1

    # The cutoff points can be found by differencing these costs with
    # The Bellman equation (J is always less than or equal to p_c_i)
    B = π_grid[np.searchsorted(
                              payoff_f1 - np.minimum(h, payoff_f0),
                              1e-10)
               - 1]
    A = π_grid[np.searchsorted(
                              np.minimum(h, payoff_f1) - payoff_f0,
                              1e-10)
               - 1]
```

(continues on next page)

(continued from previous page)

```
    return (B, A)

B, A = find_cutoff_rule(wf, h_star)
cost_L0 = (1 - wf.π_grid) * wf.L0
cost_L1 = wf.π_grid * wf.L1

fig, ax = plt.subplots(figsize=(10, 6))

ax.plot(wf.π_grid, h_star, label='continuation value')
ax.plot(wf.π_grid, cost_L1, label='choose f1')
ax.plot(wf.π_grid, cost_L0, label='choose f0')
ax.plot(wf.π_grid,
        np.amin(np.column_stack([h_star, cost_L0, cost_L1]),axis=1),
        lw=15, alpha=0.1, color='b', label='minimum cost')

ax.annotate(r"$B$", xy=(B + 0.01, 0.5), fontsize=14)
ax.annotate(r"$A$", xy=(A + 0.01, 0.5), fontsize=14)

plt.vlines(B, 0, B * wf.L0, linestyle="--")
plt.vlines(A, 0, (1 - A) * wf.L1, linestyle="--")

ax.set(xlim=(0, 1), ylim=(0, 0.5 * max(wf.L0, wf.L1)), ylabel="cost",
       xlabel=r"$\pi$", title="Value function")

plt.legend(borderpad=1.1)
plt.show()
```



The above figure portrays the value function plotted against the decision maker's Bayesian posterior.

It also shows the cutoff probabilities A and B.

The Bayesian decision rule is:

- accept H_0 if $\pi \geq A$
- accept H_1 if $\pi \leq B$
- delay deciding and draw another z if $B \leq \pi \leq A$

We can calculate two "objective" loss functions under this situation conditioning on knowing for sure that nature has selected f_0, in the first case, or f_1, in the second case.

1. under f_0,

$$V^{0}\left(\pi\right)=\begin{cases}0 & \text{if}A\leq\pi,\\ c+EV^{0}\left(\pi^{\prime}\right) & \text{if }B\leq\pi<A,\\ \bar{L}_{1} & \text{if }\pi<B.\end{cases}$$

2. under f_1

$$V^{1}\left(\pi\right)=\begin{cases}\bar{L}_{0} & \text{if }A\leq\pi,\\ c+EV^{1}\left(\pi^{\prime}\right) & \text{if }B\leq\pi<A,\\ 0 & \text{if }\pi<B.\end{cases}$$

where $\pi^{\prime}=\frac{\pi f_{0}\left(z^{\prime}\right)}{\pi f_{0}\left(z^{\prime}\right)+\left(1-\pi\right)f_{1}\left(z^{\prime}\right)}$.

Given a prior probability π_0, the expected loss for the Bayesian is

$$\bar{V}_{Bayes}\left(\pi_{0}\right)=\pi^{*}V^{0}\left(\pi_{0}\right)+\left(1-\pi^{*}\right)V^{1}\left(\pi_{0}\right).$$

Below we write some Python code that computes $V^{0}\left(\pi\right)$ and $V^{1}\left(\pi\right)$ numerically.

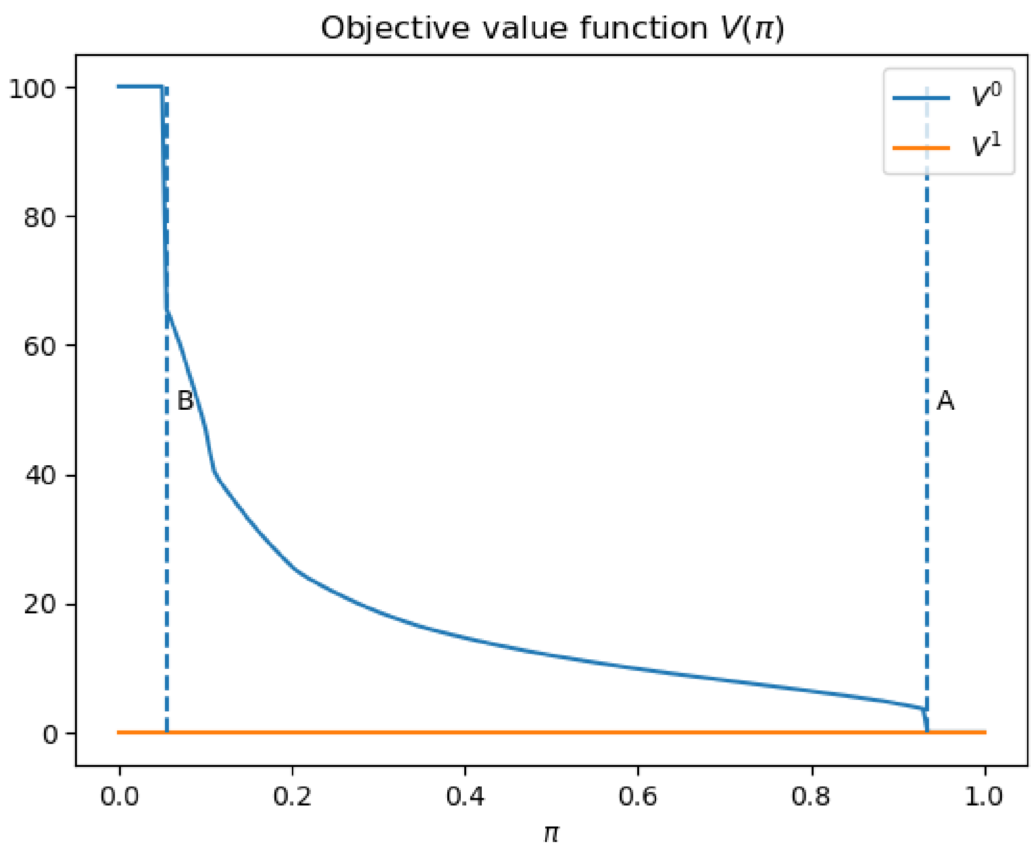
```
@jit(parallel=True)
def V_q(wf, flag):
    V = np.zeros(wf.π_grid_size)
    if flag == 0:
        z_arr = wf.z0
        V[wf.π_grid < B] = wf.L1
    else:
        z_arr = wf.z1
        V[wf.π_grid >= A] = wf.L0

    V_old = np.empty_like(V)

    while True:
        V_old[:] = V[:]
        V[(B <= wf.π_grid) & (wf.π_grid < A)] = 0

        for i in prange(len(wf.π_grid)):
            π = wf.π_grid[i]

            if π >= A or π < B:
                continue

            for j in prange(len(z_arr)):
                π_next = wf.κ(z_arr[j], π)
                V[i] += wf.c + np.interp(π_next, wf.π_grid, V_old)
```

(continues on next page)

(continued from previous page)

```
            V[i] /= wf.mc_size

        if np.abs(V - V_old).max() < 1e-5:
            break

    return V
```

```
V0 = V_q(wf, 0)
V1 = V_q(wf, 1)

plt.plot(wf.π_grid, V0, label='$V^0$')
plt.plot(wf.π_grid, V1, label='$V^1$')
plt.vlines(B, 0, wf.L0, linestyle='--')
plt.text(B+0.01, wf.L0/2, 'B')
plt.vlines(A, 0, wf.L0, linestyle='--')
plt.text(A+0.01, wf.L0/2, 'A')
plt.xlabel(r'$\pi$')
plt.title(r'Objective value function $V(\pi)$')
plt.legend()
plt.show()
```



Given an assumed value for $\pi^* = \Pr\{\text{nature selects } f_0\}$, we can then compute $\bar{V}_{Bayes}(\pi_0)$.

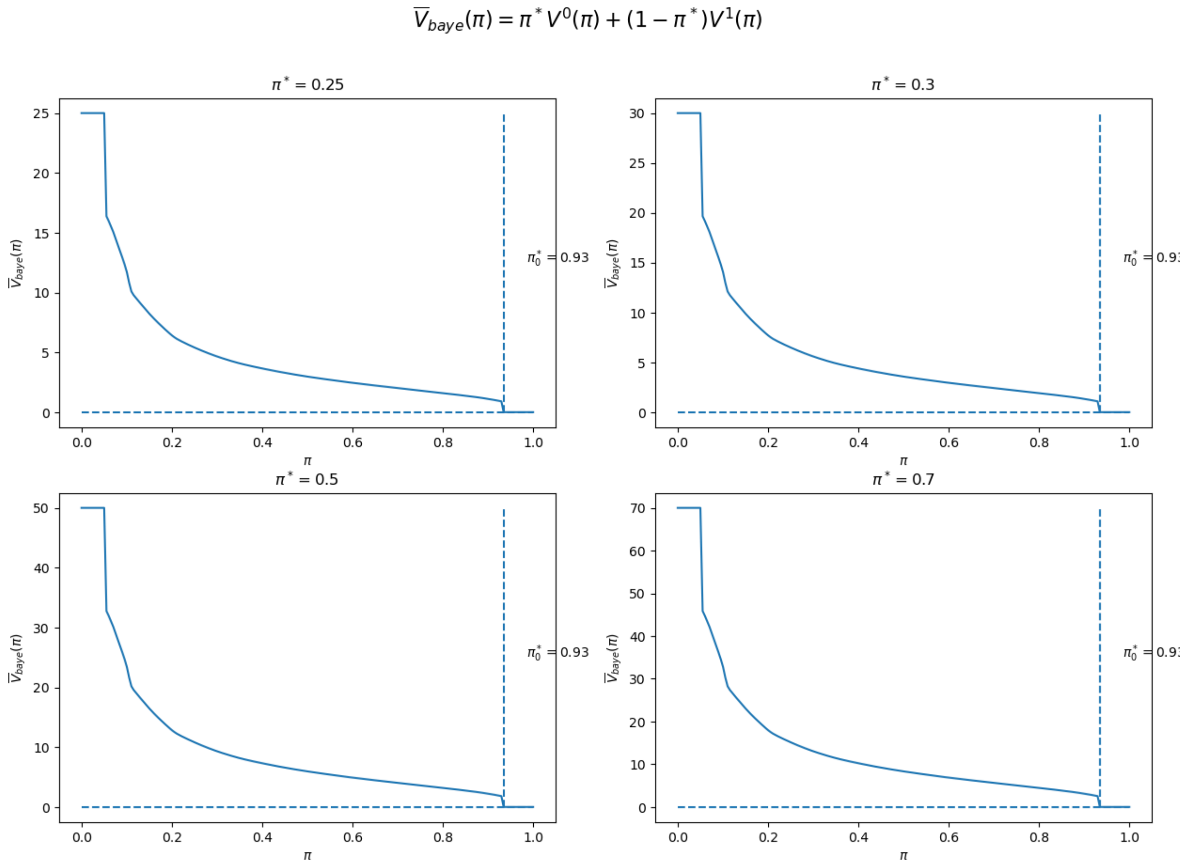
We can then determine an initial Bayesian prior π_0^* that minimizes this objective concept of expected loss.

The figure below plots four cases corresponding to $\pi^* = 0.25, 0.3, 0.5, 0.7$.

We observe that in each case π_0^* equals π^*.

```
def compute_V_baye_bar(π_star, V0, V1, wf):

    V_baye = π_star * V0 + (1 - π_star) * V1
    π_idx = np.argmin(V_baye)
    π_optimal = wf.π_grid[π_idx]
    V_baye_bar = V_baye[π_idx]

    return V_baye, π_optimal, V_baye_bar
```

```
π_star_arr = [0.25, 0.3, 0.5, 0.7]

fig, axs = plt.subplots(2, 2, figsize=(15, 10))

for i, π_star in enumerate(π_star_arr):
    row_i = i // 2
    col_i = i % 2

    V_baye, π_optimal, V_baye_bar = compute_V_baye_bar(π_star, V0, V1, wf)

    axs[row_i, col_i].plot(wf.π_grid, V_baye)
    axs[row_i, col_i].hlines(V_baye_bar, 0, 1, linestyle='--')
    axs[row_i, col_i].vlines(π_optimal, V_baye_bar, V_baye.max(), linestyle='--')
    axs[row_i, col_i].text(π_optimal+0.05, (V_baye_bar + V_baye.max()) / 2,
                        r'${\pi_0^*}=$'+f'{π_optimal:0.2f}')
    axs[row_i, col_i].set_xlabel(r'$\pi$')
    axs[row_i, col_i].set_ylabel(r'$\overline{V}_{baye}(\pi)$')
    axs[row_i, col_i].set_title(r'$\pi^*=$' + f'{π_star}')

fig.suptitle(r'$\overline{V}_{baye}(\pi)=\pi^*V^0(\pi) + (1-\pi^*)V^1(\pi)$',
 ↪fontsize=16)
plt.show()
```



This pattern of outcomes holds more generally.

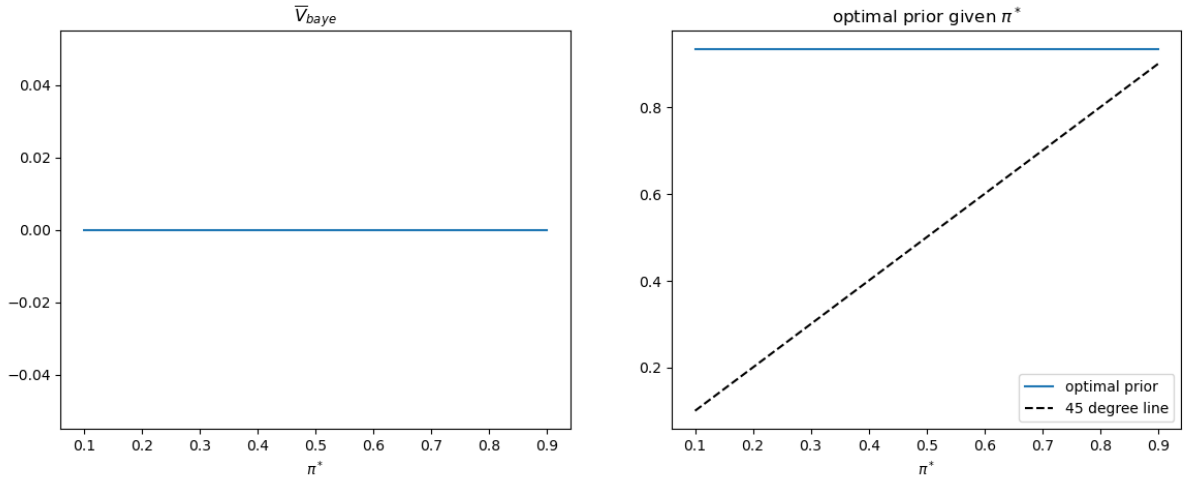
Thus, the following Python code generates the associated graph that verifies the equality of π_0^* to π^* holds for all π^*.

```
π_star_arr = np.linspace(0.1, 0.9, n_π)
V_baye_bar_arr = np.empty_like(π_star_arr)
π_optimal_arr = np.empty_like(π_star_arr)

for i, π_star in enumerate(π_star_arr):

    V_baye, π_optimal, V_baye_bar = compute_V_baye_bar(π_star, V0, V1, wf)

    V_baye_bar_arr[i] = V_baye_bar
    π_optimal_arr[i] = π_optimal

fig, axs = plt.subplots(1, 2, figsize=(14, 5))

axs[0].plot(π_star_arr, V_baye_bar_arr)
axs[0].set_xlabel(r'$\pi^*$')
axs[0].set_title(r'$\overline{V}_{baye}$')

axs[1].plot(π_star_arr, π_optimal_arr, label='optimal prior')
axs[1].plot([π_star_arr.min(), π_star_arr.max()],
            [π_star_arr.min(), π_star_arr.max()],
            c='k', linestyle='--', label='45 degree line')
axs[1].set_xlabel(r'$\pi^*$')
axs[1].set_title(r'optimal prior given $\pi^*$')
axs[1].legend()
```

(continues on next page)

(continued from previous page)

```
plt.show()
```



31.5 Was the Navy Captain's Hunch Correct?

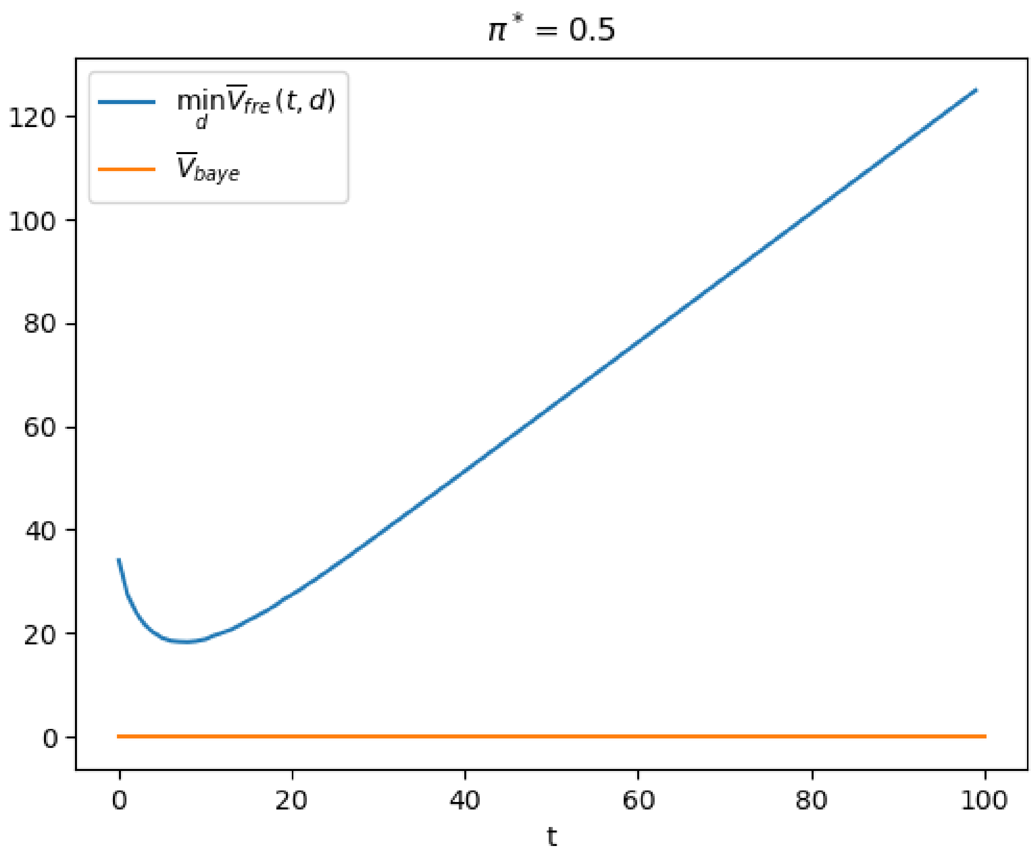
We now compare average losses obtained by our frequentist Neyman-Pearson and Bayesian decision rules.

As a starting point, let's compare average loss functions when $\pi^* = 0.5$.

```
π_star = 0.5
```

```
# frequentist
V_fre_arr, PFA_arr, PD_arr = compute_V_fre(L0_arr, L1_arr, π_star, wf)

# bayesian
V_baye = π_star * V0 + (1 - π_star) * V1
V_baye_bar = V_baye.min()
```

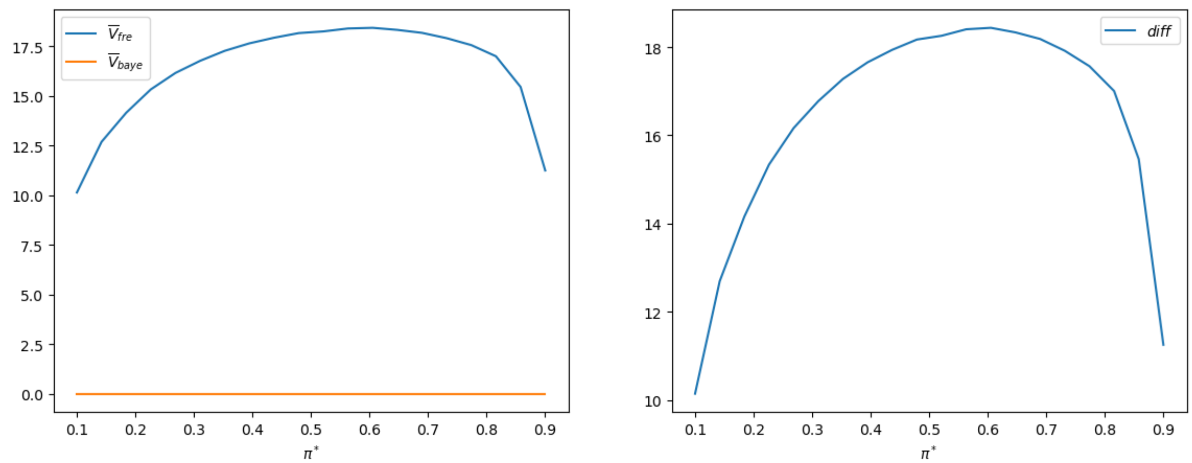
```
plt.plot(range(T), V_fre_arr, label=r'$\min_{d} \overline{V}_{fre}(t,d)$')
plt.plot([0, T], [V_baye_bar, V_baye_bar], label=r'$\overline{V}_{baye}$')
plt.xlabel('t')
plt.title(r'$\pi^*=0.5$')
plt.legend()
plt.show()
```



Evidently, there is no sample size t at which the Neyman-Pearson decision rule attains a lower loss function than does the Bayesian rule.

Furthermore, the following graph indicates that the Bayesian decision rule does better on average for all values of π^*.

```
fig, axs = plt.subplots(1, 2, figsize=(14, 5))

axs[0].plot(π_star_arr, V_fre_bar_arr, label=r'$\overline{V}_{fre}$')
axs[0].plot(π_star_arr, V_baye_bar_arr, label=r'$\overline{V}_{baye}$')
axs[0].legend()
axs[0].set_xlabel(r'$\pi^*$')

axs[1].plot(π_star_arr, V_fre_bar_arr - V_baye_bar_arr, label='$diff$')
axs[1].legend()
axs[1].set_xlabel(r'$\pi^*$')

plt.show()
```



The right panel of the above graph plots the difference $\bar{V}_{fre} - \bar{V}_{Bayes}$.

It is always positive.

31.6 More Details

We can provide more insights by focusing on the case in which $\pi^* = 0.5 = \pi_0$.

```
π_star = 0.5
```

Recall that when $\pi^* = 0.5$, the frequentist Neyman-Pearson decision rule sets a sample size `t_optimal` **ex ante**.

For our parameter settings, we can compute its value:

```
t_optimal
```

```
np.int64(9)
```

For convenience, let's define `t_idx` as the Python array index corresponding to `t_optimal` sample size.

```
t_idx = t_optimal - 1
```

31.7 Distribution of Bayesian Decision Rule's Time to Decide

We use simulations to compute the frequency distribution of the time to decide for the Bayesian decision rule and compare that time to the frequentist rule's fixed t.

The following Python code creates a graph that shows the frequency distribution of Bayesian times to decide of Bayesian decision maker, conditional on distribution $q = f_0$ or $q = f_1$ generating the data.

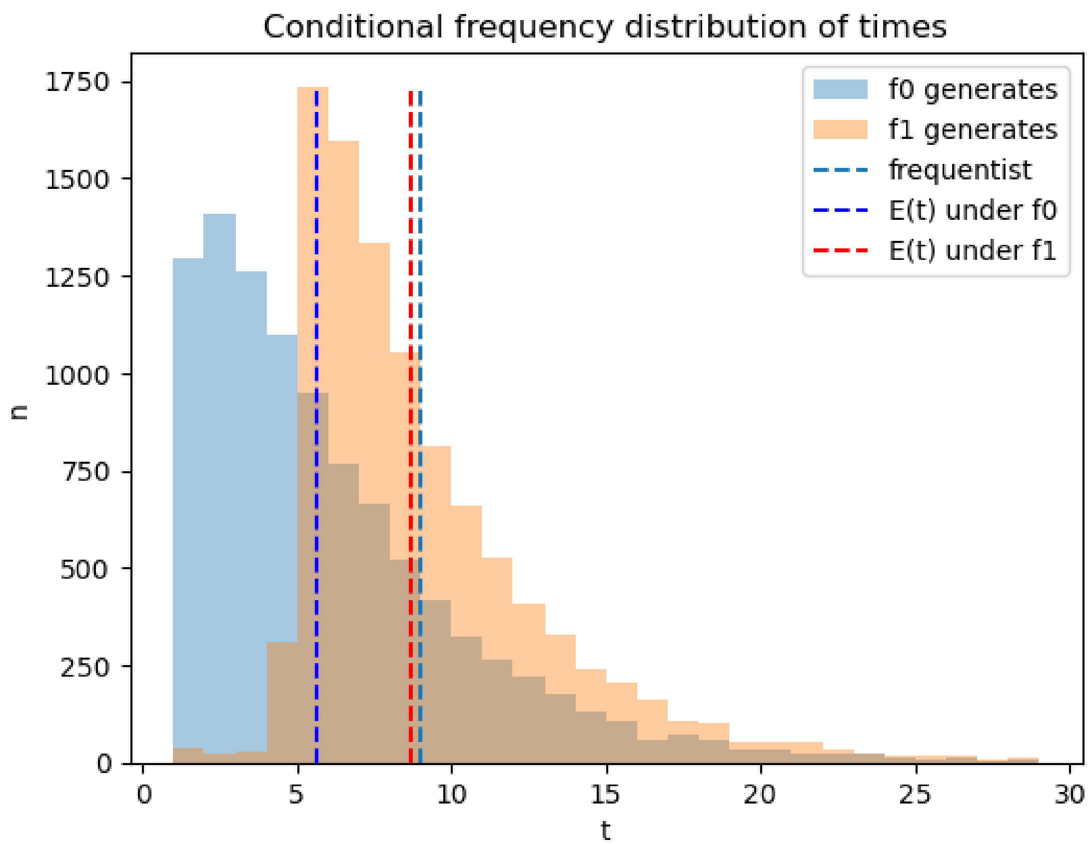
The blue and red dotted lines show averages for the Bayesian decision rule, while the black dotted line shows the frequentist optimal sample size t.

On average the Bayesian rule decides **earlier** than the frequentist rule when $q = f_0$ and **later** when $q = f_1$.

```
@jit(parallel=True)
def check_results(L_arr, A, B, flag, π0):

    N, T = L_arr.shape

    time_arr = np.empty(N)
    correctness = np.empty(N)

    π_arr = π0 * L_arr / (π0 * L_arr + 1 - π0)

    for i in prange(N):
        for t in range(T):
            if (π_arr[i, t] < B) or (π_arr[i, t] > A):
                time_arr[i] = t + 1
                correctness[i] = (flag == 0 and π_arr[i, t] > A) or (flag == 1 and π_
↪arr[i, t] < B)
                break

    return time_arr, correctness
```

```
time_arr0, correctness0 = check_results(L0_arr, A, B, 0, π_star)
time_arr1, correctness1 = check_results(L1_arr, A, B, 1, π_star)

# unconditional distribution
time_arr_u = np.concatenate((time_arr0, time_arr1))
correctness_u = np.concatenate((correctness0, correctness1))
```

```
n1 = plt.hist(time_arr0, bins=range(1, 30), alpha=0.4, label='f0 generates')[0]
n2 = plt.hist(time_arr1, bins=range(1, 30), alpha=0.4, label='f1 generates')[0]
plt.vlines(t_optimal, 0, max(n1.max(), n2.max()), linestyle='--', label='frequentist')
plt.vlines(np.mean(time_arr0), 0, max(n1.max(), n2.max()),
           linestyle='--', color='b', label='E(t) under f0')
plt.vlines(np.mean(time_arr1), 0, max(n1.max(), n2.max()),
           linestyle='--', color='r', label='E(t) under f1')
plt.legend();

plt.xlabel('t')
plt.ylabel('n')
plt.title('Conditional frequency distribution of times')

plt.show()
```



Later we'll figure out how these distributions ultimately affect objective expected values under the Neyman-Pearson and Bayesian decision rules.

To begin, let's look at simulations of the Bayesian's beliefs over time.

We can compute updated beliefs at any time t using the one-to-one mapping from L_t to π_t given π_0 described in this lecture *Likelihood Ratio Processes*.

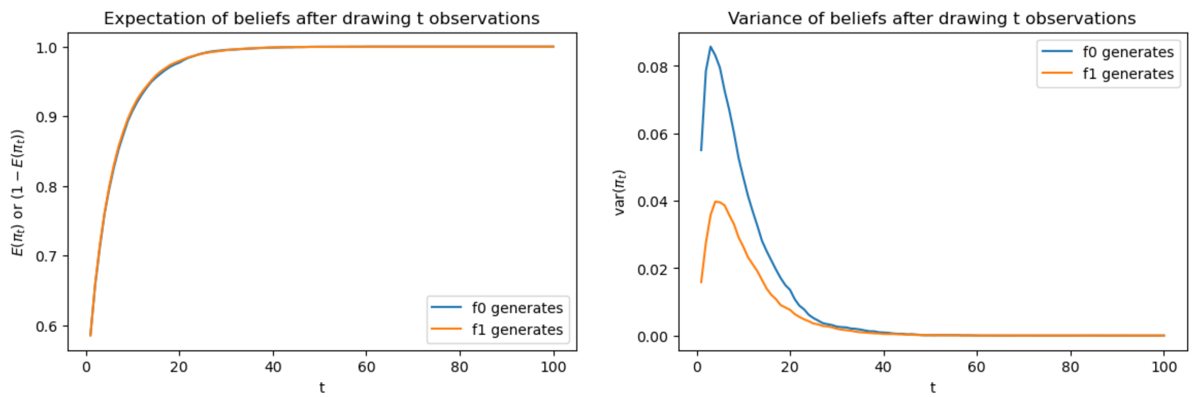
```
π0_arr = π_star * L0_arr / (π_star * L0_arr + 1 - π_star)
π1_arr = π_star * L1_arr / (π_star * L1_arr + 1 - π_star)
```

```
fig, axs = plt.subplots(1, 2, figsize=(14, 4))

axs[0].plot(np.arange(1, π0_arr.shape[1]+1), np.mean(π0_arr, 0), label='f0 generates')
axs[0].plot(np.arange(1, π1_arr.shape[1]+1), 1 - np.mean(π1_arr, 0), label='f1␣
↪generates')
axs[0].set_xlabel('t')
axs[0].set_ylabel(r'$E(\pi_t)$ or ($1 - E(\pi_t)$)')
axs[0].set_title('Expectation of beliefs after drawing t observations')
axs[0].legend()

axs[1].plot(np.arange(1, π0_arr.shape[1]+1), np.var(π0_arr, 0), label='f0 generates')
axs[1].plot(np.arange(1, π1_arr.shape[1]+1), np.var(π1_arr, 0), label='f1 generates')
axs[1].set_xlabel('t')
axs[1].set_ylabel(r'var($\pi_t$)')
axs[1].set_title('Variance of beliefs after drawing t observations')
axs[1].legend()
```

(continues on next page)

(continued from previous page)

```
plt.show()
```



The above figures compare averages and variances of updated Bayesian posteriors after t draws.

The left graph compares $E(\pi_t)$ under f_0 to $1 - E(\pi_t)$ under f_1: they lie on top of each other.

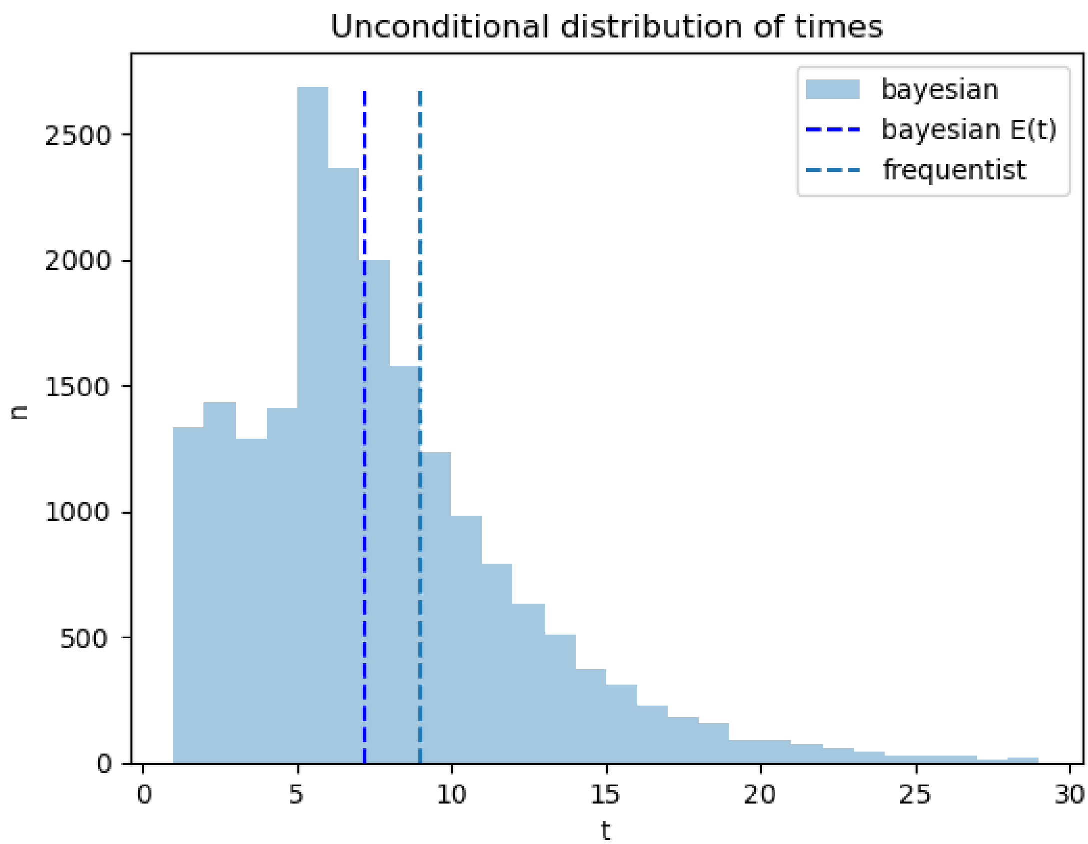
However, as the right hand side graph shows, there is significant difference in variances when t is small: the variance is lower under f_1.

The difference in variances is the reason that the Bayesian decision maker waits longer to decide when f_1 generates the data.

The code below plots outcomes of constructing an unconditional distribution by simply pooling the simulated data across the two possible distributions f_0 and f_1.

The pooled distribution describes a sense in which on average the Bayesian decides earlier, an outcome that seems at least partly to confirm the Navy Captain's hunch.

```
n = plt.hist(time_arr_u, bins=range(1, 30), alpha=0.4, label='bayesian')[0]
plt.vlines(np.mean(time_arr_u), 0, n.max(), linestyle='--',
           color='b', label='bayesian E(t)')
plt.vlines(t_optimal, 0, n.max(), linestyle='--', label='frequentist')
plt.legend()

plt.xlabel('t')
plt.ylabel('n')
plt.title('Unconditional distribution of times')

plt.show()
```



31.8 Probability of Making Correct Decision

Now we use simulations to compute the fractions of samples in which the Bayesian and the frequentist Neyman-Pearson decision rules decide correctly.

For the frequentist Neyman-Pearson rule, the probability of making the correct decision under f_1 is the optimal probability of detection given t that we defined earlier, and similarly it equals 1 minus the optimal probability of a false alarm under f_0.

Below we plot these two probabilities for the frequentist rule, along with the conditional probabilities that the Bayesian rule decides before t *and* that the decision is correct.

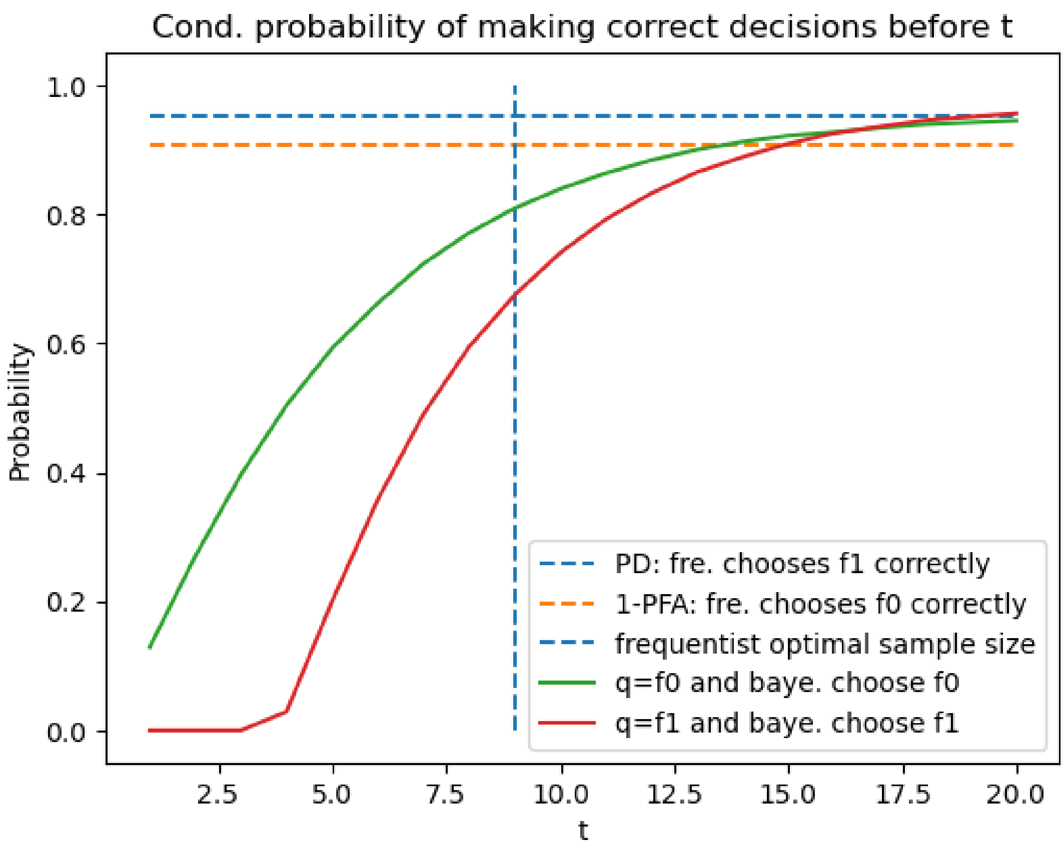
```
# optimal PFA and PD of frequentist with optimal sample size
V, PFA, PD = V_fre_t(t_optimal, L0_arr, L1_arr, π_star, wf)
```

```
plt.plot([1, 20], [PD, PD], linestyle='--', label='PD: fre. chooses f1 correctly')
plt.plot([1, 20], [1-PFA, 1-PFA], linestyle='--', label='1-PFA: fre. chooses f0␣
↪correctly')
plt.vlines(t_optimal, 0, 1, linestyle='--', label='frequentist optimal sample size')

N = time_arr0.size
T_arr = np.arange(1, 21)
plt.plot(T_arr, [np.sum(correctness0[time_arr0 <= t] == 1) / N for t in T_arr],
        label='q=f0 and baye. choose f0')
plt.plot(T_arr, [np.sum(correctness1[time_arr1 <= t] == 1) / N for t in T_arr],
```

(continues on next page)

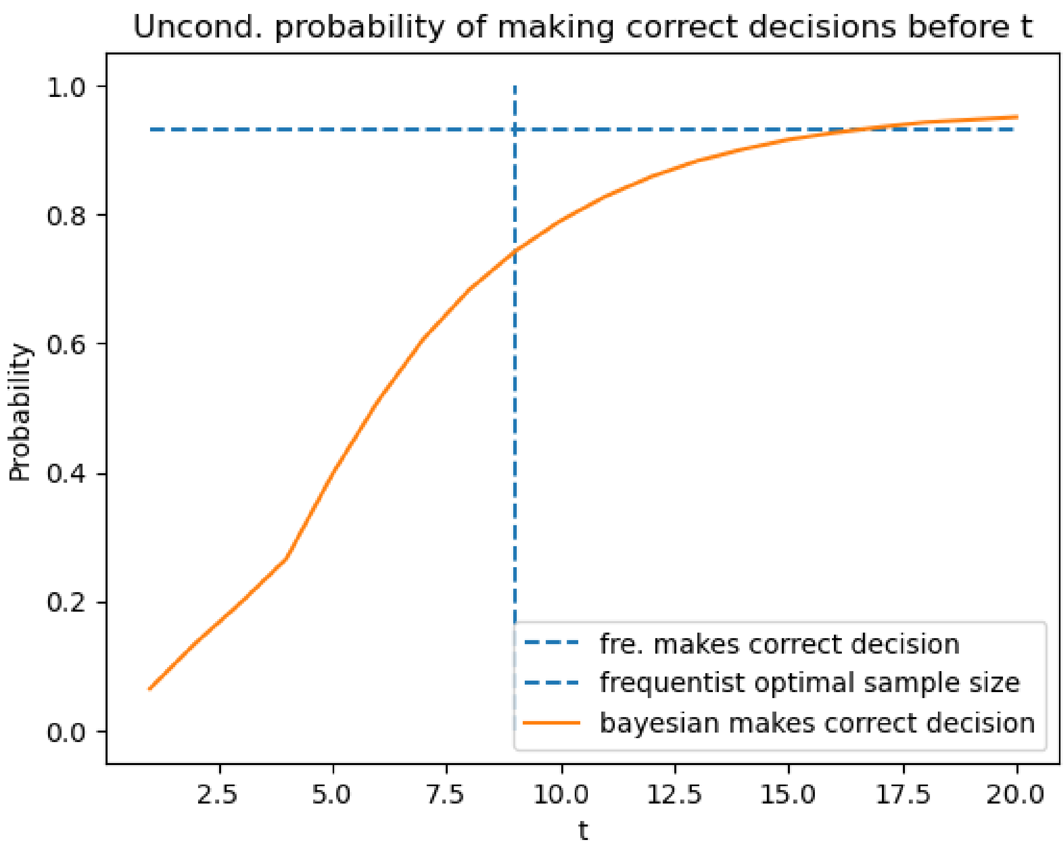
(continued from previous page)

```
        label='q=f1 and baye. choose f1')
plt.legend(loc=4)

plt.xlabel('t')
plt.ylabel('Probability')
plt.title('Cond. probability of making correct decisions before t')

plt.show()
```



By averaging using π^*, we also plot the unconditional distribution.

```
plt.plot([1, 20], [(PD + 1 - PFA) / 2, (PD + 1 - PFA) / 2],
        linestyle='--', label='fre. makes correct decision')
plt.vlines(t_optimal, 0, 1, linestyle='--', label='frequentist optimal sample size')

N = time_arr_u.size
plt.plot(T_arr, [np.sum(correctness_u[time_arr_u <= t] == 1) / N for t in T_arr],
        label="bayesian makes correct decision")
plt.legend()

plt.xlabel('t')
plt.ylabel('Probability')
plt.title('Uncond. probability of making correct decisions before t')

plt.show()
```



31.9 Distribution of Likelihood Ratios at Neyman-Pearson's t

Next we use simulations to construct distributions of likelihood ratios after t draws.

To serve as useful reference points, we also show likelihood ratios that correspond to the Bayesian cutoffs A and B.

In order to exhibit the distribution more clearly, we report logarithms of likelihood ratios.

The graphs below reports two distributions, one conditional on f_0 generating the data, the other conditional on f_1 generating the data.

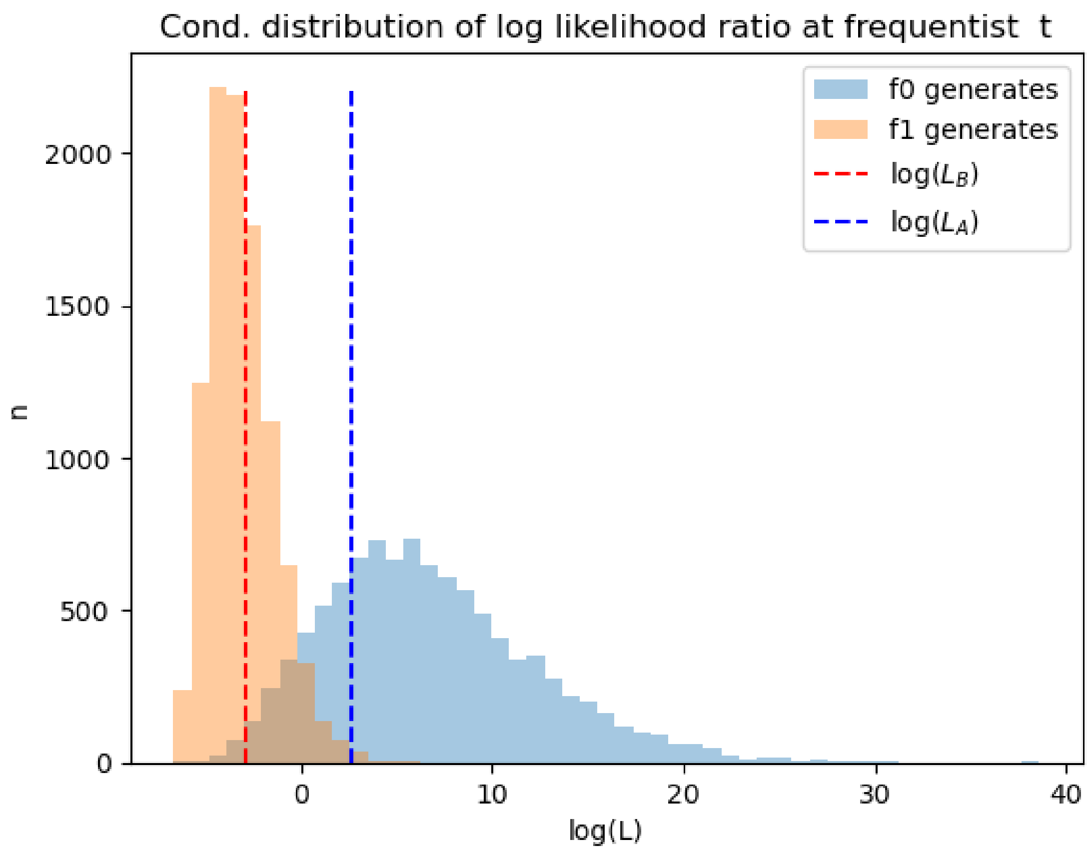
```
LA = (1 - π_star) *  A / (π_star - π_star * A)
LB = (1 - π_star) *  B / (π_star - π_star * B)
```

```
L_min = min(L0_arr[:, t_idx].min(), L1_arr[:, t_idx].min())
L_max = max(L0_arr[:, t_idx].max(), L1_arr[:, t_idx].max())
bin_range = np.linspace(np.log(L_min), np.log(L_max), 50)
n0 = plt.hist(np.log(L0_arr[:, t_idx]), bins=bin_range, alpha=0.4, label='f0 generates
↪')[0]
n1 = plt.hist(np.log(L1_arr[:, t_idx]), bins=bin_range, alpha=0.4, label='f1 generates
↪')[0]

plt.vlines(np.log(LB), 0, max(n0.max(), n1.max()), linestyle='--', color='r', label=
↪'log($L_B$)')
plt.vlines(np.log(LA), 0, max(n0.max(), n1.max()), linestyle='--', color='b', label=
```

(continues on next page)

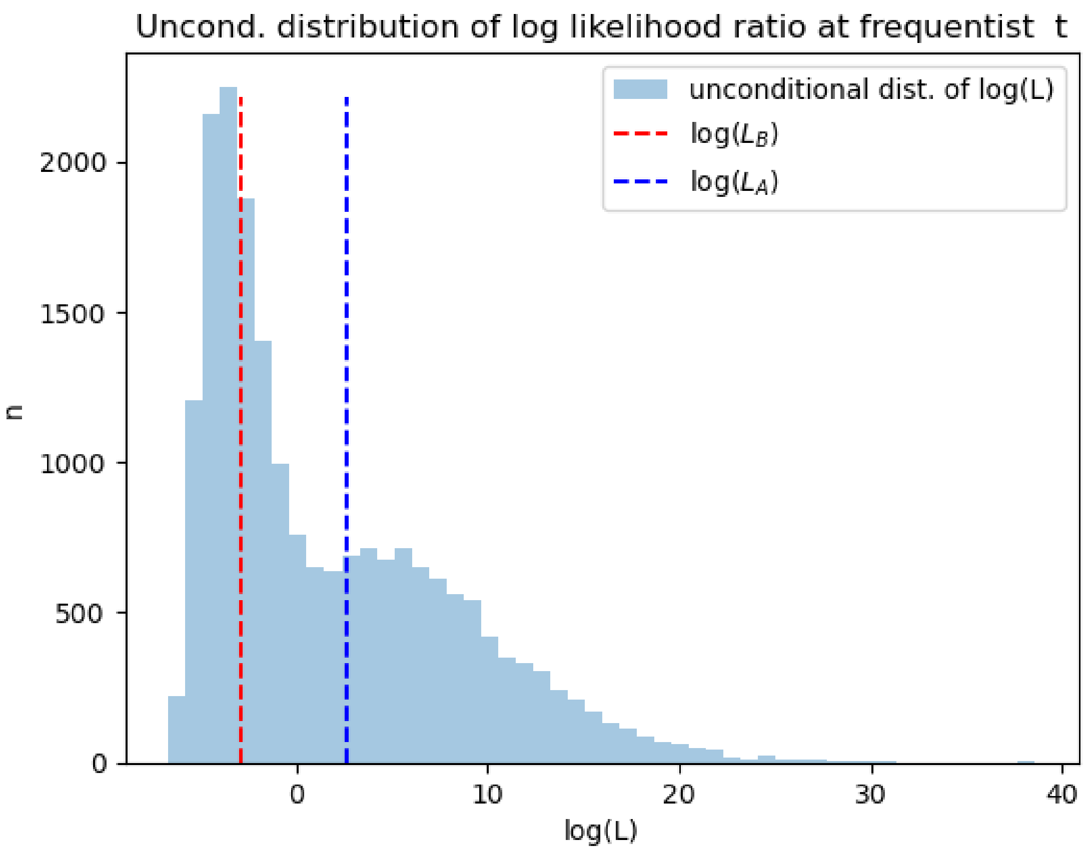
(continued from previous page)

```
↪'log($L_A$)')
plt.legend()

plt.xlabel('log(L)')
plt.ylabel('n')
plt.title('Cond. distribution of log likelihood ratio at frequentist  t')

plt.show()
```



The next graph plots the unconditional distribution of Bayesian times to decide, constructed as earlier by pooling the two conditional distributions.

```
plt.hist(np.log(np.concatenate([L0_arr[:, t_idx], L1_arr[:, t_idx]])),
        bins=50, alpha=0.4, label='unconditional dist. of log(L)')
plt.vlines(np.log(LB), 0, max(n0.max(), n1.max()), linestyle='--', color='r', label=
↪'log($L_B$)')
plt.vlines(np.log(LA), 0, max(n0.max(), n1.max()), linestyle='--', color='b', label=
↪'log($L_A$)')
plt.legend()

plt.xlabel('log(L)')
plt.ylabel('n')
plt.title('Uncond. distribution of log likelihood ratio at frequentist  t')

plt.show()
```



Uncond. distribution of log likelihood ratio at frequentist t
unconditional dist. of log(L)
log(L_B)
log(L_A)
n
log(L)
2000
1500
1000
500
0
0
10
20
30
40

Part V

Linear Programming

CHAPTER

THIRTYTWO

OPTIMAL TRANSPORT

32.1 Overview

The **transportation** or **optimal transport** problem is interesting both because of its many applications and because of its important role in the history of economic theory.

In this lecture, we describe the problem, tell how linear programming is a key tool for solving it, and then provide some examples.

We will provide other applications in followup lectures.

The optimal transport problem was studied in early work about linear programming, as summarized for example by [Dorfman *et al.*, 1958]. A modern reference about applications in economics is [Galichon, 2016].

Below, we show how to solve the optimal transport problem using several implementations of linear programming, including, in order,

1. the linprog solver from SciPy,
2. the linprog_simplex solver from QuantEcon and
3. the simplex-based solvers included in the Python Optimal Transport package.

```
!pip install --upgrade quantecon
!pip install --upgrade POT
```

Let's start with some imports.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import linprog
from quantecon.optimize.linprog_simplex import linprog_simplex
import ot
from scipy.stats import betabinom
import networkx as nx
```

32.2 The Optimal Transport Problem

Suppose that m factories produce goods that must be sent to n locations.

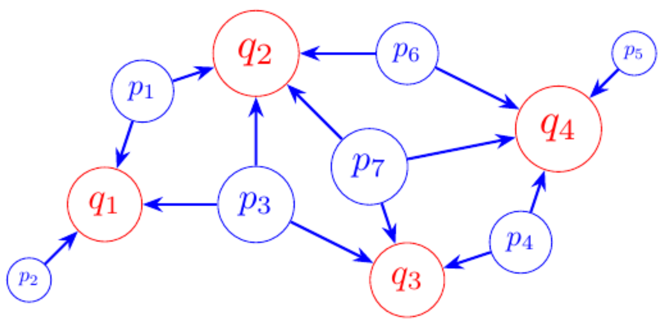
Let

- x_{ij} denote the quantity shipped from factory i to location j
- c_{ij} denote the cost of shipping one unit from factory i to location j
- p_i denote the capacity of factory i and q_j denote the amount required at location j.
- $i = 1, 2, \dots, m$ and $j = 1, 2, \dots, n$.

A planner wants to minimize total transportation costs subject to the following constraints:

- The amount shipped **from** each factory must equal its capacity.
- The amount shipped **to** each location must equal the quantity required there.

The figure below shows one visualization of this idea, when factories and target locations are distributed in the plane.



The size of the vertices in the figure are proportional to

- capacity, for the factories, and
- demand (amount required) for the target locations.

The arrows show one possible transport plan, which respects the constraints stated above.

The planner's problem can be expressed as the following constrained minimization problem:

$$
\begin{aligned}
\min_{x_{ij}} \ & \sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij} x_{ij} \\
\text{subject to } \ & \sum_{j=1}^{n} x_{ij} = p_i, \quad i = 1, 2, \dots, m \\
& \sum_{i=1}^{m} x_{ij} = q_j, \quad j = 1, 2, \dots, n \\
& x_{ij} \ge 0
\end{aligned} \tag{32.1}
$$

This is an **optimal transport problem** with

- mn decision variables, namely, the entries x_{ij} and
- $m + n$ constraints.

Summing the q_j's across all j's and the p_i's across all i's indicates that the total capacity of all the factories equals total requirements at all locations:

$$\sum_{j=1}^{n} q_j = \sum_{j=1}^{n} \sum_{i=1}^{m} x_{ij} = \sum_{i=1}^{m} \sum_{j=1}^{n} x_{ij} = \sum_{i=1}^{m} p_i \tag{32.2}$$

The presence of the restrictions in (32.2) will be the source of one redundancy in the complete set of restrictions that we describe below.

More about this later.

32.3 The Linear Programming Approach

In this section we discuss using using standard linear programming solvers to tackle the optimal transport problem.

32.3.1 Vectorizing a Matrix of Decision Variables

A *matrix* of decision variables x_{ij} appears in problem (32.1).

The SciPy function `linprog` expects to see a *vector* of decision variables.

This situation impels us to rewrite our problem in terms of a *vector* of decision variables.

Let

- X, C be $m \times n$ matrices with entries x_{ij}, c_{ij},
- p be m-dimensional vector with entries p_i,
- q be n-dimensional vector with entries q_j.

With $\mathbf{1}_n$ denoting the n-dimensional column vector $(1, 1, \dots, 1)'$, our problem can now be expressed compactly as:

$$\begin{aligned} \min_{X} \ & \operatorname{tr}(C' X) \\ \text{subject to } \ & X \ \mathbf{1}_n = p \\ & X' \ \mathbf{1}_m = q \\ & X \geq 0 \end{aligned}$$

We can convert the matrix X into a vector by stacking all of its columns into a column vector.

Doing this is called **vectorization**, an operation that we denote $\operatorname{vec}(X)$.

Similarly, we convert the matrix C into an mn-dimensional vector $\operatorname{vec}(C)$.

The objective function can be expressed as the inner product between $\operatorname{vec}(C)$ and $\operatorname{vec}(X)$:

$$\operatorname{vec}(C)' \cdot \operatorname{vec}(X).$$

To express the constraints in terms of $\operatorname{vec}(X)$, we use a **Kronecker product** denoted by $\otimes$ and defined as follows.

Suppose A is an $m \times s$ matrix with entries (a_{ij}) and that B is an $n \times t$ matrix.

The **Kronecker product** of A and B is defined, in block matrix form, by

$$A \otimes B = \begin{bmatrix} a_{11}B & a_{12}B & \dots & a_{1s}B \\ a_{21}B & a_{22}B & \dots & a_{2s}B \\ & & \vdots & \\ a_{m1}B & a_{m2}B & \dots & a_{ms}B \end{bmatrix}.$$

$A \otimes B$ is an $mn \times st$ matrix.

It has the property that for any $m \times n$ matrix X

$$\text{vec}(A'XB) = (B' \otimes A')\,\text{vec}(X). \tag{32.3}$$

We can now express our constraints in terms of $\text{vec}(X)$.

Let $A = \mathbf{I}'_m, B = \mathbf{1}_n$.

By equation (32.3)

$$X\,\mathbf{1}_n = \text{vec}(X\,\mathbf{1}_n) = \text{vec}(\mathbf{I}_m X\,\mathbf{1}_n) = (\mathbf{1}'_n \otimes \mathbf{I}_m)\,\text{vec}(X).$$

where $\mathbf{I}_m$ denotes the $m \times m$ identity matrix.

Constraint $X\,\mathbf{1}_n = p$ can now be written as:

$$(\mathbf{1}'_n \otimes \mathbf{I}_m)\,\text{vec}(X) = p.$$

Similarly, the constraint $X'\,\mathbf{1}_m = q$ can be rewriten as:

$$(\mathbf{I}_n \otimes \mathbf{1}'_m)\,\text{vec}(X) = q.$$

With $z := \text{vec}(X)$, our problem can now be expressed in terms of an mn-dimensional vector of decision variables:

$$\begin{aligned} \min_{z} \ & \text{vec}(C)'z \\ \text{subject to} \ & Az = b \\ & z \geq 0 \end{aligned} \tag{32.4}$$

where

$$A = \begin{bmatrix} \mathbf{1}'_n \otimes \mathbf{I}_m \\ \mathbf{I}_n \otimes \mathbf{1}'_m \end{bmatrix} \quad \text{and} \quad b = \begin{bmatrix} p \\ q \end{bmatrix}$$

32.3.2 An Application

We now provide an example that takes the form (32.4) that we'll solve by deploying the function `linprog`.

The table below provides numbers for the requirements vector q, the capacity vector p, and entries c_{ij} of the cost-of-shipping matrix C.

The numbers in the above table tell us to set $m = 3$, $n = 5$, and construct the following objects:

$$p = \begin{bmatrix} 50 \\ 100 \\ 150 \end{bmatrix}, \quad q = \begin{bmatrix} 25 \\ 115 \\ 60 \\ 30 \\ 70 \end{bmatrix} \quad \text{and} \quad C = \begin{bmatrix} 10 & 15 & 20 & 20 & 40 \\ 20 & 40 & 15 & 30 & 30 \\ 30 & 35 & 40 & 55 & 25 \end{bmatrix}.$$

Let's write Python code that sets up the problem and solves it.

```python
# Define parameters
m = 3
n = 5

p = np.array([50.0, 100.0, 150.0])
```

(continues on next page)

(continued from previous page)

```
q = np.array([25.0, 115.0, 60.0, 30.0, 70.0])

C = np.array([[10.0, 15.0, 20.0, 20.0, 40.0],
              [20.0, 40.0, 15.0, 30.0, 30.0],
              [30.0, 35.0, 40.0, 55.0, 25.0]])

# Vectorize matrix C
C_vec = C.reshape((m*n, 1), order='F')

# Construct matrix A by Kronecker product
A1 = np.kron(np.ones((1, n)), np.identity(m))
A2 = np.kron(np.identity(n), np.ones((1, m)))
A = np.vstack([A1, A2])

# Construct vector b
b = np.hstack([p, q])

# Solve the primal problem
res = linprog(C_vec, A_eq=A, b_eq=b)

# Print results
print("message:", res.message)
print("nit:", res.nit)
print("fun:", res.fun)
print("z:", res.x)
print("X:", res.x.reshape((m,n), order='F'))
```

```
message: Optimization terminated successfully. (HiGHS Status 7: Optimal)
nit: 8
fun: 7225.0
z: [ 0. 10. 15. 50.  0. 65.  0. 60.  0.  0. 30.  0.  0.  0. 70.]
X: [[ 0. 50.  0.  0.  0.]
 [10.  0. 60. 30.  0.]
 [15. 65.  0.  0. 70.]]
```

Notice how, in the line `C_vec = C.reshape((m*n, 1), order='F')`, we are careful to vectorize using the flag `order='F'`.

This is consistent with converting C into a vector by stacking all of its columns into a column vector.

Here `'F'` stands for "Fortran", and we are using Fortran style column-major order.

(For an alternative approach, using Python's default row-major ordering, see this lecture by Alfred Galichon.)

Interpreting the solver behavior:

Looking at matrix A, we can see that it is rank deficient.

```
np.linalg.matrix_rank(A) < min(A.shape)
```

```
np.True_
```

This indicates that the linear program has been set up to include one or more redundant constraints.

Here, the source of the redundancy is the structure of restrictions (32.2).

Let's explore this further by printing out A and staring at it.

```
A
```

```
array([[1., 0., 0., 1., 0., 0., 1., 0., 0., 1., 0., 0., 1., 0., 0.],
       [0., 1., 0., 0., 1., 0., 0., 1., 0., 0., 1., 0., 0., 1., 0.],
       [0., 0., 1., 0., 0., 1., 0., 0., 1., 0., 0., 1., 0., 0., 1.],
       [1., 1., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 1., 1., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 1., 1., 1., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 1., 1., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 1., 1.]])
```

The singularity of A reflects that the first three constraints and the last five constraints both require that "total requirements equal total capacities" expressed in (32.2).

One equality constraint here is redundant.

Fortunately, SciPy's `linprog` function handles the redundant constraints automatically without explicitly warning about rank deficiency.

But we can drop one of the equality constraints, and use only 7 of them.

After doing this, we attain the same minimized cost.

However, we find a different transportation plan.

Though it is a different plan, it attains the same cost!

```
linprog(C_vec, A_eq=A[:-1], b_eq=b[:-1])
```

```
        message: Optimization terminated successfully. (HiGHS Status 7: Optimal)
        success: True
         status: 0
            fun: 7225.0
              x: [ 0.000e+00  1.000e+01 ...  0.000e+00  7.000e+01]
            nit: 8
          lower:  residual: [ 0.000e+00  1.000e+01 ...  0.000e+00
                              7.000e+01]
                 marginals: [ 0.000e+00  0.000e+00 ...  1.500e+01
                              0.000e+00]
          upper:  residual: [       inf        inf ...        inf
                                   inf]
                 marginals: [ 0.000e+00  0.000e+00 ...  0.000e+00
                              0.000e+00]
          eqlin:  residual: [ 0.000e+00  0.000e+00  0.000e+00  0.000e+00
                              0.000e+00  0.000e+00  0.000e+00]
                 marginals: [ 5.000e+00  1.500e+01  2.500e+01  5.000e+00
                              1.000e+01 -0.000e+00  1.500e+01]
        ineqlin:  residual: []
                 marginals: []
 mip_node_count: 0
 mip_dual_bound: 0.0
        mip_gap: 0.0
```

```
%time linprog(C_vec, A_eq=A[:-1], b_eq=b[:-1])
```

```
CPU times: user 2.14 ms, sys: 17 µs, total: 2.15 ms
Wall time: 1.91 ms
```

```
        message: Optimization terminated successfully. (HiGHS Status 7: Optimal)
        success: True
         status: 0
            fun: 7225.0
              x: [ 0.000e+00  1.000e+01 ...  0.000e+00  7.000e+01]
            nit: 8
          lower:  residual: [ 0.000e+00  1.000e+01 ...  0.000e+00
                              7.000e+01]
                 marginals: [ 0.000e+00  0.000e+00 ...  1.500e+01
                              0.000e+00]
          upper:  residual: [       inf        inf ...        inf
                                    inf]
                 marginals: [ 0.000e+00  0.000e+00 ...  0.000e+00
                              0.000e+00]
          eqlin:  residual: [ 0.000e+00  0.000e+00  0.000e+00  0.000e+00
                              0.000e+00  0.000e+00  0.000e+00]
                 marginals: [ 5.000e+00  1.500e+01  2.500e+01  5.000e+00
                              1.000e+01 -0.000e+00  1.500e+01]
        ineqlin:  residual: []
                 marginals: []
 mip_node_count: 0
 mip_dual_bound: 0.0
        mip_gap: 0.0
```

```
%time linprog(C_vec, A_eq=A, b_eq=b)
```

```
CPU times: user 2.21 ms, sys: 26 µs, total: 2.23 ms
Wall time: 1.98 ms
```

```
        message: Optimization terminated successfully. (HiGHS Status 7: Optimal)
        success: True
         status: 0
            fun: 7225.0
              x: [ 0.000e+00  1.000e+01 ...  0.000e+00  7.000e+01]
            nit: 8
          lower:  residual: [ 0.000e+00  1.000e+01 ...  0.000e+00
                              7.000e+01]
                 marginals: [ 0.000e+00  0.000e+00 ...  1.500e+01
                              0.000e+00]
          upper:  residual: [       inf        inf ...        inf
                                    inf]
                 marginals: [ 0.000e+00  0.000e+00 ...  0.000e+00
                              0.000e+00]
          eqlin:  residual: [ 0.000e+00  0.000e+00  0.000e+00  0.000e+00
                              0.000e+00  0.000e+00  0.000e+00  0.000e+00]
                 marginals: [ 1.000e+01  2.000e+01  3.000e+01 -0.000e+00
                              5.000e+00 -5.000e+00  1.000e+01 -5.000e+00]
        ineqlin:  residual: []
                 marginals: []
 mip_node_count: 0
 mip_dual_bound: 0.0
        mip_gap: 0.0
```

Evidently, it is slightly quicker to work with the system that removed a redundant constraint.

Let's drill down and do some more calculations to help us understand whether or not our finding **two** different optimal transport plans reflects our having dropped a redundant equality constraint.

Hint

It will turn out that dropping a redundant equality isn't really what mattered.

To verify our hint, we shall simply use **all** of the original equality constraints (including a redundant one), but we'll just shuffle the order of the constraints.

```
arr = np.arange(m+n)
```

```
sol_found = []
cost = []

# simulate 1000 times
for i in range(1000):

    np.random.shuffle(arr)
    res_shuffle = linprog(C_vec, A_eq=A[arr], b_eq=b[arr])

    # if find a new solution
    sol = tuple(res_shuffle.x)
    if sol not in sol_found:
        sol_found.append(sol)
        cost.append(res_shuffle.fun)
```

```
for i in range(len(sol_found)):
    print(f"transportation plan {i}: ", sol_found[i])
    print(f"     minimized cost {i}: ", cost[i])
```

```
transportation plan 0:  (np.float64(0.0), np.float64(10.0), np.float64(15.0), np.
↪float64(50.0), np.float64(0.0), np.float64(65.0), np.float64(0.0), np.float64(60.
↪0), np.float64(0.0), np.float64(0.0), np.float64(30.0), np.float64(0.0), np.
↪float64(0.0), np.float64(0.0), np.float64(70.0))
     minimized cost 0:  7225.0
```

Ah hah! As you can see, putting constraints in different orders in this case uncovers two optimal transportation plans that achieve the same minimized cost.

These are the same two plans computed earlier.

Next, we show that leaving out the first constraint "accidentally" leads to the initial plan that we computed.

```
linprog(C_vec, A_eq=A[1:], b_eq=b[1:])
```

```
        message: Optimization terminated successfully. (HiGHS Status 7: Optimal)
        success: True
         status: 0
            fun: 7225.0
              x: [ 0.000e+00  1.000e+01 ...  0.000e+00  7.000e+01]
            nit: 8
          lower:  residual: [ 0.000e+00  1.000e+01 ...  0.000e+00
                              7.000e+01]
                 marginals: [ 0.000e+00  0.000e+00 ...  1.500e+01
                              0.000e+00]
          upper:  residual: [       inf        inf ...        inf
                                   inf]
```

(continues on next page)

(continued from previous page)

```
                 marginals: [ 0.000e+00  0.000e+00 ...  0.000e+00
                              0.000e+00]
          eqlin:  residual: [ 0.000e+00  0.000e+00  0.000e+00  0.000e+00
                              0.000e+00  0.000e+00  0.000e+00]
                 marginals: [ 1.000e+01  2.000e+01  1.000e+01  1.500e+01
                              5.000e+00  2.000e+01  5.000e+00]
        ineqlin:  residual: []
                 marginals: []
 mip_node_count: 0
 mip_dual_bound: 0.0
        mip_gap: 0.0
```

Let's compare this transport plan with

```
res.x
```

```
array([ 0., 10., 15., 50.,  0., 65.,  0., 60.,  0.,  0., 30.,  0.,  0.,
        0., 70.])
```

Here the matrix X contains entries x_{ij} that tell amounts shipped **from** factor $i = 1, 2, 3$ **to** location $j = 1, 2, \ldots, 5$.

The vector z evidently equals $\text{vec}(X)$.

The minimized cost from the optimal transport plan is given by the fun variable.

32.3.3 Using a Just-in-Time Compiler

We can also solve optimal transportation problems using a powerful tool from QuantEcon, namely, `quantecon.optimize.linprog_simplex`.

While `scipy.optimize.linprog` uses the HiGHS solver by default, `quantecon.optimize.linprog_simplex` implements the simplex algorithm accelerated by using a just-in-time compiler shipped in the `numba` library.

As you will see very soon, by using `quantecon.optimize.linprog_simplex` the time required to solve an optimal transportation problem can be reduced significantly.

```
# construct matrices/vectors for linprog_simplex
c = C.flatten()

# Equality constraints
A_eq = np.zeros((m+n, m*n))
for i in range(m):
    for j in range(n):
        A_eq[i, i*n+j] = 1
        A_eq[m+j, i*n+j] = 1

b_eq = np.hstack([p, q])
```

Since `quantecon.optimize.linprog_simplex` does maximization instead of minimization, we need to put a negative sign before vector `c`.

```
res_qe = linprog_simplex(-c, A_eq=A_eq, b_eq=b_eq)
```

While the two LP solvers use different algorithms (HiGHS vs. simplex), both should find optimal solutions.

The solutions differs since there are multiple optimal solutions, but the objective values are the same

```
np.allclose(-res_qe.fun, res.fun)
```

```
True
```

```
res_qe.x.reshape((m, n), order='C')
```

```
array([[15., 35.,  0.,  0.,  0.],
       [10.,  0., 60., 30.,  0.],
       [ 0., 80.,  0.,  0., 70.]])
```

```
res.x.reshape((m, n), order='F')
```

```
array([[ 0., 50.,  0.,  0.,  0.],
       [10.,  0., 60., 30.,  0.],
       [15., 65.,  0.,  0., 70.]])
```

Let's do a speed comparison between `scipy.optimize.linprog` and `quantecon.optimize.linprog_simplex`.

```
# scipy.optimize.linprog
%time res = linprog(C_vec, A_eq=A[:-1, :], b_eq=b[:-1])
```

```
CPU times: user 2.3 ms, sys: 17 µs, total: 2.31 ms
Wall time: 2.07 ms
```

```
# quantecon.optimize.linprog_simplex
%time out = linprog_simplex(-c, A_eq=A_eq, b_eq=b_eq)
```

```
CPU times: user 54 µs, sys: 3 µs, total: 57 µs
Wall time: 60.3 µs
```

As you can see, the `quantecon.optimize.linprog_simplex` is much faster.

(Note however, that the SciPy version is probably more stable than the QuantEcon version, having been tested more extensively over a longer period of time.)

32.4 The Dual Problem

Let u, v denotes vectors of dual decision variables with entries $(u_i), (v_j)$.

The **dual** to **minimization** problem (32.1) is the **maximization** problem:

$$\begin{aligned} \max_{u_i, v_j} & \sum_{i=1}^{m} p_i u_i + \sum_{j=1}^{n} q_j v_j \\ \text{subject to } & u_i + v_j \leq c_{ij}, \ i = 1, 2, \dots, m; \ j = 1, 2, \dots, n \end{aligned} \tag{32.5}$$

The dual problem is also a linear programming problem.

It has $m + n$ dual variables and mn constraints.

Vectors u and v of **values** are attached to the first and the second sets of primal constraits, respectively.

Thus, u is attached to the constraints

- $(\mathbf{1}_n' \otimes \mathbf{I}_m)\,\mathrm{vec}(X) = p$

and v is attached to constraints

- $(\mathbf{I}_n \otimes \mathbf{1}_m')\,\mathrm{vec}(X) = q.$

Components of the vectors u and v of per unit **values** are **shadow prices** of the quantities appearing on the right sides of those constraints.

We can write the dual problem as

$$\begin{aligned} \max_{u_i, v_j} \ & pu + qv \\ \text{subject to } \ & A' \begin{bmatrix} u \\ v \end{bmatrix} = \mathrm{vec}(C) \end{aligned} \tag{32.6}$$

For the same numerical example described above, let's solve the dual problem.

```
# Solve the dual problem
res_dual = linprog(-b, A_ub=A.T, b_ub=C_vec,
                   bounds=[(None, None)]*(m+n))

#Print results
print("message:", res_dual.message)
print("nit:", res_dual.nit)
print("fun:", res_dual.fun)
print("u:", res_dual.x[:m])
print("v:", res_dual.x[-n:])
```

```
message: Optimization terminated successfully. (HiGHS Status 7: Optimal)
nit: 9
fun: -7225.0
u: [-20. -10.   0.]
v: [30. 35. 25. 40. 25.]
```

`quantecon.optimize.linprog_simplex` computes and returns the dual variables alongside the primal solution.

The dual variables (shadow prices) can be extracted directly from the primal solution:

```
# The dual variables are returned by linprog_simplex
print("Dual variables from linprog_simplex:")
print("u:", -res_qe.lambd[:m])
print("v:", -res_qe.lambd[m:])
```

```
Dual variables from linprog_simplex:
u: [-20. -10.  -0.]
v: [30. 35. 25. 40. 25.]
```

We can verify these match the dual solution from SciPy:

```
print("Dual variables from SciPy linprog:")
print("u:", res_dual.x[:m])
print("v:", res_dual.x[-n:])
```

```
Dual variables from SciPy linprog:
u: [-20. -10.   0.]
v: [30. 35. 25. 40. 25.]
```

32.4.1 Interpretation of dual problem

By **strong duality** (please see this lecture Linear Programming), we know that:

$$\sum_{i=1}^{m}\sum_{j=1}^{n} c_{ij}x_{ij} = \sum_{i=1}^{m} p_i u_i + \sum_{j=1}^{n} q_j v_j$$

One unit more capacity in factory i, i.e. p_i, results in u_i more transportation costs.

Thus, u_i describes the cost of shipping one unit **from** factory i.

Call this the ship-out cost of one unit shipped from factory i.

Similarly, v_j is the cost of shipping one unit **to** location j.

Call this the ship-in cost of one unit to location j.

Strong duality implies that total transprotation costs equals total ship-out costs **plus** total ship-in costs.

It is reasonable that, for one unit of a product, ship-out cost u_i **plus** ship-in cost v_j should equal transportation cost c_{ij}.

This equality is assured by **complementary slackness** conditions that state that whenever $x_{ij} > 0$, meaning that there are positive shipments from factory i to location j, it must be true that $u_i + v_j = c_{ij}$.

32.5 The Python Optimal Transport Package

There is an excellent Python package for optimal transport that simplifies some of the steps we took above.

In particular, the package takes care of the vectorization steps before passing the data out to a linear programming routine.

(That said, the discussion provided above on vectorization remains important, since we want to understand what happens under the hood.)

32.5.1 Replicating Previous Results

The following line of code solves the example application discussed above using linear programming.

```
X = ot.emd(p, q, C)
X
```

```
array([[15., 35.,  0.,  0.,  0.],
       [10.,  0., 60., 30.,  0.],
       [ 0., 80.,  0.,  0., 70.]])
```

Sure enough, we have the same solution and the same cost

```
total_cost = np.vdot(X, C)
total_cost
```

```
np.float64(7225.0)
```

Here we use np.vdot for the trace inner product of X and C

32.5.2 A Larger Application

Now let's try using the same package on a slightly larger application.

The application has the same interpretation as above but we will also give each node (i.e., vertex) a location in the plane.

This will allow us to plot the resulting transport plan as edges in a graph.

The following class defines a node by

- its location $(x, y) \in \mathbb{R}^2$,
- its group (factory or location, denoted by p or q) and
- its mass (e.g., p_i or q_j).

```
class Node:

    def __init__(self, x, y, mass, group, name):

        self.x, self.y = x, y
        self.mass, self.group = mass, group
        self.name = name
```

Next we write a function that repeatedly calls the class above to build instances.

It allocates to the nodes it creates their location, mass, and group.

Locations are assigned randomly.

```
def build_nodes_of_one_type(group='p', n=100, seed=123):

    nodes = []
    np.random.seed(seed)

    for i in range(n):

        if group == 'p':
            m = 1/n
            x = np.random.uniform(-2, 2)
            y = np.random.uniform(-2, 2)
        else:
            m = betabinom.pmf(i, n-1, 2, 2)
            x = 0.6 * np.random.uniform(-1.5, 1.5)
            y = 0.6 * np.random.uniform(-1.5, 1.5)

        name = group + str(i)
        nodes.append(Node(x, y, m, group, name))

    return nodes
```

Now we build two lists of nodes, each one containing one type (factories or locations)

```
n_p = 32
n_q = 32
p_list = build_nodes_of_one_type(group='p', n=n_p)
q_list = build_nodes_of_one_type(group='q', n=n_q)

p_probs = [p.mass for p in p_list]
q_probs = [q.mass for q in q_list]
```

For the cost matrix C, we use the Euclidean distance between each factory and location.

```
c = np.empty((n_p, n_q))
for i in range(n_p):
    for j in range(n_q):
        x0, y0 = p_list[i].x, p_list[i].y
        x1, y1 = q_list[j].x, q_list[j].y
        c[i, j] = np.sqrt((x0-x1)**2 + (y0-y1)**2)
```

Now we are ready to apply the solver

```
%time pi = ot.emd(p_probs, q_probs, c)
```

```
CPU times: user 628 µs, sys: 0 ns, total: 628 µs
Wall time: 396 µs
```

Finally, let's plot the results using `networkx`.

In the plot below,

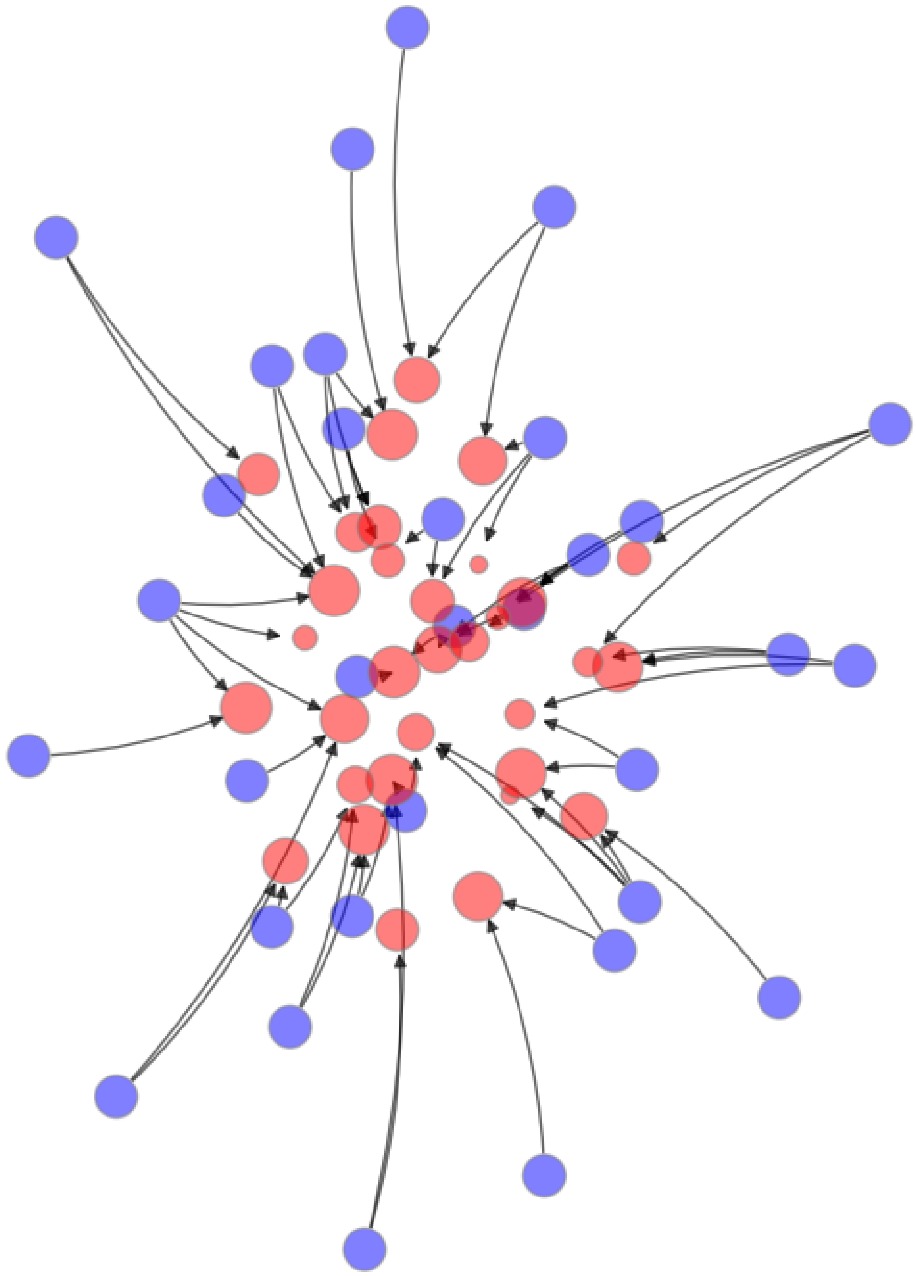
- node size is proportional to probability mass
- an edge (arrow) from i to j is drawn when a positive transfer is made from i to j under the optimal transport plan.

```
g = nx.DiGraph()
g.add_nodes_from([p.name for p in p_list])
g.add_nodes_from([q.name for q in q_list])

for i in range(n_p):
    for j in range(n_q):
        if pi[i, j] > 0:
            g.add_edge(p_list[i].name, q_list[j].name, weight=pi[i, j])

node_pos_dict={}
for p in p_list:
    node_pos_dict[p.name] = (p.x, p.y)

for q in q_list:
    node_pos_dict[q.name] = (q.x, q.y)

node_color_list = []
node_size_list = []
scale = 8_000
for p in p_list:
    node_color_list.append('blue')
    node_size_list.append(p.mass * scale)
for q in q_list:
    node_color_list.append('red')
    node_size_list.append(q.mass * scale)


fig, ax = plt.subplots(figsize=(7, 10))
plt.axis('off')

nx.draw_networkx_nodes(g,
                       node_pos_dict,
                       node_color=node_color_list,
                       node_size=node_size_list,
                       edgecolors='grey',
```

(continues on next page)

(continued from previous page)

```
                        linewidths=1,
                        alpha=0.5,
                        ax=ax)

nx.draw_networkx_edges(g,
                        node_pos_dict,
                        arrows=True,
                        connectionstyle='arc3,rad=0.1',
                        alpha=0.6)
plt.show()
```



CHAPTER

THIRTYTHREE

VON NEUMANN GROWTH MODEL (AND A GENERALIZATION)

Contents

This lecture uses the class `Neumann` to calculate key objects of a linear growth model of John von Neumann [von Neumann, 1937] that was generalized by Kemeny, Morgenstern and Thompson [Kemeny *et al.*, 1956].

Objects of interest are the maximal expansion rate (α), the interest factor (β), the optimal intensities (x), and prices (p).

In addition to watching how the towering mind of John von Neumann formulated an equilibrium model of price and quantity vectors in balanced growth, this lecture shows how fruitfully to employ the following important tools:

- a zero-sum two-player game
- linear programming
- the Perron-Frobenius theorem

We'll begin with some imports:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import fsolve, linprog
from textwrap import dedent

np.set_printoptions(precision=2)
```

The code below provides the `Neumann` class

```
class Neumann:

    """
    This class describes the Generalized von Neumann growth model as it was
    discussed in Kemeny et al. (1956, ECTA) and Gale (1960, Chapter 9.5):
```

(continues on next page)

(continued from previous page)

```
    Let:
    n ... number of goods
    m ... number of activities
    A ... input matrix is m-by-n
        a_{i,j} - amount of good j consumed by activity i
    B ... output matrix is m-by-n
        b_{i,j} - amount of good j produced by activity i

    x ... intensity vector (m-vector) with non-negative entries
        x'B - the vector of goods produced
        x'A - the vector of goods consumed
    p ... price vector (n-vector) with non-negative entries
        Bp - the revenue vector for every activity
        Ap - the cost of each activity

    Both A and B have non-negative entries. Moreover, we assume that
    (1) Assumption I (every good which is consumed is also produced):
        for all j, b_{.,j} > 0, i.e. at least one entry is strictly positive
    (2) Assumption II (no free lunch):
        for all i, a_{i,.} > 0, i.e. at least one entry is strictly positive

    Parameters
    ----------
    A : array_like or scalar(float)
        Part of the state transition equation.  It should be `n x n`
    B : array_like or scalar(float)
        Part of the state transition equation.  It should be `n x k`
    """

    def __init__(self, A, B):

        self.A, self.B = list(map(self.convert, (A, B)))
        self.m, self.n = self.A.shape

        # Check if (A, B) satisfy the basic assumptions
        assert self.A.shape == self.B.shape, 'The input and output matrices \
              must have the same dimensions!'
        assert (self.A >= 0).all() and (self.B >= 0).all(), 'The input and \
              output matrices must have only non-negative entries!'

        # (1) Check whether Assumption I is satisfied:
        if (np.sum(B, 0) <= 0).any():
            self.AI = False
        else:
            self.AI = True

        # (2) Check whether Assumption II is satisfied:
        if (np.sum(A, 1) <= 0).any():
            self.AII = False
        else:
            self.AII = True

    def __repr__(self):
        return self.__str__()

    def __str__(self):
```

(continues on next page)

(continued from previous page)

```python
        me = """
        Generalized von Neumann expanding model:
          - number of goods          : {n}
          - number of activities     : {m}

        Assumptions:
          - AI:  every column of B has a positive entry    : {AI}
          - AII: every row of A has a positive entry       : {AII}

        """
        # Irreducible                                       : {irr}
        return dedent(me.format(n=self.n, m=self.m,
                                AI=self.AI, AII=self.AII))

    def convert(self, x):
        """
        Convert array_like objects (lists of lists, floats, etc.) into
        well-formed 2D NumPy arrays
        """
        return np.atleast_2d(np.asarray(x))


    def bounds(self):
        """
        Calculate the trivial upper and lower bounds for alpha (expansion rate)
        and beta (interest factor). See the proof of Theorem 9.8 in Gale (1960)
        """

        n, m = self.n, self.m
        A, B = self.A, self.B

        f = lambda α: ((B - α * A) @ np.ones((n, 1))).max()
        g = lambda β: (np.ones((1, m)) @ (B - β * A)).min()

        UB = fsolve(f, 1).item()  # Upper bound for α, β
        LB = fsolve(g, 2).item()  # Lower bound for α, β

        return LB, UB


    def zerosum(self, γ, dual=False):
        """
        Given gamma, calculate the value and optimal strategies of a
        two-player zero-sum game given by the matrix

                M(gamma) = B - gamma * A

        Row player maximizing, column player minimizing

        Zero-sum game as an LP (primal --> α)

            max  (0', 1) @ (x', v)
            subject to
            [-M', ones(n, 1)] @ (x', v)' <= 0
            (x', v) @ (ones(m, 1), 0) = 1
            (x', v) >= (0', -inf)
```

(continues on next page)

(continued from previous page)

```
    Zero-sum game as an LP (dual --> beta)

        min (0', 1) @ (p', u)
        subject to
        [M, -ones(m, 1)] @ (p', u)' <= 0
        (p', u) @ (ones(n, 1), 0) = 1
        (p', u) >= (0', -inf)

    Outputs:
    --------
    value: scalar
        value of the zero-sum game

    strategy: vector
        if dual = False, it is the intensity vector,
        if dual = True, it is the price vector
    """

    A, B, n, m = self.A, self.B, self.n, self.m
    M = B - γ * A

    if dual == False:
        # Solve the primal LP (for details see the description)
        # (1) Define the problem for v as a maximization (linprog minimizes)
        c = np.hstack([np.zeros(m), -1])

        # (2) Add constraints :
        # ... non-negativity constraints
        bounds = tuple(m * [(0, None)] + [(None, None)])
        # ... inequality constraints
        A_iq = np.hstack([-M.T, np.ones((n, 1))])
        b_iq = np.zeros((n, 1))
        # ... normalization
        A_eq = np.hstack([np.ones(m), 0]).reshape(1, m + 1)
        b_eq = 1

        res = linprog(c, A_ub=A_iq, b_ub=b_iq, A_eq=A_eq, b_eq=b_eq,
                      bounds=bounds)

    else:
        # Solve the dual LP (for details see the description)
        # (1) Define the problem for v as a maximization (linprog minimizes)
        c = np.hstack([np.zeros(n), 1])

        # (2) Add constraints :
        # ... non-negativity constraints
        bounds = tuple(n * [(0, None)] + [(None, None)])
        # ... inequality constraints
        A_iq = np.hstack([M, -np.ones((m, 1))])
        b_iq = np.zeros((m, 1))
        # ... normalization
        A_eq = np.hstack([np.ones(n), 0]).reshape(1, n + 1)
        b_eq = 1

        res = linprog(c, A_ub=A_iq, b_ub=b_iq, A_eq=A_eq, b_eq=b_eq,
                      bounds=bounds)
```

(continues on next page)

(continued from previous page)

```
        if res.status != 0:
            print(res.message)

        # Pull out the required quantities
        value = res.x[-1]
        strategy = res.x[:-1]

        return value, strategy


    def expansion(self, tol=1e-8, maxit=1000):
        """
        The algorithm used here is described in Hamburger-Thompson-Weil
        (1967, ECTA). It is based on a simple bisection argument and utilizes
        the idea that for a given γ (= α or β), the matrix "M = B - γ * A"
        defines a two-player zero-sum game, where the optimal strategies are
        the (normalized) intensity and price vector.

        Outputs:
        --------
        alpha: scalar
            optimal expansion rate
        """

        LB, UB = self.bounds()

        for iter in range(maxit):

            γ = (LB + UB) / 2
            ZS = self.zerosum(γ=γ)
            V = ZS[0]     # value of the game with γ

            if V >= 0:
                LB = γ
            else:
                UB = γ

            if abs(UB - LB) < tol:
                γ = (UB + LB) / 2
                x = self.zerosum(γ=γ)[1]
                p = self.zerosum(γ=γ, dual=True)[1]
                break

        return γ, x, p

    def interest(self, tol=1e-8, maxit=1000):
        """
        The algorithm used here is described in Hamburger-Thompson-Weil
        (1967, ECTA). It is based on a simple bisection argument and utilizes
        the idea that for a given gamma (= alpha or beta),
        the matrix "M = B - γ * A" defines a two-player zero-sum game,
        where the optimal strategies are the (normalized) intensity and price
        vector

        Outputs:
        --------
        beta: scalar
```

(continues on next page)

(continued from previous page)

```
            optimal interest rate
        """

        LB, UB = self.bounds()

        for iter in range(maxit):
            γ = (LB + UB) / 2
            ZS = self.zerosum(γ=γ, dual=True)
            V = ZS[0]

            if V > 0:
                LB = γ
            else:
                UB = γ

            if abs(UB - LB) < tol:
                γ = (UB + LB) / 2
                p = self.zerosum(γ=γ, dual=True)[1]
                x = self.zerosum(γ=γ)[1]
                break

        return γ, x, p
```

33.1 Notation

We use the following notation.

$\mathbf{0}$ denotes a vector of zeros.

We call an n-vector **positive** and write $x \gg \mathbf{0}$ if $x_i > 0$ for all $i = 1, 2, \dots, n$.

We call a vector **non-negative** and write $x \geq \mathbf{0}$ if $x_i \geq 0$ for all $i = 1, 2, \dots, n$.

We call a vector **semi-positive** written $x > \mathbf{0}$ if $x \geq \mathbf{0}$ and $x \neq \mathbf{0}$.

For two conformable vectors x and y, $x \gg y$, $x \geq y$ and $x > y$ mean $x - y \gg \mathbf{0}$, $x - y \geq \mathbf{0}$, and $x - y > \mathbf{0}$, respectively.

We let all vectors in this lecture be column vectors; $x^\top$ denotes the transpose of x (i.e., a row vector).

Let ι_n denote a column vector composed of n ones, i.e. $\iota_n = (1, 1, \dots, 1)^\top$.

Let e^i denote a vector (of arbitrary size) containing zeros except for the i th position where it is one.

We denote matrices by capital letters. For an arbitrary matrix A, $a_{i,j}$ represents the entry in its i th row and j th column.

$a_{\cdot j}$ and $a_{i \cdot}$ denote the j th column and i th row of A, respectively.

33.2 Model Ingredients and Assumptions

A pair (A, B) of $m \times n$ non-negative matrices defines an economy.

- m is the number of **activities** (or sectors)
- n is the number of **goods** (produced and/or consumed)
- A is called the **input matrix**; $a_{i,j}$ denotes the amount of good j consumed by activity i
- B is called the **output matrix**; $b_{i,j}$ represents the amount of good j produced by activity i

Two key assumptions restrict economy (A, B):

Assumption 33.2.1 (every good that is consumed is also produced)

$$b_{.,j} > \mathbf{0} \qquad \forall j = 1, 2, \dots, n$$

Assumption 33.2.2 (no free lunch)

$$a_{i,.} > \mathbf{0} \qquad \forall i = 1, 2, \dots, m$$

A semi-positive **intensity** m-vector x denotes levels at which activities are operated.

Therefore,

- vector $x^\top A$ gives the total amount of **goods used in production**
- vector $x^\top B$ gives **total outputs**

An economy (A, B) is said to be **productive**, if there exists a non-negative intensity vector $x \geq 0$ such that $x^\top B > x^\top A$.

The semi-positive n-vector p contains prices assigned to the n goods.

The p vector implies **cost** and **revenue** vectors

- the vector Ap tells **costs** of the vector of activities
- the vector Bp tells **revenues** from the vector of activities

Satisfaction of a property of an input-output pair (A, B) called **irreducibility** (or indecomposability) determines whether an economy can be decomposed into multiple "sub-economies".

Definition 33.2.1

For an economy (A, B), the set of goods $S \subset \{1, 2, \dots, n\}$ is called an **independent subset** if it is possible to produce every good in S without consuming goods from outside S.

Formally, the set S is independent if $\exists T \subset \{1, 2, \dots, m\}$ (a subset of activities) such that $a_{i,j} = 0$, $\forall i \in T$ and $j \in S^c$ and for all $j \in S$, $\exists i \in T$ for which $b_{i,j} > 0$.

The economy is **irreducible** if there are no proper independent subsets.

We study two examples, both in Chapter 9.6 of Gale [Gale, 1989]

```
# (1) Irreducible (A, B) example: α_0 = β_0
A1 = np.array([[0, 1, 0, 0],
               [1, 0, 0, 1],
               [0, 0, 1, 0]])

B1 = np.array([[1, 0, 0, 0],
               [0, 0, 2, 0],
               [0, 1, 0, 1]])

# (2) Reducible (A, B) example: β_0 < α_0
A2 = np.array([[0, 1, 0, 0, 0, 0],
               [1, 0, 1, 0, 0, 0],
               [0, 0, 0, 1, 0, 0],
               [0, 0, 1, 0, 0, 1],
               [0, 0, 0, 0, 1, 0]])

B2 = np.array([[1, 0, 0, 1, 0, 0],
               [0, 1, 0, 0, 0, 0],
               [0, 0, 1, 0, 0, 0],
               [0, 0, 0, 0, 2, 0],
               [0, 0, 0, 1, 0, 1]])
```

The following code sets up our first Neumann economy or `Neumann` instance

```
n1 = Neumann(A1, B1)
n1
```

```
Generalized von Neumann expanding model:
  - number of goods          : 4
  - number of activities     : 3

Assumptions:
  - AI:  every column of B has a positive entry    : True
  - AII: every row of A has a positive entry       : True
```

Here is a second instance of a Neumann economy

```
n2 = Neumann(A2, B2)
n2
```

```
Generalized von Neumann expanding model:
  - number of goods          : 6
  - number of activities     : 5

Assumptions:
  - AI:  every column of B has a positive entry    : True
  - AII: every row of A has a positive entry       : True
```

33.3 Dynamic Interpretation

Attach a time index t to the preceding objects, regard an economy as a dynamic system, and study sequences

$$\{(A_t, B_t)\}_{t\geq 0}, \qquad \{x_t\}_{t\geq 0}, \qquad \{p_t\}_{t\geq 0}$$

An interesting special case holds the technology process constant and investigates the dynamics of quantities and prices only.

Accordingly, in the rest of this lecture, we assume that $(A_t, B_t) = (A, B)$ for all $t \geq 0$.

A crucial element of the dynamic interpretation involves the timing of production.

We assume that production (consumption of inputs) takes place in period t, while the consequent output materializes in period $t+1$, i.e., consumption of $x_t^\top A$ in period t results in $x_t^\top B$ amounts of output in period $t+1$.

These timing conventions imply the following feasibility condition:

$$x_t^\top B \geq x_{t+1}^\top A \qquad \forall t \geq 1$$

which asserts that no more goods can be used today than were produced yesterday.

Accordingly, Ap_t tells the costs of production in period t and Bp_t tells revenues in period $t+1$.

33.3.1 Balanced Growth

We follow John von Neumann in studying "balanced growth".

Let $./$ denote an elementwise division of one vector by another and let $\alpha > 0$ be a scalar.

Then **balanced growth** is a situation in which

$$x_{t+1}./x_t = \alpha, \quad \forall t \geq 0$$

With balanced growth, the law of motion of x is evidently $x_{t+1} = \alpha x_t$ and so we can rewrite the feasibility constraint as

$$x_t^\top B \geq \alpha x_t^\top A \qquad \forall t$$

In the same spirit, define $\beta \in \mathbb{R}$ as the **interest factor** per unit of time.

We assume that it is always possible to earn a gross return equal to the constant interest factor β by investing "outside the model".

Under this assumption about outside investment opportunities, a no-arbitrage condition gives rise to the following (no profit) restriction on the price sequence:

$$\beta A p_t \geq B p_t \qquad \forall t$$

This says that production cannot yield a return greater than that offered by the outside investment opportunity (here we compare values in period $t+1$).

The balanced growth assumption allows us to drop time subscripts and conduct an analysis purely in terms of a time-invariant growth rate α and interest factor β.

33.4 Duality

Two problems are connected by a remarkable dual relationship between technological and valuation characteristics of the economy:

Definition 33.4.1

The **technological expansion problem** (TEP) for the economy (A, B) is to find a semi-positive m-vector $x > 0$ and a number $\alpha \in \mathbb{R}$ that satisfy

$$\begin{aligned} \max_{\alpha} \quad & \alpha \\ \text{s.t.} \quad & x^\top B \geq \alpha x^\top A \end{aligned}$$

Theorem 9.3 of David Gale's book [Gale, 1989] asserts that if *Assumption 33.2.1* and *Assumption 33.2.2* are both satisfied, then a maximum value of α exists and that it is positive.

The maximal value is called the *technological expansion rate* and is denoted by α_0. The associated intensity vector x_0 is the *optimal intensity vector*.

Definition 33.4.2

The economic expansion problem (EEP) for (A, B) is to find a semi-positive n-vector $p > 0$ and a number $\beta \in \mathbb{R}$ that satisfy

$$\begin{aligned} \min_{\beta} \quad & \beta \\ \text{s.t.} \quad & Bp \leq \beta A p \end{aligned}$$

Assumption 33.2.1 and *Assumption 33.2.2* imply existence of a minimum value $\beta_0 > 0$ called the *economic expansion rate*.

The corresponding price vector p_0 is the *optimal price vector*.

Because the criterion functions in the *technological expansion* problem and the *economical expansion problem* are both linearly homogeneous, the optimality of x_0 and p_0 are defined only up to a positive scale factor.

For convenience (and to emphasize a close connection to zero-sum games), we normalize both vectors x_0 and p_0 to have unit length.

A standard duality argument (see Lemma 9.4. in (Gale, 1960) [Gale, 1989]) implies that under *Assumption 33.2.1* and *Assumption 33.2.2*, $\beta_0 \leq \alpha_0$.

But to deduce that $\beta_0 \geq \alpha_0$, *Assumption 33.2.1* and *Assumption 33.2.2* are not sufficient.

Therefore, von Neumann [von Neumann, 1937] went on to prove the following remarkable "duality" result that connects TEP and EEP.

Theorem 33.4.1 (von Neumann)

If the economy (A, B) satisfies *Assumption 33.2.1* and *Assumption 33.2.2*, then there exist (γ^*, x_0, p_0), where $\gamma^* \in [\beta_0, \alpha_0] \subset \mathbb{R}$, $x_0 > 0$ is an m-vector, $p_0 > 0$ is an n-vector, and the following arbitrage conditions hold

$$\begin{aligned} x_0^\top B &\geq \gamma^* x_0^\top A \\ Bp_0 &\leq \gamma^* A p_0 \\ x_0^\top \left(B - \gamma^* A\right) p_0 &= 0 \end{aligned}$$

Proof. (Sketch)

Assumption 33.2.1 and *Assumption 33.2.2* imply that there exist (α_0, x_0) and (β_0, p_0) that solve the TEP and EEP, respectively.

If $\gamma^* > \alpha_0$, then by definition of α_0, there cannot exist a semi-positive x that satisfies $x^\top B \geq \gamma^* x^\top A$.

Similarly, if $\gamma^* < \beta_0$, there is no semi-positive p for which $Bp \leq \gamma^* Ap$. Let $\gamma^* \in [\beta_0, \alpha_0]$, then $x_0^\top B \geq \alpha_0 x_0^\top A \geq \gamma^* x_0^\top A$.

Moreover, $Bp_0 \leq \beta_0 A p_0 \leq \gamma^* A p_0$. These two inequalities imply $x_0 \left(B - \gamma^* A\right) p_0 = 0$.

Here the constant γ^* is both an expansion factor and an interest factor (not necessarily optimal).

We have already encountered and discussed the first two inequalities that represent feasibility and no-profit conditions.

Moreover, the equality $x_0^\top \left(B - \gamma^* A\right) p_0 = 0$ concisely expresses the requirements that if any good grows at a rate larger than γ^* (i.e., if it is *oversupplied*), then its price must be zero; and that if any activity provides negative profit, it must be unused.

Therefore, the conditions stated in *Theorem 33.4.1* encode all equilibrium conditions.

So *Theorem 33.4.1* essentially states that under *Assumption 33.2.1* and *Assumption 33.2.2* there always exists an equilibrium (γ^*, x_0, p_0) with balanced growth.

Note that *Theorem 33.4.1* is silent about uniqueness of the equilibrium. In fact, it does not rule out (trivial) cases with $x_0^\top B p_0 = 0$ so that nothing of value is produced.

To exclude such uninteresting cases, Kemeny, Morgenstern and Thompson [Kemeny *et al.*, 1956] add an extra requirement

$$x_0^\top B p_0 > 0$$

and call the associated equilibria *economic solutions*.

They show that this extra condition does not affect the existence result, while it significantly reduces the number of (relevant) solutions.

33.5 Interpretation as Two-player Zero-sum Game

To compute the equilibrium (γ^*, x_0, p_0), we follow the algorithm proposed by Hamburger, Thompson and Weil (1967), building on the key insight that an equilibrium (with balanced growth) can be solved as a particular two-player zero-sum game.

First, we introduce some notation.

Consider the $m \times n$ matrix C as a payoff matrix, with the entries representing payoffs from the **minimizing** column player to the **maximizing** row player and assume that the players can use mixed strategies. Thus,

- the row player chooses the m-vector $x > \mathbf{0}$ subject to $\iota_m^\top x = 1$
- the column player chooses the n-vector $p > \mathbf{0}$ subject to $\iota_n^\top p = 1$.

Definition 33.5.1

The $m \times n$ matrix game C has the *solution* $(x^*, p^*, V(C))$ in mixed strategies if

$$(x^*)^\top C e^j \geq V(C) \quad \forall j \in \{1, \dots, n\} \qquad \text{and} \qquad (e^i)^\top C p^* \leq V(C) \quad \forall i \in \{1, \dots, m\}$$

The number $V(C)$ is called the **value** of the game.

From the above definition, it is clear that the value $V(C)$ has two alternative interpretations:

- by playing the appropriate mixed strategy, the maximizing player can assure himself at least $V(C)$ (no matter what the column player chooses)
- by playing the appropriate mixed strategy, the minimizing player can make sure that the maximizing player will not get more than $V(C)$ (irrespective of what is the maximizing player's choice)

A famous theorem of Nash (1951) tells us that there always exists a mixed strategy Nash equilibrium for any *finite* two-player zero-sum game.

Moreover, von Neumann's Minmax Theorem [von Neumann, 1928] implies that

$$V(C) = \max_x \min_p \quad x^\top C p = \min_p \max_x \quad x^\top C p = (x^*)^\top C p^*$$

33.5.1 Connection with Linear Programming (LP)

Nash equilibria of a finite two-player zero-sum game solve a linear programming problem.

To see this, we introduce the following notation

- For a fixed x, let v be the value of the minimization problem: $v \equiv \min_p x^\top C p = \min_j x^\top C e^j$
- For a fixed p, let u be the value of the maximization problem: $u \equiv \max_x x^\top C p = \max_i (e^i)^\top C p$

Then the *max-min problem* (the game from the maximizing player's point of view) can be written as the *primal* LP

$$\begin{aligned} V(C) = \max \quad & v \\ \text{s.t.} \quad v \iota_n^\top & \leq x^\top C \\ x & \geq \mathbf{0} \\ \iota_n^\top x & = 1 \end{aligned}$$

while the *min-max problem* (the game from the minimizing player's point of view) is the *dual* LP

$$\begin{aligned} V(C) = \min \quad & u \\ \text{s.t.} \quad u \iota_m & \geq C p \\ p & \geq \mathbf{0} \\ \iota_m^\top p & = 1 \end{aligned}$$

Hamburger, Thompson and Weil [Hamburger *et al.*, 1967] view the input-output pair of the economy as payoff matrices of two-player zero-sum games.

Using this interpretation, they restate *Assumption 33.2.1* and *Assumption 33.2.2* as follows

$$V(-A) < 0 \qquad \text{and} \qquad V(B) > 0$$

Proof. (Sketch)

- $\Rightarrow V(B) > 0$ implies $x_0^\top B \gg \mathbf{0}$, where x_0 is a maximizing vector. Since B is non-negative, this requires that each column of B has at least one positive entry, which is *Assumption 33.2.1*.
- $\Leftarrow$ From *Assumption 33.2.1* and the fact that $p > \mathbf{0}$, it follows that $Bp > \mathbf{0}$. This implies that the maximizing player can always choose x so that $x^\top Bp > 0$ so that it must be the case that $V(B) > 0$.

In order to (re)state *Theorem 33.4.1* in terms of a particular two-player zero-sum game, we define a matrix for $\gamma \in \mathbb{R}$

$$M(\gamma) \equiv B - \gamma A$$

For fixed γ, treating $M(\gamma)$ as a matrix game, calculating the solution of the game implies

- If $\gamma > \alpha_0$, then for all $x > 0$, there $\exists j \in \{1, \dots, n\}$, s.t. $[x^\top M(\gamma)]_j < 0$ implying that $V(M(\gamma)) < 0$.
- If $\gamma < \beta_0$, then for all $p > 0$, there $\exists i \in \{1, \dots, m\}$, s.t. $[M(\gamma)p]_i > 0$ implying that $V(M(\gamma)) > 0$.
- If $\gamma \in \{\beta_0, \alpha_0\}$, then (by *Theorem 33.4.1*) the optimal intensity and price vectors x_0 and p_0 satisfy

$$x_0^\top M(\gamma) \geq \mathbf{0}^\top \quad \text{and} \quad M(\gamma)p_0 \leq \mathbf{0}$$

That is, $(x_0, p_0, 0)$ is a solution of the game $M(\gamma)$ so that $V(M(\beta_0)) = V(M(\alpha_0)) = 0$.

- If $\beta_0 < \alpha_0$ and $\gamma \in (\beta_0, \alpha_0)$, then $V(M(\gamma)) = 0$.

Moreover, if x' is optimal for the maximizing player in $M(\gamma')$ for $\gamma' \in (\beta_0, \alpha_0)$ and p'' is optimal for the minimizing player in $M(\gamma'')$ where $\gamma'' \in (\beta_0, \gamma')$, then $(x', p'', 0)$ is a solution for $M(\gamma)$ $\forall \gamma \in (\gamma'', \gamma')$.

Proof. (Sketch) If x' is optimal for a maximizing player in game $M(\gamma')$, then $(x')^\top M(\gamma') \geq \mathbf{0}^\top$ and so for all $\gamma < \gamma'$.

$$(x')^\top M(\gamma) = (x')^\top M(\gamma') + (x')^\top(\gamma' - \gamma)A \geq \mathbf{0}^\top$$

hence $V(M(\gamma)) \geq 0$. If p'' is optimal for a minimizing player in game $M(\gamma'')$, then $M(\gamma'')p'' \leq \mathbf{0}$ and so for all $\gamma'' < \gamma$

$$M(\gamma)p'' = M(\gamma'') + (\gamma'' - \gamma)Ap'' \leq \mathbf{0}$$

hence $V(M(\gamma)) \leq 0$.

It is clear from the above argument that β_0, α_0 are the minimal and maximal γ for which $V(M(\gamma)) = 0$.

Furthermore, Hamburger et al. [Hamburger *et al.*, 1967] show that the function $\gamma \mapsto V(M(\gamma))$ is continuous and nonincreasing in γ.

This suggests an algorithm to compute (α_0, x_0) and (β_0, p_0) for a given input-output pair (A, B).

33.5.2 Algorithm

Hamburger, Thompson and Weil [Hamburger *et al.*, 1967] propose a simple bisection algorithm to find the minimal and maximal roots (i.e. β_0 and α_0) of the function $\gamma \mapsto V(M(\gamma))$.

Step 1

First, notice that we can easily find trivial upper and lower bounds for α_0 and β_0.

- TEP requires that $x^\top(B - \alpha A) \geq \mathbf{0}^\top$ and $x > \mathbf{0}$, so if α is so large that $\max_i\{[(B - \alpha A)\iota_n]_i\} < 0$, then TEP ceases to have a solution.

Accordingly, let **UB** be the α^* that solves $\max_i\{[(B - \alpha^* A)\iota_n]_i\} = 0$.

- Similar to the upper bound, if β is so low that $\min_j\{[\iota_m^\top(B - \beta A)]_j\} > 0$, then the EEP has no solution and so we can define **LB** as the β^* that solves $\min_j\{[\iota_m^\top(B - \beta^* A)]_j\} = 0$.

The *bounds* method calculates these trivial bounds for us

```
n1.bounds()
```

```
(1.0, 2.0)
```

Step 2

Compute α_0 and β_0

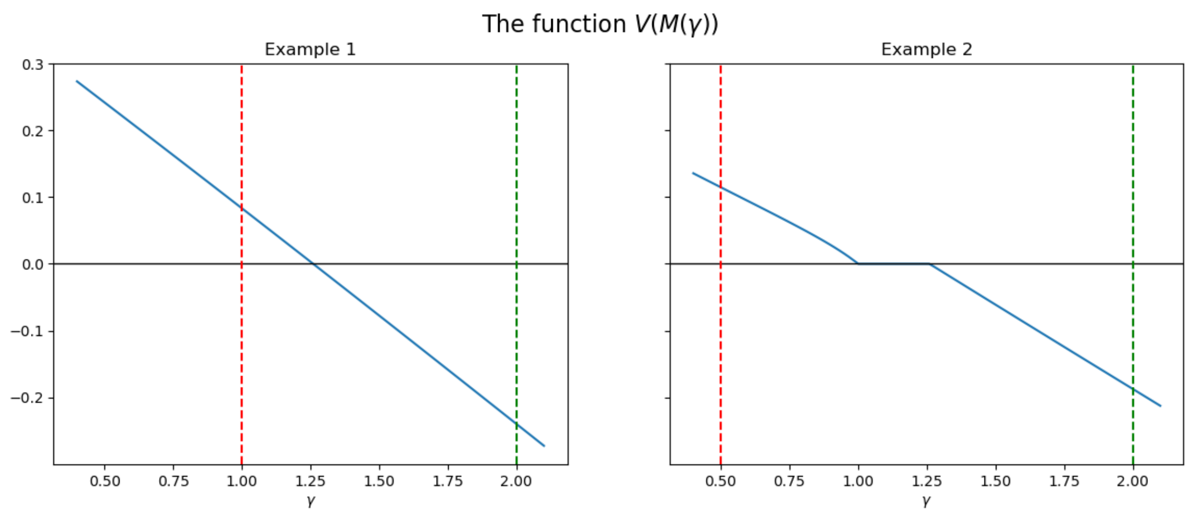
- Finding α_0
  1. Fix $\gamma = \frac{UB+LB}{2}$ and compute the solution of the two-player zero-sum game associated with $M(\gamma)$. We can use either the primal or the dual LP problem.
  2. If $V(M(\gamma)) \geq 0$, then set $LB = \gamma$, otherwise let $UB = \gamma$.
  3. Iterate on 1. and 2. until $|UB - LB| < \epsilon$.
- Finding β_0
  1. Fix $\gamma = \frac{UB+LB}{2}$ and compute the solution of the two-player zero-sum game associated with $M(\gamma)$. We can use either the primal or the dual LP problem.
  2. If $V(M(\gamma)) > 0$, then set $LB = \gamma$, otherwise let $UB = \gamma$.
  3. Iterate on 1. and 2. until $|UB - LB| < \epsilon$.
- *Existence*: Since $V(M(LB)) > 0$ and $V(M(UB)) < 0$ and $V(M(\cdot))$ is a continuous, nonincreasing function, there is at least one $\gamma \in [LB, UB]$, s.t. $V(M(\gamma)) = 0$.

The *zerosum* method calculates the value and optimal strategies associated with a given γ.

```
γ = 2

print(f'Value of the game with γ = {γ}')
print(n1.zerosum(γ=γ)[0])
print('Intensity vector (from the primal)')
print(n1.zerosum(γ=γ)[1])
print('Price vector (from the dual)')
print(n1.zerosum(γ=γ, dual=True)[1])
```

```
Value of the game with γ = 2
-0.24
Intensity vector (from the primal)
[0.32 0.28 0.4 ]
Price vector (from the dual)
[0.4  0.32 0.28 0.  ]
```

```
numb_grid = 100
γ_grid = np.linspace(0.4, 2.1, numb_grid)

value_ex1_grid = np.asarray([n1.zerosum(γ=γ_grid[i])[0]
                             for i in range(numb_grid)])
value_ex2_grid = np.asarray([n2.zerosum(γ=γ_grid[i])[0]
                             for i in range(numb_grid)])

fig, axes = plt.subplots(1, 2, figsize=(14, 5), sharey=True)
fig.suptitle(r'The function $V(M(\gamma))$', fontsize=16)

for ax, grid, N, i in zip(axes, (value_ex1_grid, value_ex2_grid),
                          (n1, n2), (1, 2)):
    ax.plot(γ_grid, grid)
    ax.set(title=f'Example {i}', xlabel=r'$\gamma$')
    ax.axhline(0, c='k', lw=1)
    ax.axvline(N.bounds()[0], c='r', ls='--', label='lower bound')
    ax.axvline(N.bounds()[1], c='g', ls='--', label='upper bound')

plt.show()
```



The *expansion* method implements the bisection algorithm for α_0 (and uses the primal LP problem for x_0)

```
α_0, x, p = n1.expansion()
print(f'α_0 = {α_0}')
print(f'x_0 = {x}')
print(f'The corresponding p from the dual = {p}')
```

```
α_0 = 1.2599210478365421
x_0 = [0.33 0.26 0.41]
The corresponding p from the dual = [0.41 0.33 0.26 0.  ]
```

The *interest* method implements the bisection algorithm for β_0 (and uses the dual LP problem for p_0)

```
β_0, x, p = n1.interest()
print(f'β_0 = {β_0}')
print(f'p_0 = {p}')
print(f'The corresponding x from the primal = {x}')
```

```
β_0 = 1.2599210478365421
p_0 = [0.41 0.33 0.26 0.  ]
The corresponding x from the primal = [0.33 0.26 0.41]
```

Of course, when γ^* is unique, it is irrelevant which one of the two methods we use – both work.

In particular, as will be shown below, in case of an irreducible (A, B) (like in Example 1), the maximal and minimal roots of $V(M(\gamma))$ necessarily coincide implying a "full duality" result, i.e. $\alpha_0 = \beta_0 = \gamma^*$ so that the expansion (and interest) rate γ^* is unique.

33.5.3 Uniqueness and Irreducibility

As an illustration, compute first the maximal and minimal roots of $V(M(\cdot))$ for our Example 2 that has a reducible input-output pair (A, B)

```
α_0, x, p = n2.expansion()
print(f'α_0 = {α_0}')
print(f'x_0 = {x}')
print(f'The corresponding p from the dual = {p}')
```

```
α_0 = 1.259921052493155
x_0 = [5.27e-10 0.00e+00 3.27e-01 2.60e-01 4.13e-01]
The corresponding p from the dual = [0.   0.21 0.33 0.26 0.21 0.  ]
```

```
β_0, x, p = n2.interest()
print(f'β_0 = {β_0}')
print(f'p_0 = {p}')
print(f'The corresponding x from the primal = {x}')
```

```
β_0 = 1.0000000009313226
p_0 = [ 5.00e-01  5.00e-01 -1.55e-09 -1.24e-09 -9.31e-10  0.00e+00]
The corresponding x from the primal = [-0.    0.    0.25  0.25  0.5 ]
```

As we can see, with a reducible (A, B), the roots found by the bisection algorithms might differ, so there might be multiple γ^* that make the value of the game with $M(\gamma^*)$ zero. (see the figure above).

Indeed, although the von Neumann theorem assures existence of the equilibrium, *Assumption 33.2.1* and *Assumption 33.2.2* are not sufficient for uniqueness. Nonetheless, Kemeny et al. (1967) show that there are at most finitely many economic solutions, meaning that there are only finitely many γ^* that satisfy $V(M(\gamma^*)) = 0$ and $x_0^\top B p_0 > 0$ and that for each such γ_i^*, there is a self-contained part of the economy (a sub-economy) that in equilibrium can expand independently with the expansion coefficient γ_i^*.

The following theorem (see Theorem 9.10. in Gale [Gale, 1989]) asserts that imposing irreducibility is sufficient for uniqueness of (γ^*, x_0, p_0).

Theorem 33.5.1

Adopt the conditions of *Theorem 33.4.1*. If the economy (A, B) is irreducible, then $\gamma^* = \alpha_0 = \beta_0$.

33.5.4 A Special Case

There is a special (A, B) that allows us to simplify the solution method significantly by invoking the powerful Perron-Frobenius theorem for non-negative matrices.

Definition 33.5.2

We call an economy *simple* if it satisfies

- $n = m$
- Each activity produces exactly one good
- Each good is produced by one and only one activity.

These assumptions imply that $B = I_n$, i.e., that B can be written as an identity matrix (possibly after reshuffling its rows and columns).

The simple model has the following special property (Theorem 9.11. in Gale [Gale, 1989]): if x_0 and $\alpha_0 > 0$ solve the TEP with (A, I_n), then

$$x_0^\top = \alpha_0 x_0^\top A \qquad \Leftrightarrow \qquad x_0^\top A = \left(\frac{1}{\alpha_0}\right) x_0^\top$$

The latter shows that $1/\alpha_0$ is a positive eigenvalue of A and x_0 is the corresponding non-negative left eigenvector.

The classic result of *Perron and Frobenius* implies that a non-negative matrix has a non-negative eigenvalue-eigenvector pair.

Moreover, if A is irreducible, then the optimal intensity vector x_0 is positive and *unique* up to multiplication by a positive scalar.

Suppose that A is reducible with k irreducible subsets $S_1, \dots, S_k$. Let A_i be the submatrix corresponding to S_i and let α_i and β_i be the associated expansion and interest factors, respectively. Then we have

$$\alpha_0 = \max_i \{\alpha_i\} \qquad \text{and} \qquad \beta_0 = \min_i \{\beta_i\}$$

Part VI

Introduction to Dynamics

CHAPTER

THIRTYFOUR

FINITE MARKOV CHAINS

Contents

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
```

34.1 Overview

Markov chains are one of the most useful classes of stochastic processes, being

- simple, flexible and supported by many elegant theoretical results
- valuable for building intuition about random dynamic models
- central to quantitative modeling in their own right

You will find them in many of the workhorse models of economics and finance.

In this lecture, we review some of the theory of Markov chains.

We will also introduce some of the high-quality routines for working with Markov chains available in QuantEcon.py.

Prerequisite knowledge is basic probability and linear algebra.

Let's start with some standard imports:

```
import matplotlib.pyplot as plt
import quantecon as qe
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
```

34.2 Definitions

The following concepts are fundamental.

34.2.1 Stochastic Matrices

A **stochastic matrix** (or **Markov matrix**) is an $n \times n$ square matrix P such that

1. each element of P is nonnegative, and
2. each row of P sums to one

Each row of P can be regarded as a probability mass function over n possible outcomes.

It is too not difficult to check[1] that if P is a stochastic matrix, then so is the k-th power P^k for all $k \in \mathbb{N}$.

34.2.2 Markov Chains

There is a close connection between stochastic matrices and Markov chains.

To begin, let S be a finite set with n elements $\{x_1, \ldots, x_n\}$.

The set S is called the **state space** and $x_1, \ldots, x_n$ are the **state values**.

A **Markov chain** $\{X_t\}$ on S is a sequence of random variables on S that have the **Markov property**.

This means that, for any date t and any state $y \in S$,

$$\mathbb{P}\{X_{t+1} = y \,|\, X_t\} = \mathbb{P}\{X_{t+1} = y \,|\, X_t, X_{t-1}, \ldots\} \tag{34.1}$$

In other words, knowing the current state is enough to know probabilities for future states.

In particular, the dynamics of a Markov chain are fully determined by the set of values

$$P(x, y) := \mathbb{P}\{X_{t+1} = y \,|\, X_t = x\} \qquad (x, y \in S) \tag{34.2}$$

By construction,

- $P(x, y)$ is the probability of going from x to y in one unit of time (one step)
- $P(x, \cdot)$ is the conditional distribution of X_{t+1} given $X_t = x$

We can view P as a stochastic matrix where

$$P_{ij} = P(x_i, x_j) \qquad 1 \leq i, j \leq n$$

Going the other way, if we take a stochastic matrix P, we can generate a Markov chain $\{X_t\}$ as follows:

- draw X_0 from a marginal distribution ψ
- for each $t = 0, 1, \ldots$, draw X_{t+1} from $P(X_t, \cdot)$

By construction, the resulting process satisfies (34.2).

[1] Hint: First show that if P and Q are stochastic matrices then so is their product — to check the row sums, try post multiplying by a column vector of ones. Finally, argue that P^n is a stochastic matrix using induction.

34.2.3 Example 1

Consider a worker who, at any given time t, is either unemployed (state 0) or employed (state 1).

Suppose that, over a one month period,

1. An unemployed worker finds a job with probability $\alpha \in (0, 1)$.
2. An employed worker loses her job and becomes unemployed with probability $\beta \in (0, 1)$.

In terms of a Markov model, we have

- $S = \{0, 1\}$
- $P(0, 1) = \alpha$ and $P(1, 0) = \beta$

We can write out the transition probabilities in matrix form as

$$P = \begin{pmatrix} 1-\alpha & \alpha \\ \beta & 1-\beta \end{pmatrix} \tag{34.3}$$

Once we have the values α and β, we can address a range of questions, such as

- What is the average duration of unemployment?
- Over the long-run, what fraction of time does a worker find herself unemployed?
- Conditional on employment, what is the probability of becoming unemployed at least once over the next 12 months?

We'll cover such applications below.

34.2.4 Example 2

From US unemployment data, Hamilton [Hamilton, 2005] estimated the stochastic matrix

$$P = \begin{pmatrix} 0.971 & 0.029 & 0 \\ 0.145 & 0.778 & 0.077 \\ 0 & 0.508 & 0.492 \end{pmatrix}$$

where

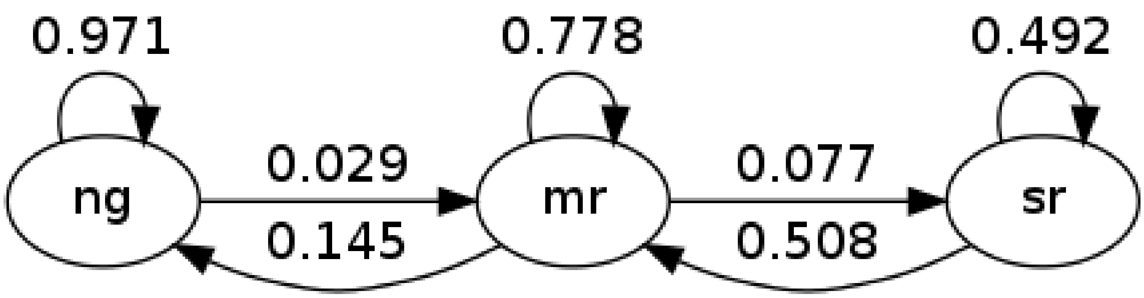
- the frequency is monthly
- the first state represents "normal growth"
- the second state represents "mild recession"
- the third state represents "severe recession"

For example, the matrix tells us that when the state is normal growth, the state will again be normal growth next month with probability 0.97.

In general, large values on the main diagonal indicate persistence in the process $\{X_t\}$.

This Markov process can also be represented as a directed graph, with edges labeled by transition probabilities

Here "ng" is normal growth, "mr" is mild recession, etc.



34.3 Simulation

One natural way to answer questions about Markov chains is to simulate them.

(To approximate the probability of event E, we can simulate many times and count the fraction of times that E occurs).

Nice functionality for simulating Markov chains exists in QuantEcon.py.

- Efficient, bundled with lots of other useful routines for handling Markov chains.

However, it's also a good exercise to roll our own routines — let's do that first and then come back to the methods in QuantEcon.py.

In these exercises, we'll take the state space to be $S = 0, \ldots, n-1$.

34.3.1 Rolling Our Own

To simulate a Markov chain, we need its stochastic matrix P and a marginal probability distribution ψ from which to draw a realization of X_0.

The Markov chain is then constructed as discussed above. To repeat:

1. At time $t = 0$, draw a realization of X_0 from ψ.
2. At each subsequent time t, draw a realization of the new state X_{t+1} from $P(X_t, \cdot)$.

To implement this simulation procedure, we need a method for generating draws from a discrete distribution.

For this task, we'll use `random.draw` from QuantEcon, which works as follows:

```
ψ = (0.3, 0.7)           # probabilities over {0, 1}
cdf = np.cumsum(ψ)       # convert into cummulative distribution
qe.random.draw(cdf, 5)   # generate 5 independent draws from ψ
```

```
array([1, 1, 1, 0, 0])
```

We'll write our code as a function that accepts the following three arguments

- A stochastic matrix `P`
- An initial state `init`
- A positive integer `sample_size` representing the length of the time series the function should return

```
def mc_sample_path(P, ψ_0=None, sample_size=1_000):

    # set up
```

(continues on next page)

(continued from previous page)

```
    P = np.asarray(P)
    X = np.empty(sample_size, dtype=int)

    # Convert each row of P into a cdf
    n = len(P)
    P_dist = [np.cumsum(P[i, :]) for i in range(n)]

    # draw initial state, defaulting to 0
    if ψ_0 is not None:
        X_0 = qe.random.draw(np.cumsum(ψ_0))
    else:
        X_0 = 0

    # simulate
    X[0] = X_0
    for t in range(sample_size - 1):
        X[t+1] = qe.random.draw(P_dist[X[t]])

    return X
```

Let's see how it works using the small matrix

```
P = [[0.4, 0.6],
     [0.2, 0.8]]
```

As we'll see later, for a long series drawn from `P`, the fraction of the sample that takes value 0 will be about 0.25.

Moreover, this is true, regardless of the initial distribution from which X_0 is drawn.

The following code illustrates this

```
X = mc_sample_path(P, ψ_0=[0.1, 0.9], sample_size=100_000)
np.mean(X == 0)
```

```
np.float64(0.25021)
```

You can try changing the initial distribution to confirm that the output is always close to 0.25, at least for the `P` matrix above.

34.3.2 Using QuantEcon's Routines

As discussed above, QuantEcon.py has routines for handling Markov chains, including simulation.

Here's an illustration using the same P as the preceding example

```
from quantecon import MarkovChain

mc = qe.MarkovChain(P)
X = mc.simulate(ts_length=1_000_000)
np.mean(X == 0)
```

```
np.float64(0.249902)
```

The QuantEcon.py routine is JIT compiled and much faster.

```
%time mc_sample_path(P, sample_size=1_000_000) # Our homemade code version
```

```
CPU times: user 1.2 s, sys: 1.98 ms, total: 1.2 s
Wall time: 1.2 s
```

```
array([0, 1, 1, ..., 1, 1, 1], shape=(1000000,))
```

```
%time mc.simulate(ts_length=1_000_000) # qe code version
```

```
CPU times: user 13.5 ms, sys: 5.03 ms, total: 18.5 ms
Wall time: 18.3 ms
```

```
array([0, 0, 1, ..., 1, 1, 1], shape=(1000000,))
```

Adding State Values and Initial Conditions

If we wish to, we can provide a specification of state values to `MarkovChain`.

These state values can be integers, floats, or even strings.

The following code illustrates

```
mc = qe.MarkovChain(P, state_values=('unemployed', 'employed'))
mc.simulate(ts_length=4, init='employed')
```

```
array(['employed', 'employed', 'employed', 'employed'], dtype='<U10')
```

```
mc.simulate(ts_length=4, init='unemployed')
```

```
array(['unemployed', 'employed', 'employed', 'employed'], dtype='<U10')
```

```
mc.simulate(ts_length=4)  # Start at randomly chosen initial state
```

```
array(['employed', 'unemployed', 'employed', 'employed'], dtype='<U10')
```

If we want to see indices rather than state values as outputs as we can use

```
mc.simulate_indices(ts_length=4)
```

```
array([0, 1, 0, 1])
```

34.4 Marginal Distributions

Suppose that

1. $\{X_t\}$ is a Markov chain with stochastic matrix P
2. the marginal distribution of X_t is known to be ψ_t

What then is the marginal distribution of X_{t+1}, or, more generally, of X_{t+m}?

To answer this, we let ψ_t be the marginal distribution of X_t for $t = 0, 1, 2, \ldots$.

Our first aim is to find ψ_{t+1} given ψ_t and P.

To begin, pick any $y \in S$.

Using the law of total probability, we can decompose the probability that $X_{t+1} = y$ as follows:

$$\mathbb{P}\{X_{t+1} = y\} = \sum_{x \in S} \mathbb{P}\{X_{t+1} = y \,|\, X_t = x\} \cdot \mathbb{P}\{X_t = x\}$$

In words, to get the probability of being at y tomorrow, we account for all ways this can happen and sum their probabilities.

Rewriting this statement in terms of marginal and conditional probabilities gives

$$\psi_{t+1}(y) = \sum_{x \in S} P(x, y) \psi_t(x)$$

There are n such equations, one for each $y \in S$.

If we think of ψ_{t+1} and ψ_t as *row vectors*, these n equations are summarized by the matrix expression

$$\psi_{t+1} = \psi_t P \tag{34.4}$$

Thus, to move a marginal distribution forward one unit of time, we postmultiply by P.

By postmultiplying m times, we move a marginal distribution forward m steps into the future.

Hence, iterating on (34.4), the expression $\psi_{t+m} = \psi_t P^m$ is also valid — here P^m is the m-th power of P.

As a special case, we see that if ψ_0 is the initial distribution from which X_0 is drawn, then $\psi_0 P^m$ is the distribution of X_m.

This is very important, so let's repeat it

$$X_0 \sim \psi_0 \quad \implies \quad X_m \sim \psi_0 P^m \tag{34.5}$$

and, more generally,

$$X_t \sim \psi_t \quad \implies \quad X_{t+m} \sim \psi_t P^m \tag{34.6}$$

34.4.1 Multiple Step Transition Probabilities

We know that the probability of transitioning from x to y in one step is $P(x, y)$.

It turns out that the probability of transitioning from x to y in m steps is $P^m(x, y)$, the (x, y)-th element of the m-th power of P.

To see why, consider again (34.6), but now with a ψ_t that puts all probability on state x so that the transition probabilities are

- 1 in the x-th position and zero elsewhere

Inserting this into (34.6), we see that, conditional on $X_t = x$, the distribution of X_{t+m} is the x-th row of P^m.

In particular

$$\mathbb{P}\{X_{t+m} = y \,|\, X_t = x\} = P^m(x, y) = (x, y)\text{-th element of } P^m$$

34.4.2 Example: Probability of Recession

Recall the stochastic matrix P for recession and growth *considered above*.

Suppose that the current state is unknown — perhaps statistics are available only at the *end* of the current month.

We guess that the probability that the economy is in state x is $\psi(x)$.

The probability of being in recession (either mild or severe) in 6 months time is given by the inner product

$$\psi P^6 \cdot \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}$$

34.4.3 Example 2: Cross-Sectional Distributions

The marginal distributions we have been studying can be viewed either as probabilities or as cross-sectional frequencies that a Law of Large Numbers leads us to anticipate for large samples.

To illustrate, recall our model of employment/unemployment dynamics for a given worker *discussed above*.

Consider a large population of workers, each of whose lifetime experience is described by the specified dynamics, with each worker's outcomes being realizations of processes that are statistically independent of all other workers' processes.

Let ψ be the current *cross-sectional* distribution over $\{0, 1\}$.

The cross-sectional distribution records fractions of workers employed and unemployed at a given moment.

- For example, $\psi(0)$ is the unemployment rate.

What will the cross-sectional distribution be in 10 periods hence?

The answer is ψP^{10}, where P is the stochastic matrix in (34.3).

This is because each worker's state evolves according to P, so ψP^{10} is a marginal distibution for a single randomly selected worker.

But when the sample is large, outcomes and probabilities are roughly equal (by an application of the Law of Large Numbers).

So for a very large (tending to infinite) population, ψP^{10} also represents fractions of workers in each state.

This is exactly the cross-sectional distribution.

34.5 Irreducibility and Aperiodicity

Irreducibility and aperiodicity are central concepts of modern Markov chain theory.

Let's see what they're about.

34.5.1 Irreducibility

Let P be a fixed stochastic matrix.

Two states x and y are said to **communicate** with each other if there exist positive integers j and k such that

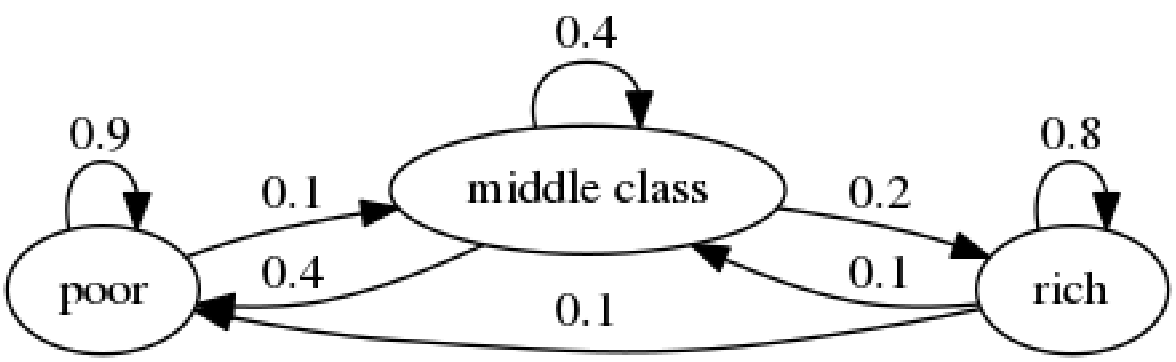
$$P^j(x,y) > 0 \quad \text{and} \quad P^k(y,x) > 0$$

In view of our discussion *above*, this means precisely that

- state x can eventually be reached from state y, and
- state y can eventually be reached from state x

The stochastic matrix P is called **irreducible** if all states communicate; that is, if x and y communicate for all (x, y) in $S \times S$.

For example, consider the following transition probabilities for wealth of a fictitious set of households



We can translate this into a stochastic matrix, putting zeros where there's no edge between nodes

$$P := \begin{pmatrix} 0.9 & 0.1 & 0 \\ 0.4 & 0.4 & 0.2 \\ 0.1 & 0.1 & 0.8 \end{pmatrix}$$

It's clear from the graph that this stochastic matrix is irreducible: we can eventually reach any state from any other state.

We can also test this using QuantEcon.py's MarkovChain class

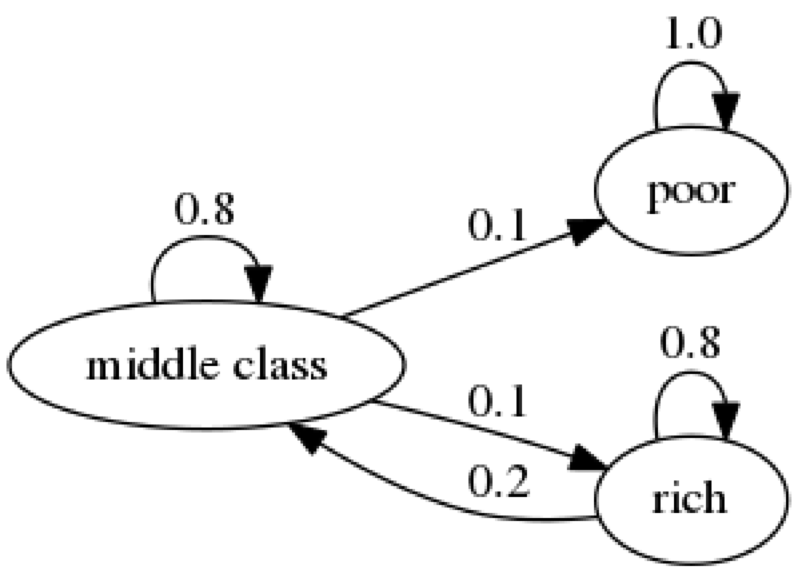
```
P = [[0.9, 0.1, 0.0],
     [0.4, 0.4, 0.2],
     [0.1, 0.1, 0.8]]

mc = qe.MarkovChain(P, ('poor', 'middle', 'rich'))
mc.is_irreducible
```

```
True
```

Here's a more pessimistic scenario in which poor people remain poor forever

This stochastic matrix is not irreducible, since, for example, rich is not accessible from poor.

Let's confirm this




```
P = [[1.0, 0.0, 0.0],
     [0.1, 0.8, 0.1],
     [0.0, 0.2, 0.8]]

mc = qe.MarkovChain(P, ('poor', 'middle', 'rich'))
mc.is_irreducible
```

```
False
```

We can also determine the "communication classes"

```
mc.communication_classes
```

```
[array(['poor'], dtype='<U6'), array(['middle', 'rich'], dtype='<U6')]
```

It might be clear to you already that irreducibility is going to be important in terms of long run outcomes.

For example, poverty is a life sentence in the second graph but not the first.

We'll come back to this a bit later.

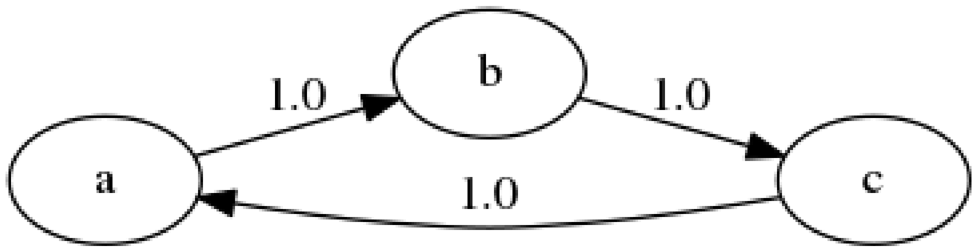
34.5.2 Aperiodicity

Loosely speaking, a Markov chain is called **periodic** if it cycles in a predictable way, and **aperiodic** otherwise.

Here's a trivial example with three states

The chain cycles with period 3:

```
P = [[0, 1, 0],
     [0, 0, 1],
     [1, 0, 0]]

mc = qe.MarkovChain(P)
mc.period
```




```
3
```

More formally, the **period** of a state x is the largest common divisor of a set of integers

$$D(x) := \{j \geq 1 : P^j(x, x) > 0\}$$

In the last example, $D(x) = \{3, 6, 9, ...\}$ for every state x, so the period is 3.

A stochastic matrix is called **aperiodic** if the period of every state is 1, and **periodic** otherwise.

For example, the stochastic matrix associated with the transition probabilities below is periodic because, for example, state a has period 2

We can confirm that the stochastic matrix is periodic with the following code

```
P = [[0.0, 1.0, 0.0, 0.0],
     [0.5, 0.0, 0.5, 0.0],
     [0.0, 0.5, 0.0, 0.5],
     [0.0, 0.0, 1.0, 0.0]]

mc = qe.MarkovChain(P)
mc.period
```

```
2
```

```
mc.is_aperiodic
```

```
False
```

34.6 Stationary Distributions

As seen in (34.4), we can shift a marginal distribution forward one unit of time via postmultiplication by P.

Some distributions are invariant under this updating process — for example,

```
P = np.array([[0.4, 0.6],
              [0.2, 0.8]])
```

(continues on next page)

(continued from previous page)

```
ψ = (0.25, 0.75)
ψ @ P
```

```
array([0.25, 0.75])
```

Such distributions are called **stationary** or **invariant**.

Formally, a marginal distribution ψ^* on S is called **stationary** for P if $\psi^* = \psi^* P$.

(This is the same notion of stationarity that we learned about in the lecture on AR(1) processes applied to a different setting.)

From this equality, we immediately get $\psi^* = \psi^* P^t$ for all t.

This tells us an important fact: If the distribution of X_0 is a stationary distribution, then X_t will have this same distribution for all t.

Hence stationary distributions have a natural interpretation as **stochastic steady states** — we'll discuss this more soon.

Mathematically, a stationary distribution is a fixed point of P when P is thought of as the map $\psi \mapsto \psi P$ from (row) vectors to (row) vectors.

Theorem. Every stochastic matrix P has at least one stationary distribution.

(We are assuming here that the state space S is finite; if not more assumptions are required)

For proof of this result, you can apply Brouwer's fixed point theorem, or see EDTC, theorem 4.3.5.

There can be many stationary distributions corresponding to a given stochastic matrix P.

- For example, if P is the identity matrix, then all marginal distributions are stationary.

To get uniqueness an invariant distribution, the transition matrix P must have the property that no nontrivial subsets of the state space are **infinitely persistent**.

A subset of the state space is infinitely persistent if other parts of the state space cannot be accessed from it.

Thus, infinite persistence of a non-trivial subset is the opposite of irreducibility.

This gives some intuition for the following fundamental theorem.

Theorem. If P is both aperiodic and irreducible, then

1. P has exactly one stationary distribution ψ^*.
2. For any initial marginal distribution ψ_0, we have $\|\psi_0 P^t - \psi^*\| \to 0$ as $t \to \infty$.

For a proof, see, for example, theorem 5.2 of [Häggström, 2002].

(Note that part 1 of the theorem only requires irreducibility, whereas part 2 requires both irreducibility and aperiodicity)

A stochastic matrix that satisfies the conditions of the theorem is sometimes called **uniformly ergodic**.

A sufficient condition for aperiodicity and irreducibility is that every element of P is strictly positive.

- Try to convince yourself of this.

34.6.1 Example

Recall our model of the employment/unemployment dynamics of a particular worker *discussed above*.

Assuming $\alpha \in (0, 1)$ and $\beta \in (0, 1)$, the uniform ergodicity condition is satisfied.

Let $\psi^* = (p, 1 - p)$ be the stationary distribution, so that p corresponds to unemployment (state 0).

Using $\psi^* = \psi^* P$ and a bit of algebra yields

$$p = \frac{\beta}{\alpha + \beta}$$

This is, in some sense, a steady state probability of unemployment — more about the interpretation of this below.

Not surprisingly it tends to zero as $\beta \to 0$, and to one as $\alpha \to 0$.

34.6.2 Calculating Stationary Distributions

As discussed above, a particular Markov matrix P can have many stationary distributions.

That is, there can be many row vectors ψ such that $\psi = \psi P$.

In fact if P has two distinct stationary distributions ψ_1, ψ_2 then it has infinitely many, since in this case, as you can verify, for any $\lambda \in [0, 1]$

$$\psi_3 := \lambda \psi_1 + (1 - \lambda) \psi_2$$

is a stationary distribution for P.

If we restrict attention to the case in which only one stationary distribution exists, one way to finding it is to solve the system

$$\psi (I_n - P) = 0 \tag{34.7}$$

for ψ, where I_n is the $n \times n$ identity.

But the zero vector solves system (34.7), so we must proceed cautiously.

We want to impose the restriction that ψ is a probability distribution.

There are various ways to do this.

One option is to regard solving system (34.7) as an eigenvector problem: a vector ψ such that $\psi = \psi P$ is a left eigenvector associated with the unit eigenvalue $\lambda = 1$.

A stable and sophisticated algorithm specialized for stochastic matrices is implemented in QuantEcon.py.

This is the one we recommend:

```
P = [[0.4, 0.6],
     [0.2, 0.8]]

mc = qe.MarkovChain(P)
mc.stationary_distributions  # Show all stationary distributions
```

```
array([[0.25, 0.75]])
```

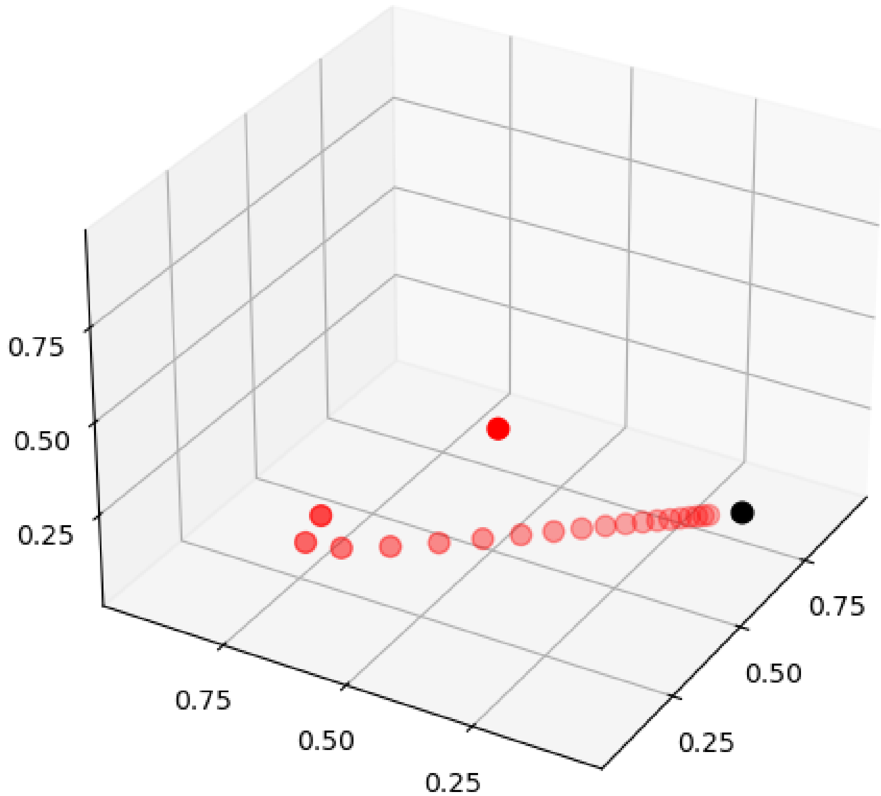
34.6.3 Convergence to Stationarity

Part 2 of the Markov chain convergence theorem *stated above* tells us that the marginal distribution of X_t converges to the stationary distribution regardless of where we begin.

This adds considerable authority to our interpretation of ψ^* as a stochastic steady state.

The convergence in the theorem is illustrated in the next figure

```
P = ((0.971, 0.029, 0.000),
     (0.145, 0.778, 0.077),
     (0.000, 0.508, 0.492))
P = np.array(P)

ψ = (0.0, 0.2, 0.8)        # Initial condition

fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')

ax.set(xlim=(0, 1), ylim=(0, 1), zlim=(0, 1),
       xticks=(0.25, 0.5, 0.75),
       yticks=(0.25, 0.5, 0.75),
       zticks=(0.25, 0.5, 0.75))

x_vals, y_vals, z_vals = [], [], []
for t in range(20):
    x_vals.append(ψ[0])
    y_vals.append(ψ[1])
    z_vals.append(ψ[2])
    ψ = ψ @ P

ax.scatter(x_vals, y_vals, z_vals, c='r', s=60)
ax.view_init(30, 210)

mc = qe.MarkovChain(P)
ψ_star = mc.stationary_distributions[0]
ax.scatter(ψ_star[0], ψ_star[1], ψ_star[2], c='k', s=60)

plt.show()
```



Here

- P is the stochastic matrix for recession and growth *considered above*.
- The highest red dot is an arbitrarily chosen initial marginal probability distribution ψ, represented as a vector in $\mathbb{R}^3$.
- The other red dots are the marginal distributions ψP^t for $t = 1, 2, \ldots$.
- The black dot is ψ^*.

You might like to try experimenting with different initial conditions.

34.7 Ergodicity

Under irreducibility, yet another important result obtains: for all $x \in S$,

$$\frac{1}{m} \sum_{t=1}^{m} \mathbf{1}\{X_t = x\} \to \psi^*(x) \quad \text{as } m \to \infty \tag{34.8}$$

Here

- $\mathbf{1}\{X_t = x\} = 1$ if $X_t = x$ and zero otherwise
- convergence is with probability one
- the result does not depend on the marginal distribution of X_0

The result tells us that the fraction of time the chain spends at state x converges to $\psi^*(x)$ as time goes to infinity.

This gives us another way to interpret the stationary distribution — provided that the convergence result in (34.8) is valid.

The convergence asserted in (34.8) is a special case of a law of large numbers result for Markov chains — see EDTC, section 4.3.4 for some additional information.

34.7.1 Example

Recall our cross-sectional interpretation of the employment/unemployment model *discussed above*.

Assume that $\alpha \in (0,1)$ and $\beta \in (0,1)$, so that irreducibility and aperiodicity both hold.

We saw that the stationary distribution is $(p, 1-p)$, where

$$p = \frac{\beta}{\alpha + \beta}$$

In the cross-sectional interpretation, this is the fraction of people unemployed.

In view of our latest (ergodicity) result, it is also the fraction of time that a single worker can expect to spend unemployed.

Thus, in the long-run, cross-sectional averages for a population and time-series averages for a given person coincide.

This is one aspect of the concept of ergodicity.

34.8 Computing Expectations

We sometimes want to compute mathematical expectations of functions of X_t of the form

$$\mathbb{E}[h(X_t)] \tag{34.9}$$

and conditional expectations such as

$$\mathbb{E}[h(X_{t+k}) \mid X_t = x] \tag{34.10}$$

where

- $\{X_t\}$ is a Markov chain generated by $n \times n$ stochastic matrix P
- h is a given function, which, in terms of matrix algebra, we'll think of as the column vector

$$h = \begin{pmatrix} h(x_1) \\ \vdots \\ h(x_n) \end{pmatrix}$$

Computing the unconditional expectation (34.9) is easy.

We just sum over the marginal distribution of X_t to get

$$\mathbb{E}[h(X_t)] = \sum_{x \in S} (\psi P^t)(x) h(x)$$

Here ψ is the distribution of X_0.

Since ψ and hence ψP^t are row vectors, we can also write this as

$$\mathbb{E}[h(X_t)] = \psi P^t h$$

For the conditional expectation (34.10), we need to sum over the conditional distribution of X_{t+k} given $X_t = x$.

We already know that this is $P^k(x, \cdot)$, so

$$\mathbb{E}[h(X_{t+k}) \mid X_t = x] = (P^k h)(x) \tag{34.11}$$

The vector $P^k h$ stores the conditional expectation $\mathbb{E}[h(X_{t+k}) \mid X_t = x]$ over all x.

34.8.1 Iterated Expectations

The **law of iterated expectations** states that

$$\mathbb{E}\left[\mathbb{E}[h(X_{t+k}) \mid X_t = x]\right] = \mathbb{E}[h(X_{t+k})]$$

where the outer $\mathbb{E}$ on the left side is an unconditional distribution taken with respect to the marginal distribution ψ_t of X_t (again see equation (34.6)).

To verify the law of iterated expectations, use equation (34.11) to substitute $(P^k h)(x)$ for $E[h(X_{t+k}) \mid X_t = x]$, write

$$\mathbb{E}\left[\mathbb{E}[h(X_{t+k}) \mid X_t = x]\right] = \psi_t P^k h,$$

and note $\psi_t P^k h = \psi_{t+k} h = \mathbb{E}[h(X_{t+k})]$.

34.8.2 Expectations of Geometric Sums

Sometimes we want to compute the mathematical expectation of a geometric sum, such as $\sum_t \beta^t h(X_t)$.

In view of the preceding discussion, this is

$$\mathbb{E}[\sum_{j=0}^{\infty} \beta^j h(X_{t+j}) \mid X_t = x\Big] = [(I - \beta P)^{-1} h](x)$$

where

$$(I - \beta P)^{-1} = I + \beta P + \beta^2 P^2 + \cdots$$

Premultiplication by $(I - \beta P)^{-1}$ amounts to "applying the **resolvent operator**".

34.9 Exercises

Exercise 34.9.1

According to the discussion *above*, if a worker's employment dynamics obey the stochastic matrix

$$P = \begin{pmatrix} 1-\alpha & \alpha \\ \beta & 1-\beta \end{pmatrix}$$

with $\alpha \in (0, 1)$ and $\beta \in (0, 1)$, then, in the long-run, the fraction of time spent unemployed will be

$$p := \frac{\beta}{\alpha + \beta}$$

In other words, if $\{X_t\}$ represents the Markov chain for employment, then $\bar{X}_m \to p$ as $m \to \infty$, where

$$\bar{X}_m := \frac{1}{m} \sum_{t=1}^{m} \mathbf{1}\{X_t = 0\}$$

This exercise asks you to illustrate convergence by computing $\bar{X}_m$ for large m and checking that it is close to p.

You will see that this statement is true regardless of the choice of initial condition or the values of α, β, provided both lie in $(0, 1)$.

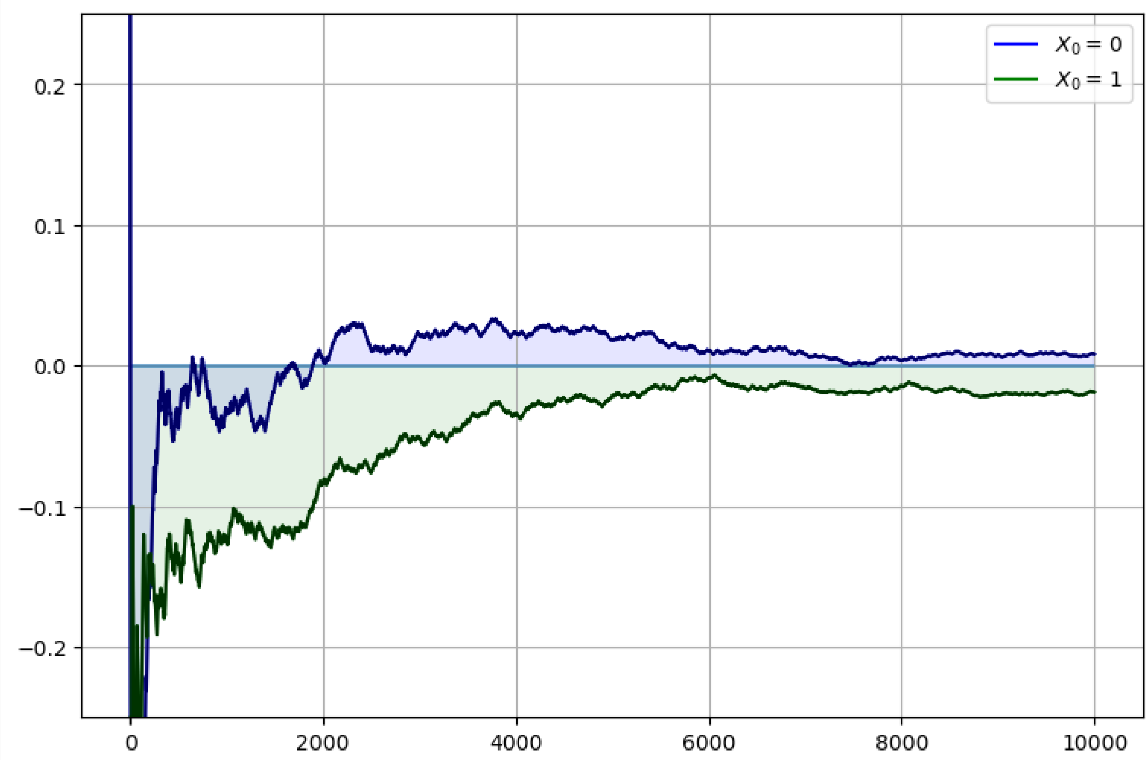
Solution

We will address this exercise graphically.

The plots show the time series of $\bar{X}_m - p$ for two initial conditions.

As m gets large, both series converge to zero.

```
α = β = 0.1
N = 10000
p = β / (α + β)

P = ((1 - α,       α),               # Careful: P and p are distinct
     (    β,   1 - β))
mc = MarkovChain(P)

fig, ax = plt.subplots(figsize=(9, 6))
ax.set_ylim(-0.25, 0.25)
ax.grid()
ax.hlines(0, 0, N, lw=2, alpha=0.6)   # Horizonal line at zero

for x0, col in ((0, 'blue'), (1, 'green')):
    # Generate time series for worker that starts at x0
    X = mc.simulate(N, init=x0)
    # Compute fraction of time spent unemployed, for each n
    X_bar = (X == 0).cumsum() / (1 + np.arange(N, dtype=float))
    # Plot
    ax.fill_between(range(N), np.zeros(N), X_bar - p, color=col, alpha=0.1)
    ax.plot(X_bar - p, color=col, label=fr'$X_0 = \, {x0} $')
    # Overlay in black--make lines clearer
    ax.plot(X_bar - p, 'k-', alpha=0.6)

ax.legend(loc='upper right')
plt.show()
```



Exercise 34.9.2

A topic of interest for economics and many other disciplines is *ranking*.

Let's now consider one of the most practical and important ranking problems — the rank assigned to web pages by search engines.

(Although the problem is motivated from outside of economics, there is in fact a deep connection between search ranking systems and prices in certain competitive equilibria — see [Du *et al.*, 2013].)

To understand the issue, consider the set of results returned by a query to a web search engine.

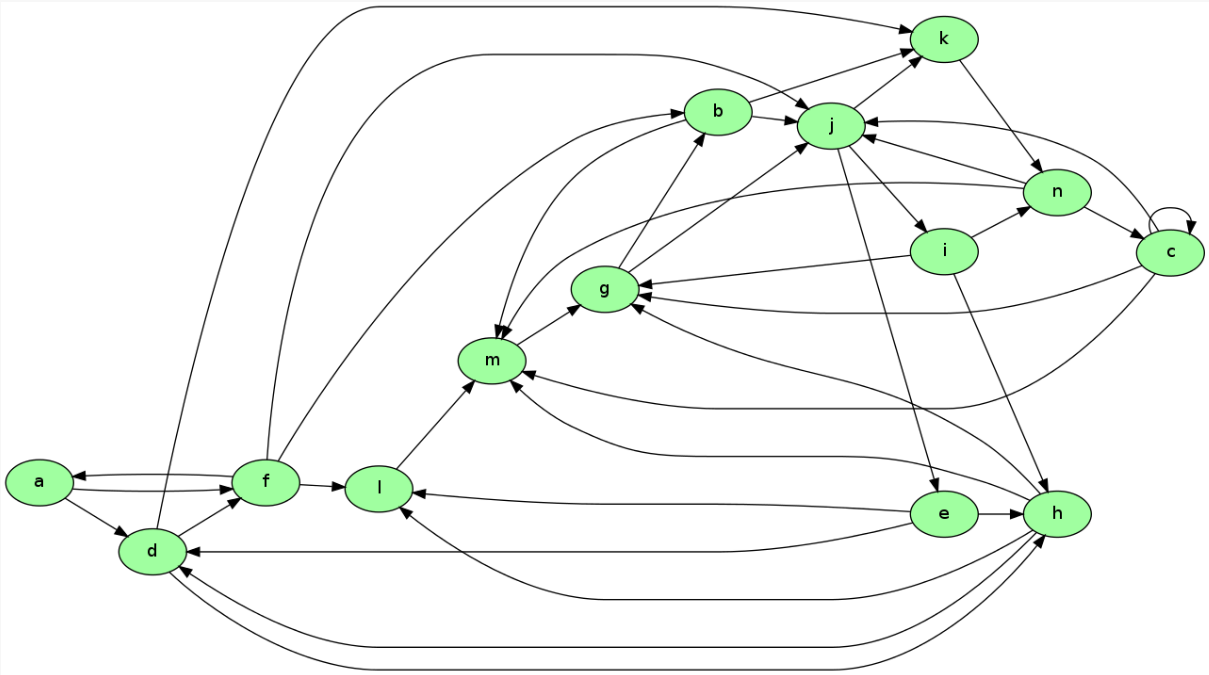
For the user, it is desirable to

1. receive a large set of accurate matches
2. have the matches returned in order, where the order corresponds to some measure of "importance"

Ranking according to a measure of importance is the problem we now consider.

The methodology developed to solve this problem by Google founders Larry Page and Sergey Brin is known as PageRank.

To illustrate the idea, consider the following diagram



Imagine that this is a miniature version of the WWW, with

- each node representing a web page
- each arrow representing the existence of a link from one page to another

Now let's think about which pages are likely to be important, in the sense of being valuable to a search engine user.

One possible criterion for the importance of a page is the number of inbound links — an indication of popularity.

By this measure, `m` and `j` are the most important pages, with 5 inbound links each.

However, what if the pages linking to `m`, say, are not themselves important?

Thinking this way, it seems appropriate to weight the inbound nodes by relative importance.

The PageRank algorithm does precisely this.

A slightly simplified presentation that captures the basic idea is as follows.

Letting j be (the integer index of) a typical page and r_j be its ranking, we set

$$r_j = \sum_{i \in L_j} \frac{r_i}{\ell_i}$$

where

- ℓ_i is the total number of outbound links from i
- L_j is the set of all pages i such that i has a link to j

This is a measure of the number of inbound links, weighted by their own ranking (and normalized by $1/\ell_i$).

There is, however, another interpretation, and it brings us back to Markov chains.

Let P be the matrix given by $P(i,j) = \mathbf{1}\{i \to j\}/\ell_i$ where $\mathbf{1}\{i \to j\} = 1$ if i has a link to j and zero otherwise.

The matrix P is a stochastic matrix provided that each page has at least one link.

With this definition of P we have

$$r_j = \sum_{i \in L_j} \frac{r_i}{\ell_i} = \sum_{\text{all } i} \mathbf{1}\{i \to j\} \frac{r_i}{\ell_i} = \sum_{\text{all } i} P(i,j) r_i$$

Writing r for the row vector of rankings, this becomes $r = rP$.

Hence r is the stationary distribution of the stochastic matrix P.

Let's think of $P(i,j)$ as the probability of "moving" from page i to page j.

The value $P(i,j)$ has the interpretation

- $P(i,j) = 1/k$ if i has k outbound links and j is one of them
- $P(i,j) = 0$ if i has no direct link to j

Thus, motion from page to page is that of a web surfer who moves from one page to another by randomly clicking on one of the links on that page.

Here "random" means that each link is selected with equal probability.

Since r is the stationary distribution of P, assuming that the uniform ergodicity condition is valid, we *can interpret* r_j as the fraction of time that a (very persistent) random surfer spends at page j.

Your exercise is to apply this ranking algorithm to the graph pictured above and return the list of pages ordered by rank.

There is a total of 14 nodes (i.e., web pages), the first named `a` and the last named `n`.

A typical line from the file has the form

```
d -> h;
```

This should be interpreted as meaning that there exists a link from `d` to `h`.

The data for this graph is shown below, and read into a file called `web_graph_data.txt` when the cell is executed.

```
%%file web_graph_data.txt
a -> d;
a -> f;
b -> j;
b -> k;
b -> m;
c -> c;
c -> g;
c -> j;
c -> m;
d -> f;
d -> h;
d -> k;
e -> d;
e -> h;
e -> l;
f -> a;
f -> b;
f -> j;
f -> l;
g -> b;
g -> j;
h -> d;
h -> g;
h -> l;
```

```
h -> m;
i -> g;
i -> h;
i -> n;
j -> e;
j -> i;
j -> k;
k -> n;
l -> m;
m -> g;
n -> c;
n -> j;
n -> m;
```

```
Overwriting web_graph_data.txt
```

To parse this file and extract the relevant information, you can use regular expressions.

The following code snippet provides a hint as to how you can go about this

```
import re
re.findall(r'\w', 'x +++ y ****** z')  # \w matches alphanumerics
```

```
['x', 'y', 'z']
```

```
re.findall(r'\w', 'a ^^ b &&& $$ c')
```

```
['a', 'b', 'c']
```

When you solve for the ranking, you will find that the highest ranked node is in fact `g`, while the lowest is `a`.

Solution

Here is one solution:

```
"""
Return list of pages, ordered by rank
"""
import re
from operator import itemgetter

infile = 'web_graph_data.txt'
alphabet = 'abcdefghijklmnopqrstuvwxyz'

n = 14 # Total number of web pages (nodes)

# Create a matrix Q indicating existence of links
#  * Q[i, j] = 1 if there is a link from i to j
#  * Q[i, j] = 0 otherwise
Q = np.zeros((n, n), dtype=int)
with open(infile) as f:
    edges = f.readlines()
for edge in edges:
    from_node, to_node = re.findall(r'\w', edge)
    i, j = alphabet.index(from_node), alphabet.index(to_node)
    Q[i, j] = 1
# Create the corresponding Markov matrix P
P = np.empty((n, n))
```

```
for i in range(n):
    P[i, :] = Q[i, :] / Q[i, :].sum()
mc = MarkovChain(P)
# Compute the stationary distribution r
r = mc.stationary_distributions[0]
ranked_pages = {alphabet[i] : r[i] for i in range(n)}
# Print solution, sorted from highest to lowest rank
print('Rankings\n ***')
for name, rank in sorted(ranked_pages.items(), key=itemgetter(1), reverse=1):
    print(f'{name}: {rank:.4}')
```

```
Rankings
 ***
g: 0.1607
j: 0.1594
m: 0.1195
n: 0.1088
k: 0.09106
b: 0.08326
e: 0.05312
i: 0.05312
c: 0.04834
h: 0.0456
l: 0.03202
d: 0.03056
f: 0.01164
a: 0.002911
```

Exercise 34.9.3

In numerical work, it is sometimes convenient to replace a continuous model with a discrete one.

In particular, Markov chains are routinely generated as discrete approximations to AR(1) processes of the form

$$y_{t+1} = \rho y_t + u_{t+1}$$

Here u_t is assumed to be IID and $N(0, \sigma_u^2)$.

The variance of the stationary probability distribution of $\{y_t\}$ is

$$\sigma_y^2 := \frac{\sigma_u^2}{1 - \rho^2}$$

Tauchen's method [Tauchen, 1986] is the most common method for approximating this continuous state process with a finite state Markov chain.

A routine for this already exists in QuantEcon.py but let's write our own version as an exercise.

As a first step, we choose

- n, the number of states for the discrete approximation
- m, an integer that parameterizes the width of the state space

Next, we create a state space $\{x_0, \ldots, x_{n-1}\} \subset \mathbb{R}$ and a stochastic $n \times n$ matrix P such that

- $x_0 = -m \, \sigma_y$
- $x_{n-1} = m \, \sigma_y$

- $x_{i+1} = x_i + s$ where $s = (x_{n-1} - x_0)/(n-1)$

Let F be the cumulative distribution function of the normal distribution $N(0, \sigma_u^2)$.

The values $P(x_i, x_j)$ are computed to approximate the AR(1) process — omitting the derivation, the rules are as follows:

1. If $j = 0$, then set

$$P(x_i, x_j) = P(x_i, x_0) = F(x_0 - \rho x_i + s/2)$$

2. If $j = n - 1$, then set

$$P(x_i, x_j) = P(x_i, x_{n-1}) = 1 - F(x_{n-1} - \rho x_i - s/2)$$

3. Otherwise, set

$$P(x_i, x_j) = F(x_j - \rho x_i + s/2) - F(x_j - \rho x_i - s/2)$$

The exercise is to write a function `approx_markov(rho, sigma_u, m=3, n=7)` that returns $\{x_0, \ldots, x_{n-1}\} \subset \mathbb{R}$ and $n \times n$ matrix P as described above.

- Even better, write a function that returns an instance of QuantEcon.py's MarkovChain class.

Solution

A solution from the QuantEcon.py library can be found here.

CHAPTER

THIRTYFIVE

INVENTORY DYNAMICS

Contents

35.1 Overview

In this lecture we will study the time path of inventories for firms that follow so-called s-S inventory dynamics.

Such firms

1. wait until inventory falls below some level s and then
2. order sufficient quantities to bring their inventory back up to capacity S.

These kinds of policies are common in practice and also optimal in certain circumstances.

A review of early literature and some macroeconomic implications can be found in [Caplin, 1985].

Here our main aim is to learn more about simulation, time series and Markov dynamics.

While our Markov environment and many of the concepts we consider are related to those found in our *lecture on finite Markov chains*, the state space is a continuum in the current application.

Let's start with some imports

```
import matplotlib.pyplot as plt
import numpy as np
from numba import jit, float64, prange
from numba.experimental import jitclass
```

35.2 Sample Paths

Consider a firm with inventory X_t.

The firm waits until $X_t \leq s$ and then restocks up to S units.

It faces stochastic demand $\{D_t\}$, which we assume is IID.

With notation $a^+ := \max\{a, 0\}$, inventory dynamics can be written as

$$X_{t+1} = \begin{cases} (S - D_{t+1})^+ & \text{if } X_t \leq s \\ (X_t - D_{t+1})^+ & \text{if } X_t > s \end{cases}$$

In what follows, we will assume that each D_t is lognormal, so that

$$D_t = \exp(\mu + \sigma Z_t)$$

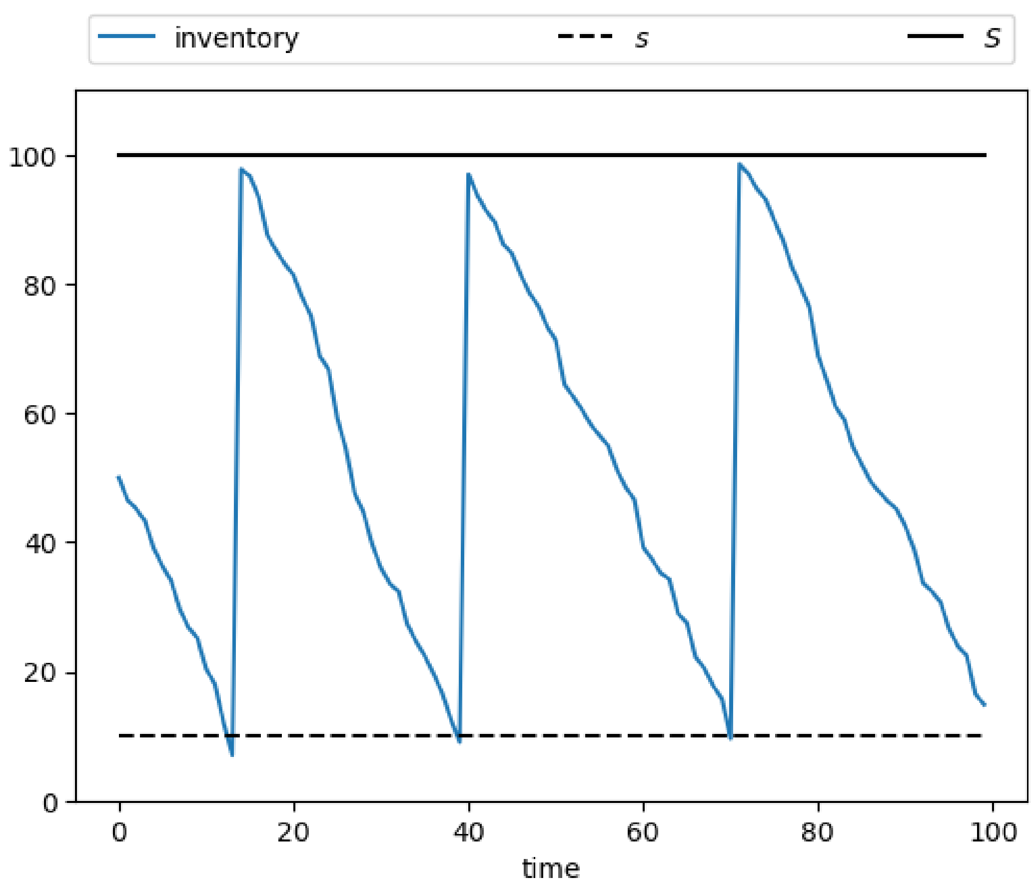
where μ and σ are parameters and $\{Z_t\}$ is IID and standard normal.

Here's a class that stores parameters and generates time paths for inventory.

```python
firm_data = [
   ('s', float64),          # restock trigger level
   ('S', float64),          # capacity
   ('mu', float64),         # shock location parameter
   ('sigma', float64)       # shock scale parameter
]


@jitclass(firm_data)
class Firm:

    def __init__(self, s=10, S=100, mu=1.0, sigma=0.5):

        self.s, self.S, self.mu, self.sigma = s, S, mu, sigma

    def update(self, x):
        "Update the state from t to t+1 given current state x."

        Z = np.random.randn()
        D = np.exp(self.mu + self.sigma * Z)
        if x <= self.s:
            return max(self.S - D, 0)
        else:
            return max(x - D, 0)

    def sim_inventory_path(self, x_init, sim_length):

        X = np.empty(sim_length)
        X[0] = x_init

        for t in range(sim_length-1):
            X[t+1] = self.update(X[t])
        return X
```

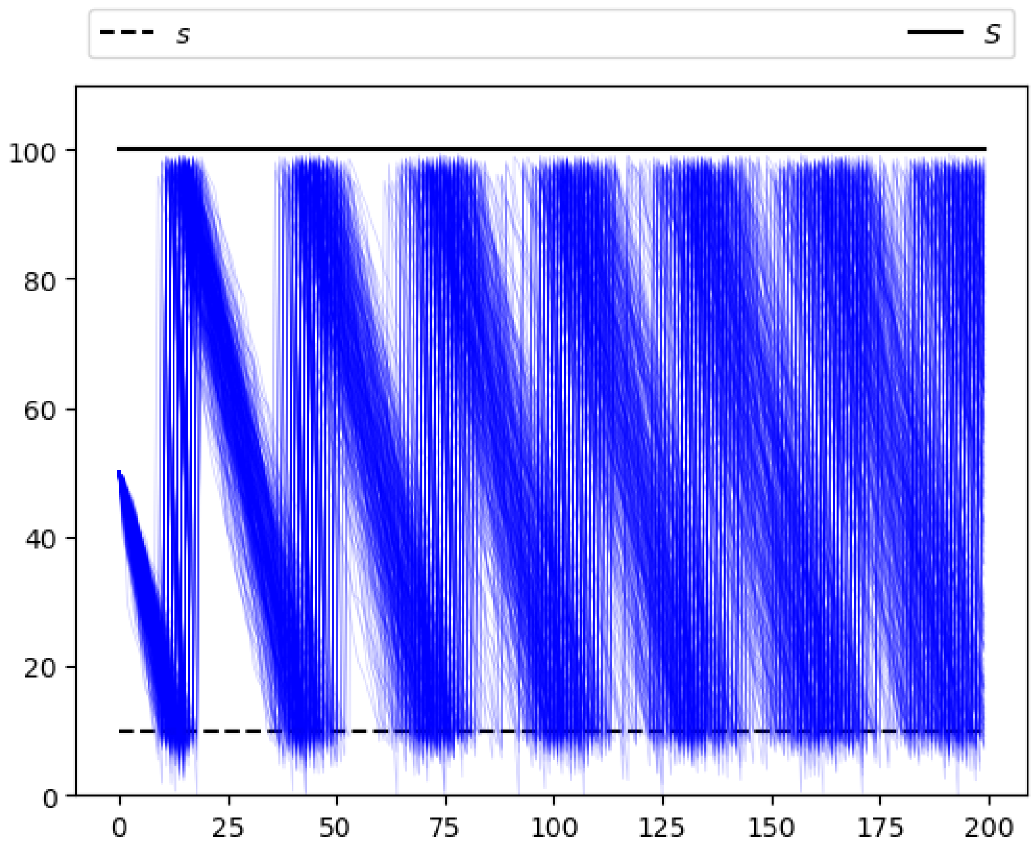
Let's run a first simulation, of a single path:

```
firm = Firm()

s, S = firm.s, firm.S
sim_length = 100
x_init = 50

X = firm.sim_inventory_path(x_init, sim_length)

fig, ax = plt.subplots()
bbox = (0., 1.02, 1., .102)
legend_args = {'ncol': 3,
               'bbox_to_anchor': bbox,
               'loc': 3,
               'mode': 'expand'}

ax.plot(X, label="inventory")
ax.plot(np.full(sim_length, s), 'k--', label="$s$")
ax.plot(np.full(sim_length, S), 'k-', label="$S$")
ax.set_ylim(0, S+10)
ax.set_xlabel("time")
ax.legend(**legend_args)

plt.show()
```



Now let's simulate multiple paths in order to build a more complete picture of the probabilities of different outcomes:

```
sim_length=200
fig, ax = plt.subplots()

ax.plot(np.full(sim_length, s), 'k--', label="$s$")
ax.plot(np.full(sim_length, S), 'k-', label="$S$")
ax.set_ylim(0, S+10)
ax.legend(**legend_args)

for i in range(400):
    X = firm.sim_inventory_path(x_init, sim_length)
    ax.plot(X, 'b', alpha=0.2, lw=0.5)

plt.show()
```



35.3 Marginal Distributions

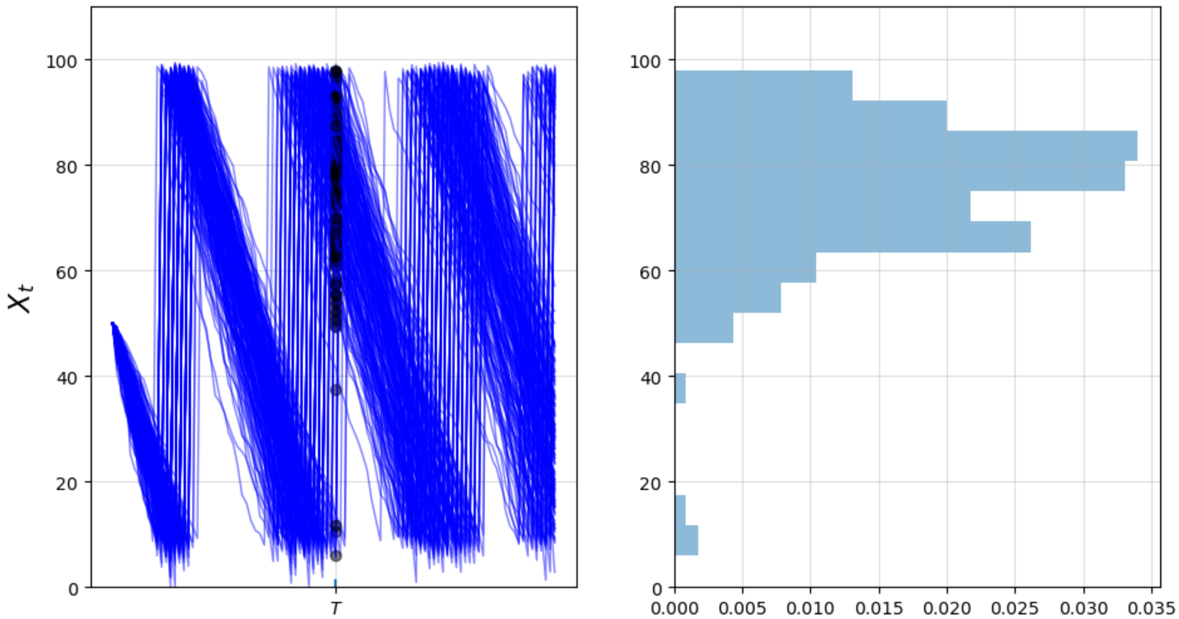
Now let's look at the marginal distribution ψ_T of X_T for some fixed T.

We will do this by generating many draws of X_T given initial condition X_0.

With these draws of X_T we can build up a picture of its distribution ψ_T.

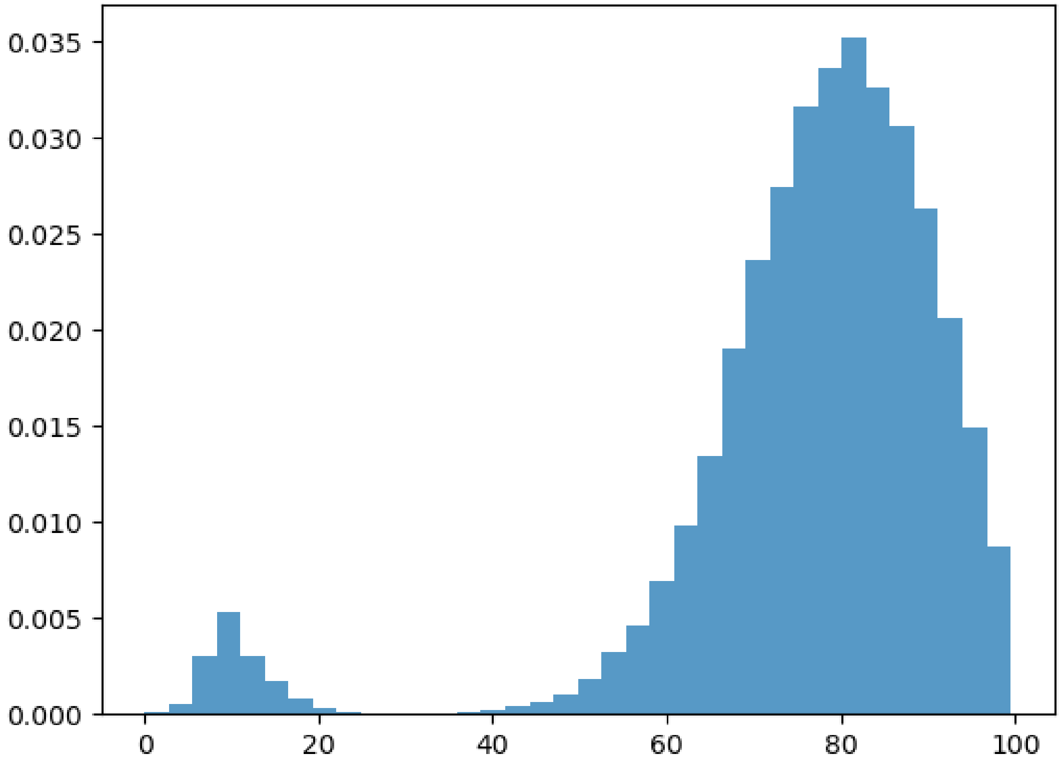
Here's one visualization, with $T = 50$.

```
T = 50
M = 200  # Number of draws

ymin, ymax = 0, S + 10

fig, axes = plt.subplots(1, 2, figsize=(11, 6))

for ax in axes:
    ax.grid(alpha=0.4)

ax = axes[0]

ax.set_ylim(ymin, ymax)
ax.set_ylabel('$X_t$', fontsize=16)
ax.vlines((T,), -1.5, 1.5)

ax.set_xticks((T,))
ax.set_xticklabels((r'$T$',))

sample = np.empty(M)
for m in range(M):
    X = firm.sim_inventory_path(x_init, 2 * T)
    ax.plot(X, 'b-', lw=1, alpha=0.5)
    ax.plot((T,), (X[T+1],), 'ko', alpha=0.5)
    sample[m] = X[T+1]

axes[1].set_ylim(ymin, ymax)

axes[1].hist(sample,
             bins=16,
             density=True,
             orientation='horizontal',
             histtype='bar',
             alpha=0.5)

plt.show()
```



We can build up a clearer picture by drawing more samples

```
T = 50
M = 50_000

fig, ax = plt.subplots()

sample = np.empty(M)
for m in range(M):
    X = firm.sim_inventory_path(x_init, T+1)
    sample[m] = X[T]

ax.hist(sample,
         bins=36,
         density=True,
         histtype='bar',
         alpha=0.75)

plt.show()
```



Note that the distribution is bimodal

- Most firms have restocked twice but a few have restocked only once (see figure with paths above).
- Firms in the second category have lower inventory.

We can also approximate the distribution using a kernel density estimator.

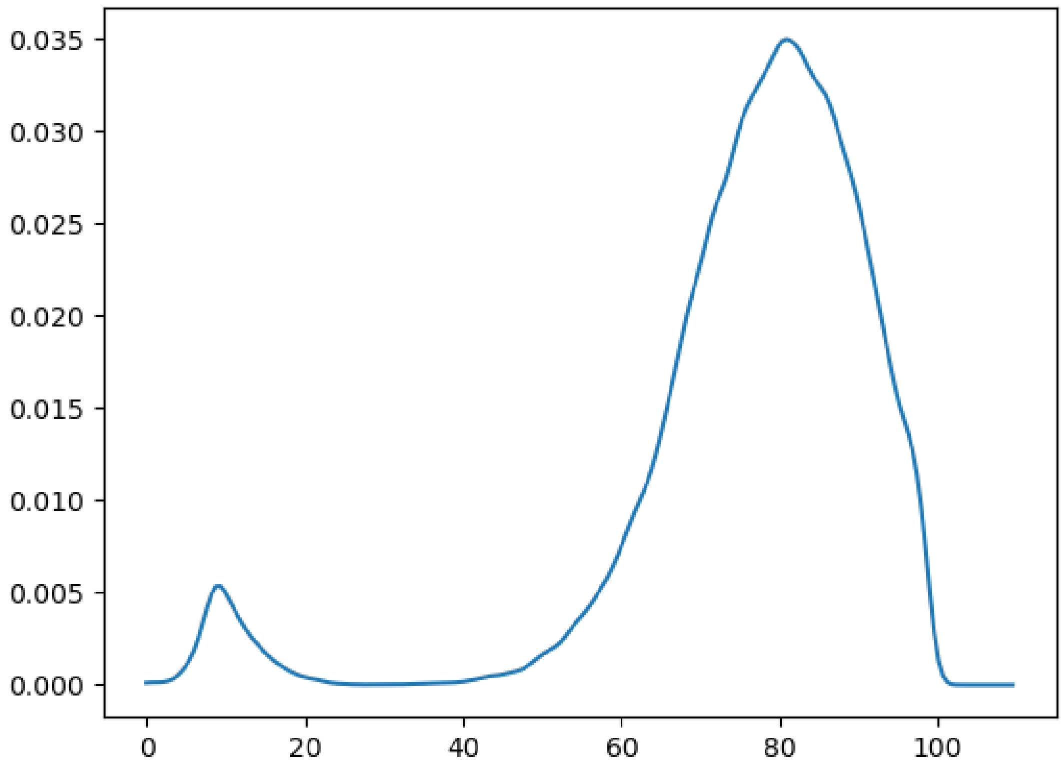
Kernel density estimators can be thought of as smoothed histograms.

They are preferable to histograms when the distribution being estimated is likely to be smooth.

We will use a kernel density estimator from scikit-learn

```
from sklearn.neighbors import KernelDensity

def plot_kde(sample, ax, label=''):

    xmin, xmax = 0.9 * min(sample), 1.1 * max(sample)
    xgrid = np.linspace(xmin, xmax, 200)
    kde = KernelDensity(kernel='gaussian').fit(sample[:, None])
    log_dens = kde.score_samples(xgrid[:, None])

    ax.plot(xgrid, np.exp(log_dens), label=label)
```

```
fig, ax = plt.subplots()
plot_kde(sample, ax)
plt.show()
```



The allocation of probability mass is similar to what was shown by the histogram just above.

35.4 Exercises

Exercise 35.4.1

This model is asymptotically stationary, with a unique stationary distribution.

(See the discussion of stationarity in our lecture on AR(1) processes for background — the fundamental concepts are the same.)

In particular, the sequence of marginal distributions $\{\psi_t\}$ is converging to a unique limiting distribution that does not depend on initial conditions.

Although we will not prove this here, we can investigate it using simulation.

Your task is to generate and plot the sequence $\{\psi_t\}$ at times $t = 10, 50, 250, 500, 750$ based on the discussion above.

(The kernel density estimator is probably the best way to present each distribution.)

You should see convergence, in the sense that differences between successive distributions are getting smaller.

Try different initial conditions to verify that, in the long run, the distribution is invariant across initial conditions.

Solution

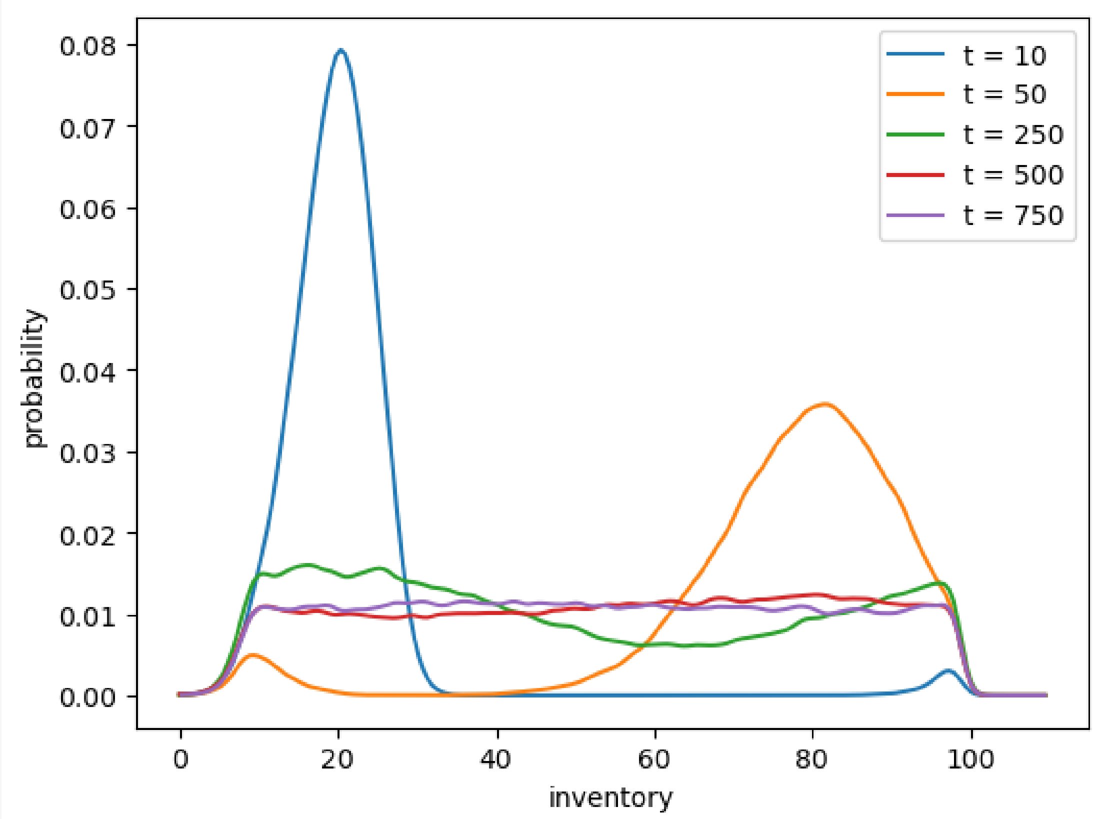
Below is one possible solution:

The computations involve a lot of CPU cycles so we have tried to write the code efficiently.

This meant writing a specialized function rather than using the class above.

```
s, S, mu, sigma = firm.s, firm.S, firm.mu, firm.sigma

@jit(parallel=True)
def shift_firms_forward(current_inventory_levels, num_periods):

    num_firms = len(current_inventory_levels)
    new_inventory_levels = np.empty(num_firms)

    for f in prange(num_firms):
        x = current_inventory_levels[f]
        for t in range(num_periods):
            Z = np.random.randn()
            D = np.exp(mu + sigma * Z)
            if x <= s:
                x = max(S - D, 0)
            else:
                x = max(x - D, 0)
        new_inventory_levels[f] = x

    return new_inventory_levels
```

```
x_init = 50
num_firms = 50_000

sample_dates = 0, 10, 50, 250, 500, 750

first_diffs = np.diff(sample_dates)

fig, ax = plt.subplots()

X = np.full(num_firms, x_init)

current_date = 0
for d in first_diffs:
    X = shift_firms_forward(X, d)
    current_date += d
    plot_kde(X, ax, label=f't = {current_date}')

ax.set_xlabel('inventory')
ax.set_ylabel('probability')
ax.legend()
plt.show()
```



Notice that by $t = 500$ or $t = 750$ the densities are barely changing.

We have reached a reasonable approximation of the stationary density.

You can convince yourself that initial conditions don't matter by testing a few of them.

For example, try rerunning the code above with all firms starting at $X_0 = 20$ or $X_0 = 80$.

Exercise 35.4.2

Using simulation, calculate the probability that firms that start with $X_0 = 70$ need to order twice or more in the first 50 periods.

You will need a large sample size to get an accurate reading.

Solution

Here is one solution.

Again, the computations are relatively intensive so we have written a a specialized function rather than using the class above.

We will also use parallelization across firms.

```
@jit(parallel=True)
def compute_freq(sim_length=50, x_init=70, num_firms=1_000_000):
```

```
    firm_counter = 0  # Records number of firms that restock 2x or more
    for m in prange(num_firms):
        x = x_init
        restock_counter = 0  # Will record number of restocks for firm m

        for t in range(sim_length):
            Z = np.random.randn()
            D = np.exp(mu + sigma * Z)
            if x <= s:
                x = max(S - D, 0)
                restock_counter += 1
            else:
                x = max(x - D, 0)

        if restock_counter > 1:
            firm_counter += 1

    return firm_counter / num_firms
```

Note the time the routine takes to run, as well as the output.

```
%%time

freq = compute_freq()
print(f"Frequency of at least two stock outs = {freq}")
```

```
Frequency of at least two stock outs = 0.447517
CPU times: user 2.82 s, sys: 11 ms, total: 2.83 s
Wall time: 679 ms
```

Try switching the `parallel` flag to `False` in the jitted function above.

Depending on your system, the difference can be substantial.

(On our desktop machine, the speed up is by a factor of 5.)

CHAPTER

THIRTYSIX

LINEAR STATE SPACE MODELS

Contents

> "We may regard the present state of the universe as the effect of its past and the cause of its future" – Marquis de Laplace

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
```

36.1 Overview

This lecture introduces the **linear state space** dynamic system.

The linear state space system is a generalization of the scalar AR(1) process we studied before.

This model is a workhorse that carries a powerful theory of prediction.

Its many applications include:

- representing dynamics of higher-order linear systems
- predicting the position of a system j steps into the future
- predicting a geometric sum of future values of a variable like
  - non-financial income

  - dividends on a stock
  - the money supply
  - a government deficit or surplus, etc.

- key ingredient of useful models
  - Friedman's permanent income model of consumption smoothing.
  - Barro's model of smoothing total tax collections.
  - Rational expectations version of Cagan's model of hyperinflation.
  - Sargent and Wallace's "unpleasant monetarist arithmetic," etc.

Let's start with some imports:

```
import matplotlib.pyplot as plt
import numpy as np
from quantecon import LinearStateSpace
from scipy.stats import norm
import random
```

36.2 The Linear State Space Model

The objects in play are:

- An $n \times 1$ vector x_t denoting the **state** at time $t = 0, 1, 2, \ldots$.
- An IID sequence of $m \times 1$ random vectors $w_t \sim N(0, I)$.
- A $k \times 1$ vector y_t of **observations** at time $t = 0, 1, 2, \ldots$.
- An $n \times n$ matrix A called the **transition matrix**.
- An $n \times m$ matrix C called the **volatility matrix**.
- A $k \times n$ matrix G sometimes called the **output matrix**.

Here is the linear state-space system

$$
\begin{aligned}
x_{t+1} &= A x_t + C w_{t+1} \\
y_t &= G x_t \\
x_0 &\sim N(\mu_0, \Sigma_0)
\end{aligned}
$$

36.2.1 Primitives

The primitives of the model are

1. the matrices A, C, G
2. shock distribution, which we have specialized to $N(0, I)$
3. the distribution of the initial condition x_0, which we have set to $N(\mu_0, \Sigma_0)$

Given A, C, G and draws of x_0 and $w_1, w_2, \ldots$, the model (36.1) pins down the values of the sequences $\{x_t\}$ and $\{y_t\}$.

Even without these draws, the primitives 1–3 pin down the **probability distributions** of $\{x_t\}$ and $\{y_t\}$.

Later we'll see how to compute these distributions and their moments.

Martingale Difference Shocks

We've made the common assumption that the shocks are independent standardized normal vectors.

But some of what we say will be valid under the assumption that $\{w_{t+1}\}$ is a **martingale difference sequence**.

A martingale difference sequence is a sequence that is zero mean when conditioned on past information.

In the present case, since $\{x_t\}$ is our state sequence, this means that it satisfies

$$\mathbb{E}[w_{t+1}|x_t, x_{t-1}, ...] = 0$$

This is a weaker condition than that $\{w_t\}$ is IID with $w_{t+1} \sim N(0, I)$.

36.2.2 Examples

By appropriate choice of the primitives, a variety of dynamics can be represented in terms of the linear state space model.

The following examples help to highlight this point.

They also illustrate the wise dictum *finding the state is an art*.

Second-order Difference Equation

Let $\{y_t\}$ be a deterministic sequence that satisfies

$$y_{t+1} = \phi_0 + \phi_1 y_t + \phi_2 y_{t-1} \quad \text{s.t.} \quad y_0, y_{-1} \text{ given} \tag{36.1}$$

To map (36.1) into our state space system (36.1), we set

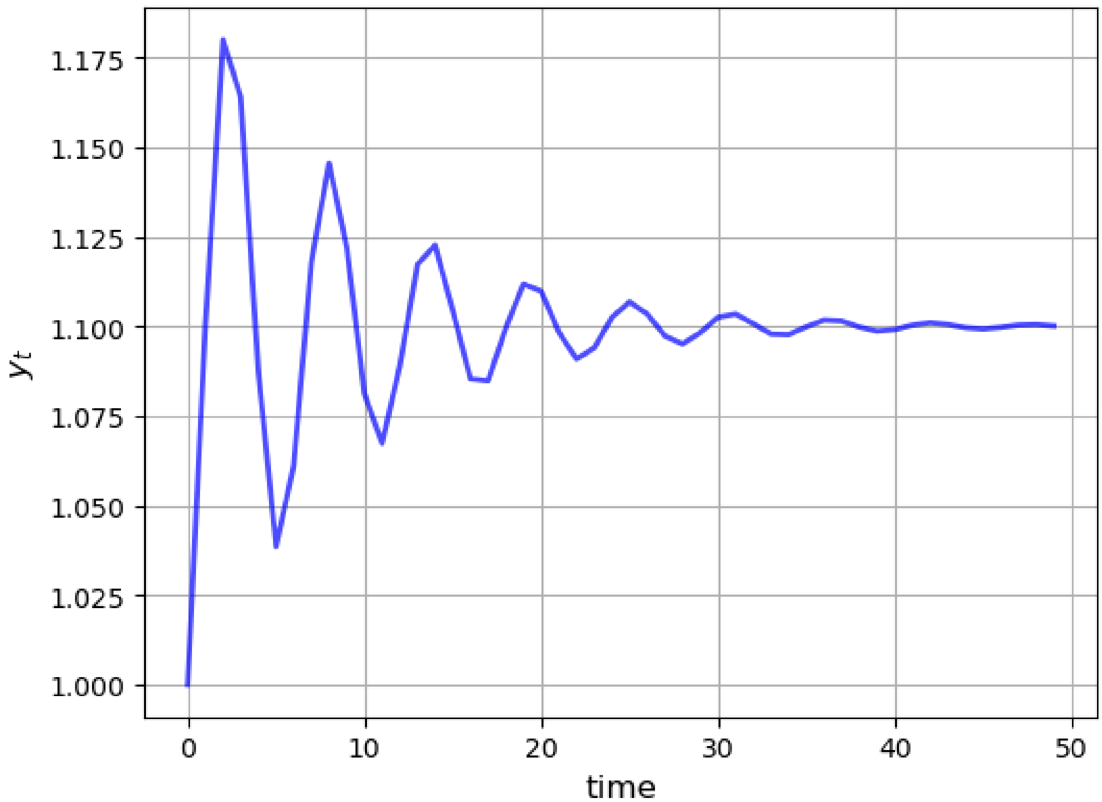
$$x_t = \begin{bmatrix} 1 \\ y_t \\ y_{t-1} \end{bmatrix} \qquad A = \begin{bmatrix} 1 & 0 & 0 \\ \phi_0 & \phi_1 & \phi_2 \\ 0 & 1 & 0 \end{bmatrix} \qquad C = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \qquad G = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}$$

You can confirm that under these definitions, (36.1) and (36.1) agree.

The next figure shows the dynamics of this process when $\phi_0 = 1.1, \phi_1 = 0.8, \phi_2 = -0.8, y_0 = y_{-1} = 1$.

```python
def plot_lss(A,
         C,
         G,
         n=3,
         ts_length=50):

    ar = LinearStateSpace(A, C, G, mu_0=np.ones(n))
    x, y = ar.simulate(ts_length)

    fig, ax = plt.subplots()
    y = y.flatten()
    ax.plot(y, 'b-', lw=2, alpha=0.7)
    ax.grid()
    ax.set_xlabel('time', fontsize=12)
    ax.set_ylabel('$y_t$', fontsize=12)
    plt.show()
```

```
ϕ_0, ϕ_1, ϕ_2 = 1.1, 0.8, -0.8

A = [[1,      0,      0  ],
     [ϕ_0,    ϕ_1,    ϕ_2],
     [0,      1,      0  ]]

C = np.zeros((3, 1))
G = [0, 1, 0]

plot_lss(A, C, G)
```



Later you'll be asked to recreate this figure.

Univariate Autoregressive Processes

We can use (36.1) to represent the model

$$y_{t+1} = \phi_1 y_t + \phi_2 y_{t-1} + \phi_3 y_{t-2} + \phi_4 y_{t-3} + \sigma w_{t+1} \tag{36.2}$$

where $\{w_t\}$ is IID and standard normal.

To put this in the linear state space format we take $x_t = \begin{bmatrix} y_t & y_{t-1} & y_{t-2} & y_{t-3} \end{bmatrix}'$ and

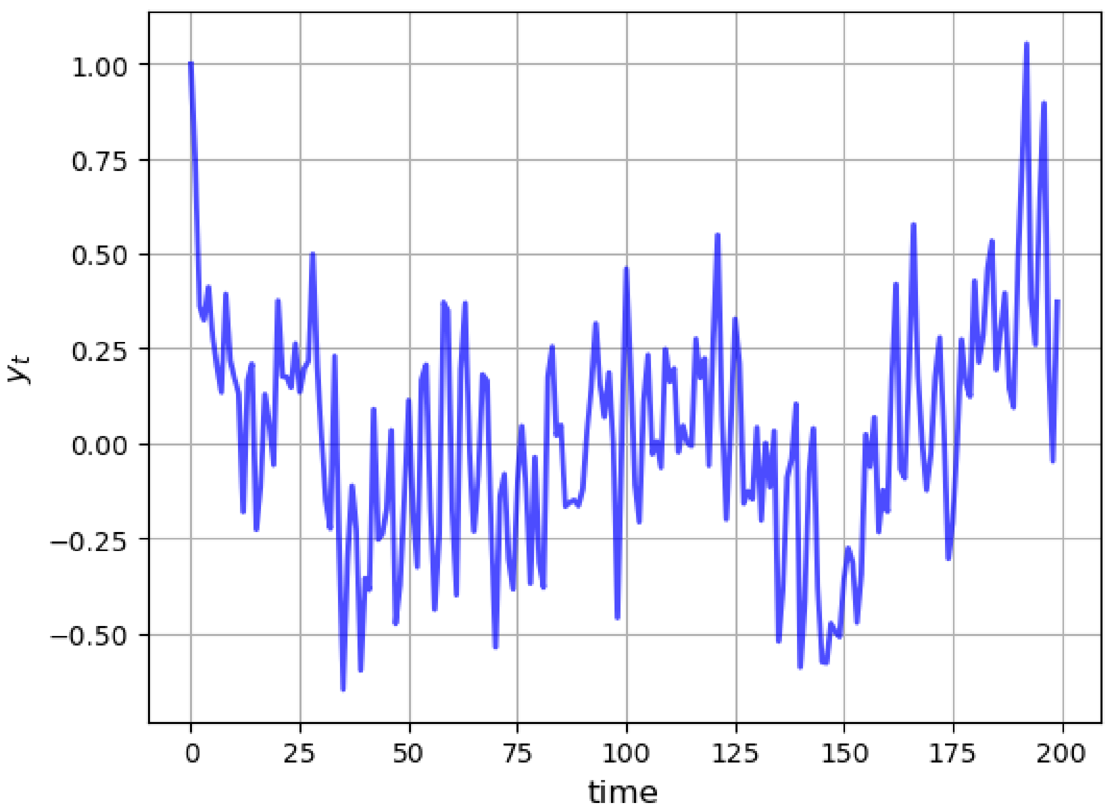
$$A = \begin{bmatrix} \phi_1 & \phi_2 & \phi_3 & \phi_4 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \qquad C = \begin{bmatrix} \sigma \\ 0 \\ 0 \\ 0 \end{bmatrix} \qquad G = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}$$

The matrix A has the form of the **companion matrix** to the vector $[\phi_1 \quad \phi_2 \quad \phi_3 \quad \phi_4]$.

The next figure shows the dynamics of this process when

$$\phi_1 = 0.5, \phi_2 = -0.2, \phi_3 = 0, \phi_4 = 0.5, \sigma = 0.2, y_0 = y_{-1} = y_{-2} = y_{-3} = 1$$

```
ϕ_1, ϕ_2, ϕ_3, ϕ_4 = 0.5, -0.2, 0, 0.5
σ = 0.2

A_1 = [[ϕ_1,    ϕ_2,    ϕ_3,    ϕ_4],
       [1,      0,      0,      0  ],
       [0,      1,      0,      0  ],
       [0,      0,      1,      0  ]]

C_1 = [[σ],
       [0],
       [0],
       [0]]

G_1 = [1, 0, 0, 0]

plot_lss(A_1, C_1, G_1, n=4, ts_length=200)
```



Vector Autoregressions

Now suppose that

- y_t is a $k \times 1$ vector
- ϕ_j is a $k \times k$ matrix and
- w_t is $k \times 1$

Then (36.2) is termed a **vector autoregression**.

To map this into (36.1), we set

$$x_t = \begin{bmatrix} y_t \\ y_{t-1} \\ y_{t-2} \\ y_{t-3} \end{bmatrix} \quad A = \begin{bmatrix} \phi_1 & \phi_2 & \phi_3 & \phi_4 \\ I & 0 & 0 & 0 \\ 0 & I & 0 & 0 \\ 0 & 0 & I & 0 \end{bmatrix} \quad C = \begin{bmatrix} \sigma \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad G = \begin{bmatrix} I & 0 & 0 & 0 \end{bmatrix}$$

where I is the $k \times k$ identity matrix and σ is a $k \times k$ matrix.

Seasonals

We can use (36.1) to represent

1. the **deterministic seasonal** $y_t = y_{t-4}$
2. the **indeterministic seasonal** $y_t = \phi_4 y_{t-4} + w_t$

In fact, both are special cases of (36.2).

With the deterministic seasonal, the transition matrix becomes

$$A = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

It is easy to check that $A^4 = I$, which implies that x_t is strictly periodic with period 4:[1]

$$x_{t+4} = x_t$$

Such an x_t process can be used to model deterministic seasonals in quarterly time series.

The *indeterministic* seasonal produces recurrent, but aperiodic, seasonal fluctuations.

Time Trends

The model $y_t = at + b$ is known as a **linear time trend**.

We can represent this model in the linear state space form by taking

$$A = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \qquad C = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \qquad G = \begin{bmatrix} a & b \end{bmatrix} \tag{36.3}$$

and starting at initial condition $x_0 = \begin{bmatrix} 0 & 1 \end{bmatrix}'$.

In fact, it's possible to use the state-space system to represent polynomial trends of any order.

[1] The eigenvalues of A are $(1, -1, i, -i)$.

For instance, we can represent the model $y_t = at^2 + bt + c$ in the linear state space form by taking

$$A = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \qquad C = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \qquad G = \begin{bmatrix} 2a & a+b & c \end{bmatrix}$$

and starting at initial condition $x_0 = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}'$.

It follows that

$$A^t = \begin{bmatrix} 1 & t & t(t-1)/2 \\ 0 & 1 & t \\ 0 & 0 & 1 \end{bmatrix}$$

Then $x_t' = \begin{bmatrix} t(t-1)/2 & t & 1 \end{bmatrix}$. You can now confirm that $y_t = Gx_t$ has the correct form.

36.2.3 Moving Average Representations

A nonrecursive expression for x_t as a function of $x_0, w_1, w_2, \dots, w_t$ can be found by using (36.1) repeatedly to obtain

$$\begin{aligned} x_t &= Ax_{t-1} + Cw_t \\ &= A^2 x_{t-2} + ACw_{t-1} + Cw_t \\ &\vdots \\ &= \sum_{j=0}^{t-1} A^j C w_{t-j} + A^t x_0 \end{aligned}$$

Representation (36.4) is a **moving average** representation.

It expresses $\{x_t\}$ as a linear function of

1. current and past values of the process $\{w_t\}$ and
2. the initial condition x_0

As an example of a moving average representation, let the model be

$$A = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \qquad C = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

You will be able to show that $A^t = \begin{bmatrix} 1 & t \\ 0 & 1 \end{bmatrix}$ and $A^j C = \begin{bmatrix} 1 & 0 \end{bmatrix}'$.

Substituting into the moving average representation (36.4), we obtain

$$x_{1t} = \sum_{j=0}^{t-1} w_{t-j} + \begin{bmatrix} 1 & t \end{bmatrix} x_0$$

where x_{1t} is the first entry of x_t.

The first term on the right is a cumulated sum of martingale differences and is therefore a martingale.

The second term is a translated linear function of time.

For this reason, x_{1t} is called a **martingale with drift**.

36.3 Distributions and Moments

36.3.1 Unconditional Moments

Using (36.1), it's easy to obtain expressions for the (unconditional) means of x_t and y_t.

We'll explain what *unconditional* and *conditional* mean soon.

Letting $\mu_t := \mathbb{E}[x_t]$ and using linearity of expectations, we find that

$$\mu_{t+1} = A\mu_t \quad \text{with} \quad \mu_0 \text{ given} \tag{36.4}$$

Here μ_0 is a primitive given in (36.1).

The variance-covariance matrix of x_t is $\Sigma_t := \mathbb{E}[(x_t - \mu_t)(x_t - \mu_t)']$.

Using $x_{t+1} - \mu_{t+1} = A(x_t - \mu_t) + Cw_{t+1}$, we can determine this matrix recursively via

$$\Sigma_{t+1} = A\Sigma_t A' + CC' \quad \text{with} \quad \Sigma_0 \text{ given} \tag{36.5}$$

As with μ_0, the matrix Σ_0 is a primitive given in (36.1).

As a matter of terminology, we will sometimes call

- μ_t the **unconditional mean** of x_t
- Σ_t the **unconditional variance-covariance matrix** of x_t

This is to distinguish μ_t and Σ_t from related objects that use conditioning information, to be defined below.

However, you should be aware that these "unconditional" moments do depend on the initial distribution $N(\mu_0, \Sigma_0)$.

Moments of the Observables

Using linearity of expectations again we have

$$\mathbb{E}[y_t] = \mathbb{E}[Gx_t] = G\mu_t \tag{36.6}$$

The variance-covariance matrix of y_t is easily shown to be

$$\text{Var}[y_t] = \text{Var}[Gx_t] = G\Sigma_t G' \tag{36.7}$$

36.3.2 Distributions

In general, knowing the mean and variance-covariance matrix of a random vector is not quite as good as knowing the full distribution.

However, there are some situations where these moments alone tell us all we need to know.

These are situations in which the mean vector and covariance matrix are all of the **parameters** that pin down the population distribution.

One such situation is when the vector in question is Gaussian (i.e., normally distributed).

This is the case here, given

1. our Gaussian assumptions on the primitives
2. the fact that normality is preserved under linear operations

In fact, it's well-known that

$$u \sim N(\bar{u}, S) \quad \text{and} \quad v = a + Bu \implies v \sim N(a + B\bar{u}, BSB') \tag{36.8}$$

In particular, given our Gaussian assumptions on the primitives and the linearity of (36.1) we can see immediately that both x_t and y_t are Gaussian for all $t \geq 0$[2].

Since x_t is Gaussian, to find the distribution, all we need to do is find its mean and variance-covariance matrix.

But in fact we've already done this, in (36.4) and (36.5).

Letting μ_t and Σ_t be as defined by these equations, we have

$$x_t \sim N(\mu_t, \Sigma_t) \tag{36.9}$$

By similar reasoning combined with (36.6) and (36.7),

$$y_t \sim N(G\mu_t, G\Sigma_t G') \tag{36.10}$$

36.3.3 Ensemble Interpretations

How should we interpret the distributions defined by (36.9)–(36.10)?

Intuitively, the probabilities in a distribution correspond to relative frequencies in a large population drawn from that distribution.

Let's apply this idea to our setting, focusing on the distribution of y_T for fixed T.

We can generate independent draws of y_T by repeatedly simulating the evolution of the system up to time T, using an independent set of shocks each time.

The next figure shows 20 simulations, producing 20 time series for $\{y_t\}$, and hence 20 draws of y_T.

The system in question is the univariate autoregressive model (36.2).

The values of y_T are represented by black dots in the left-hand figure

```python
def cross_section_plot(A,
                       C,
                       G,
                       T=20,                   # Set the time
                       ymin=-0.8,
                       ymax=1.25,
                       sample_size = 20,       # 20 observations/simulations
                       n=4):                   # The number of dimensions for the initial x0

    ar = LinearStateSpace(A, C, G, mu_0=np.ones(n))

    fig, axes = plt.subplots(1, 2, figsize=(16, 5))

    for ax in axes:
        ax.grid(alpha=0.4)
        ax.set_ylim(ymin, ymax)

    ax = axes[0]
    ax.set_ylim(ymin, ymax)
    ax.set_ylabel('$y_t$', fontsize=12)
```

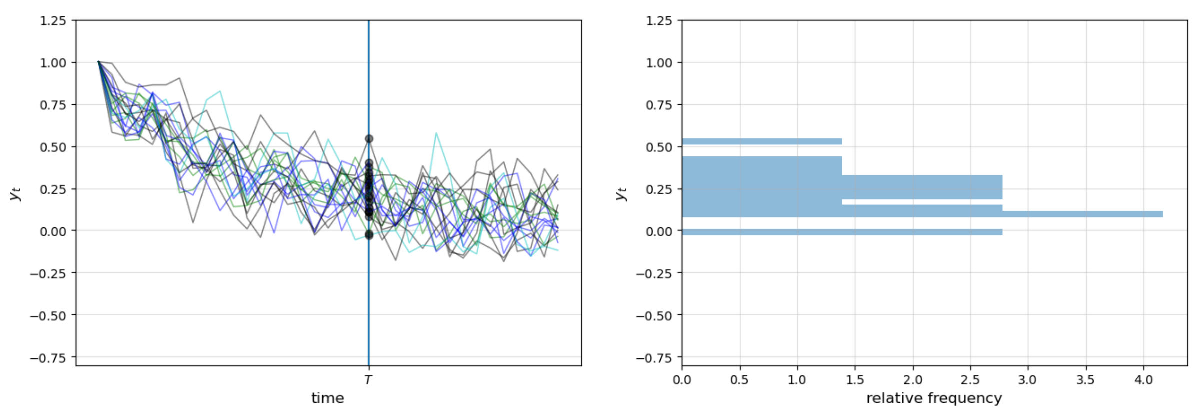
(continues on next page)

[2] The correct way to argue this is by induction. Suppose that x_t is Gaussian. Then (36.1) and (36.8) imply that x_{t+1} is Gaussian. Since x_0 is assumed to be Gaussian, it follows that every x_t is Gaussian. Evidently, this implies that each y_t is Gaussian.

(continued from previous page)

```
    ax.set_xlabel('time', fontsize=12)
    ax.vlines((T,), -1.5, 1.5)

    ax.set_xticks((T,))
    ax.set_xticklabels(('$T$',))

    sample = []
    for i in range(sample_size):
        rcolor = random.choice(('c', 'g', 'b', 'k'))
        x, y = ar.simulate(ts_length=T+15)
        y = y.flatten()
        ax.plot(y, color=rcolor, lw=1, alpha=0.5)
        ax.plot((T,), (y[T],), 'ko', alpha=0.5)
        sample.append(y[T])

    y = y.flatten()
    axes[1].set_ylim(ymin, ymax)
    axes[1].set_ylabel('$y_t$', fontsize=12)
    axes[1].set_xlabel('relative frequency', fontsize=12)
    axes[1].hist(sample, bins=16, density=True, orientation='horizontal', alpha=0.5)
    plt.show()
```

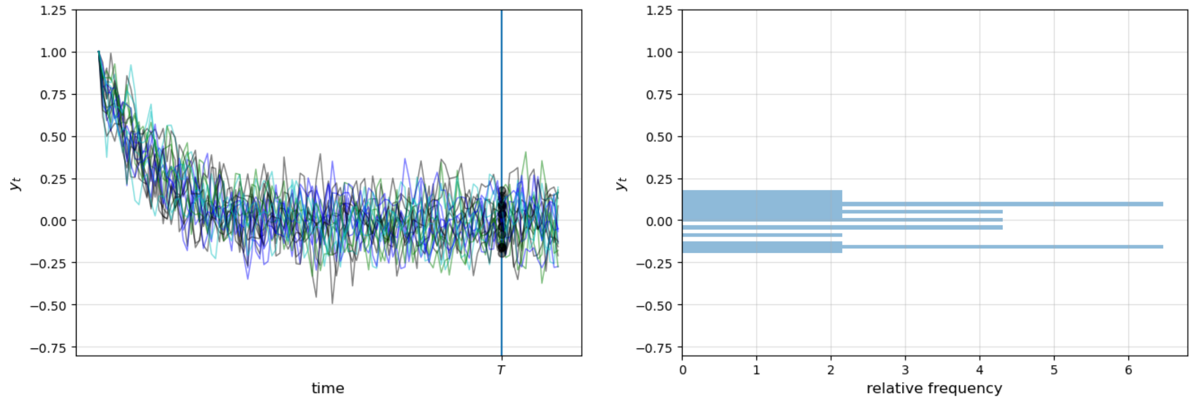
```
ϕ_1, ϕ_2, ϕ_3, ϕ_4 = 0.5, -0.2, 0, 0.5
σ = 0.1

A_2 = [[ϕ_1, ϕ_2, ϕ_3, ϕ_4],
       [1,     0,    0,    0],
       [0,     1,    0,    0],
       [0,     0,    1,    0]]

C_2 = [[σ], [0], [0], [0]]

G_2 = [1, 0, 0, 0]

cross_section_plot(A_2, C_2, G_2)
```



In the right-hand figure, these values are converted into a rotated histogram that shows relative frequencies from our sample of 20 y_T's.

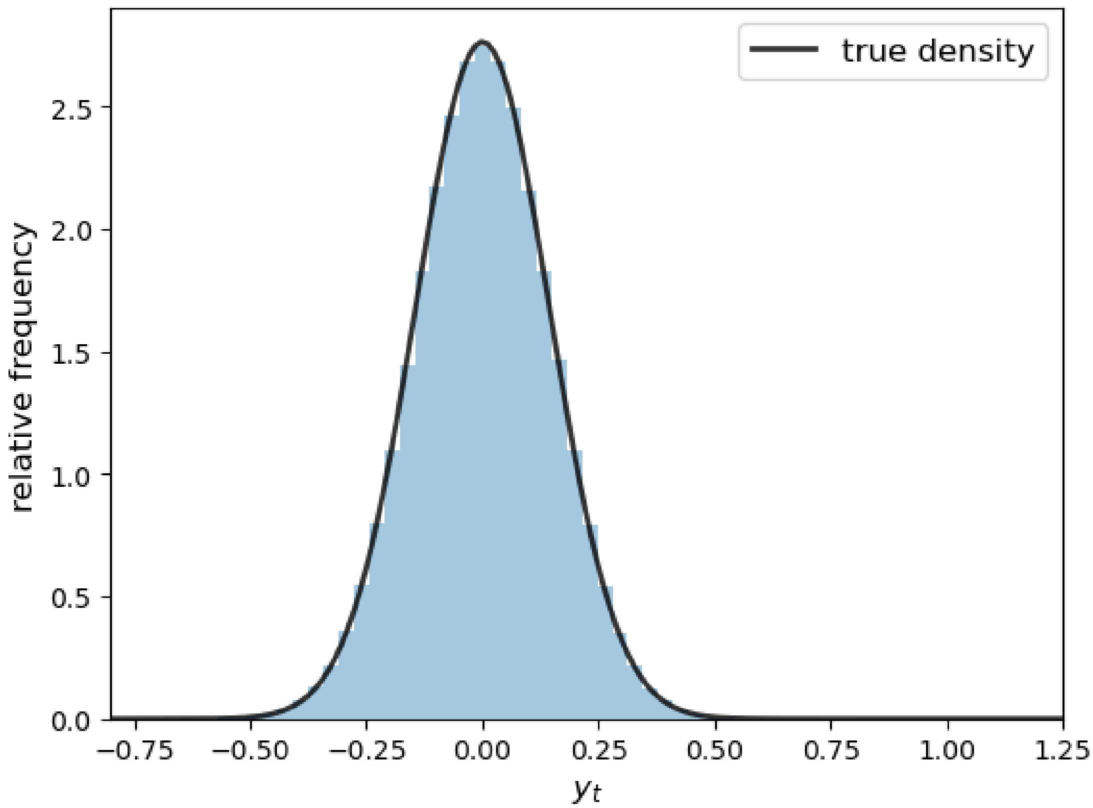
Here is another figure, this time with 100 observations

```
t = 100
cross_section_plot(A_2, C_2, G_2, T=t)
```



Let's now try with 500,000 observations, showing only the histogram (without rotation)

```
T = 100
ymin=-0.8
ymax=1.25
sample_size = 500_000

ar = LinearStateSpace(A_2, C_2, G_2, mu_0=np.ones(4))
fig, ax = plt.subplots()
x, y = ar.simulate(sample_size)
mu_x, mu_y, Sigma_x, Sigma_y, Sigma_yx = ar.stationary_distributions()
f_y = norm(loc=float(mu_y.item()), scale=float(np.sqrt(Sigma_y.item())))
y = y.flatten()
ygrid = np.linspace(ymin, ymax, 150)

ax.hist(y, bins=50, density=True, alpha=0.4)
ax.plot(ygrid, f_y.pdf(ygrid), 'k-', lw=2, alpha=0.8, label='true density')
ax.set_xlim(ymin, ymax)
ax.set_xlabel('$y_t$', fontsize=12)
ax.set_ylabel('relative frequency', fontsize=12)
ax.legend(fontsize=12)
plt.show()
```



The black line is the population density of y_T calculated from (36.10).

The histogram and population distribution are close, as expected.

By looking at the figures and experimenting with parameters, you will gain a feel for how the population distribution depends on the model primitives *listed above*, as intermediated by the distribution's parameters.

Ensemble Means

In the preceding figure, we approximated the population distribution of y_T by

1. generating I sample paths (i.e., time series) where I is a large number
2. recording each observation y^i_T
3. histogramming this sample

Just as the histogram approximates the population distribution, the **ensemble** or **cross-sectional average**

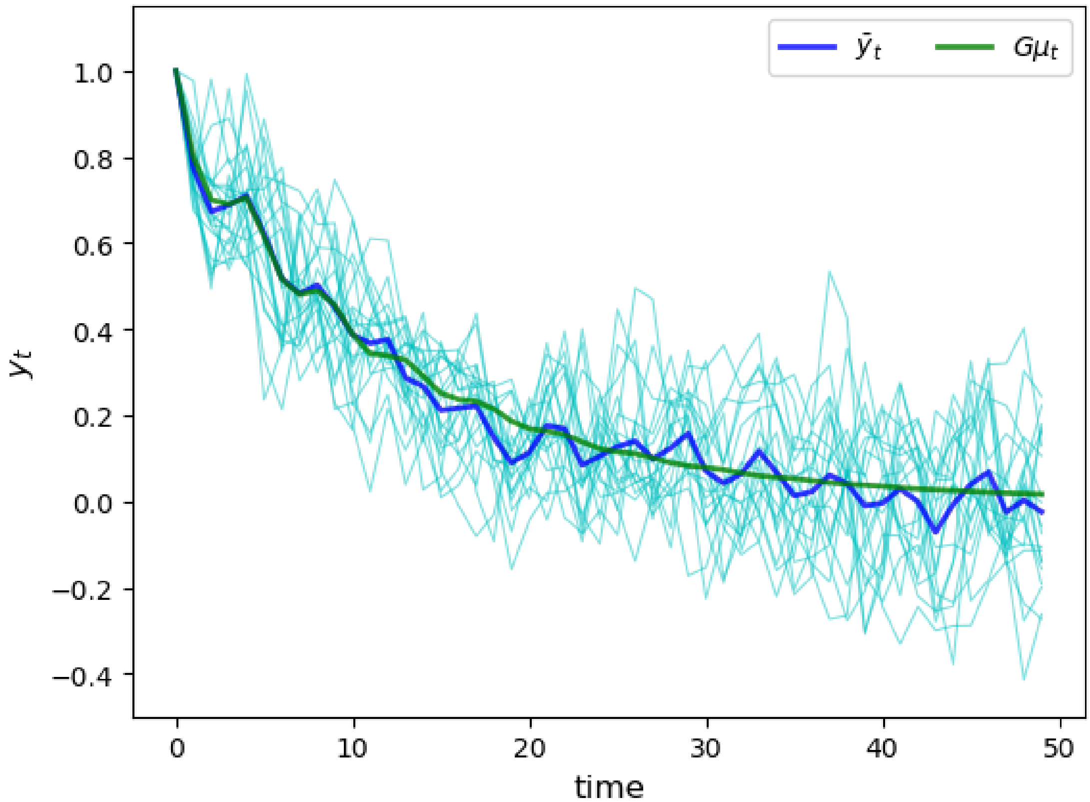
$$\bar{y}_T := \frac{1}{I} \sum_{i=1}^{I} y^i_T$$

approximates the expectation $\mathbb{E}[y_T] = G\mu_T$ (as implied by the law of large numbers).

Here's a simulation comparing the ensemble averages and population means at time points $t = 0, \ldots, 50$.

The parameters are the same as for the preceding figures, and the sample size is relatively small ($I = 20$).

```python
I = 20
T = 50
ymin = -0.5
ymax = 1.15

ar = LinearStateSpace(A_2, C_2, G_2, mu_0=np.ones(4))

fig, ax = plt.subplots()

ensemble_mean = np.zeros(T)
for i in range(I):
    x, y = ar.simulate(ts_length=T)
    y = y.flatten()
    ax.plot(y, 'c-', lw=0.8, alpha=0.5)
    ensemble_mean = ensemble_mean + y

ensemble_mean = ensemble_mean / I
ax.plot(ensemble_mean, color='b', lw=2, alpha=0.8, label='$\\bar y_t$')
m = ar.moment_sequence()

population_means = []
for t in range(T):
    μ_x, μ_y, Σ_x, Σ_y = next(m)
    population_means.append(float(μ_y.item()))

ax.plot(population_means, color='g', lw=2, alpha=0.8, label=r'$G\mu_t$')
ax.set_ylim(ymin, ymax)
ax.set_xlabel('time', fontsize=12)
ax.set_ylabel('$y_t$', fontsize=12)
ax.legend(ncol=2)
plt.show()
```



The ensemble mean for x_t is

$$\bar{x}_T := \frac{1}{I} \sum_{i=1}^{I} x_T^i \to \mu_T \qquad (I \to \infty)$$

The limit μ_T is a "long-run average".

(By *long-run average* we mean the average for an infinite ($I = \infty$) number of sample x_T's)

Another application of the law of large numbers assures us that

$$\frac{1}{I} \sum_{i=1}^{I} (x_T^i - \bar{x}_T)(x_T^i - \bar{x}_T)' \to \Sigma_T \qquad (I \to \infty)$$

36.3.4 Joint Distributions

In the preceding discussion, we looked at the distributions of x_t and y_t in isolation.

This gives us useful information but doesn't allow us to answer questions like

- what's the probability that $x_t \geq 0$ for all t?
- what's the probability that the process $\{y_t\}$ exceeds some value a before falling below b?
- etc., etc.

Such questions concern the *joint distributions* of these sequences.

To compute the joint distribution of $x_0, x_1, \ldots, x_T$, recall that joint and conditional densities are linked by the rule

$$p(x, y) = p(y \,|\, x) p(x) \qquad \text{(joint } = \text{ conditional } \times \text{ marginal)}$$

From this rule we get $p(x_0, x_1) = p(x_1 \mid x_0)p(x_0)$.

The Markov property $p(x_t \mid x_{t-1}, \ldots, x_0) = p(x_t \mid x_{t-1})$ and repeated applications of the preceding rule lead us to

$$p(x_0, x_1, \ldots, x_T) = p(x_0) \prod_{t=0}^{T-1} p(x_{t+1} \mid x_t)$$

The marginal $p(x_0)$ is just the primitive $N(\mu_0, \Sigma_0)$.

In view of (36.1), the conditional densities are

$$p(x_{t+1} \mid x_t) = N(Ax_t, CC')$$

Autocovariance Functions

An important object related to the joint distribution is the **autocovariance function**

$$\Sigma_{t+j,t} := \mathbb{E}[(x_{t+j} - \mu_{t+j})(x_t - \mu_t)'] \tag{36.11}$$

Elementary calculations show that

$$\Sigma_{t+j,t} = A^j \Sigma_t \tag{36.12}$$

Notice that $\Sigma_{t+j,t}$ in general depends on both j, the gap between the two dates, and t, the earlier date.

36.4 Stationarity and Ergodicity

Stationarity and ergodicity are two properties that, when they hold, greatly aid analysis of linear state space models.

Let's start with the intuition.

36.4.1 Visualizing Stability

Let's look at some more time series from the same model that we analyzed above.

This picture shows cross-sectional distributions for y at times T, T', T''

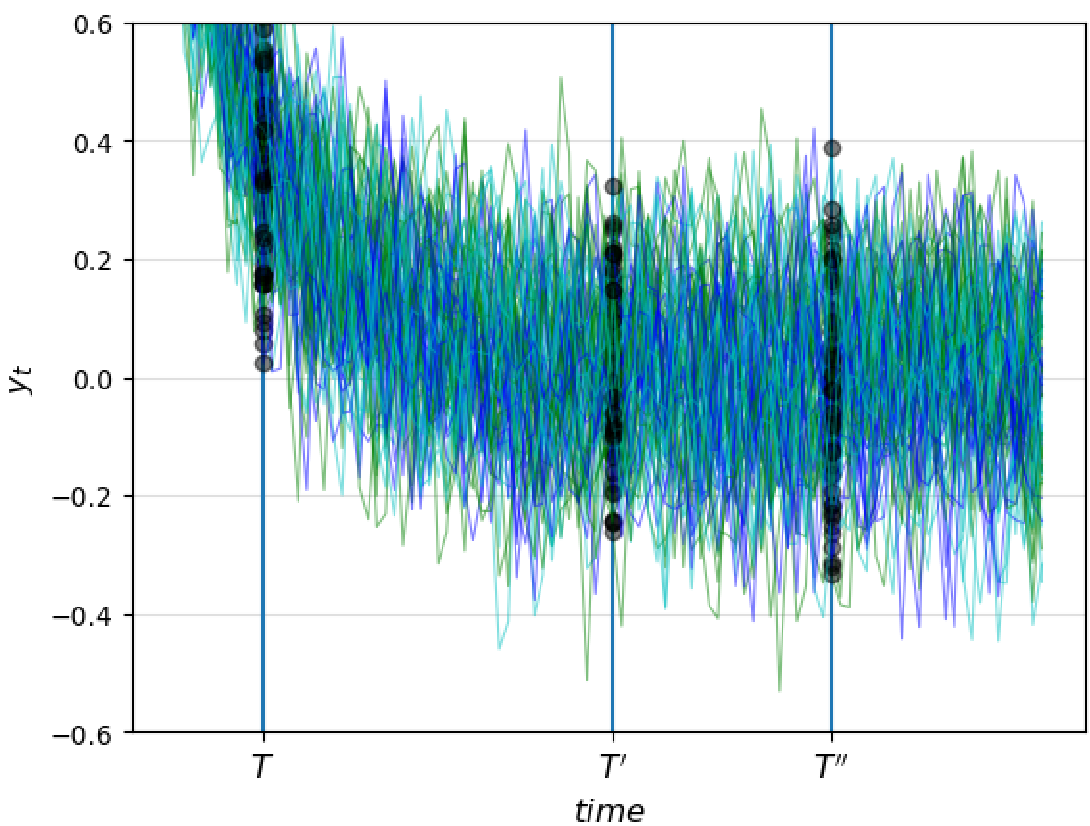
```python
def cross_plot(A,
            C,
            G,
            steady_state='False',
            T0 = 10,
            T1 = 50,
            T2 = 75,
            T4 = 100):

    ar = LinearStateSpace(A, C, G, mu_0=np.ones(4))

    if steady_state == 'True':
        μ_x, μ_y, Σ_x, Σ_y, Σ_yx = ar.stationary_distributions()
        ar_state = LinearStateSpace(A, C, G, mu_0=μ_x, Sigma_0=Σ_x)

    ymin, ymax = -0.6, 0.6
    fig, ax = plt.subplots()
```

(continues on next page)

(continued from previous page)

```
    ax.grid(alpha=0.4)
    ax.set_ylim(ymin, ymax)
    ax.set_ylabel('$y_t$', fontsize=12)
    ax.set_xlabel('$time$', fontsize=12)

    ax.vlines((T0, T1, T2), -1.5, 1.5)
    ax.set_xticks((T0, T1, T2))
    ax.set_xticklabels(("$T$", "$T'$", "$T''$"), fontsize=12)
    for i in range(80):
        rcolor = random.choice(('c', 'g', 'b'))

        if steady_state == 'True':
            x, y = ar_state.simulate(ts_length=T4)
        else:
            x, y = ar.simulate(ts_length=T4)

        y = y.flatten()
        ax.plot(y, color=rcolor, lw=0.8, alpha=0.5)
        ax.plot((T0, T1, T2), (y[T0], y[T1], y[T2],), 'ko', alpha=0.5)
    plt.show()
```

```
cross_plot(A_2, C_2, G_2)
```



Note how the time series "settle down" in the sense that the distributions at T' and T'' are relatively similar to each other — but unlike the distribution at T.

Apparently, the distributions of y_t converge to a fixed long-run distribution as $t \to \infty$.

When such a distribution exists it is called a **stationary distribution**.

36.4.2 Stationary Distributions

In our setting, a distribution ψ_∞ is said to be **stationary** for x_t if

$$x_t \sim \psi_\infty \quad \text{and} \quad x_{t+1} = Ax_t + Cw_{t+1} \quad \implies \quad x_{t+1} \sim \psi_\infty$$

Since

1. in the present case, all distributions are Gaussian
2. a Gaussian distribution is pinned down by its mean and variance-covariance matrix

we can restate the definition as follows: ψ_∞ is stationary for x_t if

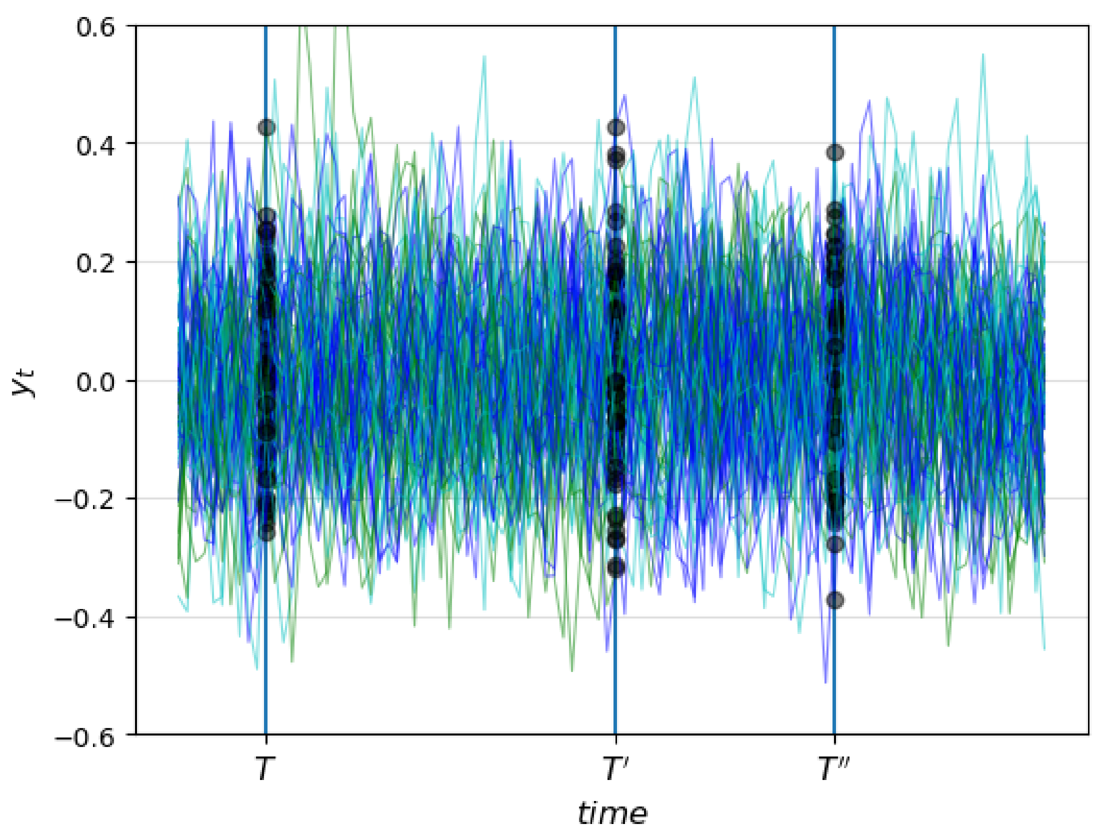
$$\psi_\infty = N(\mu_\infty, \Sigma_\infty)$$

where μ_∞ and Σ_∞ are fixed points of (36.4) and (36.5) respectively.

36.4.3 Covariance Stationary Processes

Let's see what happens to the preceding figure if we start x_0 at the stationary distribution.

```
cross_plot(A_2, C_2, G_2, steady_state='True')
```



Now the differences in the observed distributions at T, T' and T'' come entirely from random fluctuations due to the finite sample size.

By

- our choosing $x_0 \sim N(\mu_\infty, \Sigma_\infty)$
- the definitions of μ_∞ and Σ_∞ as fixed points of (36.4) and (36.5) respectively

we've ensured that

$$\mu_t = \mu_\infty \quad \text{and} \quad \Sigma_t = \Sigma_\infty \quad \text{for all } t$$

Moreover, in view of (36.12), the autocovariance function takes the form $\Sigma_{t+j,t} = A^j \Sigma_\infty$, which depends on j but not on t.

This motivates the following definition.

A process $\{x_t\}$ is said to be **covariance stationary** if

- both μ_t and Σ_t are constant in t
- $\Sigma_{t+j,t}$ depends on the time gap j but not on time t

In our setting, $\{x_t\}$ will be covariance stationary if μ_0, Σ_0, A, C assume values that imply that none of $\mu_t, \Sigma_t, \Sigma_{t+j,t}$ depends on t.

36.4.4 Conditions for Stationarity

The Globally Stable Case

The difference equation $\mu_{t+1} = A\mu_t$ is known to have *unique* fixed point $\mu_\infty = 0$ if all eigenvalues of A have moduli strictly less than unity.

That is, if `(np.absolute(np.linalg.eigvals(A)) < 1).all() == True`.

The difference equation (36.5) also has a unique fixed point in this case, and, moreover

$$\mu_t \to \mu_\infty = 0 \quad \text{and} \quad \Sigma_t \to \Sigma_\infty \quad \text{as} \quad t \to \infty$$

regardless of the initial conditions μ_0 and Σ_0.

This is the *globally stable case* — see these notes for more a theoretical treatment.

However, global stability is more than we need for stationary solutions, and often more than we want.

To illustrate, consider *our second order difference equation example*.

Here the state is $x_t = \begin{bmatrix} 1 & y_t & y_{t-1} \end{bmatrix}'$.

Because of the constant first component in the state vector, we will never have $\mu_t \to 0$.

How can we find stationary solutions that respect a constant state component?

Processes with a Constant State Component

To investigate such a process, suppose that A and C take the form

$$A = \begin{bmatrix} A_1 & a \\ 0 & 1 \end{bmatrix} \qquad C = \begin{bmatrix} C_1 \\ 0 \end{bmatrix}$$

where

- A_1 is an $(n-1) \times (n-1)$ matrix
- a is an $(n-1) \times 1$ column vector

Let $x_t = \begin{bmatrix} x_{1t}' & 1 \end{bmatrix}'$ where x_{1t} is $(n-1) \times 1$.

It follows that

$$x_{1,t+1} = A_1 x_{1t} + a + C_1 w_{t+1}$$

Let $\mu_{1t} = \mathbb{E}[x_{1t}]$ and take expectations on both sides of this expression to get

$$\mu_{1,t+1} = A_1 \mu_{1,t} + a \tag{36.13}$$

Assume now that the moduli of the eigenvalues of A_1 are all strictly less than one.

Then (36.13) has a unique stationary solution, namely,

$$\mu_{1\infty} = (I - A_1)^{-1} a$$

The stationary value of μ_t itself is then $\mu_\infty := \begin{bmatrix} \mu_{1\infty}' & 1 \end{bmatrix}'$.

The stationary values of Σ_t and $\Sigma_{t+j,t}$ satisfy

$$\begin{aligned} \Sigma_\infty &= A \Sigma_\infty A' + CC' \\ \Sigma_{t+j,t} &= A^j \Sigma_\infty \end{aligned}$$

Notice that here $\Sigma_{t+j,t}$ depends on the time gap j but not on calendar time t.

In conclusion, if

- $x_0 \sim N(\mu_\infty, \Sigma_\infty)$ and
- the moduli of the eigenvalues of A_1 are all strictly less than unity

then the $\{x_t\}$ process is covariance stationary, with constant state component.

> **Note**
>
> If the eigenvalues of A_1 are less than unity in modulus, then (a) starting from any initial value, the mean and variance-covariance matrix both converge to their stationary values; and (b) iterations on (36.5) converge to the fixed point of the *discrete Lyapunov equation* in the first line of (36.14).

36.4.5 Ergodicity

Let's suppose that we're working with a covariance stationary process.

In this case, we know that the ensemble mean will converge to μ_∞ as the sample size I approaches infinity.

Averages over Time

Ensemble averages across simulations are interesting theoretically, but in real life, we usually observe only a *single* realization $\{x_t, y_t\}_{t=0}^T$.

So now let's take a single realization and form the time-series averages

$$\bar{x} := \frac{1}{T}\sum_{t=1}^{T} x_t \quad \text{and} \quad \bar{y} := \frac{1}{T}\sum_{t=1}^{T} y_t$$

Do these time series averages converge to something interpretable in terms of our basic state-space representation?

The answer depends on something called *ergodicity*.

Ergodicity is the property that time series and ensemble averages coincide.

More formally, ergodicity implies that time series sample averages converge to their expectation under the stationary distribution.

In particular,

- $\frac{1}{T}\sum_{t=1}^{T} x_t \to \mu_\infty$
- $\frac{1}{T}\sum_{t=1}^{T} (x_t - \bar{x}_T)(x_t - \bar{x}_T)' \to \Sigma_\infty$
- $\frac{1}{T}\sum_{t=1}^{T} (x_{t+j} - \bar{x}_T)(x_t - \bar{x}_T)' \to A^j \Sigma_\infty$

In our linear Gaussian setting, any covariance stationary process is also ergodic.

36.5 Noisy Observations

In some settings, the observation equation $y_t = Gx_t$ is modified to include an error term.

Often this error term represents the idea that the true state can only be observed imperfectly.

To include an error term in the observation we introduce

- An IID sequence of $\ell \times 1$ random vectors $v_t \sim N(0, I)$.
- A $k \times \ell$ matrix H.

and extend the linear state-space system to

$$\begin{aligned} x_{t+1} &= Ax_t + Cw_{t+1} \\ y_t &= Gx_t + Hv_t \\ x_0 &\sim N(\mu_0, \Sigma_0) \end{aligned}$$

The sequence $\{v_t\}$ is assumed to be independent of $\{w_t\}$.

The process $\{x_t\}$ is not modified by noise in the observation equation and its moments, distributions and stability properties remain the same.

The unconditional moments of y_t from (36.6) and (36.7) now become

$$\mathbb{E}[y_t] = \mathbb{E}[Gx_t + Hv_t] = G\mu_t \tag{36.14}$$

The variance-covariance matrix of y_t is easily shown to be

$$\text{Var}[y_t] = \text{Var}[Gx_t + Hv_t] = G\Sigma_t G' + HH' \tag{36.15}$$

The distribution of y_t is therefore

$$y_t \sim N(G\mu_t, G\Sigma_t G' + HH')$$

36.6 Prediction

The theory of prediction for linear state space systems is elegant and simple.

36.6.1 Forecasting Formulas – Conditional Means

The natural way to predict variables is to use conditional distributions.

For example, the optimal forecast of x_{t+1} given information known at time t is

$$\mathbb{E}_t[x_{t+1}] := \mathbb{E}[x_{t+1} \mid x_t, x_{t-1}, \dots, x_0] = Ax_t$$

The right-hand side follows from $x_{t+1} = Ax_t + Cw_{t+1}$ and the fact that w_{t+1} is zero mean and independent of $x_t, x_{t-1}, \dots, x_0$.

That $\mathbb{E}_t[x_{t+1}] = \mathbb{E}[x_{t+1} \mid x_t]$ is an implication of $\{x_t\}$ having the **Markov property**.

The one-step-ahead forecast error is

$$x_{t+1} - \mathbb{E}_t[x_{t+1}] = Cw_{t+1}$$

The covariance matrix of the forecast error is

$$\mathbb{E}[(x_{t+1} - \mathbb{E}_t[x_{t+1}])(x_{t+1} - \mathbb{E}_t[x_{t+1}])'] = CC'$$

More generally, we'd like to compute the j-step ahead forecasts $\mathbb{E}_t[x_{t+j}]$ and $\mathbb{E}_t[y_{t+j}]$.

With a bit of algebra, we obtain

$$x_{t+j} = A^j x_t + A^{j-1}Cw_{t+1} + A^{j-2}Cw_{t+2} + \cdots + A^0 Cw_{t+j}$$

In view of the IID property, current and past state values provide no information about future values of the shock.

Hence $\mathbb{E}_t[w_{t+k}] = \mathbb{E}[w_{t+k}] = 0$.

It now follows from linearity of expectations that the j-step ahead forecast of x is

$$\mathbb{E}_t[x_{t+j}] = A^j x_t$$

The j-step ahead forecast of y is therefore

$$\mathbb{E}_t[y_{t+j}] = \mathbb{E}_t[Gx_{t+j} + Hv_{t+j}] = GA^j x_t$$

36.6.2 Covariance of Prediction Errors

It is useful to obtain the covariance matrix of the vector of j-step-ahead prediction errors

$$x_{t+j} - \mathbb{E}_t[x_{t+j}] = \sum_{s=0}^{j-1} A^s C w_{t-s+j} \tag{36.16}$$

Evidently,

$$V_j := \mathbb{E}_t[(x_{t+j} - \mathbb{E}_t[x_{t+j}])(x_{t+j} - \mathbb{E}_t[x_{t+j}])'] = \sum_{k=0}^{j-1} A^k C C' A^{k'} \tag{36.17}$$

V_j defined in (36.17) can be calculated recursively via $V_1 = CC'$ and

$$V_j = CC' + AV_{j-1}A', \quad j \geq 2 \tag{36.18}$$

V_j is the **conditional covariance matrix** of the errors in forecasting x_{t+j}, conditioned on time t information x_t.

Under particular conditions, V_j converges to

$$V_\infty = CC' + AV_\infty A' \tag{36.19}$$

Equation (36.19) is an example of a **discrete Lyapunov** equation in the covariance matrix V_∞.

A sufficient condition for V_j to converge is that the eigenvalues of A be strictly less than one in modulus.

Weaker sufficient conditions for convergence associate eigenvalues equaling or exceeding one in modulus with elements of C that equal 0.

36.7 Code

Our preceding simulations and calculations are based on code in the file lss.py from the QuantEcon.py package.

The code implements a class for handling linear state space models (simulations, calculating moments, etc.).

One Python construct you might not be familiar with is the use of a generator function in the method `moment_sequence()`.

Go back and read the relevant documentation if you've forgotten how generator functions work.

Examples of usage are given in the solutions to the exercises.

36.8 Exercises

Exercise 36.8.1

In several contexts, we want to compute forecasts of geometric sums of future random variables governed by the linear state-space system (36.1).

We want the following objects

- Forecast of a geometric sum of future x's, or $\mathbb{E}_t\left[\sum_{j=0}^\infty \beta^j x_{t+j}\right]$.
- Forecast of a geometric sum of future y's, or $\mathbb{E}_t\left[\sum_{j=0}^\infty \beta^j y_{t+j}\right]$.

These objects are important components of some famous and interesting dynamic models.

For example,

- if $\{y_t\}$ is a stream of dividends, then $\mathbb{E}\left[\sum_{j=0}^{\infty} \beta^j y_{t+j} | x_t\right]$ is a model of a stock price
- if $\{y_t\}$ is the money supply, then $\mathbb{E}\left[\sum_{j=0}^{\infty} \beta^j y_{t+j} | x_t\right]$ is a model of the price level

Show that:

$$\mathbb{E}_t\left[\sum_{j=0}^{\infty} \beta^j x_{t+j}\right] = [I - \beta A]^{-1} x_t$$

and

$$\mathbb{E}_t\left[\sum_{j=0}^{\infty} \beta^j y_{t+j}\right] = G[I - \beta A]^{-1} x_t$$

what must the modulus for every eigenvalue of A be less than?

Solution

Suppose that every eigenvalue of A has modulus strictly less than $\frac{1}{\beta}$.

It *then follows* that $I + \beta A + \beta^2 A^2 + \cdots = [I - \beta A]^{-1}$.

This leads to our formulas:

- Forecast of a geometric sum of future x's

$$\mathbb{E}_t\left[\sum_{j=0}^{\infty} \beta^j x_{t+j}\right] = [I + \beta A + \beta^2 A^2 + \cdots] x_t = [I - \beta A]^{-1} x_t$$

- Forecast of a geometric sum of future y's

$$\mathbb{E}_t\left[\sum_{j=0}^{\infty} \beta^j y_{t+j}\right] = G[I + \beta A + \beta^2 A^2 + \cdots] x_t = G[I - \beta A]^{-1} x_t$$

CHAPTER

THIRTYSEVEN

SAMUELSON MULTIPLIER-ACCELERATOR

Contents

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
```

37.1 Overview

This lecture creates non-stochastic and stochastic versions of Paul Samuelson's celebrated multiplier accelerator model [Samuelson, 1939].

In doing so, we extend the example of the Solow model class in our second OOP lecture.

Our objectives are to

- provide a more detailed example of OOP and classes
- review a famous model
- review linear difference equations, both deterministic and stochastic

Let's start with some standard imports:

```
import matplotlib.pyplot as plt
import numpy as np
```

We'll also use the following for various tasks described below:

```
from quantecon import LinearStateSpace
import cmath
import math
import sympy
from sympy import Symbol, init_printing
from cmath import sqrt
```

37.1.1 Samuelson's model

Samuelson used a *second-order linear difference equation* to represent a model of national output based on three components:

- a *national output identity* asserting that national output or national income is the sum of consumption plus investment plus government purchases.
- a Keynesian *consumption function* asserting that consumption at time t is equal to a constant times national output at time $t-1$.
- an investment *accelerator* asserting that investment at time t equals a constant called the *accelerator coefficient* times the difference in output between period $t-1$ and $t-2$.

Consumption plus investment plus government purchases constitute *aggregate demand,* which automatically calls forth an equal amount of *aggregate supply*.

(To read about linear difference equations see here or chapter IX of [Sargent, 1987].)

Samuelson used the model to analyze how particular values of the marginal propensity to consume and the accelerator coefficient might give rise to transient **business cycles** in national output.

Possible dynamic properties include

- smooth convergence to a constant level of output
- damped business cycles that eventually converge to a constant level of output
- persistent business cycles that neither dampen nor explode

Later we present an extension that adds a random shock to the right side of the national income identity representing random fluctuations in aggregate demand.

This modification makes national output become governed by a second-order **stochastic linear difference equation** that, with appropriate parameter values, gives rise to recurrent irregular business cycles.

(To read about stochastic linear difference equations see chapter XI of [Sargent, 1987].)

37.2 Details

Let's assume that

- $\{G_t\}$ is a sequence of levels of government expenditures – we'll start by setting $G_t = G$ for all t.
- $\{C_t\}$ is a sequence of levels of aggregate consumption expenditures, a key endogenous variable in the model.
- $\{I_t\}$ is a sequence of rates of investment, another key endogenous variable.
- $\{Y_t\}$ is a sequence of levels of national income, yet another endogenous variable.
- α is the marginal propensity to consume in the Keynesian consumption function $C_t = \alpha Y_{t-1} + \gamma$.
- β is the "accelerator coefficient" in the "investment accelerator" $I_t = \beta(Y_{t-1} - Y_{t-2})$.
- $\{\epsilon_t\}$ is an IID sequence standard normal random variables.
- $\sigma \geq 0$ is a "volatility" parameter — setting $\sigma = 0$ recovers the non-stochastic case that we'll start with.

The model combines the consumption function

$$C_t = \alpha Y_{t-1} + \gamma \tag{37.1}$$

with the investment accelerator

$$I_t = \beta(Y_{t-1} - Y_{t-2}) \tag{37.2}$$

and the national income identity

$$Y_t = C_t + I_t + G_t \tag{37.3}$$

- The parameter α is peoples' **marginal propensity to consume** out of income - equation (37.1) asserts that people consume a fraction of $\alpha \in (0, 1)$ of each additional dollar of income.
- The parameter $\beta > 0$ is the investment accelerator coefficient - equation (37.2) asserts that people invest in physical capital when income is increasing and disinvest when it is decreasing.

Equations (37.1), (37.2), and (37.3) imply the following second-order linear difference equation for national income:

$$Y_t = (\alpha + \beta)Y_{t-1} - \beta Y_{t-2} + (\gamma + G_t)$$

or

$$Y_t = \rho_1 Y_{t-1} + \rho_2 Y_{t-2} + (\gamma + G_t) \tag{37.4}$$

where $\rho_1 = (\alpha + \beta)$ and $\rho_2 = -\beta$.

To complete the model, we require two *initial conditions*.

If the model is to generate time series for $t = 0, \dots, T$, we require initial values

$$Y_{-1} = \bar{Y}_{-1}, \quad Y_{-2} = \bar{Y}_{-2}$$

We'll ordinarily set the parameters (α, β) so that starting from an arbitrary pair of initial conditions $(\bar{Y}_{-1}, \bar{Y}_{-2})$, national income Y_t converges to a constant value as t becomes large.

We are interested in studying

- the transient fluctuations in Y_t as it converges to its **steady state** level
- the *rate* at which it converges to a steady state level

The deterministic version of the model described so far — meaning that no random shocks hit aggregate demand — has only transient fluctuations.

We can convert the model to one that has persistent irregular fluctuations by adding a random shock to aggregate demand.

37.2.1 Stochastic version of the model

We create a *random* or *stochastic* version of the model by adding a random process of *shocks* or *disturbances* $\{\sigma\epsilon_t\}$ to the right side of equation (37.4), leading to the *second-order scalar linear stochastic difference equation*:

$$Y_t = (\alpha + \beta)Y_{t-1} - \beta Y_{t-2} + (\gamma + G_t) + \sigma\epsilon_t \tag{37.5}$$

37.2.2 Mathematical analysis of the model

To get started, let's set $G_t \equiv 0$, $\sigma = 0$, and $\gamma = 0$.

Then we can write equation (37.5) as

$$Y_t = \rho_1 Y_{t-1} + \rho_2 Y_{t-2}$$

or

$$Y_{t+2} - \rho_1 Y_{t+1} - \rho_2 Y_t = 0 \tag{37.6}$$

To discover the properties of the solution of (37.6), it is useful first to form the **characteristic polynomial** for (37.6):

$$z^2 - \rho_1 z - \rho_2 \tag{37.7}$$

where z is possibly a complex number.

We want to find the two **zeros** (a.k.a. **roots**) – namely λ_1, λ_2 – of the characteristic polynomial.

These are two special values of z, say $z = \lambda_1$ and $z = \lambda_2$, such that if we set z equal to one of these values in expression (37.7), the characteristic polynomial (37.7) equals zero:

$$z^2 - \rho_1 z - \rho_2 = (z - \lambda_1)(z - \lambda_2) = 0 \tag{37.8}$$

Equation (37.8) is said to *factor* the characteristic polynomial.

When the roots are complex, they will occur as a complex conjugate pair.

When the roots are complex, it is convenient to represent them in the polar form

$$\lambda_1 = re^{i\omega}, \ \lambda_2 = re^{-i\omega}$$

where r is the *amplitude* of the complex number and ω is its *angle* or *phase*.

These can also be represented as

$$\lambda_1 = r(cos(\omega) + i \sin(\omega))$$

$$\lambda_2 = r(cos(\omega) - i \sin(\omega))$$

(To read about the polar form, see here)

Given *initial conditions* Y_{-1}, Y_{-2}, we want to generate a *solution* of the difference equation (37.6).

It can be represented as

$$Y_t = \lambda_1^t c_1 + \lambda_2^t c_2$$

where c_1 and c_2 are constants that depend on the two initial conditions and on ρ_1, ρ_2.

When the roots are complex, it is useful to pursue the following calculations.

Notice that

$$
\begin{aligned}
Y_t &= c_1 (r e^{i\omega})^t + c_2 (r e^{-i\omega})^t \\
&= c_1 r^t e^{i\omega t} + c_2 r^t e^{-i\omega t} \\
&= c_1 r^t [\cos(\omega t) + i \sin(\omega t)] + c_2 r^t [\cos(\omega t) - i \sin(\omega t)] \\
&= (c_1 + c_2) r^t \cos(\omega t) + i (c_1 - c_2) r^t \sin(\omega t)
\end{aligned}
$$

The only way that Y_t can be a real number for each t is if $c_1 + c_2$ is a real number and $c_1 - c_2$ is an imaginary number.

This happens only when c_1 and c_2 are complex conjugates, in which case they can be written in the polar forms

$$
c_1 = v e^{i\theta}, \ \ c_2 = v e^{-i\theta}
$$

So we can write

$$
\begin{aligned}
Y_t = & \quad v e^{i\theta} r^t e^{i\omega t} + v e^{-i\theta} r^t e^{-i\omega t} \\
= & \quad v r^t [e^{i(\omega t + \theta)} + e^{-i(\omega t + \theta)}] \\
= & \quad 2 v r^t \cos(\omega t + \theta)
\end{aligned}
$$

where v and θ are constants that must be chosen to satisfy initial conditions for Y_{-1}, Y_{-2}.

This formula shows that when the roots are complex, Y_t displays oscillations with *period* $\check p = \frac{2\pi}{\omega}$ and *damping factor* r.

We say that $\check p$ is the *period* because in that amount of time the cosine wave $\cos(\omega t + \theta)$ goes through exactly one complete cycles.

(Draw a cosine function to convince yourself of this please)

Remark: Following [Samuelson, 1939], we want to choose the parameters α, β of the model so that the absolute values (of the possibly complex) roots λ_1, λ_2 of the characteristic polynomial are both strictly less than one:

$$
|\lambda_j| < 1 \quad \quad \text{for } j = 1, 2
$$

Remark: When both roots λ_1, λ_2 of the characteristic polynomial have absolute values strictly less than one, the absolute value of the larger one governs the rate of convergence to the steady state of the non stochastic version of the model.

37.2.3 Things this lecture does

We write a function to generate simulations of a $\{Y_t\}$ sequence as a function of time.

The function requires that we put in initial conditions for Y_{-1}, Y_{-2}.

The function checks that α, β are set so that λ_1, λ_2 are less than unity in absolute value (also called "modulus").

The function also tells us whether the roots are complex, and, if they are complex, returns both their real and complex parts.

If the roots are both real, the function returns their values.

We use our function written to simulate paths that are stochastic (when $\sigma > 0$).

We have written the function in a way that allows us to input $\{G_t\}$ paths of a few simple forms, e.g.,

- one time jumps in G at some time
- a permanent jump in G that occurs at some time

We proceed to use the Samuelson multiplier-accelerator model as a laboratory to make a simple OOP example.

The "state" that determines next period's Y_{t+1} is now not just the current value Y_t but also the once lagged value Y_{t-1}.

This involves a little more bookkeeping than is required in the Solow model class definition.

We use the Samuelson multiplier-accelerator model as a vehicle for teaching how we can gradually add more features to the class.

We want to have a method in the class that automatically generates a simulation, either non-stochastic ($\sigma = 0$) or stochastic ($\sigma > 0$).

We also show how to map the Samuelson model into a simple instance of the `LinearStateSpace` class described here.

We can use a `LinearStateSpace` instance to do various things that we did above with our homemade function and class.

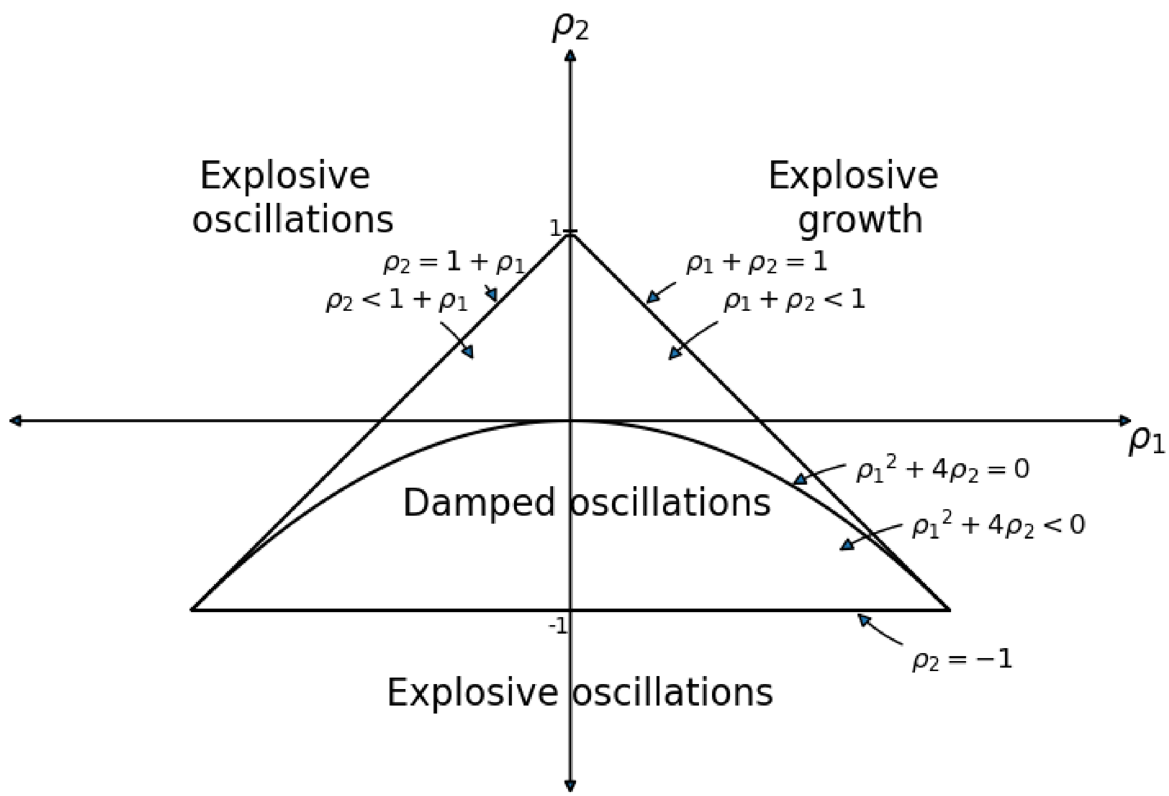
Among other things, we show by example that the eigenvalues of the matrix A that we use to form the instance of the `LinearStateSpace` class for the Samuelson model equal the roots of the characteristic polynomial (37.7) for the Samuelson multiplier accelerator model.

Here is the formula for the matrix A in the linear state space system in the case that government expenditures are a constant G:

$$A = \begin{bmatrix} 1 & 0 & 0 \\ \gamma + G & \rho_1 & \rho_2 \\ 0 & 1 & 0 \end{bmatrix}$$

37.3 Implementation

We'll start by drawing an informative graph from page 189 of [Sargent, 1987]



The graph portrays regions in which the (λ_1, λ_2) root pairs implied by the $(\rho_1 = (\alpha + \beta), \rho_2 = -\beta)$ difference equation parameter pairs in the Samuelson model are such that:

- (λ_1, λ_2) are complex with modulus less than 1 - in this case, the $\{Y_t\}$ sequence displays damped oscillations.
- (λ_1, λ_2) are both real, but one is strictly greater than 1 - this leads to explosive growth.
- (λ_1, λ_2) are both real, but one is strictly less than -1 - this leads to explosive oscillations.
- (λ_1, λ_2) are both real and both are less than 1 in absolute value - in this case, there is smooth convergence to the steady state without damped cycles.

Later we'll present the graph with a red mark showing the particular point implied by the setting of (α, β).

37.3.1 Function to describe implications of characteristic polynomial

```python
def categorize_solution(ρ1, ρ2):
    """
    This function takes values of ρ1 and ρ2 and uses them
    to classify the type of solution.
    """
    discriminant = ρ1**2 + 4 * ρ2
    if ρ2 > 1 + ρ1 or ρ2 < -1:
        return "Explosive oscillations"
    elif ρ1 + ρ2 > 1:
        return "Explosive growth"
    elif discriminant < 0:
        return "Damped oscillations"
    else:
        return "Steady state convergence"

def analyze_roots(α, β, verbose=True):
    """
    Unified function to calculate roots and analyze their properties.
    """
    ρ1 = α + β
    ρ2 = -β

    # Compute characteristic polynomial roots
    roots = np.roots([1, -ρ1, -ρ2])

    # Classify solution type
    solution_type = categorize_solution(ρ1, ρ2)

    # Determine root properties
    is_complex = all(isinstance(root, complex) for root in roots)
    is_stable = all(abs(root) < 1 for root in roots)

    if verbose:
        print(f"ρ1 = {ρ1:.2f}, ρ2 = {ρ2:.2f}")
        print(f"Roots: {[f'{root:.2f}' for root in roots]}")
        print(f"Root type: {'Complex' if is_complex else 'Real'}")
        print(f"Stability: {'Stable' if is_stable else 'Unstable'}")
        print(f"Solution type: {solution_type}")

    return {
        'roots': roots,
        'rho1': ρ1,
        'rho2': ρ2,
        'is_complex': is_complex,
        'is_stable': is_stable,
```

(continues on next page)

(continued from previous page)

```
        'solution_type': solution_type
    }
```

We also write a unified simulation function that can handle both non-stochastic and stochastic versions of the model.

It allows for government spending paths of a few simple forms which we specify via a dictionary `g_params`

```
def simulate_samuelson(
    y_0, y_1, α, β, γ=10, n=100, σ=0, g_params=None, seed=0
):
    """
    Unified simulation function for Samuelson model.

    Parameters:
    g_params: dict with keys 'g', 'g_t', 'duration' for government spending
    seed: random seed for reproducible results
    """
    analysis = analyze_roots(α, β, verbose=False)
    ρ1, ρ2 = analysis['rho1'], analysis['rho2']

    # Initialize time series
    y_t = [y_0, y_1]

    # Generate shocks if stochastic
    if σ > 0:
        np.random.seed(seed)
        ϵ = np.random.normal(0, 1, n)

    # Simulate forward
    for t in range(2, n):

        # Determine government spending
        g = 0
        if g_params:
            g_val, g_t_val = g_params.get('g', 0), g_params.get('g_t', 0)
            duration = g_params.get('duration', None)
            if duration == 'permanent' and t >= g_t_val:
                g = g_val
            elif duration == 'one-off' and t == g_t_val:
                g = g_val
            elif duration is None:
                g = g_val

        # Calculate next value
        y_next = ρ1 * y_t[t-1] + ρ2 * y_t[t-2] + γ + g
        if σ > 0:
            y_next += σ * ϵ[t]

        y_t.append(y_next)

    return y_t, analysis
```

We will use this function to run simulations of the model.

But before doing that, let's test the analysis function

```
analysis = analyze_roots(α=1.3, β=0.4)
```

```
ρ1 = 1.70, ρ2 = -0.40
Roots: ['1.42', '0.28']
Root type: Real
Stability: Unstable
Solution type: Explosive growth
```

37.3.2 Function for plotting paths

A useful function for our work below is

```
def plot_y(function=None):
    """Function plots path of Y_t"""

    plt.subplots(figsize=(10, 6))
    plt.plot(function)
    plt.xlabel("$t$")
    plt.ylabel("$Y_t$")
    plt.show()
```

37.3.3 Manual or "by hand" root calculations

The following function calculates roots of the characteristic polynomial using high school algebra.

(We'll calculate the roots in other ways later using `analyze_roots`.)

The function also plots a Y_t starting from initial conditions that we set

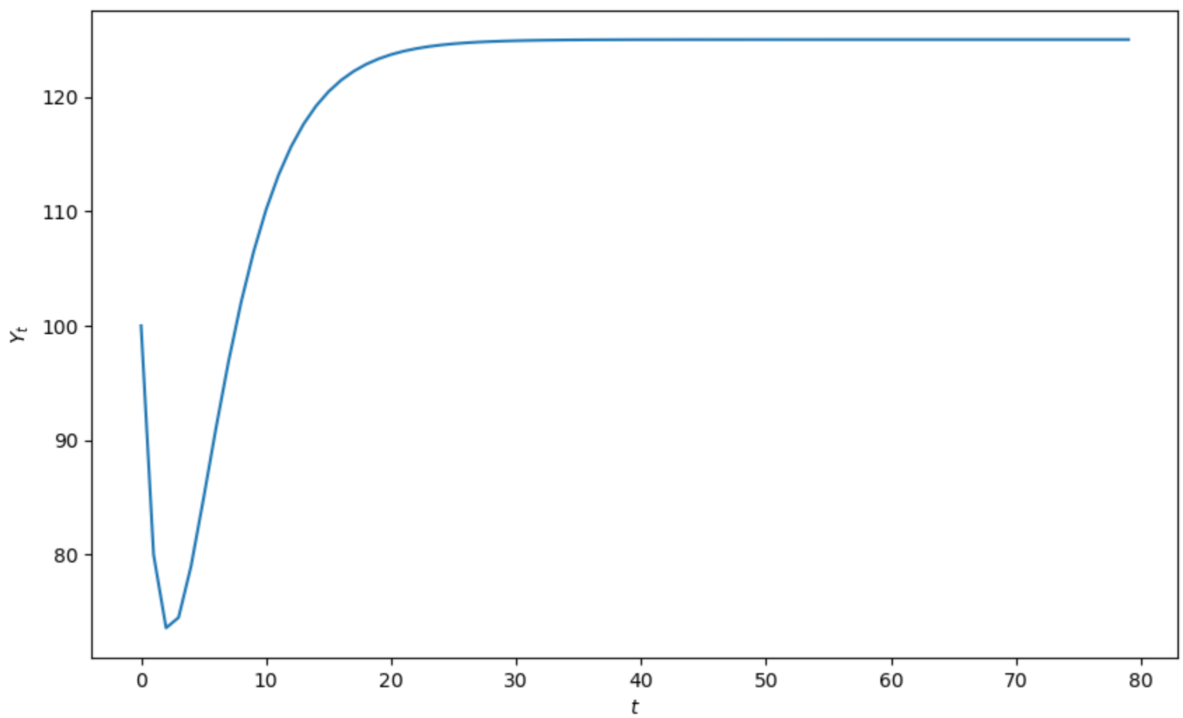
```
def y_nonstochastic(y_0=100, y_1=80, α=0.92, β=0.5, γ=10, n=80):
    """
    This function calculates the roots of the characteristic polynomial
    by hand and returns a path of y_t starting from initial conditions
    """
    roots = []

    ρ1 = α + β
    ρ2 = -β

    print(f"ρ_1 is {ρ1:.2f}")
    print(f"ρ_2 is {ρ2:.2f}")

    discriminant = ρ1**2 + 4 * ρ2

    if discriminant == 0:
        roots.append(-ρ1 / 2)
        print("Single real root: ")
        print("".join(f"{r:.2f}" for r in roots))
    elif discriminant > 0:
        roots.append((-ρ1 + sqrt(discriminant).real) / 2)
        roots.append((-ρ1 - sqrt(discriminant).real) / 2)
        print("Two real roots: ")
        print(" ".join(f"{r:.2f}" for r in roots))
    else:
        roots.append((-ρ1 + sqrt(discriminant)) / 2)
        roots.append((-ρ1 - sqrt(discriminant)) / 2)
        print("Two complex roots: ")
```

(continues on next page)

(continued from previous page)

```
        print(" ".join(f"{r.real:.2f}{r.imag:+.2f}j" for r in roots))

    if all(abs(root) < 1 for root in roots):
        print("Absolute values of roots are less than one")
    else:
        print("Absolute values of roots are not less than one")

    def transition(x, t):
        return ρ1 * x[t - 1] + ρ2 * x[t - 2] + γ

    y_t = [y_0, y_1]

    for t in range(2, n):
        y_t.append(transition(y_t, t))

    return y_t


plot_y(y_nonstochastic())
```

```
ρ_1 is 1.42
ρ_2 is -0.50
Two real roots:
-0.65 -0.77
Absolute values of roots are less than one
```



37.3.4 Reverse-engineering parameters to generate damped cycles

The next cell writes code that takes as inputs the modulus r and phase ϕ of a conjugate pair of complex numbers in polar form

$$\lambda_1 = r\exp(i\phi), \quad \lambda_2 = r\exp(-i\phi)$$

- The code assumes that these two complex numbers are the roots of the characteristic polynomial
- It then reverse-engineers (α, β) and (ρ_1, ρ_2), pairs that would generate those roots

```
def f(r, ϕ):
    """
    Takes modulus r and angle ϕ of complex number r exp(j ϕ)
    and creates ρ1 and ρ2 of characteristic polynomial for which
    r exp(j ϕ) and r exp(- j ϕ) are complex roots.

    Returns the multiplier coefficient $\alpha$
    and the accelerator coefficient $\beta$
    that verifies those roots.
    """
    # Create complex conjugate pair from polar coordinates
    g1 = cmath.rect(r, ϕ)
    g2 = cmath.rect(r, -ϕ)

    # Calculate corresponding ρ1, ρ2 parameters
    ρ1 = g1 + g2
    ρ2 = -g1 * g2

    # Derive α and β coefficients from ρ parameters
    β = -ρ2
    α = ρ1 - β
    return ρ1, ρ2, α, β
```

Now let's use the function in an example.

Here are the example parameters

```
r = 0.95

# Cycle period in time units
period = 10
ϕ = 2 * math.pi / period

# Apply the reverse-engineering function
ρ1, ρ2, α, β = f(r, ϕ)

print(f"α, β = {α:.2f}, {β:.2f}")
print(f"ρ1, ρ2 = {ρ1:.2f}, {ρ2:.2f}")
```

```
α, β = 0.63+0.00j, 0.90-0.00j
ρ1, ρ2 = 1.54+0.00j, -0.90+0.00j
```

The real parts of the roots are

```
print(f"ρ1 = {ρ1.real:.2f}, ρ2 = {ρ2.real:.2f}")
```

```
ρ1 = 1.54, ρ2 = -0.90
```

37.3.5 Root finding using numpy

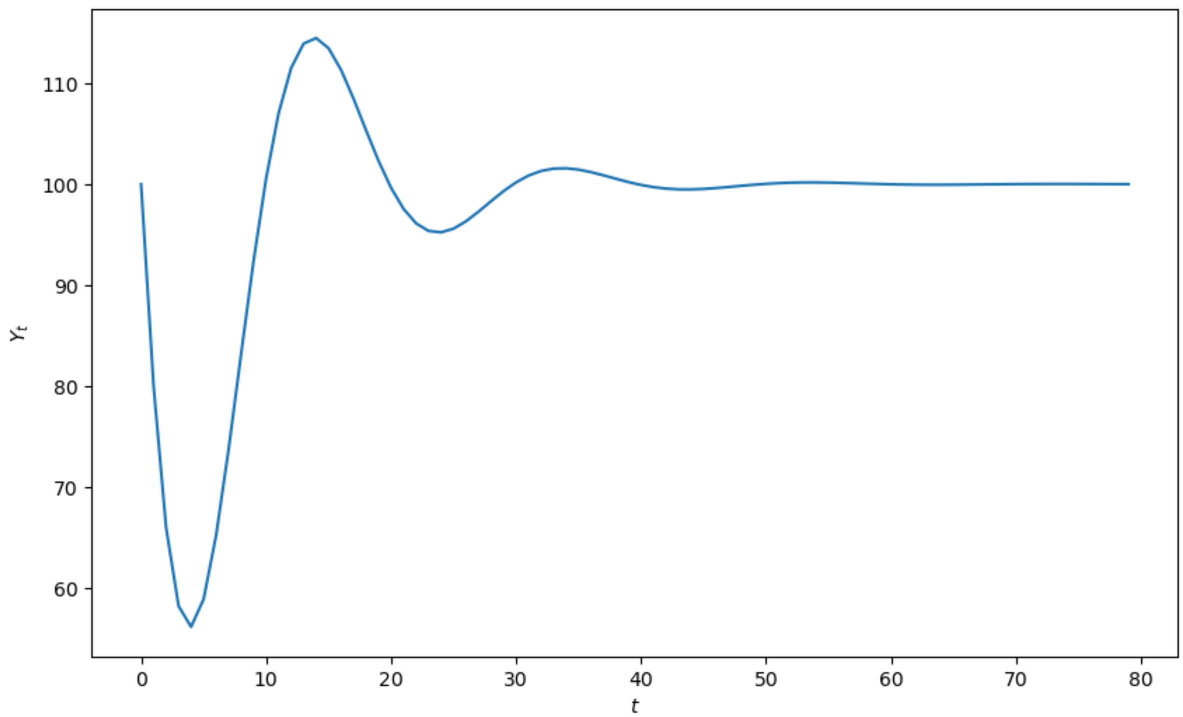
Here we'll use numpy to compute the roots of the characteristic polynomial

```
r1, r2 = np.roots([1, -ρ1, -ρ2])

p1 = cmath.polar(r1)
p2 = cmath.polar(r2)

print(f"r, φ = {r:.2f}, {φ:.2f}")
print(f"p1, p2 = ({p1[0]:.2f}, {p1[1]:.2f}), ({p2[0]:.2f}, {p2[1]:.2f})")

print(f"α, β = {α:.2f}, {β:.2f}")
print(f"ρ1, ρ2 = {ρ1:.2f}, {ρ2:.2f}")
```

```
r, φ = 0.95, 0.63
p1, p2 = (0.95, 0.63), (0.95, -0.63)
α, β = 0.63+0.00j, 0.90-0.00j
ρ1, ρ2 = 1.54+0.00j, -0.90+0.00j
```

```
def y_nonstochastic(y_0=100, y_1=80, α=0.9, β=0.8, γ=10, n=80):
    """
    This function enlists numpy to calculate the roots of the characteristic
    polynomial.
    """

    y_series, analysis = simulate_samuelson(y_0, y_1, α, β, γ, n, 0, None, 42)

    print(f"Solution type: {analysis['solution_type']}")
    print(f"Roots are {analysis['roots']}")
    print(f"Root type: {'Complex' if analysis['is_complex'] else 'Real'}")
    print(f"Stability: {'Stable' if analysis['is_stable'] else 'Unstable'}")

    return y_series


plot_y(y_nonstochastic())
```

```
Solution type: Damped oscillations
Roots are [0.85+0.27838822j 0.85-0.27838822j]
Root type: Complex
Stability: Stable
```



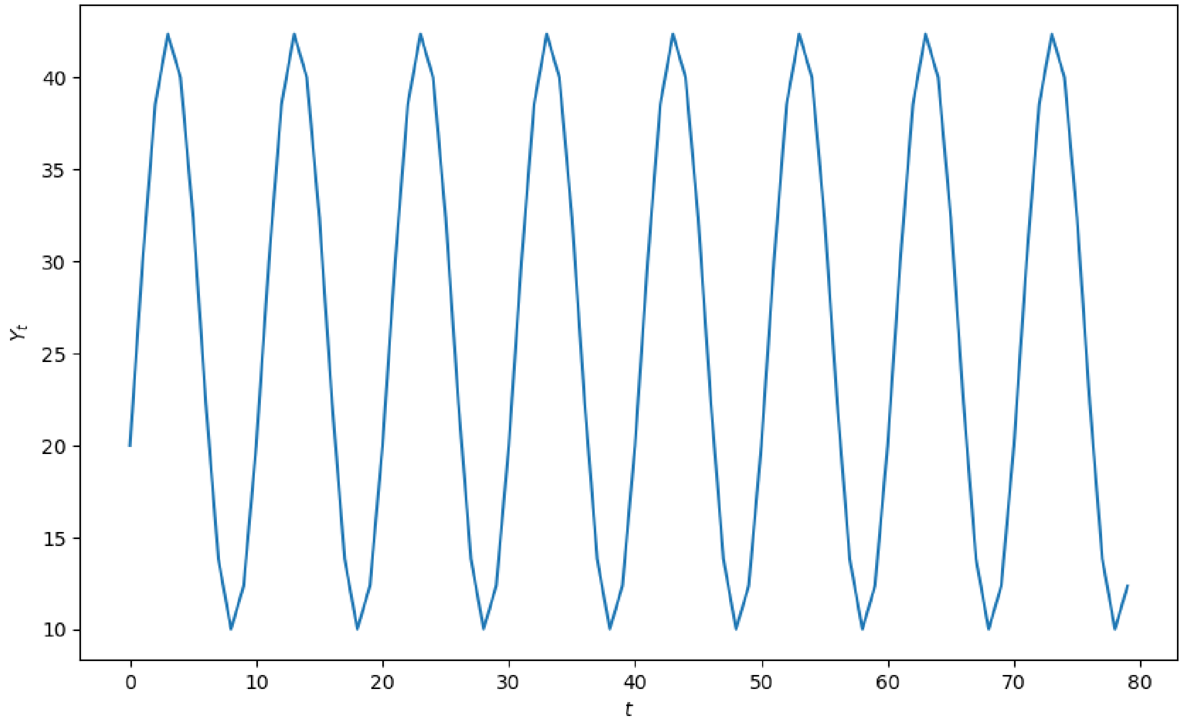
37.3.6 Reverse-engineered complex roots: Example

The next cell studies the implications of reverse-engineered complex roots.

We'll generate an *undamped* cycle of period 10

```
r = 1  # Generates undamped, nonexplosive cycles

period = 10  # Length of cycle in units of time
ϕ = 2 * math.pi / period

# Apply the reverse-engineering function f
ρ1, ρ2, α, β = f(r, ϕ)

# Extract real parts for numerical computation
α = α.real
β = β.real

print(f"α, β = {α:.2f}, {β:.2f}")

ytemp = y_nonstochastic(α=α, β=β, y_0=20, y_1=30)
plot_y(ytemp)
```

```
α, β = 0.62, 1.00
Solution type: Damped oscillations
Roots are [0.80901699+0.58778525j 0.80901699-0.58778525j]
Root type: Complex
Stability: Stable
```



37.3.7 Digression: Using Sympy to find roots

We can also use sympy to compute analytic formulas for the roots

```
init_printing()

r1 = Symbol("ρ_1")
r2 = Symbol("ρ_2")
z = Symbol("z")

sympy.solve(z**2 - r1 * z - r2, z)
```

$$\left[\rho_1/2 - \sqrt{\rho_1^2 + 4\rho_2}/2, \ \rho_1/2 + \sqrt{\rho_1^2 + 4\rho_2}/2\right]$$

```
α = Symbol("α")
β = Symbol("β")
r1 = α + β
r2 = -β

sympy.solve(z**2 - r1 * z - r2, z)
```

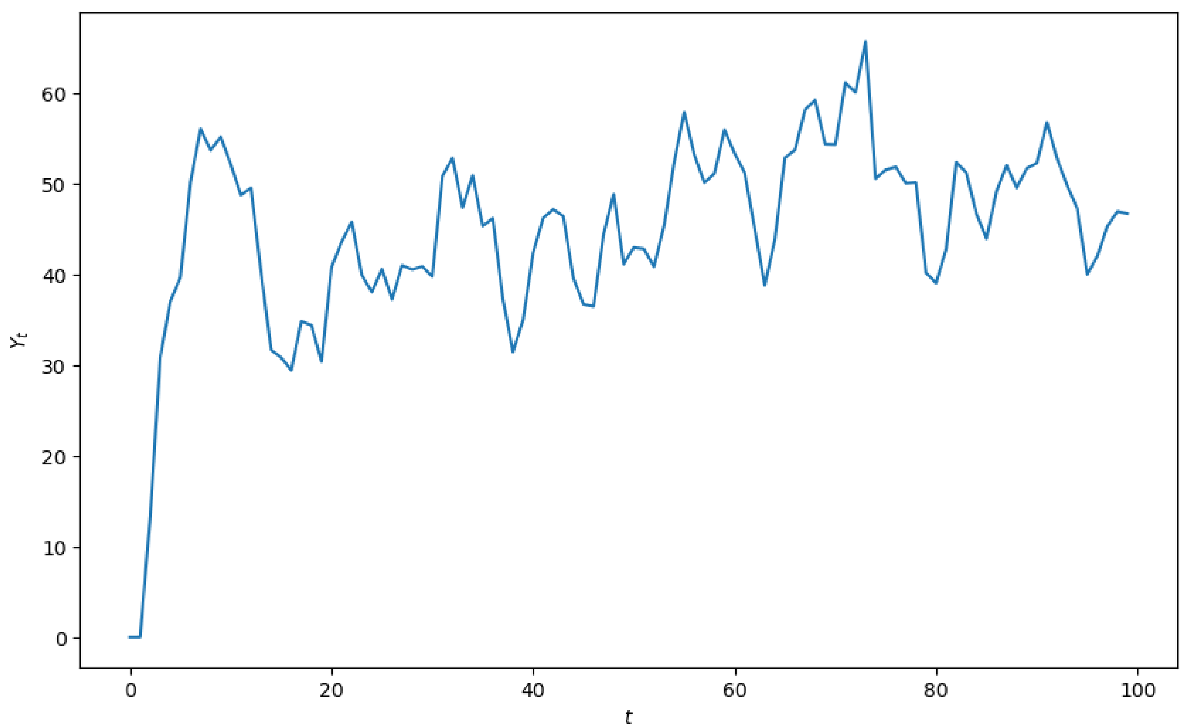
$$\left[\alpha/2 + \beta/2 - \sqrt{\alpha^2 + 2\alpha\beta + \beta^2 - 4\beta}/2, \ \alpha/2 + \beta/2 + \sqrt{\alpha^2 + 2\alpha\beta + \beta^2 - 4\beta}/2\right]$$

37.4 Stochastic shocks

Now we'll construct some code to simulate the stochastic version of the model that emerges when we add a random shock process to aggregate demand

```
def y_stochastic(y_0=0, y_1=0, α=0.8, β=0.2, γ=10, n=100, σ=5):
    """
    This function takes parameters of a stochastic version of
    the model, analyzes the roots of the characteristic
    polynomial and generates a simulation.
    """

    y_series, analysis = simulate_samuelson(y_0, y_1, α, β, γ, n, σ, None, 42)

    print(f"Solution type: {analysis['solution_type']}")
    print(f"Roots are {[f'{root:.2f}' for root in analysis['roots']]}")
    print(f"Root type: {'Complex' if analysis['is_complex'] else 'Real'}")
    print(f"Stability: {'Stable' if analysis['is_stable'] else 'Unstable'}")

    return y_series


plot_y(y_stochastic())
```

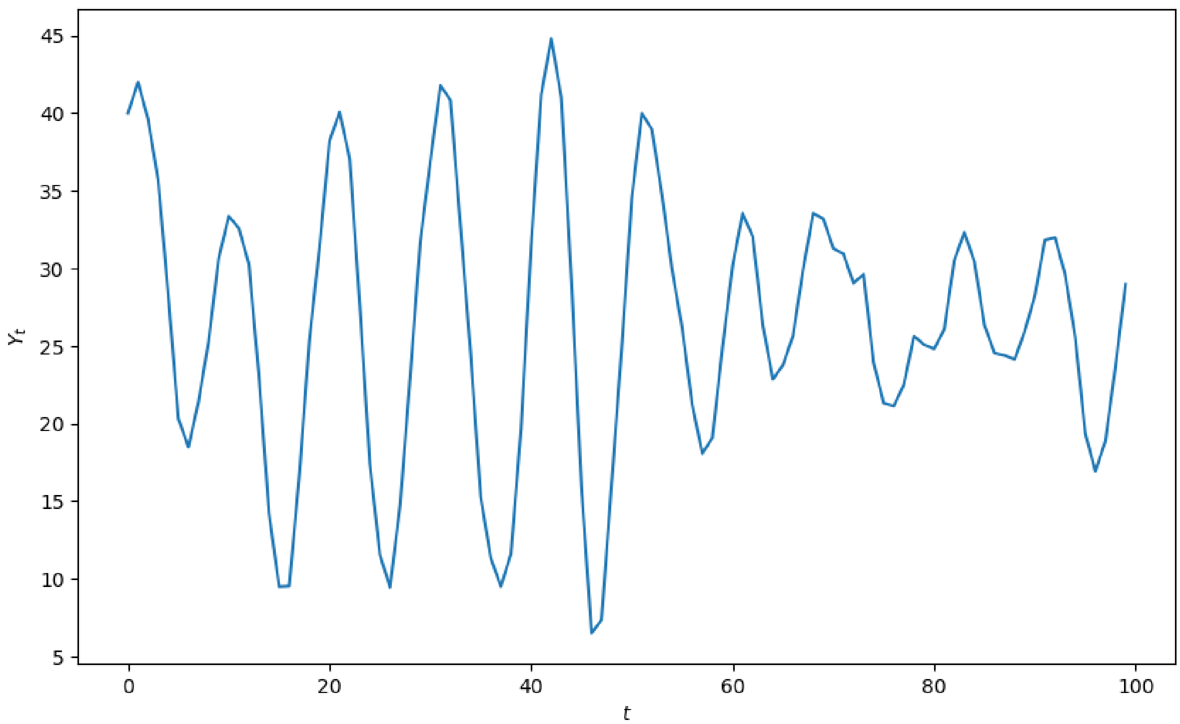
```
Solution type: Steady state convergence
Roots are ['0.72', '0.28']
Root type: Real
Stability: Stable
```



Let's do a simulation in which there are shocks and the characteristic polynomial has complex roots

```
r = 0.97

period = 10  # Length of cycle in units of time
ϕ = 2 * math.pi / period

# Apply the reverse-engineering function f
ρ1, ρ2, α, β = f(r, ϕ)

# Extract real parts for numerical computation
α = α.real
β = β.real

print(f"α, β = {α:.2f}, {β:.2f}")
plot_y(y_stochastic(y_0=40, y_1=42, α=α, β=β, σ=2, n=100))
```

```
α, β = 0.63, 0.94
Solution type: Damped oscillations
Roots are ['0.78+0.57j', '0.78-0.57j']
Root type: Complex
Stability: Stable
```



37.5 Government spending

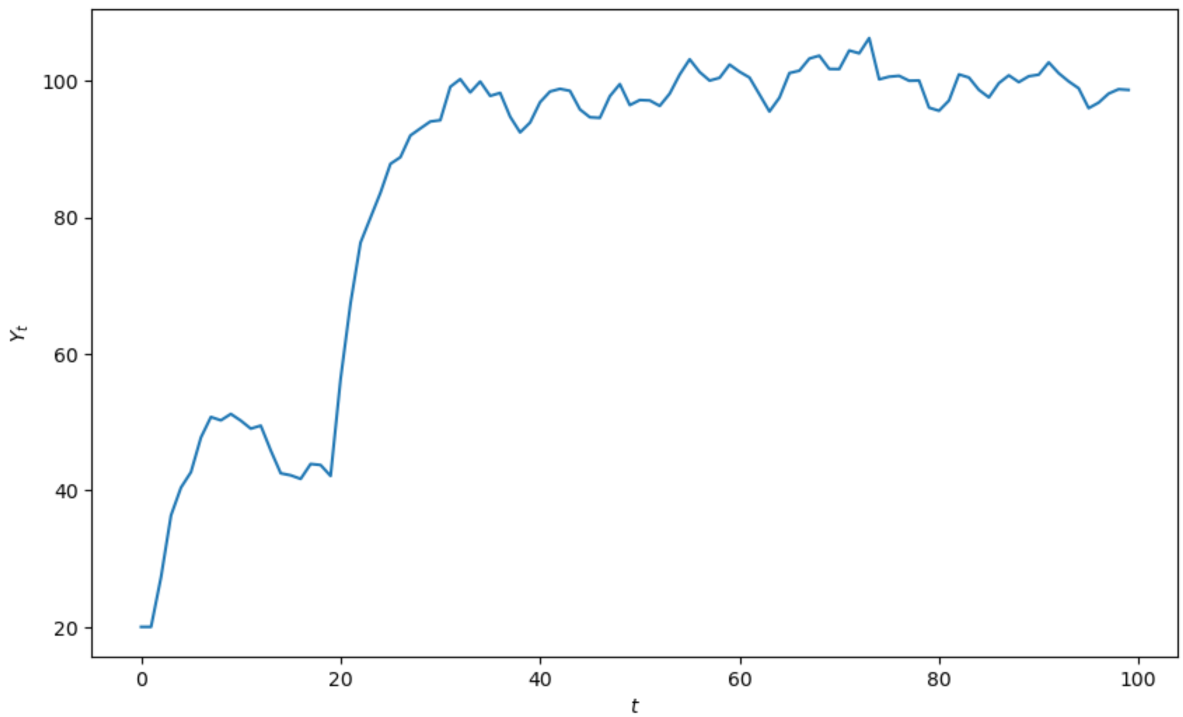
This function computes a response to either a permanent or one-off increase in government expenditures

```
def y_stochastic_g(
    y_0=20, y_1=20, α=0.8, β=0.2, γ=10,
    n=100, σ=2, g=0, g_t=0, duration="permanent"
):
    """
    This program computes a response to a permanent increase
    in government expenditures that occurs at time 20
    """

    g_params = (
        {'g': g, 'g_t': g_t, 'duration': duration} if g != 0 else None
    )
    y_series, analysis = simulate_samuelson(
        y_0, y_1, α, β, γ, n, σ, g_params, 42
    )

    print(f"Solution type: {analysis['solution_type']}")
    print(f"Roots: {analysis['roots']}")
    print(f"Root type: {'Complex' if analysis['is_complex'] else 'Real'}")
    print(f"Stability: {'Stable' if analysis['is_stable'] else 'Unstable'}")

    return y_series
```

A permanent government spending shock can be simulated as follows

```
plot_y(y_stochastic_g(g=10, g_t=20, duration="permanent"))
```

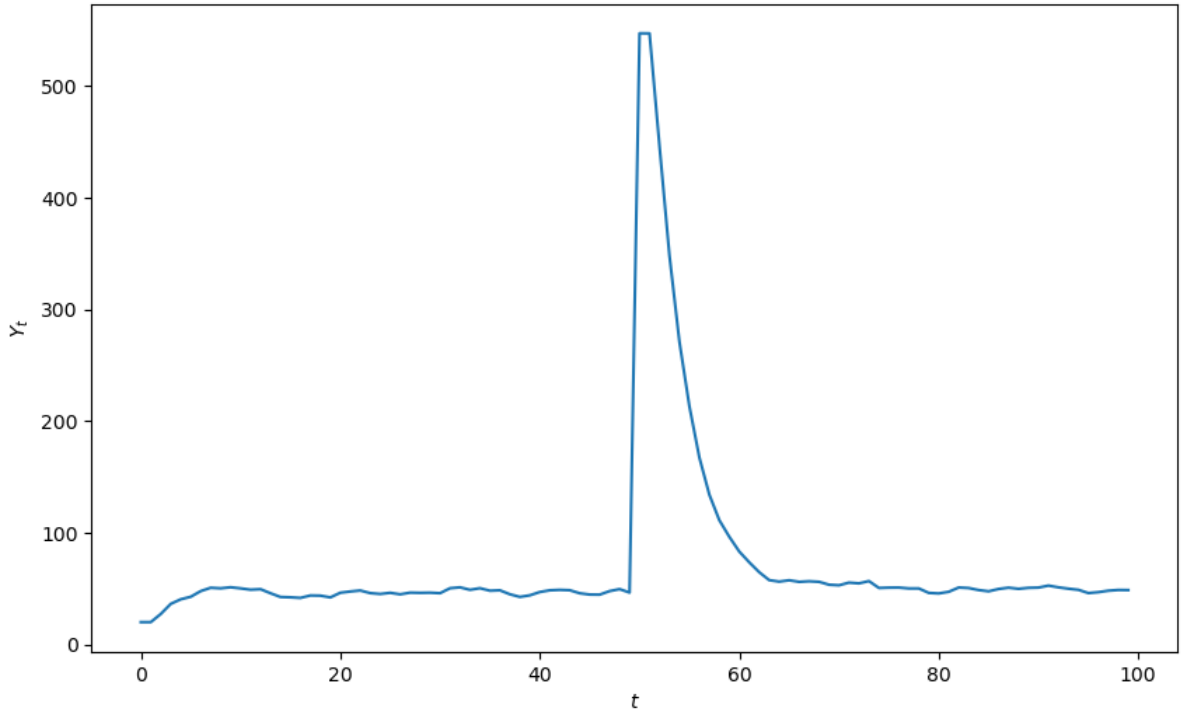
```
Solution type: Steady state convergence
Roots: [0.7236068 0.2763932]
Root type: Real
Stability: Stable
```



We can also see the response to a one time jump in government expenditures

```
plot_y(y_stochastic_g(g=500, g_t=50, duration="one-off"))
```

```
Solution type: Steady state convergence
Roots: [0.7236068 0.2763932]
Root type: Real
Stability: Stable
```



37.6 Wrapping everything into a class

Up to now, we have written functions to do the work.

Now we'll roll up our sleeves and write a Python class called `Samuelson` for the Samuelson model

```
class Samuelson:
    """
    This class represents the Samuelson model, otherwise known as the
    multiplier-accelerator model.
    The model combines the Keynesian multiplier
    with the accelerator theory of investment.

    The path of output is governed by a linear
    second-order difference equation

    .. math::

        Y_t = \alpha (1 + \beta) Y_{t-1} - \alpha \beta Y_{t-2}

    Parameters
    ----------
    y_0 : scalar
        Initial condition for Y_0
    y_1 : scalar
        Initial condition for Y_1
    α : scalar
        Marginal propensity to consume
    β : scalar
        Accelerator coefficient
```

(continues on next page)

(continued from previous page)

```
    n : int
        Number of iterations
    σ : scalar
        Volatility parameter. It must be greater than or equal to 0. Set
        equal to 0 for a non-stochastic model.
    g : scalar
        Government spending shock
    g_t : int
        Time at which government spending shock occurs. Must be specified
        when duration != None.
    duration : {None, 'permanent', 'one-off'}
        Specifies type of government spending shock. If none, government
        spending equal to g for all t.

    """

    def __init__(
        self, y_0=100, y_1=50,
        α=1.3, β=0.2, γ=10, n=100, σ=0, g=0, g_t=0, duration=None
    ):

        self.y_0, self.y_1, self.α, self.β = y_0, y_1, α, β
        self.n, self.g, self.g_t, self.duration = n, g, g_t, duration
        self.γ, self.σ = γ, σ

        # Use unified analysis function
        self.analysis = analyze_roots(α, β, verbose=False)
        self.ρ1, self.ρ2 = self.analysis['rho1'], self.analysis['rho2']
        self.roots = self.analysis['roots']

    def root_type(self):
        return "Complex conjugate" if self.analysis['is_complex'] else "Real"

    def root_less_than_one(self):
        return self.analysis['is_stable']

    def solution_type(self):
        return self.analysis['solution_type']

    def generate_series(self, seed=0):
        g_params = (
            {'g': self.g, 'g_t': self.g_t, 'duration': self.duration}
            if self.g != 0 else None
        )
        y_series, _ = simulate_samuelson(
            self.y_0, self.y_1, self.α, self.β, self.γ,
            self.n, self.σ, g_params, seed
        )
        return y_series

    def summary(self):
        print("Summary\n" + "-" * 50)
        print(f"Root type: {self.root_type()}")
        print(f"Solution type: {self.solution_type()}")
        print(f"Roots: {str(self.roots)}")

        if self.root_less_than_one() == True:
```

(continues on next page)

(continued from previous page)

```
            print("Absolute value of roots is less than one")
        else:
            print("Absolute value of roots is not less than one")

        if self.σ > 0:
            print("Stochastic series with σ = " + str(self.σ))
        else:
            print("Non-stochastic series")

        if self.g != 0:
            print("Government spending equal to " + str(self.g))

        if self.duration != None:
            print(
                self.duration.capitalize()
                + " government spending shock at t = "
                + str(self.g_t)
            )

    def plot(self, seed=0):
        fig, ax = plt.subplots(figsize=(10, 6))
        ax.plot(self.generate_series(seed))
        ax.set(xlabel="iteration", xlim=(0, self.n))
        ax.set_ylabel("$Y_t$", rotation=0)

        # Display model parameters on the plot
        paramstr = (
            f"$α={self.α:.2f}$ \n $β={self.β:.2f}$ \n "
            f"$\\gamma={self.γ:.2f}$ \n $\\sigma={self.σ:.2f}$ \n "
            f"$\\rho_1={self.ρ1:.2f}$ \n $\\rho_2={self.ρ2:.2f}$"
        )
        props = dict(fc="white", pad=10, alpha=0.5)
        ax.text(
            0.87,
            0.05,
            paramstr,
            transform=ax.transAxes,
            fontsize=12,
            bbox=props,
            va="bottom",
        )

        return fig

    def param_plot(self):


        fig = param_plot()
        ax = fig.gca()

        # Display eigenvalues in the legend
        for i, root in enumerate(self.roots):
            if isinstance(root, complex):

                # Handle sign formatting for complex number display
                operator = ["+", ""]
                root_real = self.roots[i].real
```

(continues on next page)

(continued from previous page)

```
                root_imag = self.roots[i].imag
                label = (
                    rf"$\lambda_{i+1} = {root_real:.2f}"
                    rf"{operator[i]} {root_imag:.2f}i$"
                )
            else:
                label = rf"$\lambda_{i+1} = {self.roots[i].real:.2f}$"

            # Add invisible point for legend entry
            ax.scatter(
                0, 0, s=0, label=label
            )

        # Mark current parameter values on the stability diagram
        ax.scatter(
            self.ρ1,
            self.ρ2,
            s=100,
            c="red",
            marker="+",
            label=r"$(\rho_1, \rho_2)$",
            zorder=5,
        )

        plt.legend(fontsize=12, loc=3)

        return fig
```

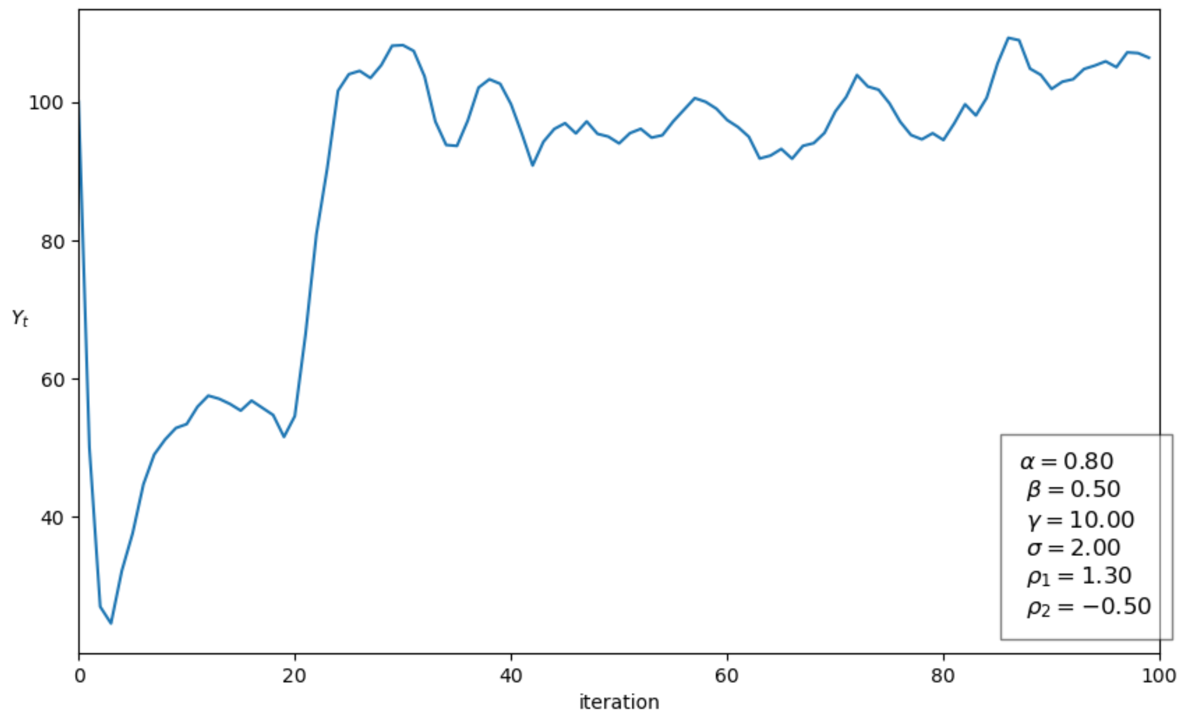
37.6.1 Illustration of Samuelson class

Now we'll put our Samuelson class to work on an example

```
sam = Samuelson(α=0.8, β=0.5, σ=2, g=10, g_t=20, duration="permanent")
sam.summary()
```

```
Summary
---------------------------------------------------
Root type: Complex conjugate
Solution type: Damped oscillations
Roots: [0.65+0.27838822j 0.65-0.27838822j]
Absolute value of roots is less than one
Stochastic series with σ = 2
Government spending equal to 10
Permanent government spending shock at t = 20
```

```
sam.plot()
plt.show()
```

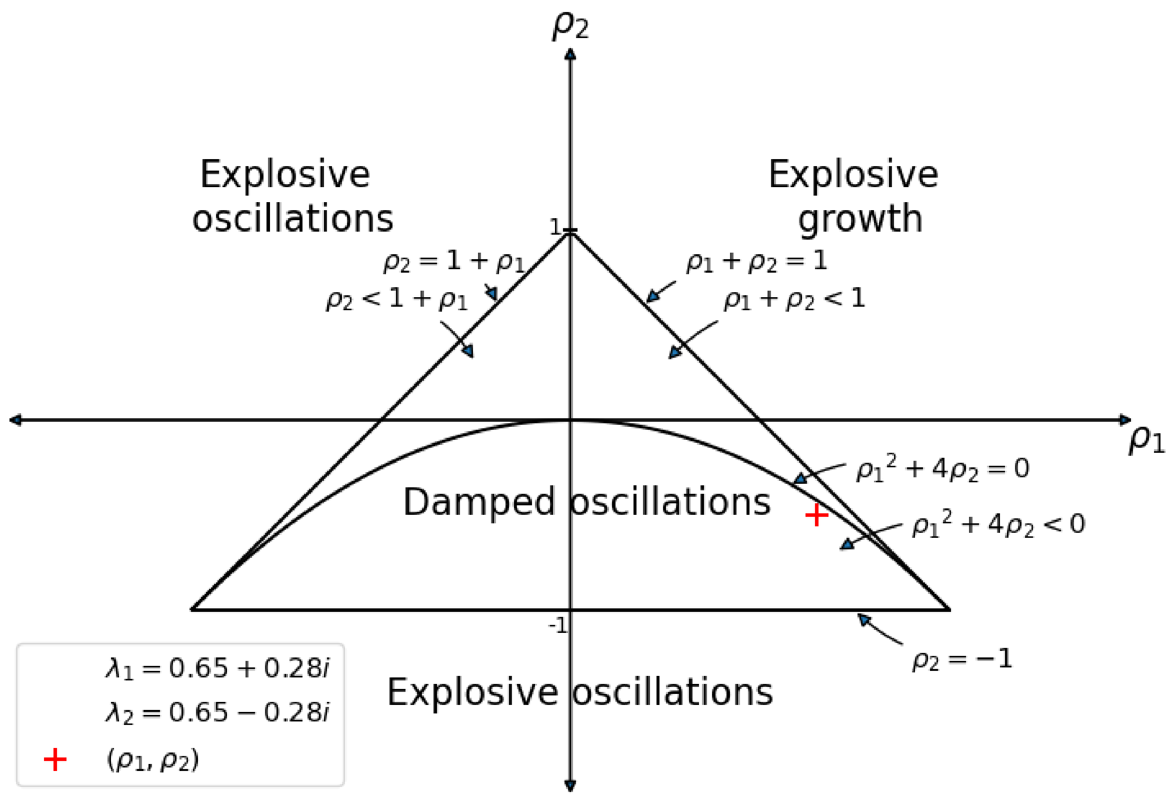


37.6.2 Using the graph

We'll use our graph to show where the roots lie and how their location is consistent with the behavior of the path just graphed.

The red $+$ sign shows the location of the roots

```
sam.param_plot()
plt.show()
```



37.7 Using the LinearStateSpace class

It turns out that we can use the QuantEcon.py LinearStateSpace class to do much of the work that we have done from scratch above.

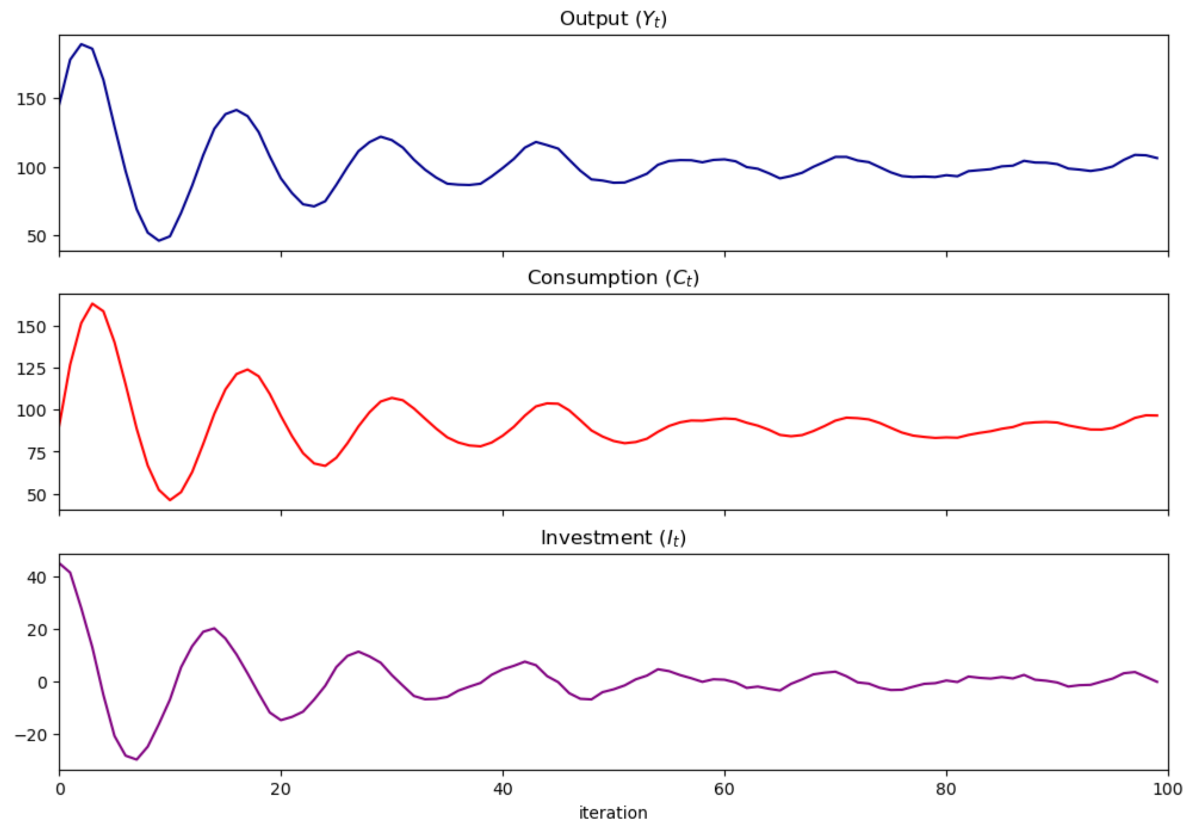
Here is how we map the Samuelson model into an instance of a `LinearStateSpace` class

```
α = 0.8
β = 0.9
ρ1 = α + β
ρ2 = -β
γ = 10
σ = 1
g = 10
n = 100

A = [[1, 0, 0], [γ + g, ρ1, ρ2], [0, 1, 0]]

G = [
    [γ + g, ρ1, ρ2],   # Y_{t+1}
    [γ, α, 0],         # C_{t+1}
    [0, β, -β],        # I_{t+1}
]

μ_0 = [1, 100, 50]
C = np.zeros((3, 1))
C[1] = σ  # Shock variance
```

(continues on next page)

(continued from previous page)

```
sam_t = LinearStateSpace(A, C, G, mu_0=μ_0)

x, y = sam_t.simulate(ts_length=n)

fig, axes = plt.subplots(3, 1, sharex=True, figsize=(12, 8))
titles = ["Output ($Y_t$)", "Consumption ($C_t$)", "Investment ($I_t$)"]
colors = ["darkblue", "red", "purple"]
for ax, series, title, color in zip(axes, y, titles, colors):
    ax.plot(series, color=color)
    ax.set(title=title, xlim=(0, n))

axes[-1].set_xlabel("iteration")

plt.show()
```



37.7.1 Other methods in the `LinearStateSpace` class

Let's plot *impulse response functions* for the instance of the Samuelson model using a method in the `LinearStateSpace` class

```
imres = sam_t.impulse_response()
imres = np.asarray(imres)
y1 = imres[:, :, 0]
y2 = imres[:, :, 1]
y1.shape
```

$$(2, \ 6, \ 1)$$

Now let's compute the zeros of the characteristic polynomial by simply calculating the eigenvalues of A

```
A = np.asarray(A)
w, v = np.linalg.eig(A)
print(np.round(w, 2))
```

```
[0.85+0.42j 0.85-0.42j 1.  +0.j  ]
```

37.7.2 Inheriting methods from `LinearStateSpace`

We could also create a subclass of `LinearStateSpace` (inheriting all its methods and attributes) to add more functions to use

```
class SamuelsonLSS(LinearStateSpace):
    """
    This subclass creates a Samuelson multiplier-accelerator model
    as a linear state space system.
    """

    def __init__(self, y_0=100, y_1=50, α=0.8, β=0.9, γ=10, σ=1, g=10):

        self.α, self.β = α, β
        self.y_0, self.y_1, self.g = y_0, y_1, g
        self.γ, self.σ = γ, σ

        # Set initial state vector
        self.initial_μ = [1, y_0, y_1]

        self.ρ1 = α + β
        self.ρ2 = -β

        # Construct state transition matrix
        self.A = [[1, 0, 0], [γ + g, self.ρ1, self.ρ2], [0, 1, 0]]

        # Construct observation matrix
        self.G = [
            [γ + g, self.ρ1, self.ρ2],  # Y_{t+1}
            [γ, α, 0],                  # C_{t+1}
            [0, β, -β],                 # I_{t+1}
        ]

        self.C = np.zeros((3, 1))
        self.C[1] = σ  # Shock variance

        # Initialize the LinearStateSpace instance
        LinearStateSpace.__init__(
            self, self.A, self.C, self.G, mu_0=self.initial_μ
        )

    # Create unicode aliases for mu_0 and Sigma_0 in the parent class
    @property
    def μ_0(self):
        return self.mu_0
```

(continues on next page)

(continued from previous page)

```
    @μ_0.setter
    def μ_0(self, value):
        self.mu_0 = value

    @property
    def Σ_0(self):
        return self.Sigma_0

    @Σ_0.setter
    def Σ_0(self, value):
        self.Sigma_0 = value

    def plot_simulation(self, ts_length=100, stationary=True, seed=0):

        # Store original distribution parameters
        temp_μ = self.μ_0
        temp_Σ = self.Σ_0

        # Use stationary distribution for simulation
        if stationary == True:
            try:
                (self.μ_x, self.μ_y, self.Σ_x, self.Σ_y, self.Σ_yx
                ) = self.stationary_distributions()
                self.μ_0 = self.μ_x
                self.Σ_0 = self.Σ_x

            # Handle case where stationary distribution doesn't exist
            except ValueError:
                print("Stationary distribution does not exist")

        np.random.seed(seed)
        x, y = self.simulate(ts_length)

        fig, axes = plt.subplots(3, 1, sharex=True, figsize=(12, 8))
        titles = ["Output ($Y_t$)",
                  "Consumption ($C_t$)",
                  "Investment ($I_t$)"]
        colors = ["darkblue", "red", "purple"]
        for ax, series, title, color in zip(axes, y, titles, colors):
            ax.plot(series, color=color)
            ax.set(title=title, xlim=(0, n))

        axes[-1].set_xlabel("iteration")
        plt.show()

        # Restore original distribution parameters
        self.μ_0 = temp_μ
        self.Σ_0 = temp_Σ

    def plot_irf(self, j=5):

        x, y = self.impulse_response(j)

        # Reshape impulse responses for plotting
        yimf = np.array(y).flatten().reshape(j + 1, 3).T

        fig, axes = plt.subplots(3, 1, sharex=True, figsize=(12, 8))
```

(continues on next page)

(continued from previous page)

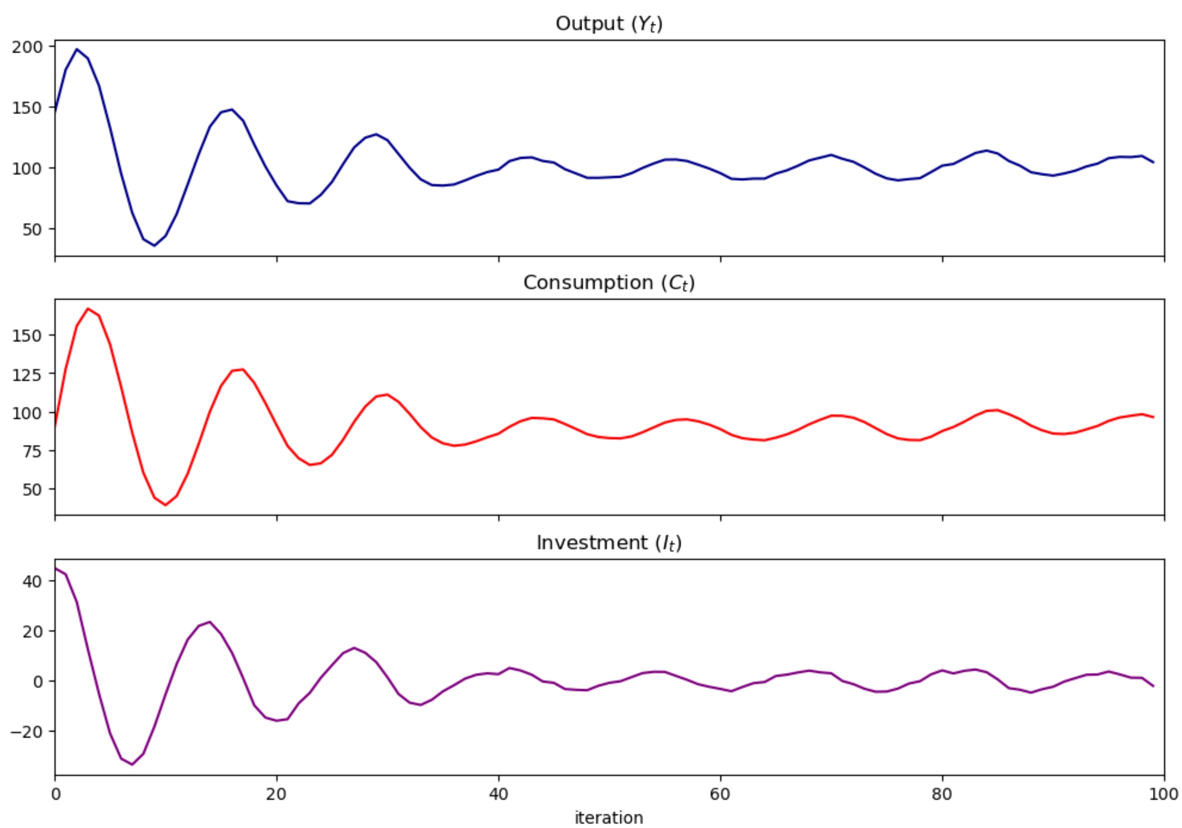
```
        labels = ["$Y_t$", "$C_t$", "$I_t$"]
        colors = ["darkblue", "red", "purple"]
        for ax, series, label, color in zip(axes, yimf, labels, colors):
            ax.plot(series, color=color)
            ax.set(xlim=(0, j))
            ax.set_ylabel(label, rotation=0, fontsize=14, labelpad=10)

        axes[0].set_title("Impulse response functions")
        axes[-1].set_xlabel("iteration")
        plt.show()

    def multipliers(self, j=5):
        x, y = self.impulse_response(j)
        return np.sum(np.array(y).flatten().reshape(j + 1, 3), axis=0)
```

37.7.3 Illustrations

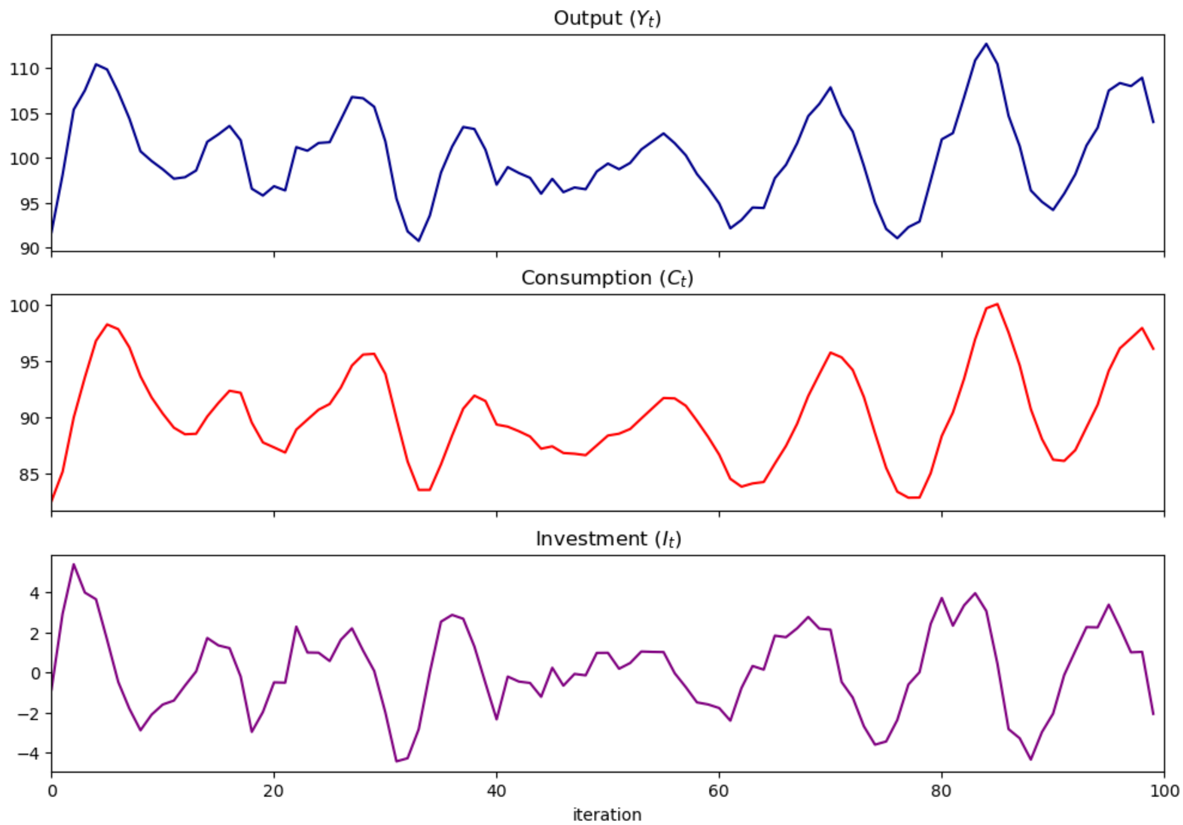
Let's show how we can use the `SamuelsonLSS`

```
samlss = SamuelsonLSS()
```

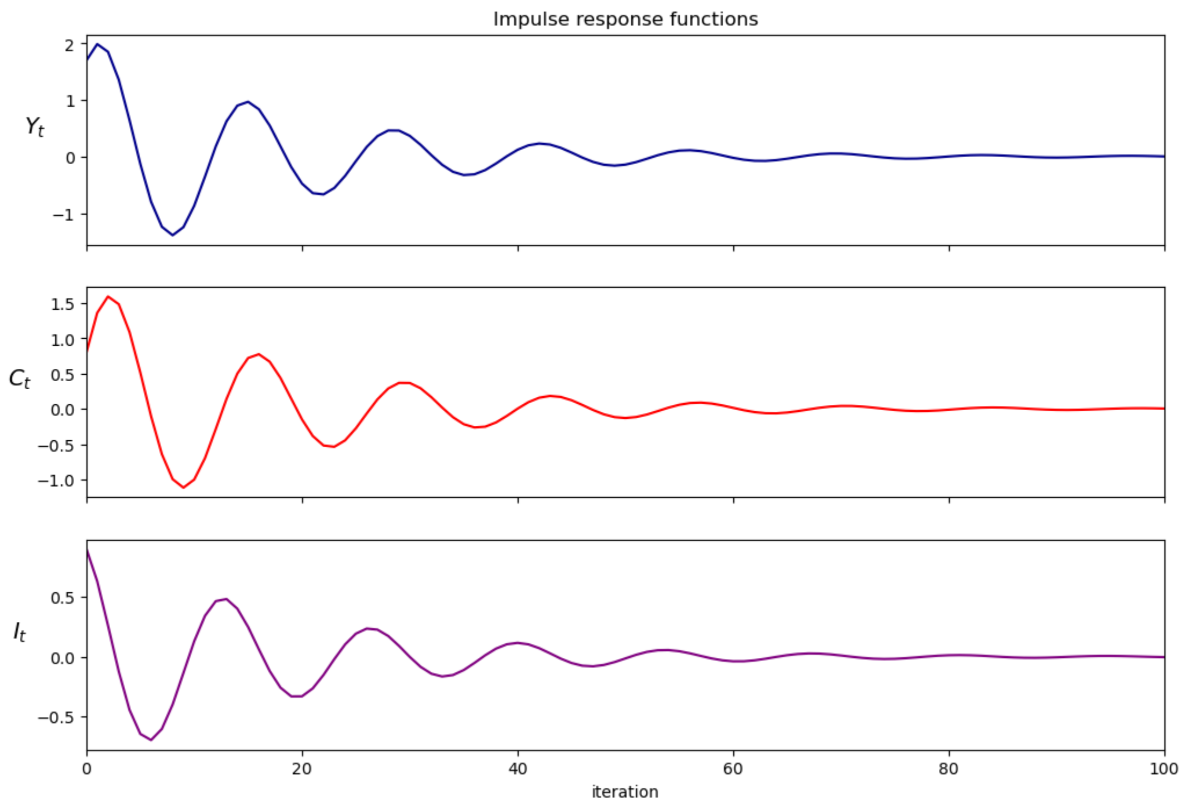
```
samlss.plot_simulation(100, stationary=False)
plt.show()
```



```
samlss.plot_simulation(100, stationary=True)
plt.show()
```




```
samlss.plot_irf(100)
plt.show()
```




```
samlss.multipliers()
```

```
array([7.414389, 6.835896, 0.578493])
```

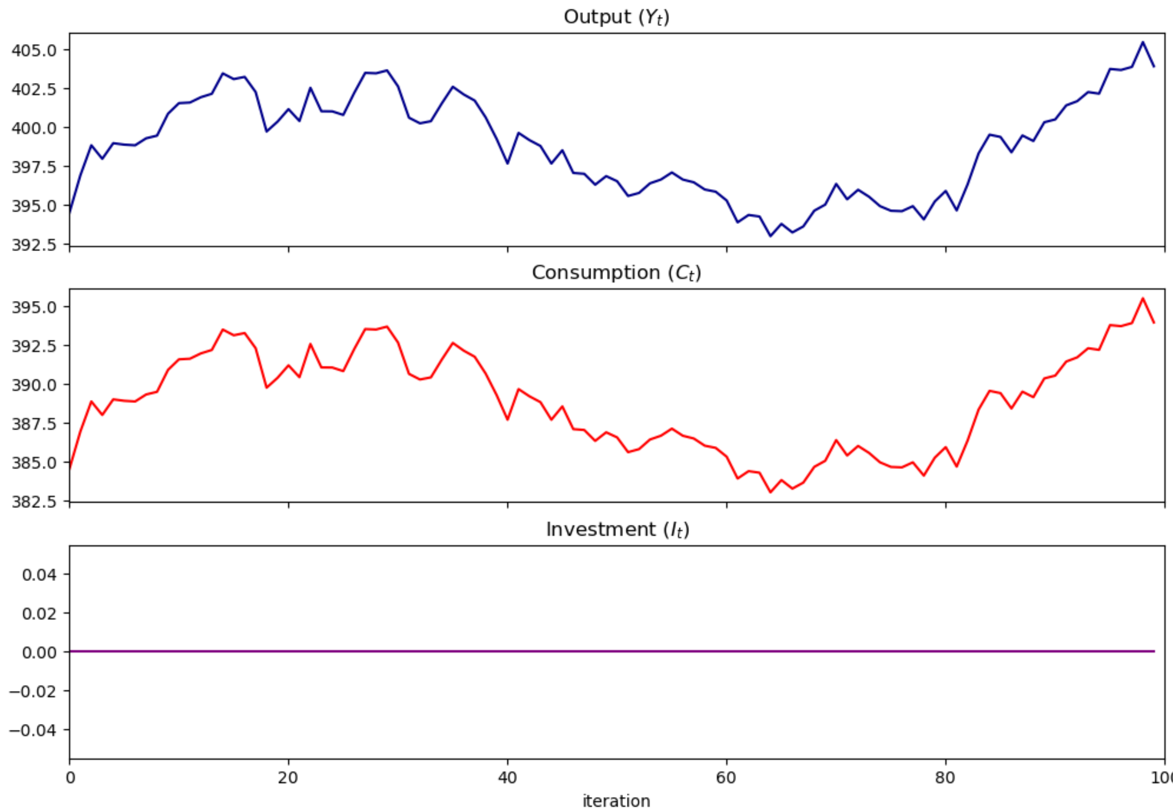
37.8 Pure multiplier model

Let's shut down the accelerator by setting $b = 0$ to get a pure multiplier model

- the absence of cycles gives an idea about why Samuelson included the accelerator

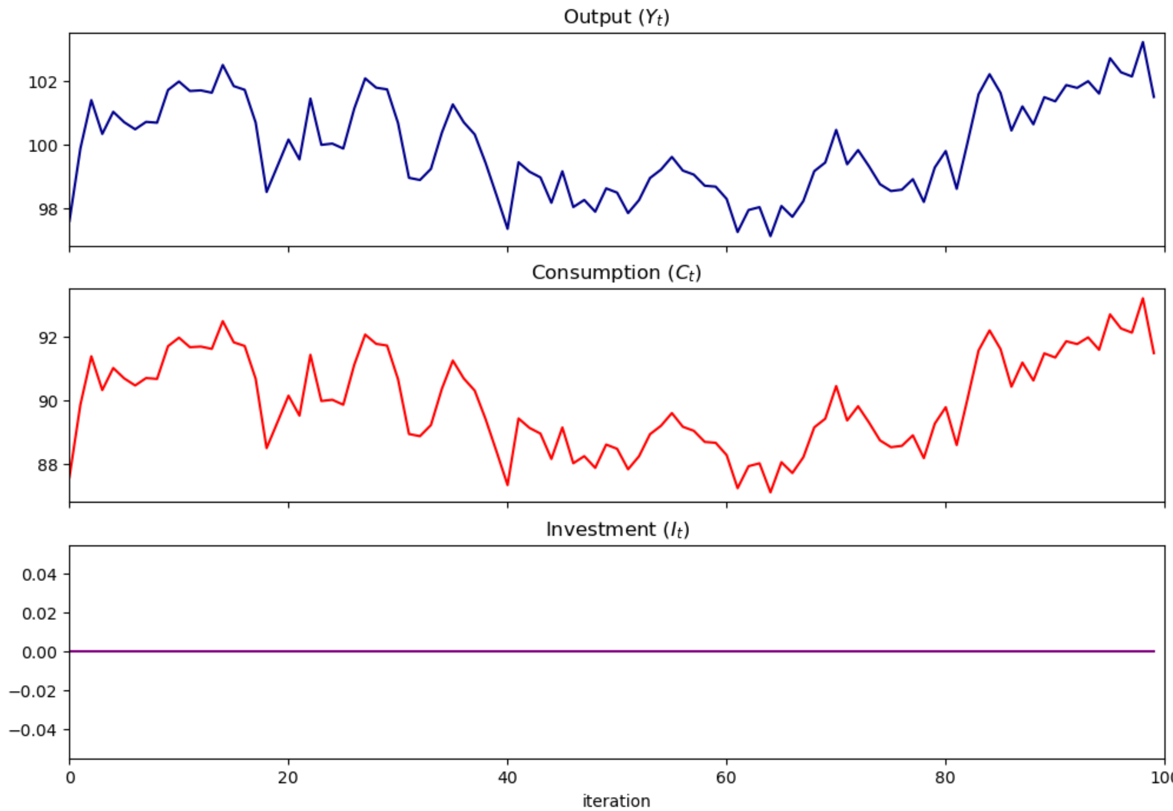
```
pure_multiplier = SamuelsonLSS(α=0.95, β=0)
```

```
pure_multiplier.plot_simulation()
```

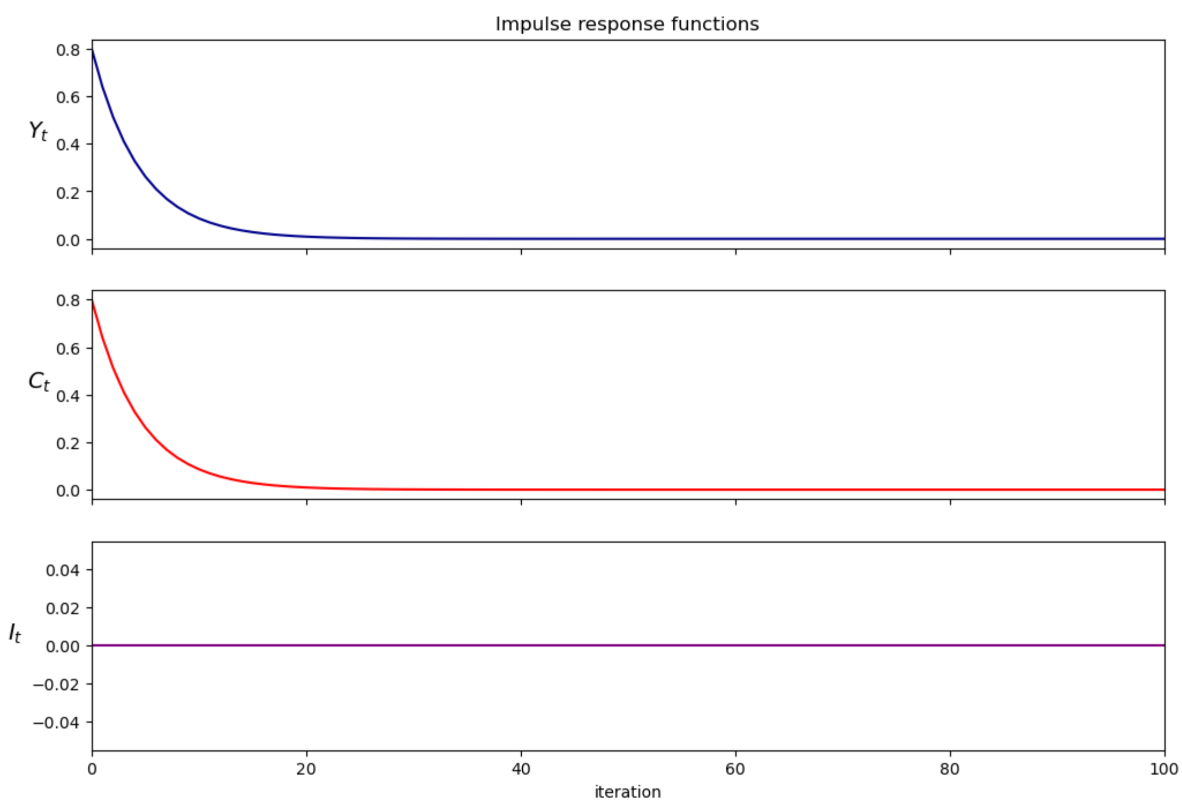



```
pure_multiplier = SamuelsonLSS(α=0.8, β=0)
```

```
pure_multiplier.plot_simulation()
```




```
pure_multiplier.plot_irf(100)
```



37.9 Summary

In this lecture, we wrote functions and classes to represent non-stochastic and stochastic versions of the Samuelson (1939) multiplier-accelerator model, described in [Samuelson, 1939].

We saw that different parameter values led to different output paths, which could either be stationary, explosive, or oscillating.

We also were able to represent the model using the QuantEcon.py LinearStateSpace class.

CHAPTER

THIRTYEIGHT

THE ACCELERATION PRINCIPLE AND THE NATURE OF BUSINESS CYCLES

Contents

38.1 Overview

This lecture studies two classic papers by Gregory Chow:

- Chow [1968] presents empirical evidence for the acceleration principle, describes how acceleration promotes oscillations, and analyzes conditions for the emergence of spectral peaks in linear difference equation subjected to random shocks
- Chow and Levitan [1969] presents a spectral analysis of a calibrated US macroeconometric model and teaches about spectral gains, coherences, and lead–lag patterns

These papers are related to ideas in the following lectures:

- The multiplier–accelerator mechanism in *Samuelson Multiplier-Accelerator*
- Linear stochastic difference equations and autocovariances in *Linear State Space Models*
- Eigenmodes of multivariate dynamics in *VARs and DMDs*

- Fourier ideas in *Circulant Matrices* (and, for empirical estimation, the advanced lecture Estimation of Spectra)

Chow [1968] builds on earlier empirical work testing the acceleration principle on US investment data.

We start with that empirical evidence before developing the theoretical framework.

We will keep returning to three ideas:

- In deterministic models, oscillations indicate complex eigenvalues of a transition matrix.
- In stochastic models, a "cycle" shows up as a local peak in a (univariate) spectral density.
- Spectral peaks depend on eigenvalues, but also on how shocks enter and on how observables load on eigenmodes.

Let's start with some standard imports:

```
import numpy as np
import matplotlib.pyplot as plt
```

```
Matplotlib is building the font cache; this may take a moment.
```

We will use the following helper functions throughout the lecture

```
def spectral_density_var1(A, V, ω_grid):
    """Spectral density matrix for VAR(1): y_t = A y_{t-1} + u_t."""
    A, V = np.asarray(A), np.asarray(V)
    n = A.shape[0]
    I = np.eye(n)
    F = np.empty((len(ω_grid), n, n), dtype=complex)
    for k, ω in enumerate(ω_grid):
        H = np.linalg.inv(I - np.exp(-1j * ω) * A)
        F[k] = (H @ V @ H.conj().T) / (2 * np.pi)
    return F

def spectrum_of_linear_combination(F, b):
    """Spectrum of x_t = b'y_t given the spectral matrix F(ω)."""
    b = np.asarray(b).reshape(-1, 1)
    return np.array([np.real((b.T @ F[k] @ b).item())
                            for k in range(F.shape[0])])

def simulate_var1(A, V, T, burn=200, seed=1234):
    r"""Simulate y_t = A y_{t-1} + u_t with u_t \sim N(0, V)."""
    rng = np.random.default_rng(seed)
    A, V = np.asarray(A), np.asarray(V)
    n = A.shape[0]
    chol = np.linalg.cholesky(V)
    y = np.zeros((T + burn, n))

    for t in range(1, T + burn):
        y[t] = A @ y[t - 1] + chol @ rng.standard_normal(n)

    return y[burn:]

def sample_autocorrelation(x, max_lag):
    """Sample autocorrelation of a 1d array from lag 0 to max_lag."""
    x = np.asarray(x)
    x = x - x.mean()
    denom = np.dot(x, x)
    acf = np.empty(max_lag + 1)
    for k in range(max_lag + 1):
```

(continues on next page)

(continued from previous page)

```
        acf[k] = np.dot(x[:-k] if k else x, x[k:]) / denom
    return acf
```

38.2 Empirical foundation for the acceleration principle

Chow [1968] opens by reviewing empirical evidence for the acceleration principle from earlier macroeconometric work.

Using annual observations for 1931–40 and 1948–63, Chow tested the acceleration equation on three investment categories:

- new construction
- gross private domestic investment in producers' durable equipment combined with change in business inventories
- the last two variables separately

In each case, when the regression included both Y_t and Y_{t-1} (where Y is gross national product minus taxes net of transfers), the coefficient on Y_{t-1} was of *opposite sign* and slightly smaller in absolute value than the coefficient on Y_t.

Equivalently, when expressed in terms of ΔY_t and Y_{t-1}, the coefficient on Y_{t-1} was a small fraction of the coefficient on ΔY_t.

38.2.1 An example: automobile demand

Chow presents a clean illustration using data on net investment in automobiles from his earlier work on automobile demand.

Using annual data for 1922–41 and 1948–57, he estimates by least squares:

$$y_t^n = \underset{(0.0022)}{0.0155} Y_t \underset{(0.0020)}{-0.0144} Y_{t-1} \underset{(0.0056)}{-0.0239} p_t \underset{(0.0040)}{+0.0199} p_{t-1} + \underset{(0.101)}{0.351} y_{t-1}^n + \text{const.} \tag{38.1}$$

where:

- Y_t is real disposable personal income per capita
- p_t is a relative price index for automobiles
- y_t^n is per capita net investment in passenger automobiles
- standard errors appear in parentheses

The key observation: the coefficients on Y_{t-1} and p_{t-1} are *the negatives* of the coefficients on Y_t and p_t.

This pattern is exactly what the acceleration principle predicts.

38.2.2 From stock adjustment to acceleration

The empirical support for acceleration should not be surprising once we accept a stock-adjustment demand equation for capital:

$$s_{it} = a_i Y_t + b_i s_{i,t-1} \tag{38.2}$$

where s_{it} is the stock of capital good i.

The acceleration equation (38.1) is essentially the *first difference* of (38.2).

Net investment is the change in stock, $y_{it}^n = \Delta s_{it}$, and differencing (38.2) gives:

$$y_{it}^n = a_i \Delta Y_t + b_i y_{i,t-1}^n \tag{38.3}$$

The coefficients on Y_t and Y_{t-1} in the level form are a_i and $-a_i(1-b_i)$ respectively.

They are opposite in sign and similar in magnitude when b_i is not too far from unity.

This connection between stock adjustment and acceleration is central to Chow's argument about why acceleration matters for business cycles.

38.3 Acceleration enables oscillations

Having established the empirical evidence for acceleration, we now examine why it matters theoretically for generating oscillations.

Chow [1968] asks a fundamental question: if we build a macro model using only standard demand equations with simple distributed lags, can the system generate sustained oscillations?

He shows that, under natural sign restrictions, the answer is no.

Stock-adjustment demand for durable goods leads to investment equations where the coefficient on Y_{t-1} is negative.

This negative coefficient captures the **acceleration effect**: investment responds not just to the level of income, but to its rate of change.

This negative coefficient is also what makes complex roots possible in the characteristic equation.

Without it, Chow proves that demand systems with only positive coefficients have real positive roots, and hence no oscillatory dynamics.

The *Samuelson Multiplier-Accelerator* lecture explores this mechanism in detail through the Hansen-Samuelson multiplier-accelerator model.

Here we briefly illustrate the effect.

Take the multiplier–accelerator law of motion:

$$Y_t = cY_{t-1} + v(Y_{t-1} - Y_{t-2}),$$

and rewrite it as a first-order system in (Y_t, Y_{t-1}).

```python
def samuelson_transition(c, v):
    return np.array([[c + v, -v], [1.0, 0.0]])

# Compare weak vs strong acceleration
# Weak: c=0.8, v=0.1 gives real roots (discriminant > 0)
# Strong: c=0.6, v=0.8 gives complex roots (discriminant < 0)
cases = [("weak acceleration", 0.8, 0.1),
         ("strong acceleration", 0.6, 0.8)]
A_list = [samuelson_transition(c, v) for _, c, v in cases]

for (label, c, v), A in zip(cases, A_list):
    eig = np.linalg.eigvals(A)
    disc = (c + v)**2 - 4*v
    print(
        f"{label}: c={c}, v={v}, discriminant={disc:.2f}, eigenvalues={eig}")
```

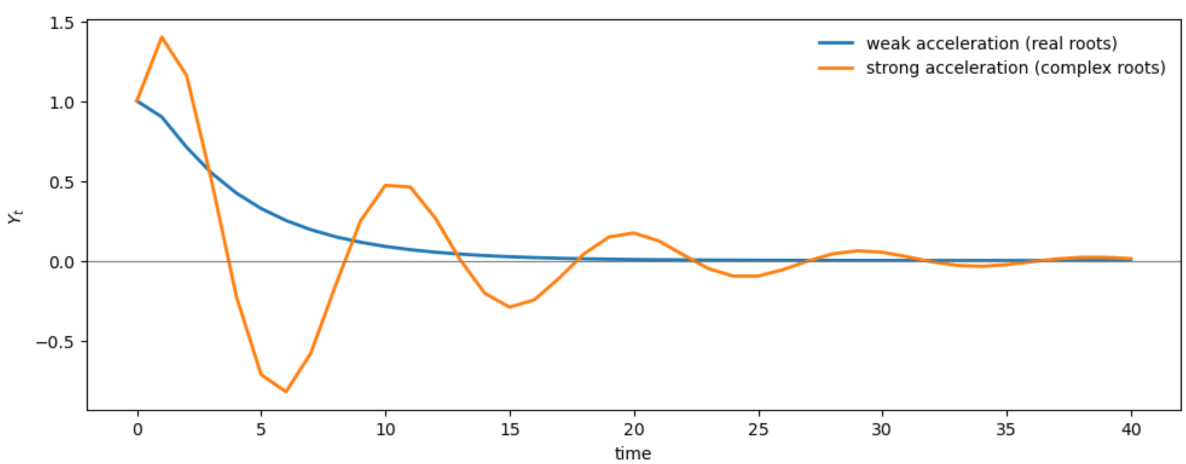
```
weak acceleration: c=0.8, v=0.1, discriminant=0.41, eigenvalues=[0.77015621 0.
↪12984379]
strong acceleration: c=0.6, v=0.8, discriminant=-1.24, eigenvalues=[0.7+0.
↪55677644j 0.7-0.55677644j]
```

With weak acceleration ($v = 0.1$), the discriminant is positive and the roots are real.

With strong acceleration ($v = 0.8$), the discriminant is negative and the roots are complex conjugates that enable oscillatory dynamics.

Now let's see how these different eigenvalue structures affect the impulse responses to a one-time shock in Y

```python
T = 40
s0 = np.array([1.0, 0.0])
irfs = []
for A in A_list:
    s = s0.copy()
    path = np.empty(T + 1)
    for t in range(T + 1):
        path[t] = s[0]
        s = A @ s
    irfs.append(path)

fig, ax = plt.subplots(figsize=(10, 4))
ax.plot(range(T + 1), irfs[0], lw=2,
                label="weak acceleration (real roots)")
ax.plot(range(T + 1), irfs[1], lw=2,
                label="strong acceleration (complex roots)")
ax.axhline(0.0, lw=0.8, color='gray')
ax.set_xlabel("time")
ax.set_ylabel(r"$Y_t$")
ax.legend(frameon=False)
plt.tight_layout()
plt.show()
```



With weak acceleration, the impulse response decays monotonically.

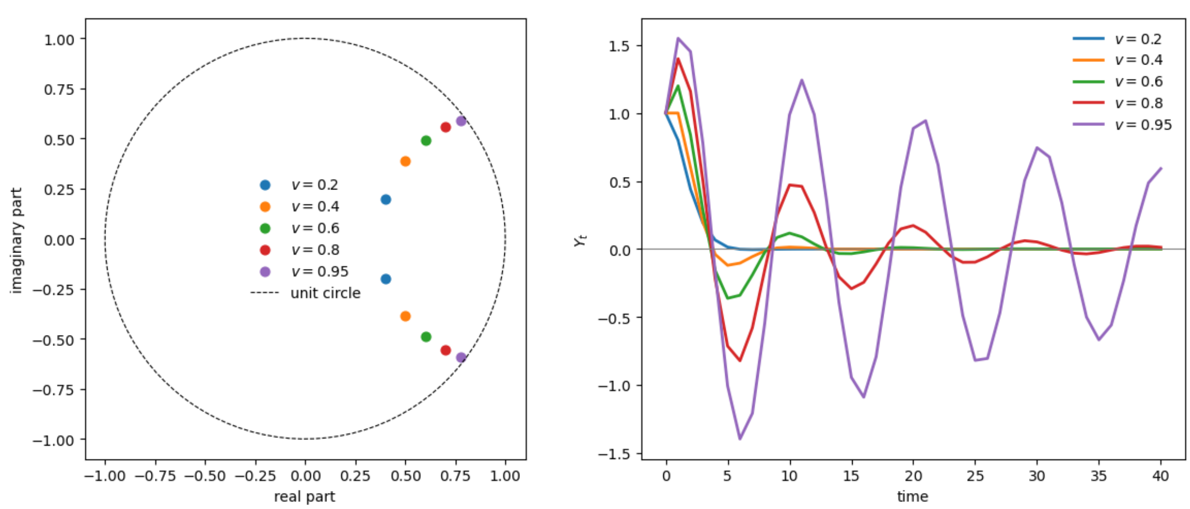
With strong acceleration, it oscillates.

We can ask how the eigenvalues change as we increase the accelerator v.

As we increase the accelerator v, the eigenvalues move further from the origin.

For this model, the eigenvalue modulus is $|\lambda| = \sqrt{v}$, so the stability boundary is $v = 1$.

```
v_grid = [0.2, 0.4, 0.6, 0.8, 0.95]
c = 0.6
T_irf = 40  # periods for impulse response

fig, axes = plt.subplots(1, 2, figsize=(12, 5))

for v in v_grid:
    A = samuelson_transition(c, v)
    eig = np.linalg.eigvals(A)

    # Eigenvalues (left panel)
    axes[0].scatter(eig.real, eig.imag, s=40, label=f'$v={v}$')

    # Impulse response (right panel)
    s = np.array([1.0, 0.0])
    irf = np.empty(T_irf + 1)
    for t in range(T_irf + 1):
        irf[t] = s[0]
        s = A @ s
    axes[1].plot(range(T_irf + 1), irf, lw=2, label=f'$v={v}$')

# Visualize the eigenvalue locations and the unit circle
θ_circle = np.linspace(0, 2*np.pi, 100)
axes[0].plot(np.cos(θ_circle), np.sin(θ_circle),
                'k--', lw=0.8, label='unit circle')
axes[0].set_xlabel('real part')
axes[0].set_ylabel('imaginary part')
axes[0].set_aspect('equal')
axes[0].legend(frameon=False)

# impulse response panel
axes[1].axhline(0, lw=0.8, color='gray')
axes[1].set_xlabel('time')
axes[1].set_ylabel(r'$Y_t$')
axes[1].legend(frameon=False)

plt.tight_layout()
plt.show()
```



As v increases, eigenvalues approach the unit circle and oscillations become more persistent.

This illustrates that acceleration creates complex eigenvalues, which are necessary for oscillatory dynamics in deterministic systems.

But what happens when we add random shocks?

An insight of Ragnar Frisch [Frisch, 1933] was that damped oscillations can be "maintained" when the system is continuously perturbed by random disturbances.

To study this formally, we need to introduce the stochastic framework.

38.4 A linear system with shocks

We analyze a first-order linear stochastic system

$$y_t = Ay_{t-1} + u_t, \qquad \mathbb{E}[u_t] = 0, \qquad \mathbb{E}[u_t u_t^\top] = V, \qquad \mathbb{E}[u_t u_{t-k}^\top] = 0, \quad k \neq 0. \tag{38.4}$$

When the eigenvalues of A are strictly inside the unit circle, the process is covariance stationary and its autocovariances exist.

In the notation of *Linear State Space Models*, this is the same stability condition that guarantees a unique solution to a discrete Lyapunov equation.

Define the lag-k autocovariance matrices

$$\Gamma_k := \mathbb{E}[y_t y_{t-k}^\top]. \tag{38.5}$$

Standard calculations (also derived in [Chow, 1968]) give the recursion

$$\Gamma_k = A\Gamma_{k-1}, \quad k \geq 1, \qquad \text{and} \qquad \Gamma_0 = A\Gamma_0 A^\top + V. \tag{38.6}$$

The second equation is the discrete Lyapunov equation for Γ_0.

Chow [1968] motivates the stochastic analysis with a quote from Ragnar Frisch:

> The examples we have discussed … show that when a [deterministic] economic system gives rise to oscillations, these will most frequently be damped. But in reality the cycles … are generally not damped. How can the maintenance of the swings be explained? … One way which I believe is particularly fruitful and promising is to study what would become of the solution of a determinate dynamic system if it were exposed to a stream of erratic shocks … Thus, by connecting the two ideas: (1) the continuous solution of a determinate dynamic system and (2) the discontinuous shocks intervening and supplying the energy that may maintain the swings—we get a theoretical setup which seems to furnish a rational interpretation of those movements which we have been accustomed to see in our statistical time data.
>
> — Ragnar Frisch (1933) [Frisch, 1933]

Chow's main insight is that oscillations in the deterministic system are *neither necessary nor sufficient* for producing "cycles" in the stochastic system.

We have to bring the stochastic element into the picture.

We will show that even when eigenvalues are real (no deterministic oscillations), the stochastic system can exhibit cyclical patterns in its autocovariances and spectral densities.

38.4.1 Autocovariances in terms of eigenvalues

Let $\lambda_1, \dots, \lambda_p$ be the distinct, possibly complex, eigenvalues of A, and let B be the matrix whose columns are the corresponding right eigenvectors:

$$AB = BD_\lambda, \quad \text{or equivalently} \quad A = BD_\lambda B^{-1} \tag{38.7}$$

where $D_\lambda = \text{diag}(\lambda_1, \dots, \lambda_p)$.

Define canonical variables $z_t = B^{-1} y_t$.

These satisfy the decoupled dynamics:

$$z_t = D_\lambda z_{t-1} + \varepsilon_t \tag{38.8}$$

where $\varepsilon_t = B^{-1} u_t$ has covariance matrix $W = B^{-1} V (B^{-1})^\top$.

The autocovariance matrix of the canonical variables, denoted Γ_k^*, satisfies

$$\Gamma_k^* = D_\lambda^k \Gamma_0^*, \quad k = 1, 2, 3, \dots \tag{38.9}$$

and

$$\Gamma_0^* = \left(\frac{w_{ij}}{1 - \lambda_i \lambda_j} \right) \tag{38.10}$$

where w_{ij} are elements of W.

The autocovariance matrices of the original variables are then

$$\Gamma_k = B \Gamma_k^* B^\top = B D_\lambda^k \Gamma_0^* B^\top, \quad k = 0, 1, 2, \dots \tag{38.11}$$

The scalar autocovariance $\gamma_{ij,k} = \mathbb{E}[y_{it} y_{j,t-k}]$ is a *linear combination* of powers of the eigenvalues:

$$\gamma_{ij,k} = \sum_m \sum_n b_{im} b_{jn} \gamma_{mn,0}^* \lambda_m^k = \sum_m d_{ij,m} \lambda_m^k \tag{38.12}$$

Compare this to the deterministic time path from initial condition y_0:

$$y_{it} = \sum_j b_{ij} z_{j0} \lambda_j^t \tag{38.13}$$

Both the autocovariance function (38.12) and the deterministic path (38.13) are linear combinations of λ_m^k (or λ_j^t).

38.4.2 Complex roots and damped oscillations

When eigenvalues come in complex conjugate pairs $\lambda = re^{\pm i\theta}$ with $r < 1$, their contribution to the autocovariance function is a **damped cosine**:

$$2sr^k \cos(\theta k + \phi) \tag{38.14}$$

for appropriate amplitude s and phase ϕ determined by the eigenvector loadings.

In the deterministic model, such complex roots generate damped oscillatory time paths.

In the stochastic model, they generate damped oscillatory autocovariance functions.

It is in this sense that deterministic oscillations could be "maintained" in the stochastic model, but as we will see, the connection between eigenvalues and spectral peaks is more subtle than this suggests.

38.5 From autocovariances to spectra

Chow's key step is to translate the autocovariance sequence $\{\Gamma_k\}$ into a frequency-domain object.

The **spectral density matrix** is the Fourier transform of Γ_k:

$$F(\omega) := \frac{1}{2\pi} \sum_{k=-\infty}^{\infty} \Gamma_k e^{-i\omega k}, \qquad \omega \in [0, \pi]. \tag{38.15}$$

For the VAR(1) system (38.4), this sum has a closed form

$$F(\omega) = \frac{1}{2\pi} \left(I - Ae^{-i\omega}\right)^{-1} V \left(I - A^\top e^{i\omega}\right)^{-1}. \tag{38.16}$$

$F(\omega)$ tells us how much variation in y_t is associated with cycles of (angular) frequency ω.

Higher frequencies correspond to rapid oscillations, meaning short cycles where the series completes many up-and-down movements per unit of time.

Lower frequencies correspond to slower oscillations, meaning long cycles that unfold over extended periods.

The corresponding cycle length (or period) is

$$T(\omega) = \frac{2\pi}{\omega}. \tag{38.17}$$

Thus, a frequency of $\omega = \pi$ corresponds to the shortest possible cycle of $T = 2$ periods, while frequencies near zero correspond to very long cycles.

When the spectral density $F(\omega)$ is concentrated at particular frequencies, it indicates that the time series exhibits pronounced cyclical behavior at those frequencies.

The advanced lecture Estimation of Spectra explains how to estimate $F(\omega)$ from data.

Here we focus on the model-implied spectrum.

We saw earlier that acceleration creates complex eigenvalues, which enable oscillatory impulse responses.

But do complex roots guarantee a spectral peak?

Are they necessary for one?

Chow provides precise answers for the Hansen-Samuelson model.

38.6 Spectral peaks in the Hansen-Samuelson model

Chow [1968] provides a detailed spectral analysis of the Hansen-Samuelson multiplier-accelerator model, deriving exact conditions for when spectral peaks occur.

The analysis reveals that in this specific model, complex roots are *necessary* for a peak, but as we will see later, this is not true in general.

38.6.1 The model as a first-order system

The second-order Hansen-Samuelson equation can be written as a first-order system:

$$\begin{bmatrix} y_{1t} \\ y_{2t} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ 1 & 0 \end{bmatrix} \begin{bmatrix} y_{1,t-1} \\ y_{2,t-1} \end{bmatrix} + \begin{bmatrix} u_{1t} \\ 0 \end{bmatrix} \tag{38.18}$$

where $y_{2t} = y_{1,t-1}$ is simply the lagged value of y_{1t}.

This structure implies a special relationship among the autocovariances:

$$\gamma_{11,k} = \gamma_{22,k} = \gamma_{12,k-1} = \gamma_{21,k+1} \tag{38.19}$$

Using the autocovariance recursion, Chow shows that this leads to the condition

$$\gamma_{11,-1} = d_{11,1}\lambda_1^{-1} + d_{11,2}\lambda_2^{-1} = \gamma_{11,1} = d_{11,1}\lambda_1 + d_{11,2}\lambda_2 \tag{38.20}$$

which constrains the spectral density in a useful way.

38.6.2 The spectral density formula

From equations (38.12) and the scalar kernel $g_i(\omega) = (1-\lambda_i^2)/(1+\lambda_i^2 - 2\lambda_i \cos\omega)$, the spectral density of y_{1t} is:

$$f_{11}(\omega) = d_{11,1}g_1(\omega) + d_{11,2}g_2(\omega) \tag{38.21}$$

which can be written in the combined form:

$$f_{11}(\omega) = \frac{d_{11,1}(1-\lambda_1^2)(1+\lambda_2^2) + d_{11,2}(1-\lambda_2^2)(1+\lambda_1^2) - 2[d_{11,1}(1-\lambda_1^2)\lambda_2 + d_{11,2}(1-\lambda_2^2)\lambda_1]\cos\omega}{(1+\lambda_1^2 - 2\lambda_1\cos\omega)(1+\lambda_2^2-2\lambda_2\cos\omega)} \tag{38.22}$$

A key observation: due to condition (38.20), the *numerator is not a function of* $\cos\omega$.

Therefore, to find a maximum of $f_{11}(\omega)$, we need only find a minimum of the denominator.

38.6.3 Conditions for a spectral peak

The first derivative of the denominator with respect to ω is:

$$2[(1+\lambda_1^2)\lambda_2 + (1+\lambda_2^2)\lambda_1]\sin\omega - 8\lambda_1\lambda_2\cos\omega\sin\omega \tag{38.23}$$

For $0 < \omega < \pi$, we have $\sin\omega > 0$, so the derivative equals zero if and only if:

$$(1+\lambda_1^2)\lambda_2 + (1+\lambda_2^2)\lambda_1 = 4\lambda_1\lambda_2\cos\omega \tag{38.24}$$

For *complex conjugate roots* $\lambda_1 = re^{i\theta}$, $\lambda_2 = re^{-i\theta}$, substitution into (38.24) gives:

$$\cos\omega = \frac{1+r^2}{2r}\cos\theta \tag{38.25}$$

The second derivative confirms this is a maximum when $\omega < \frac{3\pi}{4}$.

The necessary condition for a valid solution is:

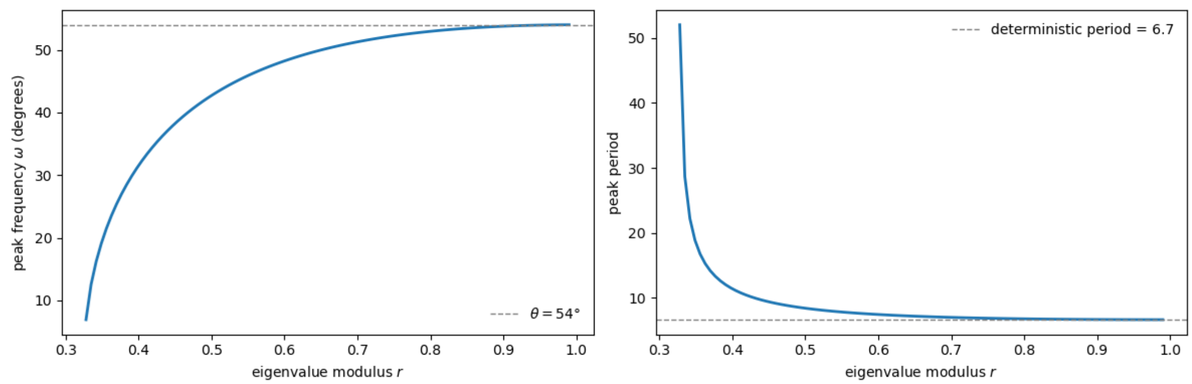
$$-1 < \frac{1+r^2}{2r}\cos\theta < 1 \tag{38.26}$$

We can interpret it as:

- When $r \approx 1$, the factor $(1 + r^2)/2r \approx 1$, so $\omega \approx \theta$
- When r is small (e.g., 0.3 or 0.4), condition (38.26) can only be satisfied if $\cos\theta \approx 0$, meaning $\theta \approx \pi/2$ (cycles of approximately 4 periods)

If $\theta = 54^{\circ}$ (corresponding to cycles of 6.67 periods) and $r = 0.4$, then $(1+r^2)/2r = 1.45$, giving $\cos\omega = 1.45 \times 0.588 = 0.85$, or $\omega = 31.5^{\circ}$, corresponding to cycles of 11.4 periods, which is much longer than the deterministic cycle.

```python
def peak_condition_factor(r):
    """Compute (1 + r^2) / (2r)"""
    return (1 + r**2) / (2 * r)

θ_deg = 54
θ = np.deg2rad(θ_deg)
r_grid = np.linspace(0.3, 0.99, 100)

# For each r, compute the implied peak frequency
ω_peak = []
for r in r_grid:
    factor = peak_condition_factor(r)
    cos_ω = factor * np.cos(θ)
    if -1 < cos_ω < 1:
        ω_peak.append(np.arccos(cos_ω))
    else:
        ω_peak.append(np.nan)

ω_peak = np.array(ω_peak)
period_peak = 2 * np.pi / ω_peak

fig, axes = plt.subplots(1, 2, figsize=(12, 4))

axes[0].plot(r_grid, np.rad2deg(ω_peak), lw=2)
axes[0].axhline(θ_deg, ls='--', lw=1.0, color='gray',
        label=rf'$\theta = {θ_deg}°$')
axes[0].set_xlabel('eigenvalue modulus $r$')
axes[0].set_ylabel(r'peak frequency $\omega$ (degrees)')
axes[0].legend(frameon=False)

axes[1].plot(r_grid, period_peak, lw=2)
axes[1].axhline(360/θ_deg, ls='--', lw=1.0, color='gray',
        label=rf'deterministic period = {360/θ_deg:.1f}')
axes[1].set_xlabel('eigenvalue modulus $r$')
axes[1].set_ylabel('peak period')
axes[1].legend(frameon=False)

plt.tight_layout()
plt.show()

r_example = 0.4
factor = peak_condition_factor(r_example)
cos_ω = factor * np.cos(θ)
ω_example = np.arccos(cos_ω)
print(f"Chow's example: r = {r_example}, θ = {θ_deg}°")
print(f"  cos(ω) = {cos_ω:.3f}")
print(f"  ω = {np.rad2deg(ω_example):.1f}°")
print(f"  Peak period = {360/np.rad2deg(ω_example):.1f}")
```




```
Chow's example: r = 0.4, θ = 54°
  cos(ω) = 0.852
  ω = 31.5°
  Peak period = 11.4
```

As $r \to 1$, the peak frequency converges to θ.

For smaller r, the peak frequency can differ substantially from the deterministic oscillation frequency.

38.6.4 Real positive roots cannot produce peaks

For *real and positive roots* $\lambda_1, \lambda_2 > 0$, the first-order condition (38.24) cannot be satisfied.

To see why, recall that a spectral peak at an interior frequency $\omega \in (0, \pi)$ requires

$$\cos\omega = \frac{(1+\lambda_1^2)\lambda_2 + (1+\lambda_2^2)\lambda_1}{4\lambda_1\lambda_2}.$$

For this to have a solution, we need the right-hand side to lie in $[-1, 1]$.

But for positive λ_1, λ_2, the numerator exceeds $4\lambda_1\lambda_2$:

$$(1+\lambda_1^2)\lambda_2 + (1+\lambda_2^2)\lambda_1 - 4\lambda_1\lambda_2 = \lambda_1(1-\lambda_2)^2 + \lambda_2(1-\lambda_1)^2. \tag{38.27}$$

The right-hand side is a sum of two non-negative terms (each is a positive number times a square).

It equals zero only if both $\lambda_1 = 1$ and $\lambda_2 = 1$, which violates the stability condition $|\lambda_i| < 1$.

For any stable system with real positive roots, this expression is strictly positive, so

$$\cos\omega = \frac{(1+\lambda_1^2)\lambda_2 + (1+\lambda_2^2)\lambda_1}{4\lambda_1\lambda_2} > 1, \tag{38.28}$$

which is impossible.

This is a key result: in the Hansen-Samuelson model, *complex roots are necessary* for a spectral peak at interior frequencies.

The following figure illustrates the difference in spectra between a case with complex roots and a case with real roots

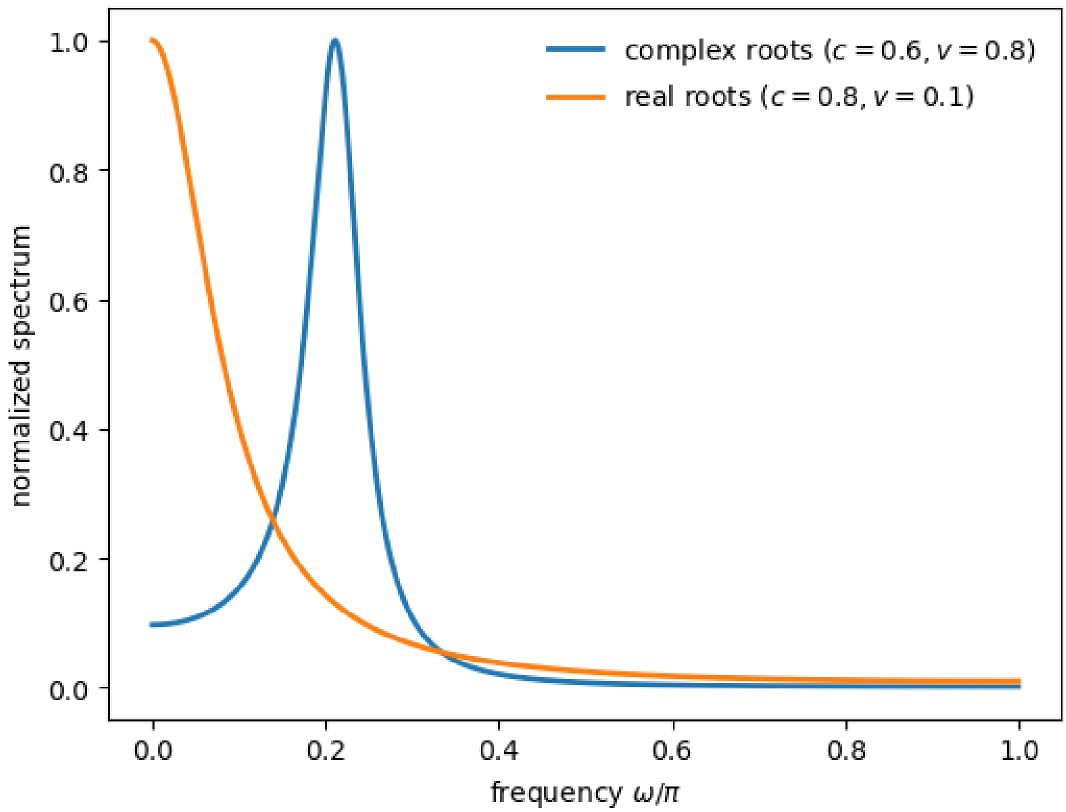
```
ω_grid = np.linspace(1e-3, np.pi - 1e-3, 800)
V_hs = np.array([[1.0, 0.0], [0.0, 0.0]])  # shock only in first equation

# Case 1: Complex roots (c=0.6, v=0.8)
c_complex, v_complex = 0.6, 0.8
A_complex = samuelson_transition(c_complex, v_complex)
```

(continues on next page)

(continued from previous page)

```
eig_complex = np.linalg.eigvals(A_complex)

# Case 2: Real roots (c=0.8, v=0.1)
c_real, v_real = 0.8, 0.1
A_real = samuelson_transition(c_real, v_real)
eig_real = np.linalg.eigvals(A_real)

print(
    f"Complex case (c={c_complex}, v={v_complex}): eigenvalues = {eig_complex}")
print(
    f"Real case (c={c_real}, v={v_real}): eigenvalues = {eig_real}")

F_complex = spectral_density_var1(A_complex, V_hs, ω_grid)
F_real = spectral_density_var1(A_real, V_hs, ω_grid)

f11_complex = np.real(F_complex[:, 0, 0])
f11_real = np.real(F_real[:, 0, 0])

fig, ax = plt.subplots()
ax.plot(ω_grid / np.pi, f11_complex / np.max(f11_complex), lw=2,
        label=fr'complex roots ($c={c_complex}, v={v_complex}$)')
ax.plot(ω_grid / np.pi, f11_real / np.max(f11_real), lw=2,
        label=fr'real roots ($c={c_real}, v={v_real}$)')
ax.set_xlabel(r'frequency $\omega/\pi$')
ax.set_ylabel('normalized spectrum')
ax.legend(frameon=False)
plt.show()
```

```
Complex case (c=0.6, v=0.8): eigenvalues = [0.7+0.55677644j 0.7-0.55677644j]
Real case (c=0.8, v=0.1): eigenvalues = [0.77015621 0.12984379]
```



With complex roots, the spectrum has a clear interior peak.

With real roots, the spectrum is monotonically decreasing and no interior peak is possible.

38.7 Real roots can produce peaks in general models

While real positive roots cannot produce spectral peaks in the Hansen-Samuelson model, Chow [1968] emphasizes that this is *not true in general*.

In multivariate systems, the spectral density of a linear combination of variables can have interior peaks even when all eigenvalues are real and positive.

38.7.1 Example

Chow constructs the following explicit example with two real positive eigenvalues:

$$\lambda_1 = 0.1, \quad \lambda_2 = 0.9 \tag{38.29}$$

$$w_{11} = w_{22} = 1, \quad w_{12} = 0.8 \tag{38.30}$$

$$b_{m1} = 1, \quad b_{m2} = -0.01 \tag{38.31}$$

The spectral density of the linear combination $x_t = b_m^\top y_t$ is:

$$f_{mm}(\omega) = \frac{0.9913}{1.01 - 0.2\cos\omega} - \frac{0.001570}{1.81 - 1.8\cos\omega} \tag{38.32}$$

Chow tabulates the values:

ω	0	$\pi/8$	$2\pi/8$	$3\pi/8$	$4\pi/8$	$5\pi/8$	$6\pi/8$	$7\pi/8$	π
$f_{mm}(\omega)$	1.067	1.183	1.191	1.138	1.061	0.981	0.912	0.860	0.829

The peak at ω slightly below $\pi/8$ (corresponding to periods around 11) is "quite pronounced."

In the following figure, we reproduce this table, but with Python, we can plot a finer grid to find the peak more accurately

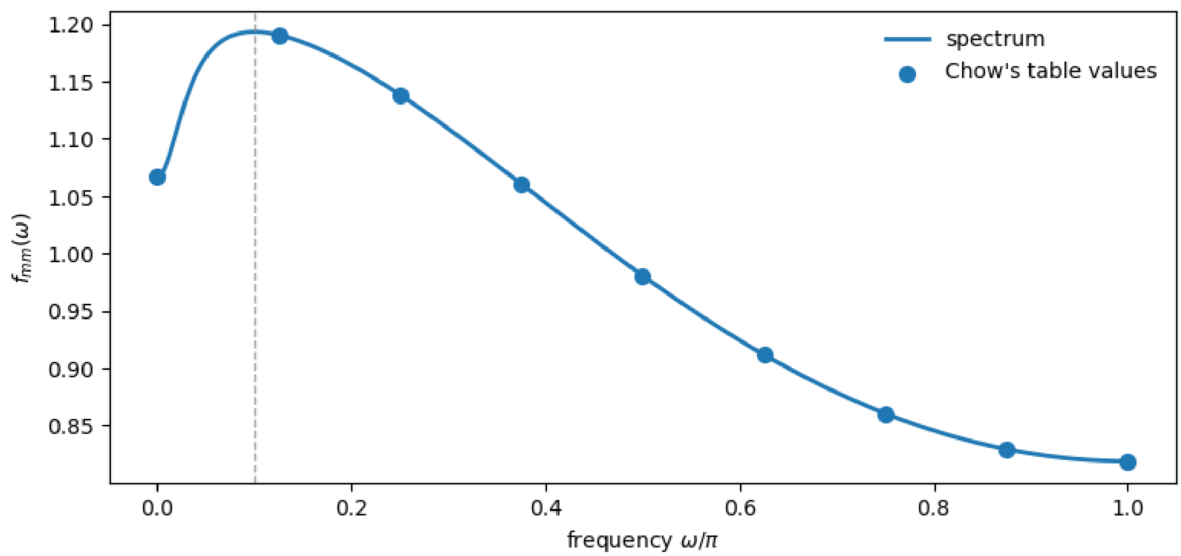
```
λ1, λ2 = 0.1, 0.9
w11, w22, w12 = 1.0, 1.0, 0.8
bm1, bm2 = 1.0, -0.01

# Construct the system
A_chow_ex = np.diag([λ1, λ2])

# W is the canonical shock covariance; we need V = B W B^T
# For diagonal A with distinct eigenvalues, B = I, so V = W
V_chow_ex = np.array([[w11, w12], [w12, w22]])
b_chow_ex = np.array([bm1, bm2])

# Chow's formula
def chow_spectrum_formula(ω):
    term1 = 0.9913 / (1.01 - 0.2 * np.cos(ω))
    term2 = 0.001570 / (1.81 - 1.8 * np.cos(ω))
    return term1 - term2

# Compute via formula and via our general method
ω_table = np.array([0, np.pi/8, 2*np.pi/8, 3*np.pi/8, 4*np.pi/8,
                    5*np.pi/8, 6*np.pi/8, 7*np.pi/8, np.pi])
f_formula = np.array([chow_spectrum_formula(ω) for ω in ω_table])

# General method
ω_grid_fine = np.linspace(1e-4, np.pi, 1000)
F_chow_ex = spectral_density_var1(A_chow_ex, V_chow_ex, ω_grid_fine)
f_general = spectrum_of_linear_combination(F_chow_ex, b_chow_ex)

# Normalize to match Chow's table scale
scale = f_formula[0] / spectrum_of_linear_combination(
    spectral_density_var1(
        A_chow_ex, V_chow_ex, np.array([0.0])), b_chow_ex)[0]

print("Chow's Table (equation 67):")
print("ω/π:        ", "  ".join([f"{ω/np.pi:.3f}" for ω in ω_table]))
print("f_mm(ω):    ", "  ".join([f"{f:.3f}" for f in f_formula]))

fig, ax = plt.subplots(figsize=(9, 4))
ax.plot(ω_grid_fine / np.pi, f_general * scale, lw=2,
            label='spectrum')
ax.scatter(ω_table / np.pi, f_formula, s=50, zorder=3,
            label="Chow's table values")

# Mark the peak
i_peak = np.argmax(f_general)
ω_peak = ω_grid_fine[i_peak]
ax.axvline(ω_peak / np.pi, ls='--', lw=1.0, color='gray', alpha=0.7)
```

(continues on next page)

(continued from previous page)

```
ax.set_xlabel(r'frequency $\omega/\pi$')
ax.set_ylabel(r'$f_{mm}(\omega)$')
ax.legend(frameon=False)
plt.show()

print(f"\nPeak at ω/π ≈ {ω_peak/np.pi:.3f}, period ≈ {2*np.pi/ω_peak:.1f}")
```

```
Chow's Table (equation 67):
ω/π:        0.000  0.125  0.250  0.375  0.500  0.625  0.750  0.875  1.000
f_mm(ω):    1.067  1.191  1.138  1.061  0.981  0.912  0.860  0.829  0.819

Peak at ω/π ≈ 0.100, period ≈ 20.0
```



The peak appears at $\omega/\pi \approx 0.10$, which corresponds to a cycle length of approximately 20 periods, again much longer than the deterministic cycles implied by the eigenvalues.

38.7.2 The Slutsky connection

Chow connects this result to Slutsky's [Slutsky, 1927] finding that moving averages of a random series have recurrent cycles.

The VAR(1) model can be written as an infinite moving average:

$$y_t = u_t + Au_{t-1} + A^2 u_{t-2} + \cdots \tag{38.33}$$

This amounts to taking an infinite moving average of the random vectors u_t with "geometrically declining" weights $A^0, A^1, A^2, \ldots$

For a scalar process with $0 < \lambda < 1$, no distinct cycles can emerge.

But for a matrix A with real roots between 0 and 1, cycles *can* emerge in linear combinations of the variables.

As Chow puts it: "When neither of two (canonical) variables has distinct cycles… a linear combination can have a peak in its spectral density."

38.7.3 The general lesson

The examples above illustrate the following central points:

1. In the *Hansen-Samuelson model specifically*, complex roots are necessary for a spectral peak
2. But in *general multivariate systems*, complex roots are neither necessary nor sufficient
3. The full spectral shape depends on:
   - The eigenvalues of A
   - The shock covariance structure V
   - How the observable of interest loads on the eigenmodes (the vector b)

38.8 A calibrated model in the frequency domain

Chow and Levitan [1969] use the frequency-domain objects from Chow [1968] to study a calibrated annual macroeconometric model.

They work with five annual aggregates:

- $y_1 = C$ (consumption),
- $y_2 = I_1$ (equipment plus inventories),
- $y_3 = I_2$ (construction),
- $y_4 = R_a$ (long rate),
- $y_5 = Y_1 = C + I_1 + I_2$ (private-domestic GNP),

and add $y_6 = y_{1,t-1}$ to rewrite the original system in first-order form.

Throughout this section, frequency is measured in cycles per year, $f = \omega/2\pi \in [0, 1/2]$.

Following the paper, we normalize each spectrum to have area 1 over $[0, 1/2]$ so plots compare shape rather than scale.

Our goal is to reconstruct the transition matrix A and then compute and interpret the model-implied spectra, gains/coherences, and phase differences.

38.8.1 The cycle subsystem

The paper starts from a reduced form with exogenous inputs,

$$y_t = Ay_{t-1} + Cx_t + u_t. \tag{38.34}$$

To study cycles, they remove the deterministic component attributable to x_t and focus on the zero-mean subsystem

$$y_t = Ay_{t-1} + u_t. \tag{38.35}$$

For second moments, the only additional ingredient is the covariance matrix $V = \mathbb{E}[u_t u_t^\top]$.

Chow and Levitan compute it from structural parameters via

$$V = M^{-1}\Sigma(M^{-1})^\top \tag{38.36}$$

where Σ is the covariance of structural residuals and M is the matrix of contemporaneous structural coefficients.

Here we take A and V as given and ask what they imply for spectra and cross-spectra.

The 6×6 reduced-form shock covariance matrix V (scaled by 10^{-7}) reported by Chow and Levitan is:

$$V = \begin{bmatrix} 8.250 & 7.290 & 2.137 & 2.277 & 17.68 & 0 \\ 7.290 & 7.135 & 1.992 & 2.165 & 16.42 & 0 \\ 2.137 & 1.992 & 0.618 & 0.451 & 4.746 & 0 \\ 2.277 & 2.165 & 0.451 & 1.511 & 4.895 & 0 \\ 17.68 & 16.42 & 4.746 & 4.895 & 38.84 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}. \tag{38.37}$$

The sixth row and column are zeros because y_6 is an identity (lagged y_1).

The transition matrix A has six characteristic roots:

$$\begin{aligned} &\lambda_1 = 0.9999725, \quad \lambda_2 = 0.9999064, \quad \lambda_3 = 0.4838, \\ &\lambda_4 = 0.0761 + 0.1125i, \quad \lambda_5 = 0.0761 - 0.1125i, \quad \lambda_6 = -0.00004142. \end{aligned} \tag{38.38}$$

Two roots are near unity because two structural equations are in first differences.

One root (λ_6) is theoretically zero because of the identity $y_5 = y_1 + y_2 + y_3$.

The complex conjugate pair $\lambda_{4,5}$ has modulus $|\lambda_4| = \sqrt{0.0761^2 + 0.1125^2} \approx 0.136$.

The right eigenvector matrix B (columns are eigenvectors corresponding to $\lambda_1, \dots, \lambda_6$):

$$B = \begin{bmatrix} -0.008 & 1.143 & 0.320 & 0.283 + 0.581i & 0.283 - 0.581i & 0.000 \\ -0.000 & 0.013 & -0.586 & -2.151 + 0.742i & -2.151 - 0.742i & 2.241 \\ -0.001 & 0.078 & 0.889 & -0.215 + 0.135i & -0.215 - 0.135i & 0.270 \\ 1.024 & 0.271 & 0.069 & -0.231 + 0.163i & -0.231 - 0.163i & 0.307 \\ -0.009 & 1.235 & 0.623 & -2.082 + 1.468i & -2.082 - 1.468i & 2.766 \\ -0.008 & 1.143 & 0.662 & 4.772 + 0.714i & 4.772 - 0.714i & -4.399 \end{bmatrix}. \tag{38.39}$$

Together, V, $\{\lambda_i\}$, and B are sufficient to compute all spectral and cross-spectral densities.

38.8.2 Reconstructing A and computing $F(\omega)$

The paper reports (λ, B, V), which is enough to reconstruct $A = B\,\mathrm{diag}(\lambda_1, \dots, \lambda_6)\,B^{-1}$ and then compute the model-implied spectral objects.

```
λ = np.array([
    0.9999725, 0.9999064, 0.4838,
    0.0761 + 0.1125j, 0.0761 - 0.1125j, -0.00004142
], dtype=complex)

B = np.array([
    [-0.008, 1.143, 0.320, 0.283+0.581j, 0.283-0.581j, 0.000],
    [-0.000, 0.013, -0.586, -2.151+0.742j, -2.151-0.742j, 2.241],
    [-0.001, 0.078, 0.889, -0.215+0.135j, -0.215-0.135j, 0.270],
    [1.024, 0.271, 0.069, -0.231+0.163j, -0.231-0.163j, 0.307],
    [-0.009, 1.235, 0.623, -2.082+1.468j, -2.082-1.468j, 2.766],
    [-0.008, 1.143, 0.662, 4.772+0.714j, 4.772-0.714j, -4.399]
], dtype=complex)

V = np.array([
    [8.250, 7.290, 2.137, 2.277, 17.68, 0],
    [7.290, 7.135, 1.992, 2.165, 16.42, 0],
    [2.137, 1.992, 0.618, 0.451, 4.746, 0],
    [2.277, 2.165, 0.451, 1.511, 4.895, 0],
```

(continues on next page)

(continued from previous page)

```
    [17.68, 16.42, 4.746, 4.895, 38.84, 0],
    [0, 0, 0, 0, 0, 0]
]) * 1e-7

D_λ = np.diag(λ)
A_chow = B @ D_λ @ np.linalg.inv(B)
A_chow = np.real(A_chow)
print("eigenvalues of reconstructed A:")
print(np.linalg.eigvals(A_chow).round(6))
```

```
eigenvalues of reconstructed A:
[-4.10000e-05+0.j      7.61000e-02+0.1125j  7.61000e-02-0.1125j
  4.83800e-01+0.j      9.99906e-01+0.j      9.99973e-01+0.j    ]
```

38.8.3 Canonical coordinates

Chow and Levitan's canonical transformation uses $z_t = B^{-1} y_t$, giving dynamics $z_t = D_\lambda z_{t-1} + e_t$.

Accordingly, the canonical shock covariance is

$$W = B^{-1} V (B^{-1})^\top.$$

```
B_inv = np.linalg.inv(B)
W = B_inv @ V @ B_inv.T
print("diagonal of W:")
print(np.diag(W).round(10))
```

```
diagonal of W:
[ 8.560e-08-0.00e+00j  3.638e-07-0.00e+00j  1.300e-08-0.00e+00j
 -7.880e-08+6.26e-08j -7.880e-08-6.26e-08j -1.650e-08+0.00e+00j]
```

Chow and Levitan derive the following closed-form formula for the spectral density matrix:

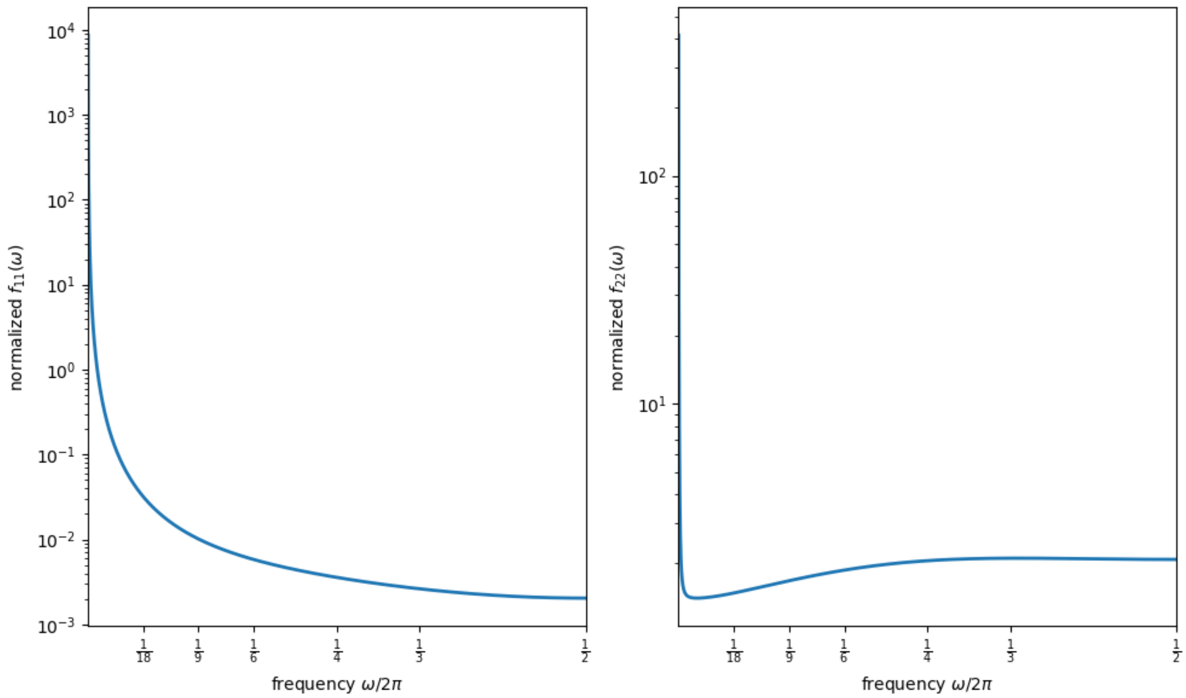
$$F(\omega) = B \left[\frac{w_{ij}}{(1 - \lambda_i e^{-i\omega})(1 - \lambda_j e^{i\omega})} \right] B^\top, \tag{38.40}$$

where w_{ij} are elements of the canonical shock covariance W.

```
def spectral_density_chow(λ, B, W, ω_grid):
    """Spectral density via Chow's eigendecomposition formula."""
    p = len(λ)
    F = np.zeros((len(ω_grid), p, p), dtype=complex)
    for k, ω in enumerate(ω_grid):
        F_star = np.zeros((p, p), dtype=complex)
        for i in range(p):
            for j in range(p):
                denom = (1 - λ[i] * np.exp(-1j * ω)) \
                    * (1 - λ[j] * np.exp(1j * ω))
                F_star[i, j] = W[i, j] / denom
        F[k] = B @ F_star @ B.T
    return F / (2 * np.pi)

freq = np.linspace(1e-4, 0.5, 5000)     # cycles/year in [0, 1/2]
ω_grid = 2 * np.pi * freq               # radians in [0, π]
F_chow = spectral_density_chow(λ, B, W, ω_grid)
```

Let's plot the univariate spectra of consumption (y_1) and equipment plus inventories (y_2)

```
variable_names = ['$C$', '$I_1$', '$I_2$', '$R_a$', '$Y_1$']
freq_ticks = [1/18, 1/9, 1/6, 1/4, 1/3, 1/2]
freq_labels = [r'$\frac{1}{18}$', r'$\frac{1}{9}$', r'$\frac{1}{6}$',
               r'$\frac{1}{4}$', r'$\frac{1}{3}$', r'$\frac{1}{2}$']

def paper_frequency_axis(ax):
    ax.set_xlim([0.0, 0.5])
    ax.set_xticks(freq_ticks)
    ax.set_xticklabels(freq_labels)
    ax.set_xlabel(r'frequency $\omega/2\pi$')

# Normalized spectra (areas set to 1)
S = np.real(np.diagonal(F_chow, axis1=1, axis2=2))[:, :5]
df = np.diff(freq)
areas = np.sum(0.5 * (S[1:] + S[:-1]) * df[:, None], axis=0)
S_norm = S / areas
mask = freq >= 0.0

fig, axes = plt.subplots(1, 2, figsize=(10, 6))

# Figure I.1: consumption (log scale)
axes[0].plot(freq[mask], S_norm[mask, 0], lw=2)
axes[0].set_yscale('log')
paper_frequency_axis(axes[0])
axes[0].set_ylabel(r'normalized $f_{11}(\omega)$')

# Figure I.2: equipment + inventories (log scale)
axes[1].plot(freq[mask], S_norm[mask, 1], lw=2)
axes[1].set_yscale('log')
paper_frequency_axis(axes[1])
axes[1].set_ylabel(r'normalized $f_{22}(\omega)$')

plt.tight_layout()
plt.show()

i_peak = np.argmax(S_norm[mask, 1])
f_peak = freq[mask][i_peak]
```



The left panel corresponds to consumption and declines monotonically with frequency.

It illustrates Granger's "typical spectral shape" for macroeconomic time series.

The right panel corresponds to equipment plus inventories and shows the clearest (but still very flat) interior-frequency bump.

Chow and Levitan associate the dominance of very low frequencies in both plots with strong persistence and long-run movements.

Very large low-frequency power can arise from eigenvalues extremely close to one, which occurs mechanically when some equations are written in first differences.

Local peaks are not automatic: complex roots may have small modulus, and multivariate interactions can generate peaks even when all roots are real.

The interior bump in the right panel corresponds to cycles of roughly three years, with the spectrum nearly flat over cycles between about two and four years.

(This discussion follows Section II of [Chow and Levitan, 1969].)

38.8.4 How variables move together across frequencies

Beyond univariate spectra, we can ask how pairs of variables covary at each frequency.

The **cross-spectrum** $f_{ij}(\omega) = c_{ij}(\omega) - i \cdot q_{ij}(\omega)$ decomposes into the cospectrum c_{ij} and the quadrature spectrum q_{ij}.

The **cross-amplitude** is $g_{ij}(\omega) = |f_{ij}(\omega)| = \sqrt{c_{ij}^2 + q_{ij}^2}$.

The **squared coherence** measures linear association at frequency ω:

$$R_{ij}^2(\omega) = \frac{|f_{ij}(\omega)|^2}{f_{ii}(\omega) f_{jj}(\omega)} \in [0, 1]. \tag{38.41}$$

Coherence measures how much of the variance of y_i at frequency ω can be "explained" by y_j at the same frequency.

High coherence means the two series move together tightly at that frequency.

The **gain** is the frequency-response coefficient when regressing y_i on y_j:

$$G_{ij}(\omega) = \frac{|f_{ij}(\omega)|}{f_{jj}(\omega)}. \tag{38.42}$$

It measures how much y_i responds to a unit change in y_j at frequency ω.

For instance, a gain of 0.9 at low frequencies means long-cycle movements in y_j translate almost one-for-one to y_i, and a gain of 0.3 at high frequencies means short-cycle movements are dampened.

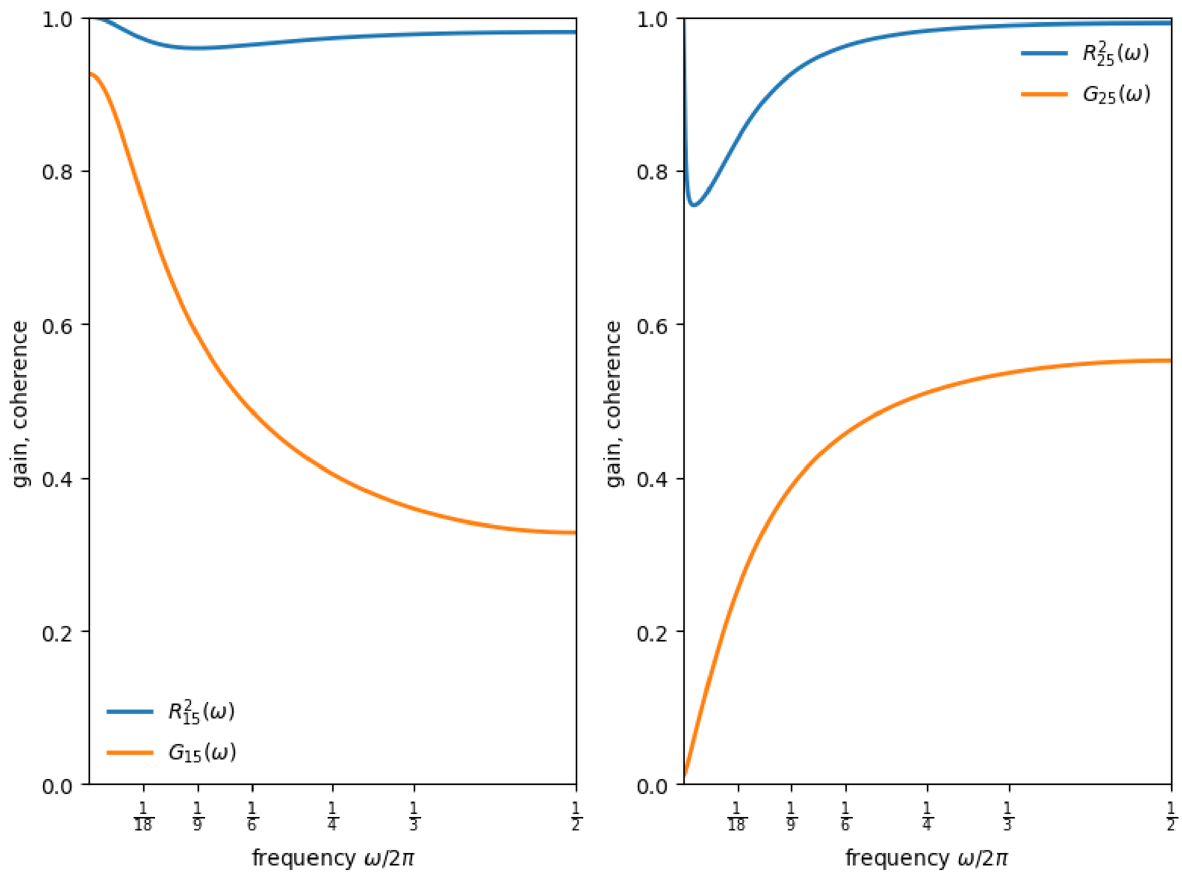
The **phase** captures lead-lag relationships (in radians):

$$\Delta_{ij}(\omega) = \tan^{-1}\left(\frac{q_{ij}(\omega)}{c_{ij}(\omega)}\right). \tag{38.43}$$

```
def cross_spectral_measures(F, i, j):
    """Compute coherence, gain (y_i on y_j), and phase between variables i and j."""
    f_ij = F[:, i, j]
    f_ii, f_jj = np.real(F[:, i, i]), np.real(F[:, j, j])
    g_ij = np.abs(f_ij)
    coherence = (g_ij**2) / (f_ii * f_jj)
    gain = g_ij / f_jj
    phase = np.arctan2(-np.imag(f_ij), np.real(f_ij))
    return coherence, gain, phase
```

We now plot gain and coherence as in Figures II.1–II.4 of [Chow and Levitan, 1969].

```
gnp_idx = 4

fig, axes = plt.subplots(1, 2, figsize=(8, 6))

for idx, var_idx in enumerate([0, 1]):
    coherence, gain, phase = cross_spectral_measures(F_chow, var_idx, gnp_idx)
    ax = axes[idx]

    ax.plot(freq[mask], coherence[mask],
            lw=2, label=rf'$R^2_{{{var_idx+1}5}}(\omega)$')
    ax.plot(freq[mask], gain[mask],
            lw=2, label=rf'$G_{{{var_idx+1}5}}(\omega)$')
    paper_frequency_axis(ax)
    ax.set_ylim([0, 1.0])
    ax.set_ylabel('gain, coherence')
    ax.legend(frameon=False, loc='best')

plt.tight_layout()
plt.show()
```



The gain and coherence patterns differ across components (Figures II.1–II.2 of [Chow and Levitan, 1969]):

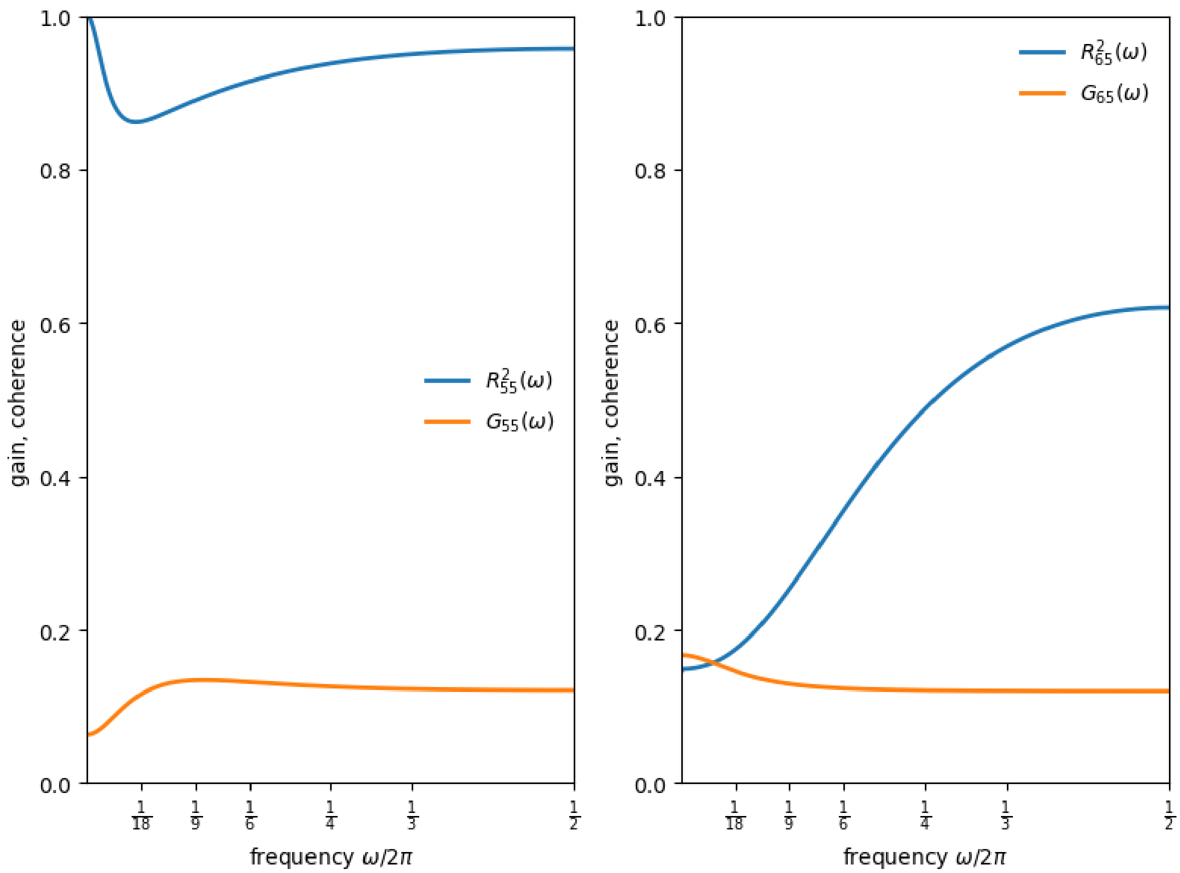
- Consumption vs private-domestic GNP (left panel):
  - Gain is about 0.9 at very low frequencies but falls below 0.4 for cycles shorter than four years.
  - This is evidence that short-cycle income movements translate less into consumption than long-cycle movements, consistent with permanent-income interpretations.
  - Coherence remains high throughout.
- Equipment plus inventories vs private-domestic GNP (right panel):
  - Gain *rises* with frequency, exceeding 0.5 for short cycles.
  - This is the frequency-domain signature of acceleration and volatile short-run inventory movements.

```
fig, axes = plt.subplots(1, 2, figsize=(8, 6))

for idx, var_idx in enumerate([2, 3]):
    coherence, gain, phase = cross_spectral_measures(F_chow, var_idx, gnp_idx)
    ax = axes[idx]

    ax.plot(freq[mask], coherence[mask],
            lw=2, label=rf'$R^2_{{{var_idx+3}5}}(\omega)$')
    ax.plot(freq[mask], gain[mask],
            lw=2, label=rf'$G_{{{var_idx+3}5}}(\omega)$')
    paper_frequency_axis(ax)
    ax.set_ylim([0, 1.0])
```

(continues on next page)

(continued from previous page)

```
    ax.set_ylabel('gain, coherence')
    ax.legend(frameon=False, loc='best')

plt.tight_layout()
plt.show()
```



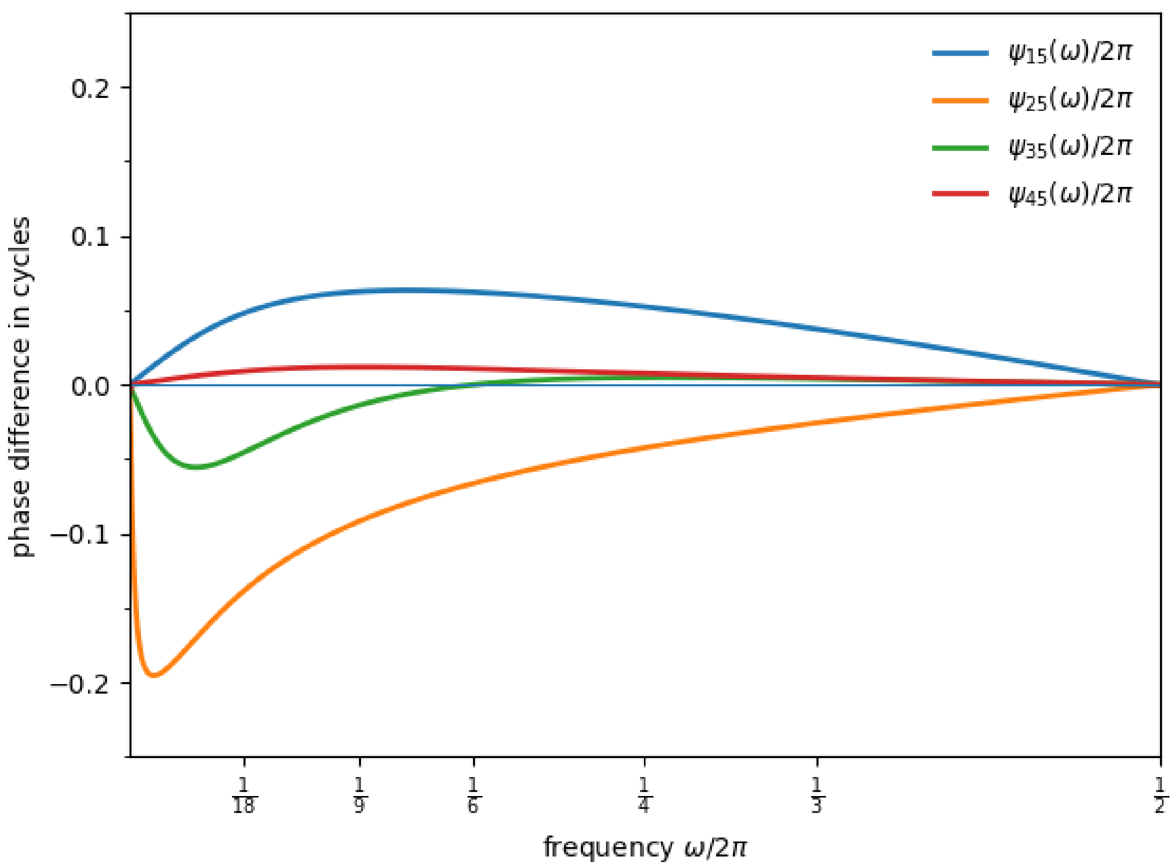
- New construction vs private-domestic GNP (left panel):
  - Gain peaks at medium cycle lengths (around 0.1 for short cycles).
  - Coherence for both investment series stays fairly high across frequencies.
- Long-bond yield vs private-domestic GNP (right panel):
  - Gain varies less across frequencies than real activity series.
  - Coherence with output is comparatively low at business-cycle frequencies, making it hard to explain interest-rate movements by inverting a money-demand equation.

38.8.5 Lead-lag relationships

The phase tells us which variable leads at each frequency.

Positive phase means output leads the component; negative phase means the component leads output.

```
fig, ax = plt.subplots()

labels = [r'$\psi_{15}(\omega)/2\pi$', r'$\psi_{25}(\omega)/2\pi$',
          r'$\psi_{35}(\omega)/2\pi$', r'$\psi_{45}(\omega)/2\pi$']

for var_idx in range(4):
    coherence, gain, phase = cross_spectral_measures(F_chow, var_idx, gnp_idx)
    phase_cycles = phase / (2 * np.pi)
    ax.plot(freq[mask], phase_cycles[mask], lw=2, label=labels[var_idx])

ax.axhline(0, lw=0.8)
paper_frequency_axis(ax)
ax.set_ylabel('phase difference in cycles')
ax.set_ylim([-0.25, 0.25])
ax.set_yticks(np.arange(-0.25, 0.3, 0.05), minor=True)
ax.legend(frameon=False)
plt.tight_layout()
plt.show()
```



The phase relationships reveal that:

- Output leads consumption by a small fraction of a cycle (about 0.06 cycles at a 6-year period, 0.04 cycles at a 3-year period).

- Equipment plus inventories tends to lead output (by about 0.07 cycles at a 6-year period, 0.03 cycles at a 3-year period).
- New construction leads at low frequencies and is close to coincident at higher frequencies.
- The bond yield lags output slightly, remaining close to coincident in timing.

These implied leads and lags are broadly consistent with turning-point timing summaries reported elsewhere, and simulations of the same model deliver similar lead–lag ordering at turning points (Figure III of [Chow and Levitan, 1969]).

38.8.6 Building blocks of spectral shape

Each eigenvalue contributes a characteristic spectral shape through the *scalar kernel*

$$g_i(\omega) = \frac{1 - |\lambda_i|^2}{|1 - \lambda_i e^{-i\omega}|^2} = \frac{1 - |\lambda_i|^2}{1 + |\lambda_i|^2 - 2\mathrm{Re}(\lambda_i)\cos\omega + 2\mathrm{Im}(\lambda_i)\sin\omega}. \tag{38.44}$$

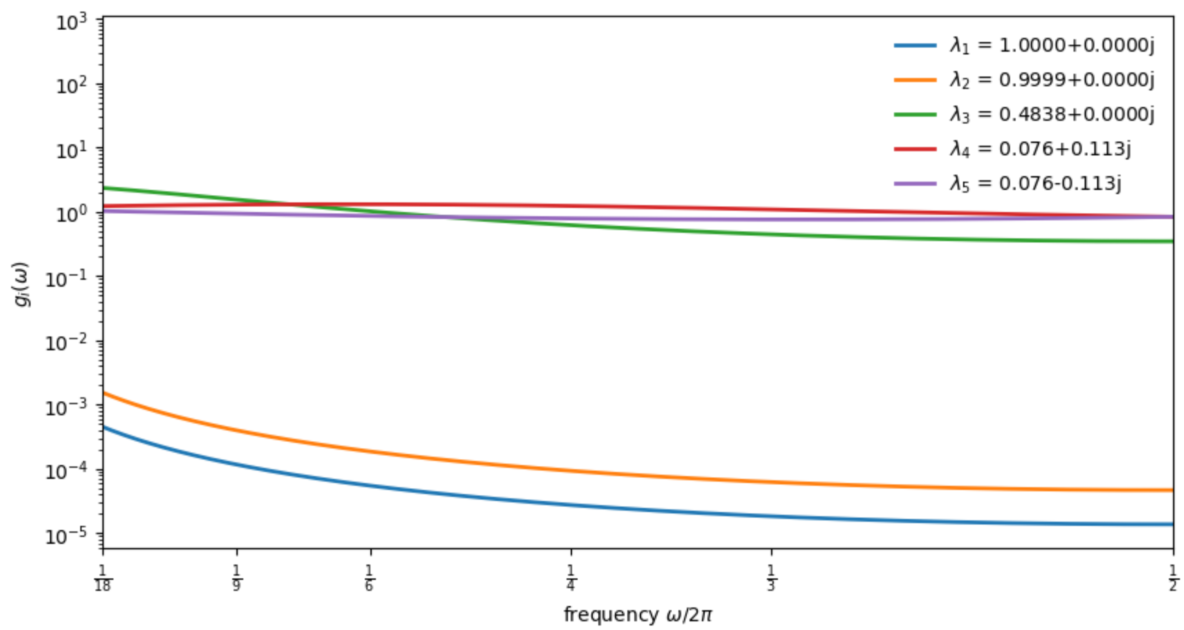
For real λ_i, this simplifies to

$$g_i(\omega) = \frac{1 - \lambda_i^2}{1 + \lambda_i^2 - 2\lambda_i \cos\omega}.$$

Each observable spectral density is a linear combination of these kernels (plus cross-terms).

Below, we plot the scalar kernels for each eigenvalue to see how they shape the overall spectra

```
def scalar_kernel(λ_i, ω_grid):
    """scalar spectral kernel g_i(ω)."""
    λ_i = complex(λ_i)
    mod_sq = np.abs(λ_i)**2
    return np.array(
        [(1 - mod_sq) / np.abs(1 - λ_i * np.exp(-1j * ω))**2
        for ω in ω_grid])

fig, ax = plt.subplots(figsize=(10, 5))
for i, λ_i in enumerate(λ):
    if np.abs(λ_i) > 0.01:
        g_i = scalar_kernel(λ_i, ω_grid)
        label = f'$\\lambda_{i+1}$ = {λ_i:.4f}' \
        if np.isreal(λ_i) else f'$\\lambda_{i+1}$ = {λ_i:.3f}'
        ax.semilogy(freq, g_i, label=label, lw=2)

ax.set_xlabel(r'frequency $\omega/2\pi$')
ax.set_ylabel('$g_i(\\omega)$')
ax.set_xlim([1/18, 0.5])
ax.set_xticks(freq_ticks)
ax.set_xticklabels(freq_labels)
ax.legend(frameon=False)
plt.show()
```



The figure reveals how eigenvalue magnitude shapes spectral contributions:

- *Near-unit eigenvalues* ($\lambda_1, \lambda_2 \approx 1$) produce kernels sharply peaked at low frequencies as these drive the strong low-frequency power seen in the spectra above.
- *The moderate eigenvalue* ($\lambda_3 \approx 0.48$) contributes a flatter component that spreads power more evenly across frequencies.
- *The complex pair* ($\lambda_{4,5}$) has such small modulus ($|\lambda_{4,5}| \approx 0.136$) that its kernel is nearly flat, which is too weak to generate a pronounced interior peak.

This decomposition explains why the spectra look the way they do: the near-unit eigenvalues dominate, concentrating variance at very low frequencies.

The complex pair, despite enabling oscillatory dynamics in principle, has insufficient modulus to produce a visible spectral peak.

38.9 Summary

Chow [1968] draws several conclusions that remain relevant for understanding business cycles.

The acceleration principle receives strong empirical support: the negative coefficient on lagged output in investment equations is a robust finding across datasets.

The relationship between eigenvalues and spectral peaks is more subtle than it first appears:

- Complex roots guarantee oscillatory autocovariances, but they are neither necessary nor sufficient for a pronounced spectral peak.
- In the Hansen–Samuelson model specifically, complex roots *are* necessary for a peak.
- But in general multivariate systems, even real roots can produce peaks through the interaction of shocks and eigenvector loadings.

Chow and Levitan [1969] demonstrate what these objects look like in a calibrated system: strong low-frequency power from near-unit eigenvalues, frequency-dependent gains and coherences, and lead–lag relations that vary with cycle length.

Their results are consistent with Granger's "typical spectral shape" for economic time series.

That is a monotonically decreasing function of frequency, driven by the near-unit eigenvalues that arise when some equations are specified in first differences.

Understanding whether this shape reflects the true data-generating process requires analyzing the spectral densities implied by structural econometric models.

38.10 Exercises

Exercise 38.10.1

Plot impulse responses and spectra side-by-side for several values of the accelerator v in the Hansen-Samuelson model, showing how acceleration strength affects both the time-domain and frequency-domain signatures.

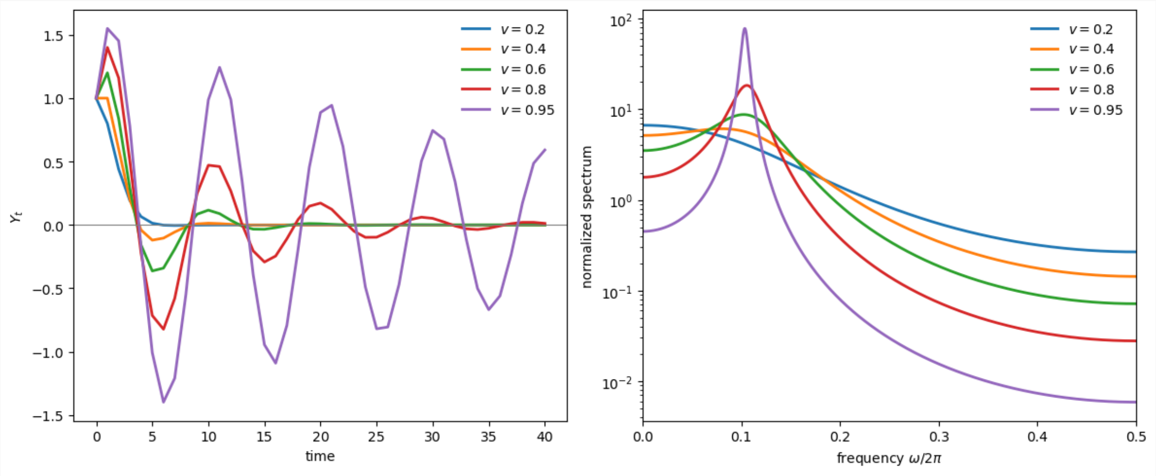
Use the same v values as in the main text: $v \in \{0.2, 0.4, 0.6, 0.8, 0.95\}$ with $c = 0.6$.

Solution

Here is one solution:

```
v_grid_ex1 = [0.2, 0.4, 0.6, 0.8, 0.95]
c_ex1 = 0.6
freq_ex1 = np.linspace(1e-4, 0.5, 2000)
ω_grid_ex1 = 2 * np.pi * freq_ex1
V_ex1 = np.array([[1.0, 0.0], [0.0, 0.0]])
T_irf_ex1 = 40

fig, axes = plt.subplots(1, 2, figsize=(12, 5))

for v in v_grid_ex1:
    A = samuelson_transition(c_ex1, v)

    # impulse response (left panel)
    s = np.array([1.0, 0.0])
    irf = np.empty(T_irf_ex1 + 1)
    for t in range(T_irf_ex1 + 1):
        irf[t] = s[0]
        s = A @ s
    axes[0].plot(range(T_irf_ex1 + 1), irf, lw=2, label=f'$v={v}$')

    # spectrum (right panel)
    F = spectral_density_var1(A, V_ex1, ω_grid_ex1)
    f11 = np.real(F[:, 0, 0])
    df = np.diff(freq_ex1)
    area = np.sum(0.5 * (f11[1:] + f11[:-1]) * df)
    f11_norm = f11 / area
    axes[1].plot(freq_ex1, f11_norm, lw=2, label=f'$v={v}$')

axes[0].axhline(0, lw=0.8, color='gray')
axes[0].set_xlabel('time')
axes[0].set_ylabel(r'$Y_t$')
axes[0].legend(frameon=False)

axes[1].set_xlabel(r'frequency $\omega/2\pi$')
```

```
axes[1].set_ylabel('normalized spectrum')
axes[1].set_xlim([0, 0.5])
axes[1].set_yscale('log')
axes[1].legend(frameon=False)

plt.tight_layout()
plt.show()
```



As v increases, eigenvalues approach the unit circle: oscillations become more persistent in the time domain (left), and the spectral peak becomes sharper in the frequency domain (right).

Complex roots produce a pronounced peak at interior frequencies—the spectral signature of business cycles.

Exercise 38.10.2

Verify spectral peak condition (38.25) numerically for the Hansen-Samuelson model.

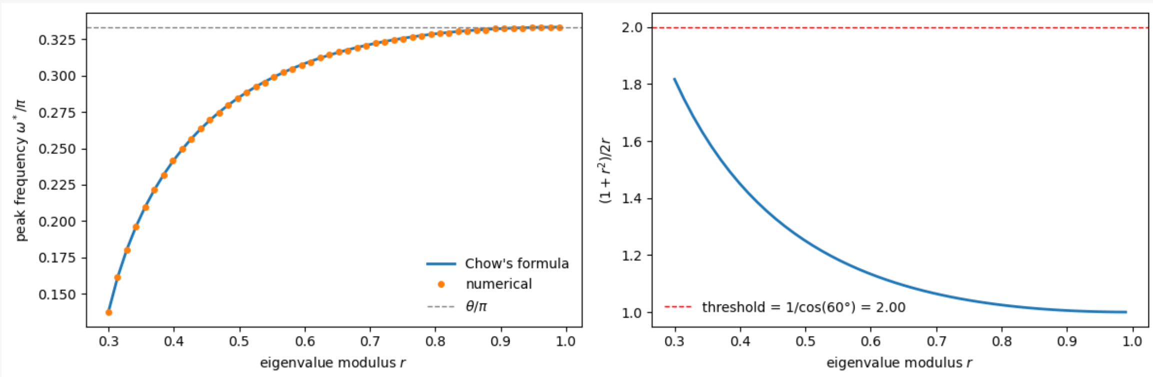
1. For a range of eigenvalue moduli $r \in [0.3, 0.99]$ with fixed $\theta = 60°$, compute:
   - The theoretical peak frequency from formula: $\cos\omega = \frac{1+r^2}{2r}\cos\theta$
   - The actual peak frequency by numerically maximizing the spectral density
2. Plot both on the same graph and verify they match.
3. Identify the range of r for which no valid peak exists (when the condition (38.26) is violated).

Solution

Here is one solution:

```
θ_ex = np.pi / 3  # 60 degrees
r_grid = np.linspace(0.3, 0.99, 50)
ω_grid_ex = np.linspace(1e-3, np.pi - 1e-3, 1000)
V_hs_ex = np.array([[1.0, 0.0], [0.0, 0.0]])

ω_theory = []
ω_numerical = []

for r in r_grid:
```

```
    # Theoretical peak
    factor = (1 + r**2) / (2 * r)
    cos_ω = factor * np.cos(θ_ex)
    if -1 < cos_ω < 1:
        ω_theory.append(np.arccos(cos_ω))
    else:
        ω_theory.append(np.nan)

    # Numerical peak from spectral density
    # Construct Hansen-Samuelson with eigenvalues r*exp(+-iθ)
    # This corresponds to c + v = 2r*cos(θ), v = r^2
    v = r**2
    c = 2 * r * np.cos(θ_ex) - v
    A_ex = samuelson_transition(c, v)
    F_ex = spectral_density_var1(A_ex, V_hs_ex, ω_grid_ex)
    f11 = np.real(F_ex[:, 0, 0])
    i_max = np.argmax(f11)

    # Only count as a peak if it's not at the boundary
    if 5 < i_max < len(ω_grid_ex) - 5:
        ω_numerical.append(ω_grid_ex[i_max])
    else:
        ω_numerical.append(np.nan)

ω_theory = np.array(ω_theory)
ω_numerical = np.array(ω_numerical)

fig, axes = plt.subplots(1, 2, figsize=(12, 4))

# Plot peak frequencies
axes[0].plot(r_grid, ω_theory / np.pi, lw=2, label="Chow's formula")
axes[0].plot(r_grid, ω_numerical / np.pi, 'o', markersize=4, label='numerical')
axes[0].axhline(θ_ex / np.pi, ls='--', lw=1.0, color='gray', label=r'$\theta/\pi$')
axes[0].set_xlabel('eigenvalue modulus $r$')
axes[0].set_ylabel(r'peak frequency $\omega^*/\pi$')
axes[0].legend(frameon=False)

# Plot the factor (1+r^2)/2r to show when peaks are valid
axes[1].plot(r_grid, (1 + r_grid**2) / (2 * r_grid), lw=2)
axes[1].axhline(1 / np.cos(θ_ex), ls='--', lw=1.0, color='red',
                label=f'threshold = 1/cos({np.rad2deg(θ_ex):.0f}°) = {1/np.cos(θ_
↪ex):.2f}')
axes[1].set_xlabel('eigenvalue modulus $r$')
axes[1].set_ylabel(r'$(1+r^2)/2r$')
axes[1].legend(frameon=False)

plt.tight_layout()
plt.show()

# Find threshold r below which no peak exists
valid_mask = ~np.isnan(ω_theory)
if valid_mask.any():
    r_threshold = r_grid[valid_mask][0]
    print(f"Peak exists for r >= {r_threshold:.2f}")
```




```
Peak exists for r >= 0.30
```

The theoretical and numerical peak frequencies match closely.

As $r \to 1$, the peak frequency converges to θ.

For smaller r, the factor $(1 + r^2)/2r$ exceeds the threshold, and no valid peak exists.

Exercise 38.10.3

In the "real roots but a peak" example, hold A fixed and vary the shock correlation (the off-diagonal entry of V) between 0 and 0.99.

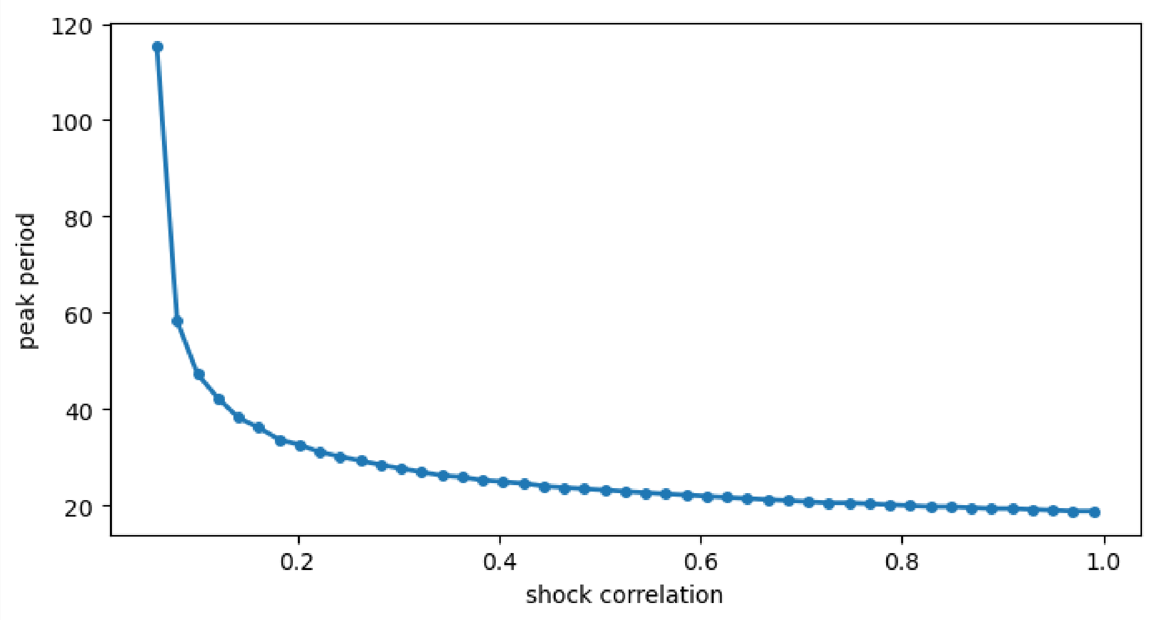
When does the interior-frequency peak appear, and how does its location change?

Solution

Here is one solution:

```
A_ex3 = np.diag([0.1, 0.9])
b_ex3 = np.array([1.0, -0.01])
corr_grid = np.linspace(0, 0.99, 50)
peak_periods = []
for corr in corr_grid:
    V_ex3 = np.array([[1.0, corr], [corr, 1.0]])
    F_ex3 = spectral_density_var1(A_ex3, V_ex3, ω_grid_ex)
    f_x = spectrum_of_linear_combination(F_ex3, b_ex3)
    i_max = np.argmax(f_x)
    if 5 < i_max < len(ω_grid_ex) - 5:
        peak_periods.append(2 * np.pi / ω_grid_ex[i_max])
    else:
        peak_periods.append(np.nan)

fig, ax = plt.subplots(figsize=(8, 4))
ax.plot(corr_grid, peak_periods, marker='o', lw=2, markersize=4)
ax.set_xlabel('shock correlation')
ax.set_ylabel('peak period')
plt.show()

threshold_idx = np.where(~np.isnan(peak_periods))[0]
if len(threshold_idx) > 0:
    print(
```

```
    f"interior peak when correlation >= {corr_grid[threshold_idx[0]]:.2f}")
```




```
interior peak when correlation >= 0.06
```

The interior peak appears only when the shock correlation exceeds a threshold.

This illustrates that spectral peaks depend on the full system structure, not just eigenvalues.

Exercise 38.10.4

Using the calibrated Chow-Levitan parameters, compute the autocovariance matrices $\Gamma_0, \Gamma_1, \ldots, \Gamma_{10}$ using:

1. The recursion $\Gamma_k = A\Gamma_{k-1}$ with Γ_0 from the Lyapunov equation.
2. Chow's eigendecomposition formula $\Gamma_k = BD_\lambda^k \Gamma_0^* B^\top$ where Γ_0^* is the canonical covariance.

Verify that both methods give the same result.

Solution

Here is one solution:

```
from scipy.linalg import solve_discrete_lyapunov

Γ_0_lyap = solve_discrete_lyapunov(A_chow, V)
Γ_recursion = [Γ_0_lyap]
for k in range(1, 11):
    Γ_recursion.append(A_chow @ Γ_recursion[-1])

p = len(λ)
Γ_0_star = np.zeros((p, p), dtype=complex)
for i in range(p):
    for j in range(p):
        Γ_0_star[i, j] = W[i, j] / (1 - λ[i] * λ[j])
```

```python
Γ_eigen = []
for k in range(11):
    D_k = np.diag(λ**k)
    Γ_eigen.append(np.real(B @ D_k @ Γ_0_star @ B.T))

print("Comparison of Γ_5 (first 3x3 block):")
print("\nRecursion method:")
print(np.real(Γ_recursion[5][:3, :3]).round(10))
print("\nEigendecomposition method:")
print(Γ_eigen[5][:3, :3].round(10))
print("\nMax absolute difference:",
        np.max(np.abs(np.real(Γ_recursion[5]) - Γ_eigen[5])))
```

```
Comparison of Γ_5 (first 3x3 block):

Recursion method:
[[2.5417901e-03 2.9310700e-05 1.7370210e-04]
 [2.8886900e-05 3.3220000e-07 1.9737000e-06]
 [1.7356020e-04 2.0028000e-06 1.1861700e-05]]

Eigendecomposition method:
[[2.5417901e-03 2.9310700e-05 1.7370210e-04]
 [2.8886900e-05 3.3220000e-07 1.9737000e-06]
 [1.7356020e-04 2.0028000e-06 1.1861700e-05]]

Max absolute difference: 9.500386588534582e-14
```

Both methods produce essentially identical results, up to numerical precision.

Exercise 38.10.5

Modify the Chow-Levitan model by changing λ_3 from 0.4838 to 0.95.

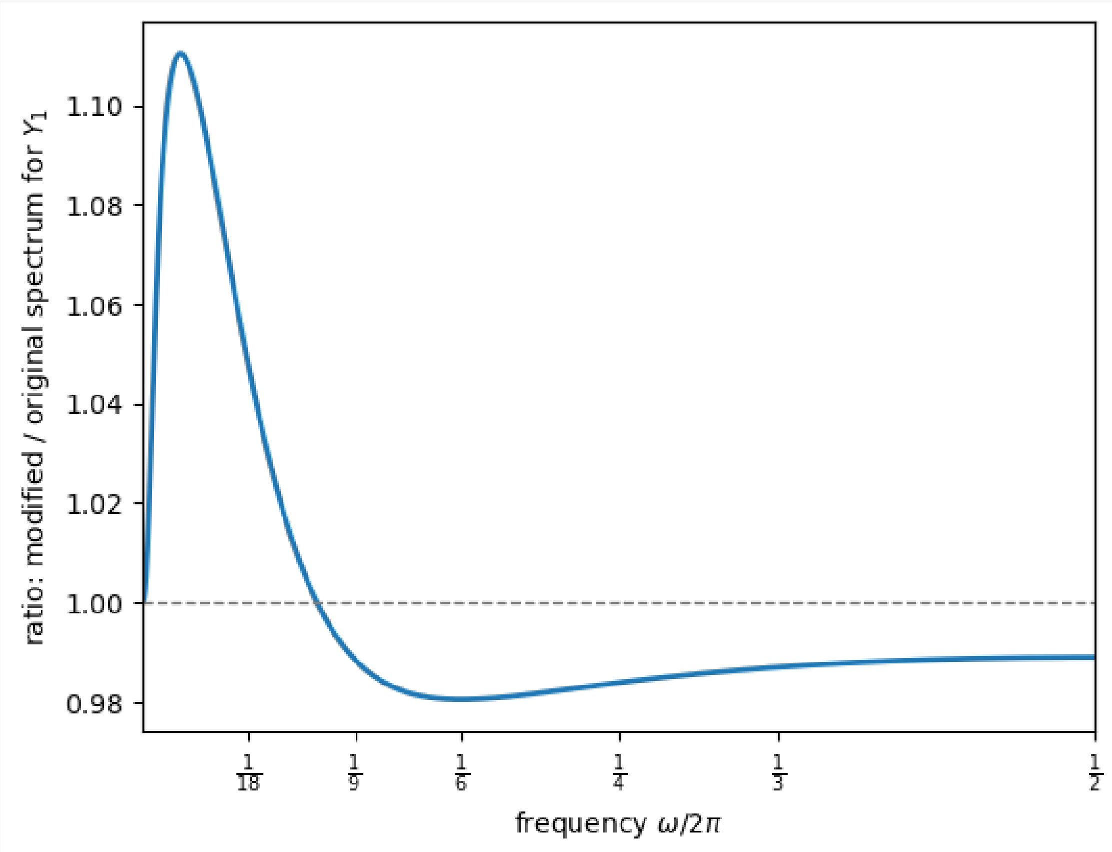
1. Recompute the spectral densities.
2. How does this change affect the spectral shape for each variable?
3. What economic interpretation might correspond to this parameter change?

Solution

Here is one solution:

```python
# Modify λ_3 and reconstruct the transition matrix
λ_modified = λ.copy()
λ_modified[2] = 0.95
D_λ_mod = np.diag(λ_modified)
A_mod = np.real(B @ D_λ_mod @ np.linalg.inv(B))

# Compute spectra using the VAR(1) formula with original V
F_mod = spectral_density_var1(A_mod, V, ω_grid)
F_orig = spectral_density_var1(A_chow, V, ω_grid)

# Plot ratio of spectra for output (Y_1)
f_orig = np.real(F_orig[:, 4, 4])
f_mod = np.real(F_mod[:, 4, 4])
```

```
fig, ax = plt.subplots()
ax.plot(freq, f_mod / f_orig, lw=2)
ax.axhline(1.0, ls='--', lw=1, color='gray')
paper_frequency_axis(ax)
ax.set_ylabel(r"ratio: modified / original spectrum for $Y_1$")
plt.show()
```



The near-unit eigenvalues ($\lambda_1, \lambda_2 \approx 0.9999$) dominate the output spectrum so heavily that changing λ_3 from 0.48 to 0.95 produces only a small relative effect.

The ratio plot reveals the change: the modified spectrum has slightly more power at low-to-medium frequencies and slightly less at high frequencies.

Economically, increasing λ_3 adds persistence to the mode it governs.

CHAPTER

THIRTYNINE

KESTEN PROCESSES AND FIRM DYNAMICS

Contents

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
!pip install --upgrade yfinance
```

39.1 Overview

Previously we learned about linear scalar-valued stochastic processes (AR(1) models).

Now we generalize these linear models slightly by allowing the multiplicative coefficient to be stochastic.

Such processes are known as Kesten processes after German–American mathematician Harry Kesten (1931–2019)

Although simple to write down, Kesten processes are interesting for at least two reasons:

1. A number of significant economic processes are or can be described as Kesten processes.
2. Kesten processes generate interesting dynamics, including, in some cases, heavy-tailed cross-sectional distributions.

We will discuss these issues as we go along.

Let's start with some imports:

```python
import matplotlib.pyplot as plt
import numpy as np
import quantecon as qe
```

The following two lines are only added to avoid a `FutureWarning` caused by compatibility issues between pandas and matplotlib.

```
from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters()
```

Additional technical background related to this lecture can be found in the monograph of [Buraczewski *et al.*, 2016].

39.2 Kesten Processes

A **Kesten process** is a stochastic process of the form

$$X_{t+1} = a_{t+1} X_t + \eta_{t+1} \tag{39.1}$$

where $\{a_t\}_{t \geq 1}$ and $\{\eta_t\}_{t \geq 1}$ are IID sequences.

We are interested in the dynamics of $\{X_t\}_{t \geq 0}$ when X_0 is given.

We will focus on the nonnegative scalar case, where X_t takes values in $\mathbb{R}_+$.

In particular, we will assume that

- the initial condition X_0 is nonnegative,
- $\{a_t\}_{t \geq 1}$ is a nonnegative IID stochastic process and
- $\{\eta_t\}_{t \geq 1}$ is another nonnegative IID stochastic process, independent of the first.

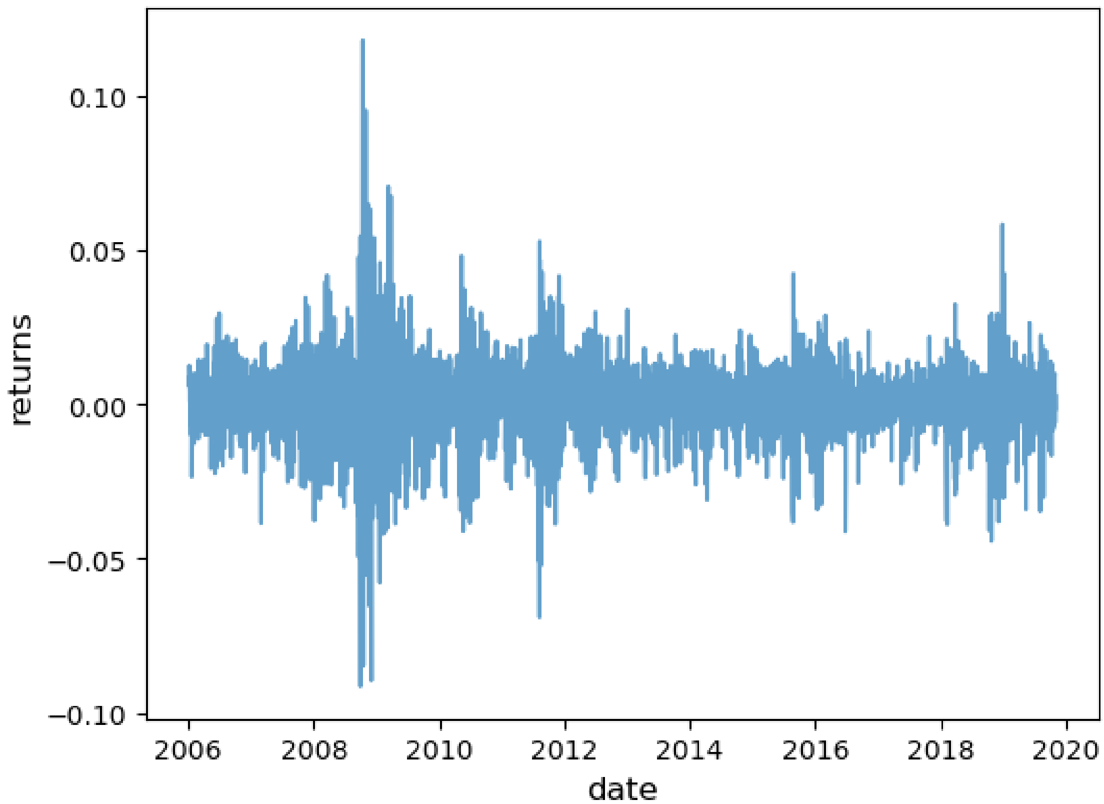
39.2.1 Example: GARCH Volatility

The GARCH model is common in financial applications, where time series such as asset returns exhibit time varying volatility.

For example, consider the following plot of daily returns on the Nasdaq Composite Index for the period 1st January 2006 to 1st November 2019.

```
import yfinance as yf

s = yf.download('^IXIC', '2006-1-1', '2019-11-1', auto_adjust=False)['Adj Close']

r = s.pct_change()

fig, ax = plt.subplots()

ax.plot(r, alpha=0.7)

ax.set_ylabel('returns', fontsize=12)
ax.set_xlabel('date', fontsize=12)

plt.show()
```

```
[*********************100%***********************]  1 of 1 completed
```



Notice how the series exhibits bursts of volatility (high variance) and then settles down again.

GARCH models can replicate this feature.

The GARCH(1, 1) volatility process takes the form

$$\sigma_{t+1}^2 = \alpha_0 + \sigma_t^2(\alpha_1 \xi_{t+1}^2 + \beta) \tag{39.2}$$

where $\{\xi_t\}$ is IID with $\mathbb{E}\xi_t^2 = 1$ and all parameters are positive.

Returns on a given asset are then modeled as

$$r_t = \sigma_t \zeta_t \tag{39.3}$$

where $\{\zeta_t\}$ is again IID and independent of $\{\xi_t\}$.

The volatility sequence $\{\sigma_t^2\}$, which drives the dynamics of returns, is a Kesten process.

39.2.2 Example: Wealth Dynamics

Suppose that a given household saves a fixed fraction s of its current wealth in every period.

The household earns labor income y_t at the start of time t.

Wealth then evolves according to

$$w_{t+1} = R_{t+1} s w_t + y_{t+1} \tag{39.4}$$

where $\{R_t\}$ is the gross rate of return on assets.

If $\{R_t\}$ and $\{y_t\}$ are both IID, then (39.4) is a Kesten process.

39.2.3 Stationarity

In earlier lectures, such as the one on AR(1) processes, we introduced the notion of a stationary distribution.

In the present context, we can define a stationary distribution as follows:

The distribution F^* on $\mathbb{R}$ is called **stationary** for the Kesten process (39.1) if

$$X_t \sim F^* \quad \implies \quad a_{t+1} X_t + \eta_{t+1} \sim F^* \tag{39.5}$$

In other words, if the current state X_t has distribution F^*, then so does the next period state X_{t+1}.

We can write this alternatively as

$$F^*(y) = \int \mathbb{P}\{a_{t+1} x + \eta_{t+1} \leq y\} F^*(dx) \quad \text{for all } y \geq 0. \tag{39.6}$$

The left hand side is the distribution of the next period state when the current state is drawn from F^*.

The equality in (39.6) states that this distribution is unchanged.

39.2.4 Cross-Sectional Interpretation

There is an important cross-sectional interpretation of stationary distributions, discussed previously but worth repeating here.

Suppose, for example, that we are interested in the wealth distribution — that is, the current distribution of wealth across households in a given country.

Suppose further that

- the wealth of each household evolves independently according to (39.4),
- F^* is a stationary distribution for this stochastic process and
- there are many households.

Then F^* is a steady state for the cross-sectional wealth distribution in this country.

In other words, if F^* is the current wealth distribution then it will remain so in subsequent periods, *ceteris paribus*.

To see this, suppose that F^* is the current wealth distribution.

What is the fraction of households with wealth less than y next period?

To obtain this, we sum the probability that wealth is less than y tomorrow, given that current wealth is w, weighted by the fraction of households with wealth w.

Noting that the fraction of households with wealth in interval dw is $F^*(dw)$, we get

$$\int \mathbb{P}\{R_{t+1} s w + y_{t+1} \leq y\} F^*(dw)$$

By the definition of stationarity and the assumption that F^* is stationary for the wealth process, this is just $F^*(y)$.

Hence the fraction of households with wealth in $[0, y]$ is the same next period as it is this period.

Since y was chosen arbitrarily, the distribution is unchanged.

39.2.5 Conditions for Stationarity

The Kesten process $X_{t+1} = a_{t+1} X_t + \eta_{t+1}$ does not always have a stationary distribution.

For example, if $a_t \equiv \eta_t \equiv 1$ for all t, then $X_t = X_0 + t$, which diverges to infinity.

To prevent this kind of divergence, we require that $\{a_t\}$ is strictly less than 1 most of the time.

In particular, if

$$\mathbb{E} \ln a_t < 0 \quad \text{and} \quad \mathbb{E} \eta_t < \infty \tag{39.7}$$

then a unique stationary distribution exists on $\mathbb{R}_+$.

- See, for example, theorem 2.1.3 of [Buraczewski *et al.*, 2016], which provides slightly weaker conditions.

As one application of this result, we see that the wealth process (39.4) will have a unique stationary distribution whenever labor income has finite mean and $\mathbb{E} \ln R_t + \ln s < 0$.

39.3 Heavy Tails

Under certain conditions, the stationary distribution of a Kesten process has a Pareto tail.

(See our earlier lecture on heavy-tailed distributions for background.)

This fact is significant for economics because of the prevalence of Pareto-tailed distributions.

39.3.1 The Kesten–Goldie Theorem

To state the conditions under which the stationary distribution of a Kesten process has a Pareto tail, we first recall that a random variable is called **nonarithmetic** if its distribution is not concentrated on $\{\dots, -2t, -t, 0, t, 2t, \dots\}$ for any $t \geq 0$.

For example, any random variable with a density is nonarithmetic.

The famous Kesten–Goldie Theorem (see, e.g., [Buraczewski *et al.*, 2016], theorem 2.4.4) states that if

1. the stationarity conditions in (39.7) hold,
2. the random variable a_t is positive with probability one and nonarithmetic,
3. $\mathbb{P}\{a_t x + \eta_t = x\} < 1$ for all $x \in \mathbb{R}_+$ and
4. there exists a positive constant α such that

$$\mathbb{E} a_t^\alpha = 1, \quad \mathbb{E} \eta_t^\alpha < \infty, \quad \text{and} \quad \mathbb{E}[a_t^{\alpha+1}] < \infty$$

then the stationary distribution of the Kesten process has a Pareto tail with tail index α.

More precisely, if F^* is the unique stationary distribution and $X^* \sim F^*$, then

$$\lim_{x \to \infty} x^\alpha \mathbb{P}\{X^* > x\} = c$$

for some positive constant c.

39.3.2 Intuition

Later we will illustrate the Kesten–Goldie Theorem using rank-size plots.

Prior to doing so, we can give the following intuition for the conditions.

Two important conditions are that $\mathbb{E} \ln a_t < 0$, so the model is stationary, and $\mathbb{E} a_t^\alpha = 1$ for some $\alpha > 0$.

The first condition implies that the distribution of a_t has a large amount of probability mass below 1.

The second condition implies that the distribution of a_t has at least some probability mass at or above 1.

The first condition gives us existence of the stationary condition.

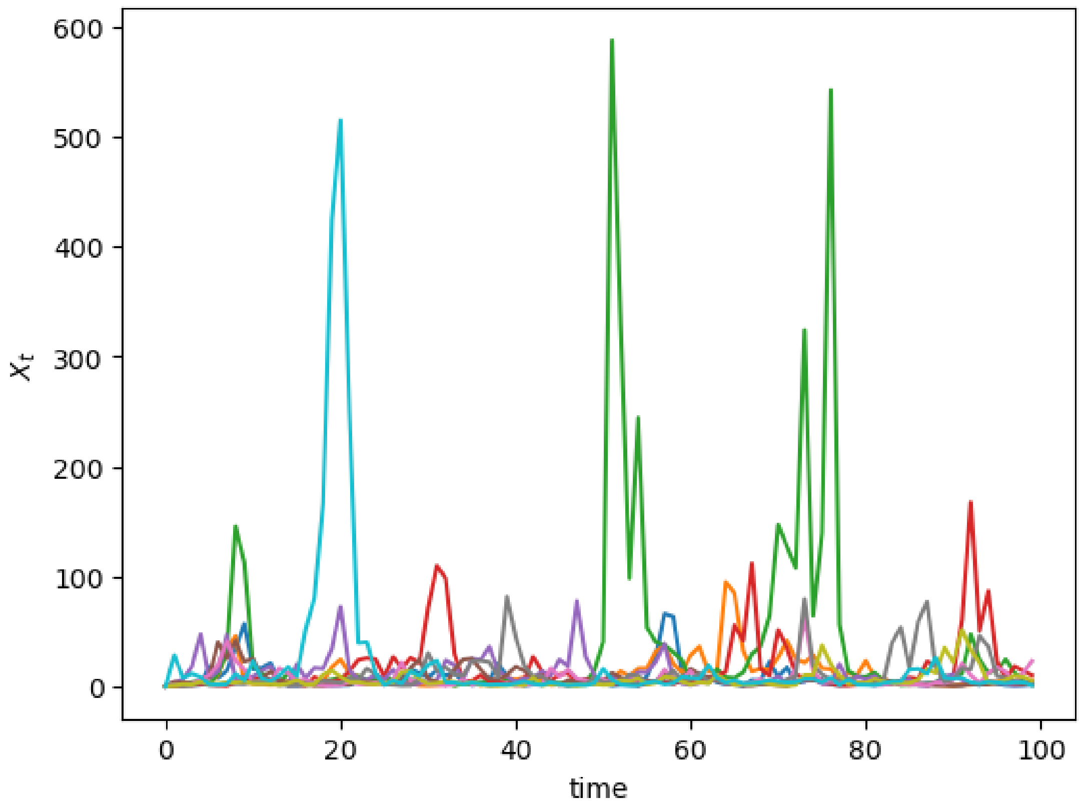
The second condition means that the current state can be expanded by a_t.

If this occurs for several concurrent periods, the effects compound each other, since a_t is multiplicative.

This leads to spikes in the time series, which fill out the extreme right hand tail of the distribution.

The spikes in the time series are visible in the following simulation, which generates of 10 paths when a_t and b_t are lognormal.

```
μ = -0.5
σ = 1.0

def kesten_ts(ts_length=100):
    x = np.zeros(ts_length)
    for t in range(ts_length-1):
        a = np.exp(μ + σ * np.random.randn())
        b = np.exp(np.random.randn())
        x[t+1] = a * x[t] + b
    return x

fig, ax = plt.subplots()

num_paths = 10
np.random.seed(12)

for i in range(num_paths):
    ax.plot(kesten_ts())

ax.set(xlabel='time', ylabel='$X_t$')
plt.show()
```



39.4 Application: Firm Dynamics

As noted in our lecture on heavy tails, for common measures of firm size such as revenue or employment, the US firm size distribution exhibits a Pareto tail (see, e.g., [Axtell, 2001], [Gabaix, 2016]).

Let us try to explain this rather striking fact using the Kesten–Goldie Theorem.

39.4.1 Gibrat's Law

It was postulated many years ago by Robert Gibrat [Gibrat, 1931] that firm size evolves according to a simple rule whereby size next period is proportional to current size.

This is now known as Gibrat's law of proportional growth.

We can express this idea by stating that a suitably defined measure s_t of firm size obeys

$$\frac{s_{t+1}}{s_t} = a_{t+1} \tag{39.8}$$

for some positive IID sequence $\{a_t\}$.

One implication of Gibrat's law is that the growth rate of individual firms does not depend on their size.

However, over the last few decades, research contradicting Gibrat's law has accumulated in the literature.

For example, it is commonly found that, on average,

1. small firms grow faster than large firms (see, e.g., [Evans, 1987] and [Hall, 1987]) and

2. the growth rate of small firms is more volatile than that of large firms [Dunne *et al.*, 1989].

On the other hand, Gibrat's law is generally found to be a reasonable approximation for large firms [Evans, 1987].

We can accommodate these empirical findings by modifying (39.8) to

$$s_{t+1} = a_{t+1} s_t + b_{t+1} \tag{39.9}$$

where $\{a_t\}$ and $\{b_t\}$ are both IID and independent of each other.

In the exercises you are asked to show that (39.9) is more consistent with the empirical findings presented above than Gibrat's law in (39.8).

39.4.2 Heavy Tails

So what has this to do with Pareto tails?

The answer is that (39.9) is a Kesten process.

If the conditions of the Kesten–Goldie Theorem are satisfied, then the firm size distribution is predicted to have heavy tails — which is exactly what we see in the data.

In the exercises below we explore this idea further, generalizing the firm size dynamics and examining the corresponding rank-size plots.

We also try to illustrate why the Pareto tail finding is significant for quantitative analysis.

39.5 Exercises

Exercise 39.5.1

Simulate and plot 15 years of daily returns (consider each year as having 250 working days) using the GARCH(1, 1) process in (39.2)–(39.3).

Take ξ_t and ζ_t to be independent and standard normal.

Set $\alpha_0 = 0.00001, \alpha_1 = 0.1, \beta = 0.9$ and $\sigma_0 = 0$.

Compare visually with the Nasdaq Composite Index returns *shown above*.

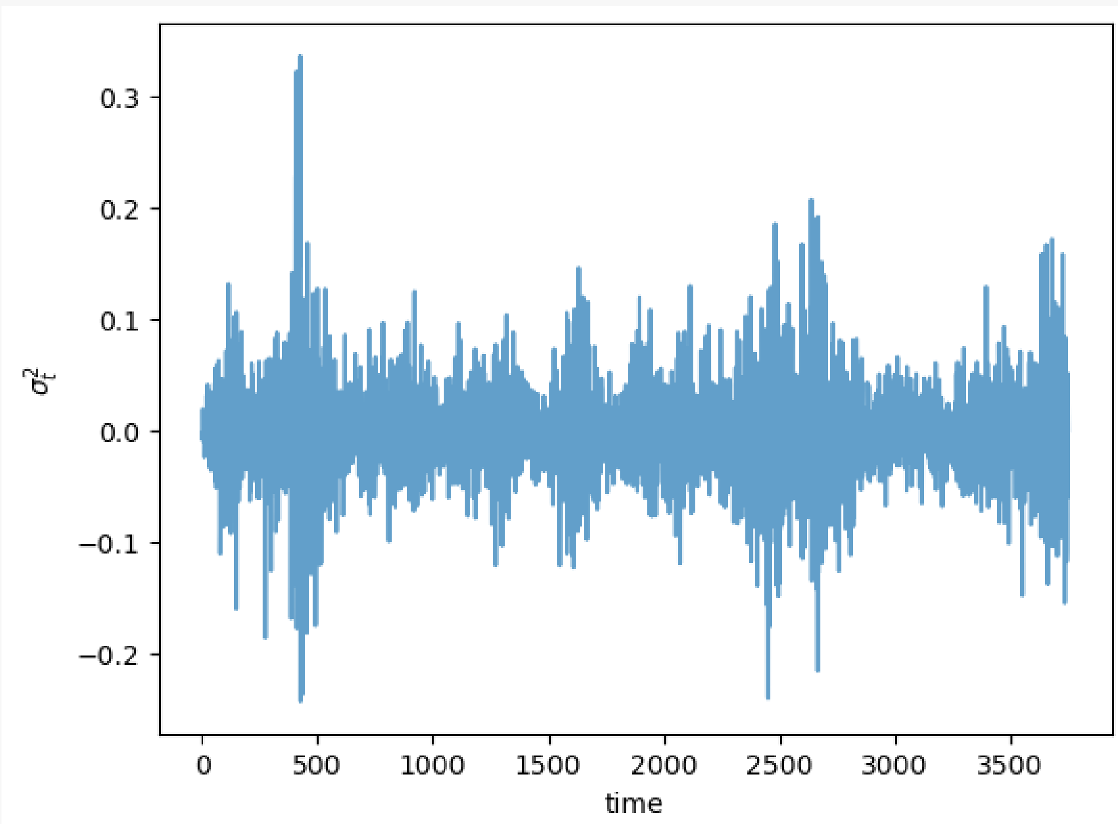
While the time path differs, you should see bursts of high volatility.

Solution

Here is one solution:

```python
α_0 = 1e-5
α_1 = 0.1
β = 0.9

years = 15
days = years * 250

def garch_ts(ts_length=days):
    σ2 = 0
    r = np.zeros(ts_length)
```

```
    for t in range(ts_length-1):
        ξ = np.random.randn()
        σ2 = α_0 + σ2 * (α_1 * ξ**2 + β)
        r[t] = np.sqrt(σ2) * np.random.randn()
    return r

fig, ax = plt.subplots()

np.random.seed(12)

ax.plot(garch_ts(), alpha=0.7)

ax.set(xlabel='time', ylabel='$\\sigma_t^2$')
plt.show()
```



Exercise 39.5.2

In our discussion of firm dynamics, it was claimed that (39.9) is more consistent with the empirical literature than Gibrat's law in (39.8).

(The empirical literature was reviewed immediately above (39.9).)

In what sense is this true (or false)?

Solution

The empirical findings are that

1. small firms grow faster than large firms and
2. the growth rate of small firms is more volatile than that of large firms.

Also, Gibrat's law is generally found to be a reasonable approximation for large firms than for small firms

The claim is that the dynamics in (39.9) are more consistent with points 1-2 than Gibrat's law.

To see why, we rewrite (39.9) in terms of growth dynamics:

$$\frac{s_{t+1}}{s_t} = a_{t+1} + \frac{b_{t+1}}{s_t} \tag{39.10}$$

Taking $s_t = s$ as given, the mean and variance of firm growth are

$$\mathbb{E}a + \frac{\mathbb{E}b}{s} \quad \text{and} \quad \mathbb{V}a + \frac{\mathbb{V}b}{s^2}$$

Both of these decline with firm size s, consistent with the data.

Moreover, the law of motion (39.10) clearly approaches Gibrat's law (39.8) as s_t gets large.

Exercise 39.5.3

Consider an arbitrary Kesten process as given in (39.1).

Suppose that $\{a_t\}$ is lognormal with parameters (μ, σ).

In other words, each a_t has the same distribution as $\exp(\mu + \sigma Z)$ when Z is standard normal.

Suppose further that $\mathbb{E}\eta_t^r < \infty$ for every $r > 0$, as would be the case if, say, η_t is also lognormal.

Show that the conditions of the Kesten–Goldie theorem are satisfied if and only if $\mu < 0$.

Obtain the value of α that makes the Kesten–Goldie conditions hold.

Solution

Since a_t has a density it is nonarithmetic.

Since a_t has the same density as $a = \exp(\mu + \sigma Z)$ when Z is standard normal, we have

$$\mathbb{E} \ln a_t = \mathbb{E}(\mu + \sigma Z) = \mu,$$

and since η_t has finite moments of all orders, the stationarity condition holds if and only if $\mu < 0$.

Given the properties of the lognormal distribution (which has finite moments of all orders), the only other condition in doubt is existence of a positive constant α such that $\mathbb{E} a_t^\alpha = 1$.

This is equivalent to the statement

$$\exp\left(\alpha \mu + \frac{\alpha^2 \sigma^2}{2}\right) = 1.$$

Solving for α gives $\alpha = -2\mu/\sigma^2$.

Exercise 39.5.4

One unrealistic aspect of the firm dynamics specified in (39.9) is that it ignores entry and exit.

In any given period and in any given market, we observe significant numbers of firms entering and exiting the market.

Empirical discussion of this can be found in a famous paper by Hugo Hopenhayn [Hopenhayn, 1992].

In the same paper, Hopenhayn builds a model of entry and exit that incorporates profit maximization by firms and market clearing quantities, wages and prices.

In his model, a stationary equilibrium occurs when the number of entrants equals the number of exiting firms.

In this setting, firm dynamics can be expressed as

$$s_{t+1} = e_{t+1} \mathbb{1}\{s_t < \bar{s}\} + (a_{t+1} s_t + b_{t+1}) \mathbb{1}\{s_t \geq \bar{s}\} \tag{39.11}$$

Here

- the state variable s_t represents productivity (which is a proxy for output and hence firm size),
- the IID sequence $\{e_t\}$ is thought of as a productivity draw for a new entrant and
- the variable $\bar{s}$ is a threshold value that we take as given, although it is determined endogenously in Hopenhayn's model.

The idea behind (39.11) is that firms stay in the market as long as their productivity s_t remains at or above $\bar{s}$.

- In this case, their productivity updates according to (39.9).

Firms choose to exit when their productivity s_t falls below $\bar{s}$.

- In this case, they are replaced by a new firm with productivity e_{t+1}.

What can we say about dynamics?

Although (39.11) is not a Kesten process, it does update in the same way as a Kesten process when s_t is large.

So perhaps its stationary distribution still has Pareto tails?

Your task is to investigate this question via simulation and rank-size plots.

The approach will be to

1. generate M draws of s_T when M and T are large and
2. plot the largest 1,000 of the resulting draws in a rank-size plot.

(The distribution of s_T will be close to the stationary distribution when T is large.)

In the simulation, assume that

- each of a_t, b_t and e_t is lognormal,
- the parameters are

```
μ_a = -0.5        # location parameter for a
σ_a = 0.1         # scale parameter for a
μ_b = 0.0         # location parameter for b
σ_b = 0.5         # scale parameter for b
μ_e = 0.0         # location parameter for e
σ_e = 0.5         # scale parameter for e
s_bar = 1.0       # threshold
T = 500           # sampling date
M = 1_000_000     # number of firms
s_init = 1.0      # initial condition for each firm
```

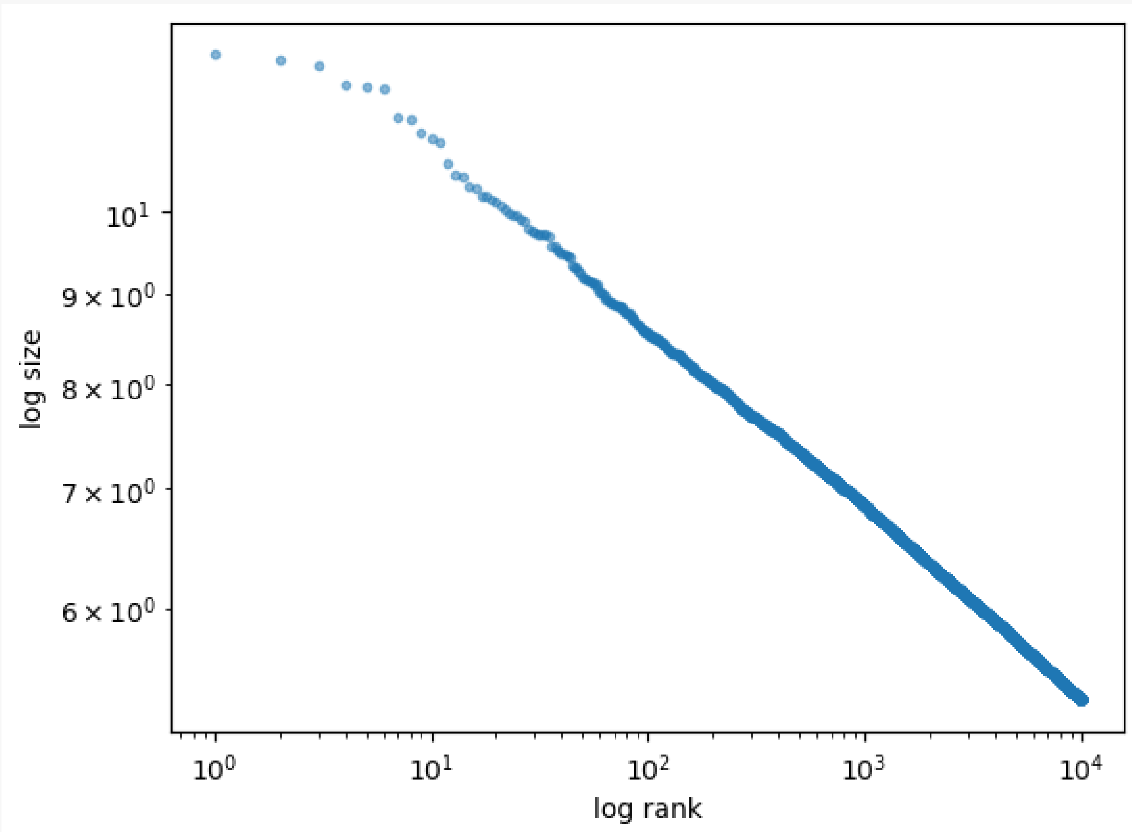
Solution

Here's one solution. First we generate the observations:

```
from numba import jit, prange
from numpy.random import randn


@jit(parallel=True)
def generate_draws(μ_a=-0.5,
                   σ_a=0.1,
                   μ_b=0.0,
                   σ_b=0.5,
                   μ_e=0.0,
                   σ_e=0.5,
                   s_bar=1.0,
                   T=500,
                   M=1_000_000,
                   s_init=1.0):

    draws = np.empty(M)
    for m in prange(M):
        s = s_init
        for t in range(T):
            if s < s_bar:
                new_s = np.exp(μ_e + σ_e *  randn())
            else:
                a = np.exp(μ_a + σ_a * randn())
                b = np.exp(μ_b + σ_b * randn())
                new_s = a * s + b
            s = new_s
        draws[m] = s

    return draws

data = generate_draws()
```

Now we produce the rank-size plot:

```
fig, ax = plt.subplots()

rank_data, size_data = qe.rank_size(data, c=0.01)
ax.loglog(rank_data, size_data, 'o', markersize=3.0, alpha=0.5)
ax.set_xlabel("log rank")
ax.set_ylabel("log size")

plt.show()
```



The plot produces a straight line, consistent with a Pareto tail.

CHAPTER

FORTY

WEALTH DISTRIBUTION DYNAMICS

Contents

See also

A version of this lecture using JAX is available here

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
```

40.1 Overview

This notebook gives an introduction to wealth distribution dynamics, with a focus on

- modeling and computing the wealth distribution via simulation,
- measures of inequality such as the Lorenz curve and Gini coefficient, and
- how inequality is affected by the properties of wage income and returns on assets.

One interesting property of the wealth distribution we discuss is Pareto tails.

The wealth distribution in many countries exhibits a Pareto tail

- See this lecture for a definition.
- For a review of the empirical evidence, see, for example, [Benhabib and Bisin, 2018].

This is consistent with high concentration of wealth amongst the richest households.

It also gives us a way to quantify such concentration, in terms of the tail index.

One question of interest is whether or not we can replicate Pareto tails from a relatively simple model.

40.1.1 A Note on Assumptions

The evolution of wealth for any given household depends on their savings behavior.

Modeling such behavior will form an important part of this lecture series.

However, in this particular lecture, we will be content with rather ad hoc (but plausible) savings rules.

We do this to more easily explore the implications of different specifications of income dynamics and investment returns.

At the same time, all of the techniques discussed here can be plugged into models that use optimization to obtain savings rules.

We will use the following imports.

```
import matplotlib.pyplot as plt
import numpy as np
import quantecon as qe
from numba import jit, float64, prange
from numba.experimental import jitclass
```

40.2 Lorenz Curves and the Gini Coefficient

Before we investigate wealth dynamics, we briefly review some measures of inequality.

40.2.1 Lorenz Curves

One popular graphical measure of inequality is the Lorenz curve.

The package QuantEcon.py, already imported above, contains a function to compute Lorenz curves.

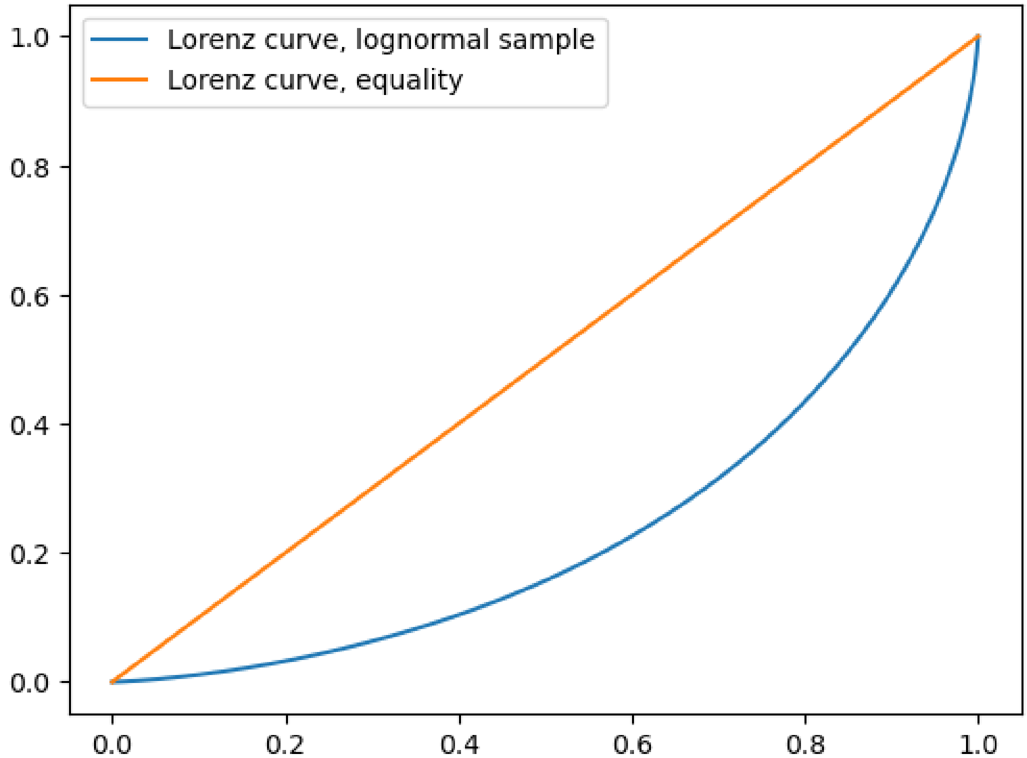
To illustrate, suppose that

```
n = 10_000                      # size of sample
w = np.exp(np.random.randn(n))  # lognormal draws
```

is data representing the wealth of 10,000 households.

We can compute and plot the Lorenz curve as follows:

```
f_vals, l_vals = qe.lorenz_curve(w)

fig, ax = plt.subplots()
ax.plot(f_vals, l_vals, label='Lorenz curve, lognormal sample')
ax.plot(f_vals, f_vals, label='Lorenz curve, equality')
ax.legend()
plt.show()
```



This curve can be understood as follows: if point (x, y) lies on the curve, it means that, collectively, the bottom $(100x)\%$ of the population holds $(100y)\%$ of the wealth.

The "equality" line is the 45 degree line (which might not be exactly 45 degrees in the figure, depending on the aspect ratio).

A sample that produces this line exhibits perfect equality.

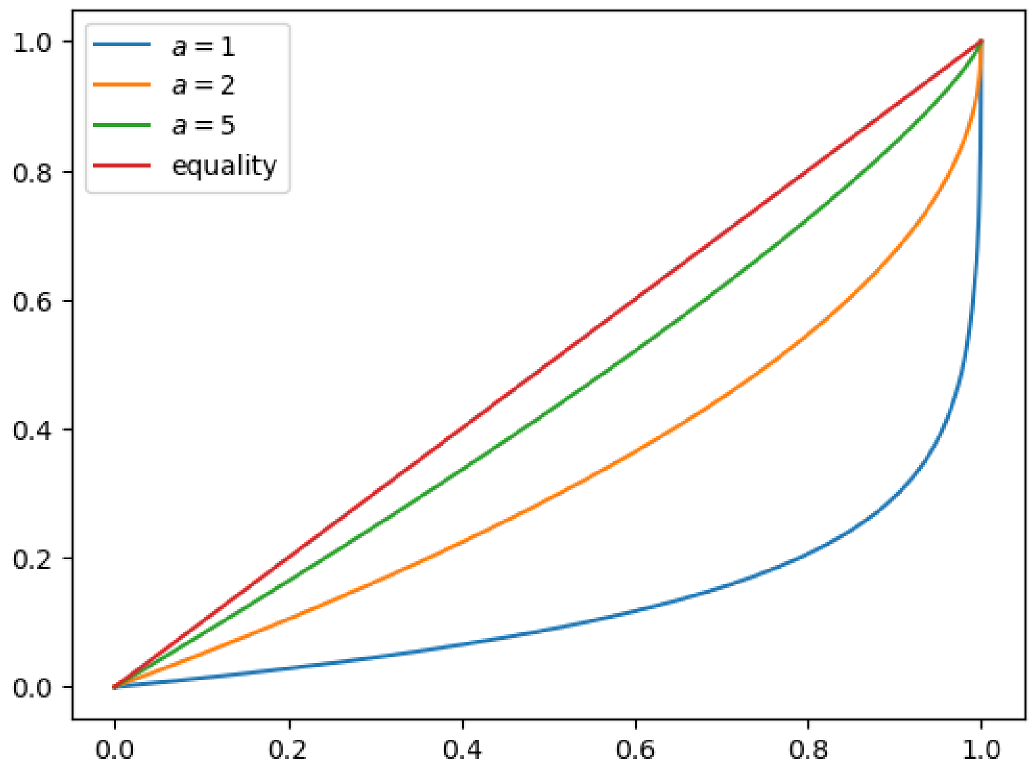
The other line in the figure is the Lorenz curve for the lognormal sample, which deviates significantly from perfect equality.

For example, the bottom 80% of the population holds around 40% of total wealth.

Here is another example, which shows how the Lorenz curve shifts as the underlying distribution changes.

We generate 10,000 observations using the Pareto distribution with a range of parameters, and then compute the Lorenz curve corresponding to each set of observations.

```
a_vals = (1, 2, 5)              # Pareto tail index
n = 10_000                      # size of each sample
fig, ax = plt.subplots()
for a in a_vals:
    u = np.random.uniform(size=n)
    y = u**(-1/a)               # distributed as Pareto with tail index a
    f_vals, l_vals = qe.lorenz_curve(y)
    ax.plot(f_vals, l_vals, label=f'$a = {a}$')
ax.plot(f_vals, f_vals, label='equality')
ax.legend()
plt.show()
```



You can see that, as the tail parameter of the Pareto distribution increases, inequality decreases.

This is to be expected, because a higher tail index implies less weight in the tail of the Pareto distribution.

40.2.2 The Gini Coefficient

The definition and interpretation of the Gini coefficient can be found on the corresponding Wikipedia page.

A value of 0 indicates perfect equality (corresponding the case where the Lorenz curve matches the 45 degree line) and a value of 1 indicates complete inequality (all wealth held by the richest household).

The QuantEcon.py library contains a function to calculate the Gini coefficient.

We can test it on the Weibull distribution with parameter a, where the Gini coefficient is known to be

$$G = 1 - 2^{-1/a}$$

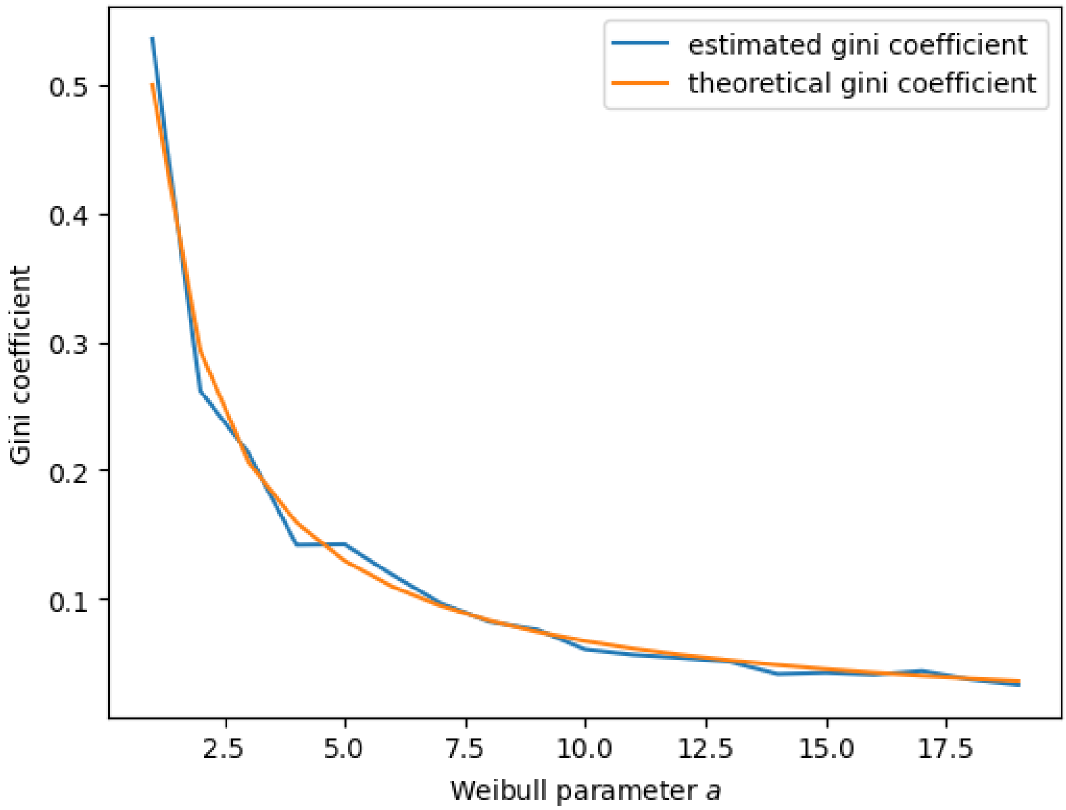
Let's see if the Gini coefficient computed from a simulated sample matches this at each fixed value of a.

```
a_vals = range(1, 20)
ginis = []
ginis_theoretical = []
n = 100

fig, ax = plt.subplots()
for a in a_vals:
    y = np.random.weibull(a, size=n)
    ginis.append(qe.gini_coefficient(y))
    ginis_theoretical.append(1 - 2**(-1/a))
ax.plot(a_vals, ginis, label='estimated gini coefficient')
```

(continues on next page)

(continued from previous page)

```
ax.plot(a_vals, ginis_theoretical, label='theoretical gini coefficient')
ax.legend()
ax.set_xlabel("Weibull parameter $a$")
ax.set_ylabel("Gini coefficient")
plt.show()
```



The simulation shows that the fit is good.

40.3 A Model of Wealth Dynamics

Having discussed inequality measures, let us now turn to wealth dynamics.

The model we will study is

$$w_{t+1} = (1 + r_{t+1})s(w_t) + y_{t+1} \tag{40.1}$$

where

- w_t is wealth at time t for a given household,
- r_t is the rate of return of financial assets,
- y_t is current non-financial (e.g., labor) income and
- $s(w_t)$ is current wealth net of consumption

Letting $\{z_t\}$ be a correlated state process of the form

$$z_{t+1} = a z_t + b + \sigma_z \epsilon_{t+1}$$

we'll assume that

$$R_t := 1 + r_t = c_r \exp(z_t) + \exp(\mu_r + \sigma_r \xi_t)$$

and

$$y_t = c_y \exp(z_t) + \exp(\mu_y + \sigma_y \zeta_t)$$

Here $\{(\epsilon_t, \xi_t, \zeta_t)\}$ is IID and standard normal in $\mathbb{R}^3$.

The value of c_r should be close to zero, since rates of return on assets do not exhibit large trends.

When we simulate a population of households, we will assume all shocks are idiosyncratic (i.e., specific to individual households and independent across them).

Regarding the savings function s, our default model will be

$$s(w) = s_0 w \cdot \mathbb{1}\{w \geq \hat{w}\} \tag{40.2}$$

where s_0 is a positive constant.

Thus, for $w < \hat{w}$, the household saves nothing. For $w \geq \bar{w}$, the household saves a fraction s_0 of their wealth.

We are using something akin to a fixed savings rate model, while acknowledging that low wealth households tend to save very little.

40.4 Implementation

Here's some type information to help Numba.

```
wealth_dynamics_data = [
    ('w_hat',  float64),    # savings parameter
    ('s_0',    float64),    # savings parameter
    ('c_y',    float64),    # labor income parameter
    ('μ_y',    float64),    # labor income paraemter
    ('σ_y',    float64),    # labor income parameter
    ('c_r',    float64),    # rate of return parameter
    ('μ_r',    float64),    # rate of return parameter
    ('σ_r',    float64),    # rate of return parameter
    ('a',      float64),    # aggregate shock parameter
    ('b',      float64),    # aggregate shock parameter
    ('σ_z',    float64),    # aggregate shock parameter
    ('z_mean', float64),    # mean of z process
    ('z_var', float64),     # variance of z process
    ('y_mean', float64),    # mean of y process
    ('R_mean', float64)     # mean of R process
]
```

Here's a class that stores instance data and implements methods that update the aggregate state and household wealth.

```
@jitclass(wealth_dynamics_data)
class WealthDynamics:

    def __init__(self,
                 w_hat=1.0,
                 s_0=0.75,
                 c_y=1.0,
```

(continues on next page)

(continued from previous page)

```
                 μ_y=1.0,
                 σ_y=0.2,
                 c_r=0.05,
                 μ_r=0.1,
                 σ_r=0.5,
                 a=0.5,
                 b=0.0,
                 σ_z=0.1):

        self.w_hat, self.s_0 = w_hat, s_0
        self.c_y, self.μ_y, self.σ_y = c_y, μ_y, σ_y
        self.c_r, self.μ_r, self.σ_r = c_r, μ_r, σ_r
        self.a, self.b, self.σ_z = a, b, σ_z

        # Record stationary moments
        self.z_mean = b / (1 - a)
        self.z_var = σ_z**2 / (1 - a**2)
        exp_z_mean = np.exp(self.z_mean + self.z_var / 2)
        self.R_mean = c_r * exp_z_mean + np.exp(μ_r + σ_r**2 / 2)
        self.y_mean = c_y * exp_z_mean + np.exp(μ_y + σ_y**2 / 2)

        # Test a stability condition that ensures wealth does not diverge
        # to infinity.
        α = self.R_mean * self.s_0
        if α >= 1:
            raise ValueError("Stability condition failed.")

    def parameters(self):
        """
        Collect and return parameters.
        """
        parameters = (self.w_hat, self.s_0,
                      self.c_y, self.μ_y, self.σ_y,
                      self.c_r, self.μ_r, self.σ_r,
                      self.a, self.b, self.σ_z)
        return parameters

    def update_states(self, w, z):
        """
        Update one period, given current wealth w and persistent
        state z.
        """

        # Simplify names
        params = self.parameters()
        w_hat, s_0, c_y, μ_y, σ_y, c_r, μ_r, σ_r, a, b, σ_z = params
        zp = a * z + b + σ_z * np.random.randn()

        # Update wealth
        y = c_y * np.exp(zp) + np.exp(μ_y + σ_y * np.random.randn())
        wp = y
        if w >= w_hat:
            R = c_r * np.exp(zp) + np.exp(μ_r + σ_r * np.random.randn())
            wp += R * s_0 * w
        return wp, zp
```

Here's function to simulate the time series of wealth for in individual households.

```
@jit
def wealth_time_series(wdy, w_0, n):
    """
    Generate a single time series of length n for wealth given
    initial value w_0.

    The initial persistent state z_0 for each household is drawn from
    the stationary distribution of the AR(1) process.

        * wdy: an instance of WealthDynamics
        * w_0: scalar
        * n: int


    """
    z = wdy.z_mean + np.sqrt(wdy.z_var) * np.random.randn()
    w = np.empty(n)
    w[0] = w_0
    for t in range(n-1):
        w[t+1], z = wdy.update_states(w[t], z)
    return w
```

Now here's function to simulate a cross section of households forward in time.

Note the use of parallelization to speed up computation.

```
@jit(parallel=True)
def update_cross_section(wdy, w_distribution, shift_length=500):
    """
    Shifts a cross-section of household forward in time

    * wdy: an instance of WealthDynamics
    * w_distribution: array_like, represents current cross-section

    Takes a current distribution of wealth values as w_distribution
    and updates each w_t in w_distribution to w_{t+j}, where
    j = shift_length.

    Returns the new distribution.

    """
    new_distribution = np.empty_like(w_distribution)

    # Update each household
    for i in prange(len(new_distribution)):
        z = wdy.z_mean + np.sqrt(wdy.z_var) * np.random.randn()
        w = w_distribution[i]
        for t in range(shift_length-1):
            w, z = wdy.update_states(w, z)
        new_distribution[i] = w
    return new_distribution
```

Parallelization is very effective in the function above because the time path of each household can be calculated independently once the path for the aggregate state is known.

40.5 Applications

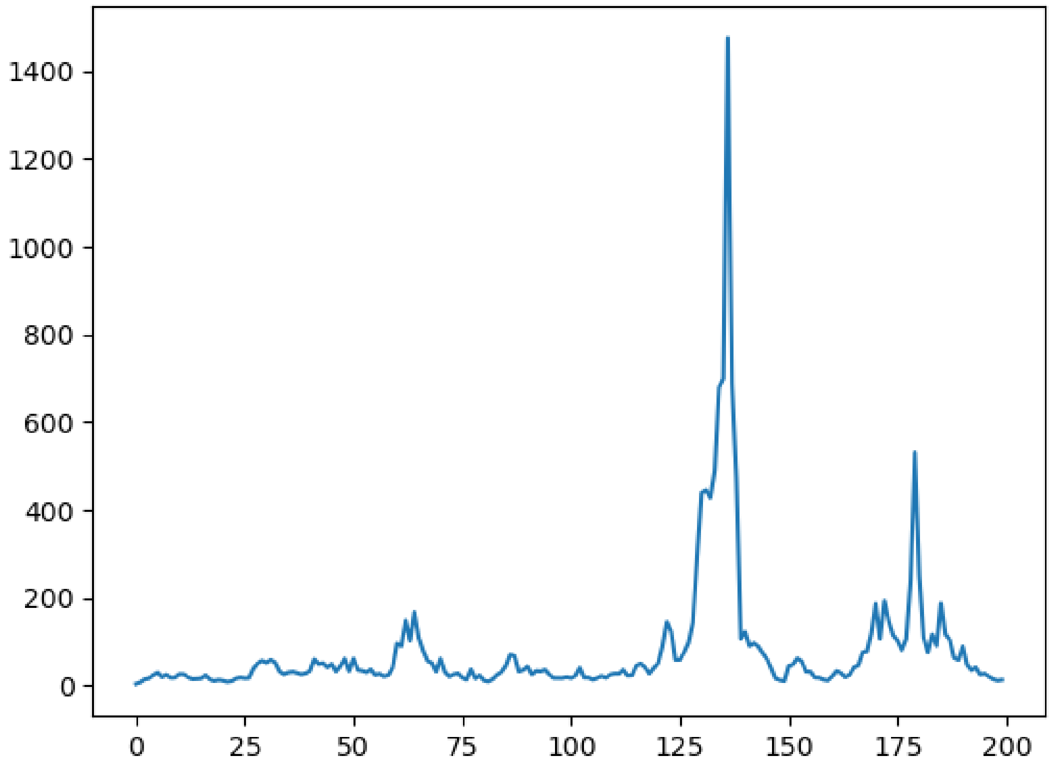
Let's try simulating the model at different parameter values and investigate the implications for the wealth distribution.

40.5.1 Time Series

Let's look at the wealth dynamics of an individual household.

```
wdy = WealthDynamics()
ts_length = 200
w = wealth_time_series(wdy, wdy.y_mean, ts_length)
```

```
fig, ax = plt.subplots()
ax.plot(w)
plt.show()
```



Notice the large spikes in wealth over time.

Such spikes are similar to what we observed in time series when *we studied Kesten processes*.

40.5.2 Inequality Measures

Let's look at how inequality varies with returns on financial assets.

The next function generates a cross section and then computes the Lorenz curve and Gini coefficient.

```
def generate_lorenz_and_gini(wdy, num_households=100_000, T=500):
    """
    Generate the Lorenz curve data and gini coefficient corresponding to a
    WealthDynamics mode by simulating num_households forward to time T.
    """
    ψ_0 = np.full(num_households, wdy.y_mean)
    z_0 = wdy.z_mean

    ψ_star = update_cross_section(wdy, ψ_0, shift_length=T)
    return qe.gini_coefficient(ψ_star), qe.lorenz_curve(ψ_star)
```

Now we investigate how the Lorenz curves associated with the wealth distribution change as return to savings varies.

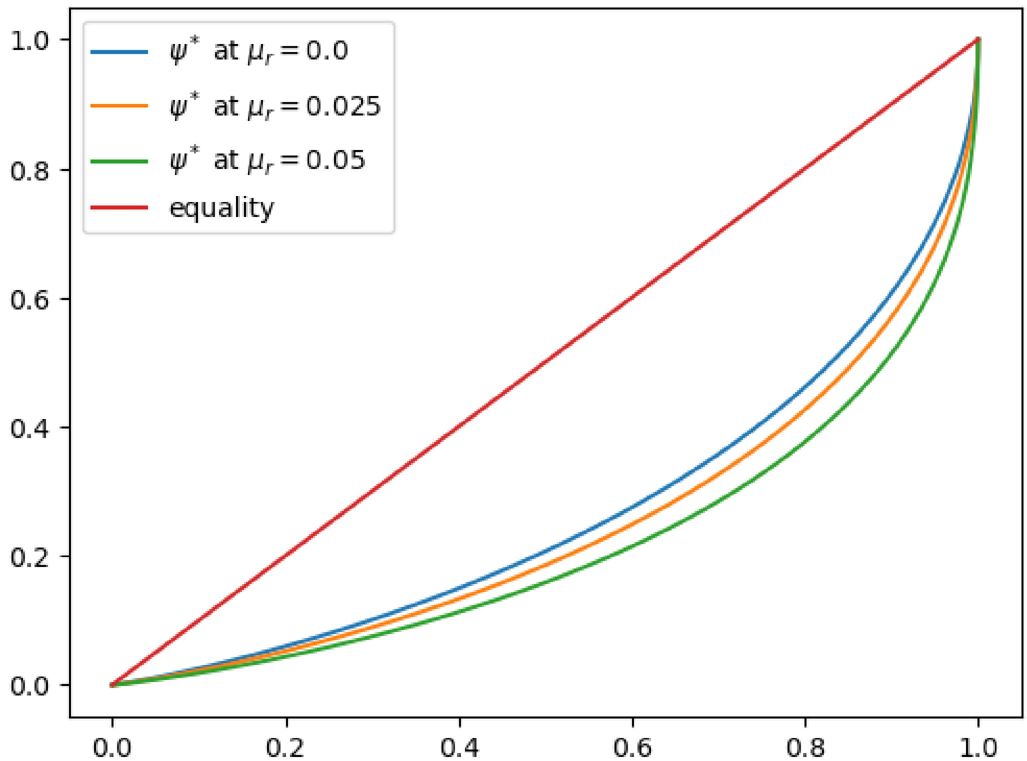
The code below plots Lorenz curves for three different values of μ_r.

If you are running this yourself, note that it will take one or two minutes to execute.

This is unavoidable because we are executing a CPU intensive task.

In fact the code, which is JIT compiled and parallelized, runs extremely fast relative to the number of computations.

```
%%time

fig, ax = plt.subplots()
μ_r_vals = (0.0, 0.025, 0.05)
gini_vals = []

for μ_r in μ_r_vals:
    wdy = WealthDynamics(μ_r=μ_r)
    gv, (f_vals, l_vals) = generate_lorenz_and_gini(wdy)
    ax.plot(f_vals, l_vals, label=fr'$\psi^*$ at $\mu_r = {μ_r:0.2}$')
    gini_vals.append(gv)

ax.plot(f_vals, f_vals, label='equality')
ax.legend(loc="upper left")
plt.show()
```




```
CPU times: user 1min 14s, sys: 62.6 ms, total: 1min 14s
Wall time: 10 s
```

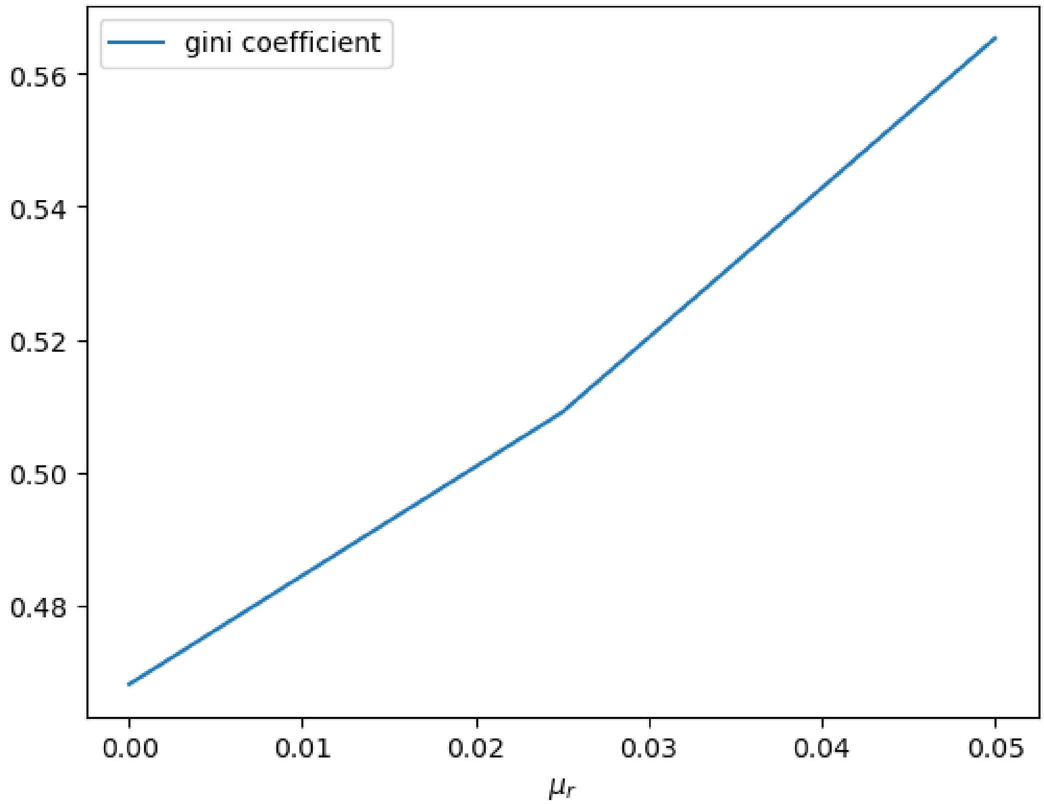
The Lorenz curve shifts downwards as returns on financial income rise, indicating a rise in inequality.

We will look at this again via the Gini coefficient immediately below, but first consider the following image of our system resources when the code above is executing:

Since the code is both efficiently JIT compiled and fully parallelized, it's close to impossible to make this sequence of tasks run faster without changing hardware.

Now let's check the Gini coefficient.

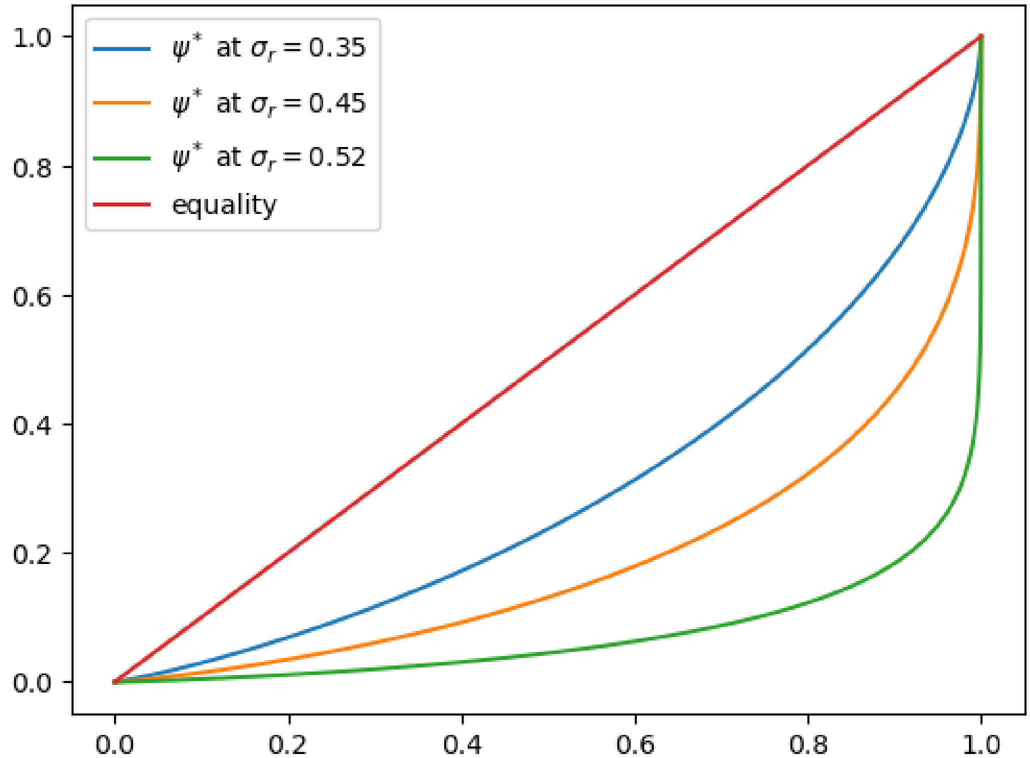
```
fig, ax = plt.subplots()
ax.plot(μ_r_vals, gini_vals, label='gini coefficient')
ax.set_xlabel(r"$\mu_r$")
ax.legend()
plt.show()
```



Once again, we see that inequality increases as returns on financial income rise.

Let's finish this section by investigating what happens when we change the volatility term σ_r in financial returns.

```
%%time

fig, ax = plt.subplots()
σ_r_vals = (0.35, 0.45, 0.52)
gini_vals = []

for σ_r in σ_r_vals:
    wdy = WealthDynamics(σ_r=σ_r)
    gv, (f_vals, l_vals) = generate_lorenz_and_gini(wdy)
    ax.plot(f_vals, l_vals, label=fr'$\psi^*$ at $\sigma_r = {σ_r:0.2}$')
    gini_vals.append(gv)

ax.plot(f_vals, f_vals, label='equality')
ax.legend(loc="upper left")
plt.show()
```




```
CPU times: user 1min 15s, sys: 50 ms, total: 1min 15s
Wall time: 9.66 s
```

We see that greater volatility has the effect of increasing inequality in this model.

40.6 Exercises

Exercise 40.6.1

For a wealth or income distribution with Pareto tail, a higher tail index suggests lower inequality.

Indeed, it is possible to prove that the Gini coefficient of the Pareto distribution with tail index a is $1/(2a - 1)$.

To the extent that you can, confirm this by simulation.

In particular, generate a plot of the Gini coefficient against the tail index using both the theoretical value just given and the value computed from a sample via `qe.gini_coefficient`.

For the values of the tail index, use `a_vals = np.linspace(1, 10, 25)`.

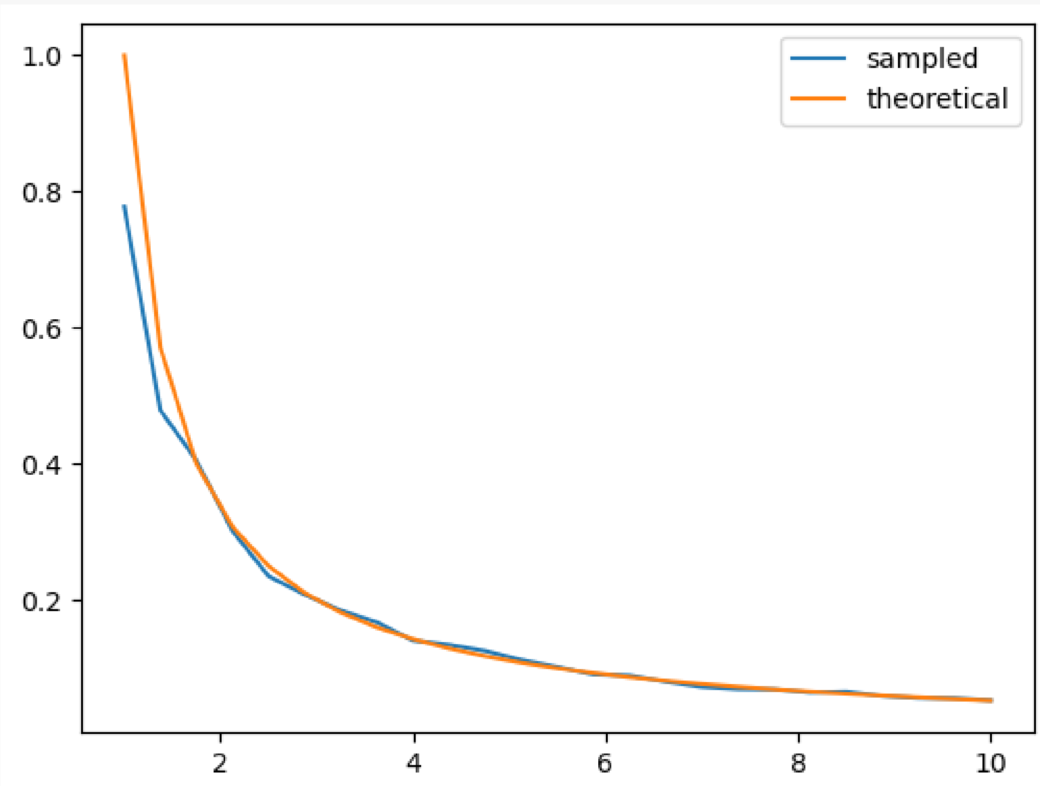
Use sample of size 1,000 for each a and the sampling method for generating Pareto draws employed in the discussion of Lorenz curves for the Pareto distribution.

To the extent that you can, interpret the monotone relationship between the Gini index and a.

Solution

Here is one solution, which produces a good match between theory and simulation.

```
a_vals = np.linspace(1, 10, 25)  # Pareto tail index
ginis = np.empty_like(a_vals)

n = 1000                         # size of each sample
fig, ax = plt.subplots()
for i, a in enumerate(a_vals):
    y = np.random.uniform(size=n)**(-1/a)
    ginis[i] = qe.gini_coefficient(y)
ax.plot(a_vals, ginis, label='sampled')
ax.plot(a_vals, 1/(2*a_vals - 1), label='theoretical')
ax.legend()
plt.show()
```



In general, for a Pareto distribution, a higher tail index implies less weight in the right hand tail.

This means less extreme values for wealth and hence more equality.

More equality translates to a lower Gini index.

Exercise 40.6.2

The wealth process (40.1) is similar to a *Kesten process*.

This is because, according to (40.2), savings is constant for all wealth levels above $\hat{w}$.

When savings is constant, the wealth process has the same quasi-linear structure as a Kesten process, with multiplicative and additive shocks.

The Kesten–Goldie theorem tells us that Kesten processes have Pareto tails under a range of parameterizations.

The theorem does not directly apply here, since savings is not always constant and since the multiplicative and additive terms in (40.1) are not IID.

At the same time, given the similarities, perhaps Pareto tails will arise.

To test this, run a simulation that generates a cross-section of wealth and generate a rank-size plot.

If you like, you can use the function `rank_size` from the `quantecon` library (documentation here).

In viewing the plot, remember that Pareto tails generate a straight line. Is this what you see?

For sample size and initial conditions, use

```
num_households = 250_000
T = 500                                    # shift forward T periods
ψ_0 = np.full(num_households, wdy.y_mean)    # initial distribution
z_0 = wdy.z_mean
```

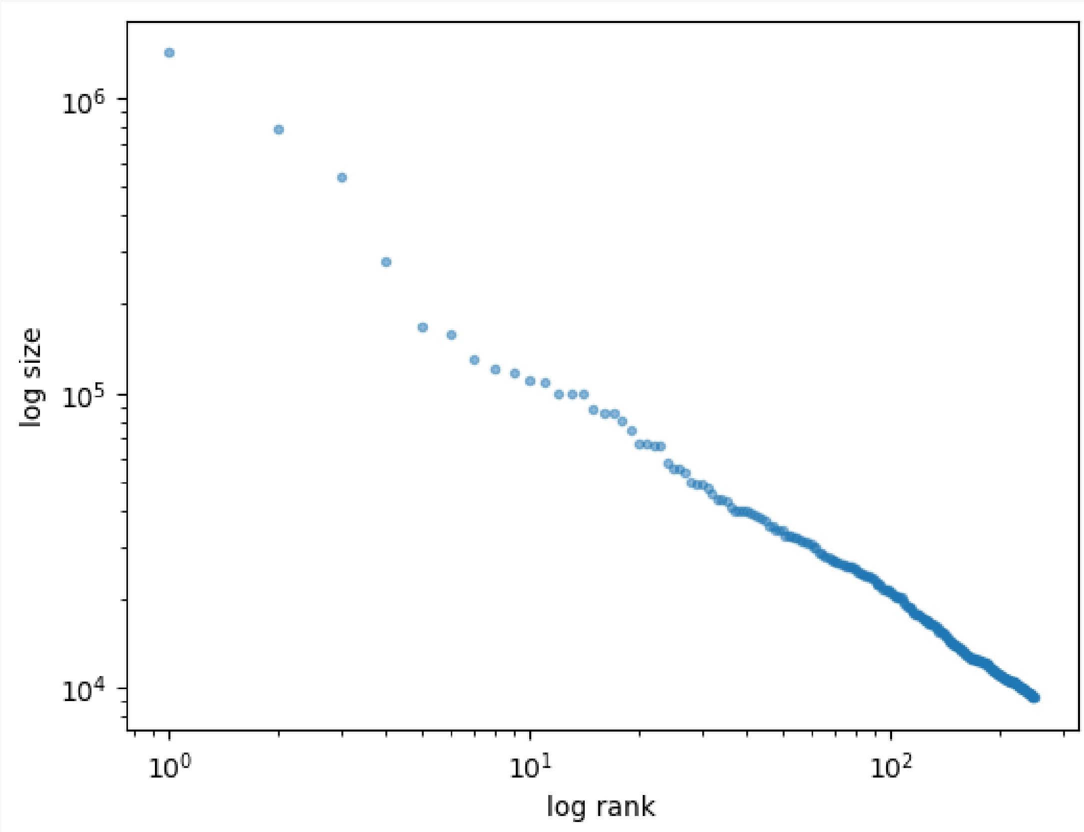
Solution

First let's generate the distribution:

```
num_households = 250_000
T = 500  # how far to shift forward in time
wdy = WealthDynamics()
ψ_0 = np.full(num_households, wdy.y_mean)
z_0 = wdy.z_mean

ψ_star = update_cross_section(wdy, ψ_0, shift_length=T)
```

Now let's see the rank-size plot:

```
fig, ax = plt.subplots()

rank_data, size_data = qe.rank_size(ψ_star, c=0.001)
ax.loglog(rank_data, size_data, 'o', markersize=3.0, alpha=0.5)
ax.set_xlabel("log rank")
ax.set_ylabel("log size")

plt.show()
```


log size
10^6
10^5
10^4
10^0
10^1
10^2
log rank

CHAPTER

FORTYONE

A FIRST LOOK AT THE KALMAN FILTER

Contents

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
```

41.1 Overview

This lecture provides a simple and intuitive introduction to the Kalman filter, for those who either

- have heard of the Kalman filter but don't know how it works, or
- know the Kalman filter equations, but don't know where they come from

For additional (more advanced) reading on the Kalman filter, see

- [Ljungqvist and Sargent, 2018], section 2.7
- [Anderson and Moore, 2005]

The second reference presents a comprehensive treatment of the Kalman filter.

Required knowledge: Familiarity with matrix manipulations, multivariate normal distributions, covariance matrices, etc.

We'll need the following imports:

```
import matplotlib.pyplot as plt
from scipy import linalg
import numpy as np
import matplotlib.cm as cm
from quantecon import Kalman, LinearStateSpace
```

(continues on next page)

(continued from previous page)

```
from scipy.stats import norm
from scipy.integrate import quad
from scipy.linalg import eigvals
```

41.2 The Basic Idea

The Kalman filter has many applications in economics, but for now let's pretend that we are rocket scientists.

A missile has been launched from country Y and our mission is to track it.

Let $x \in \mathbb{R}^2$ denote the current location of the missile—a pair indicating latitude-longitude coordinates on a map.

At the present moment in time, the precise location x is unknown, but we do have some beliefs about x.

One way to summarize our knowledge is a point prediction $\hat{x}$

- But what if the President wants to know the probability that the missile is currently over the Sea of Japan?
- Then it is better to summarize our initial beliefs with a bivariate probability density p
  - $\int_E p(x)dx$ indicates the probability that we attach to the missile being in region E.

The density p is called our **prior** for the random variable x.

To keep things tractable in our example, we assume that our prior is Gaussian.

In particular, we take

$$p = N(\hat{x}, \Sigma) \tag{41.1}$$

where $\hat{x}$ is the mean of the distribution and Σ is a 2×2 covariance matrix. In our simulations, we will suppose that

$$\hat{x} = \begin{pmatrix} 0.2 \\ -0.2 \end{pmatrix}, \qquad \Sigma = \begin{pmatrix} 0.4 & 0.3 \\ 0.3 & 0.45 \end{pmatrix} \tag{41.2}$$

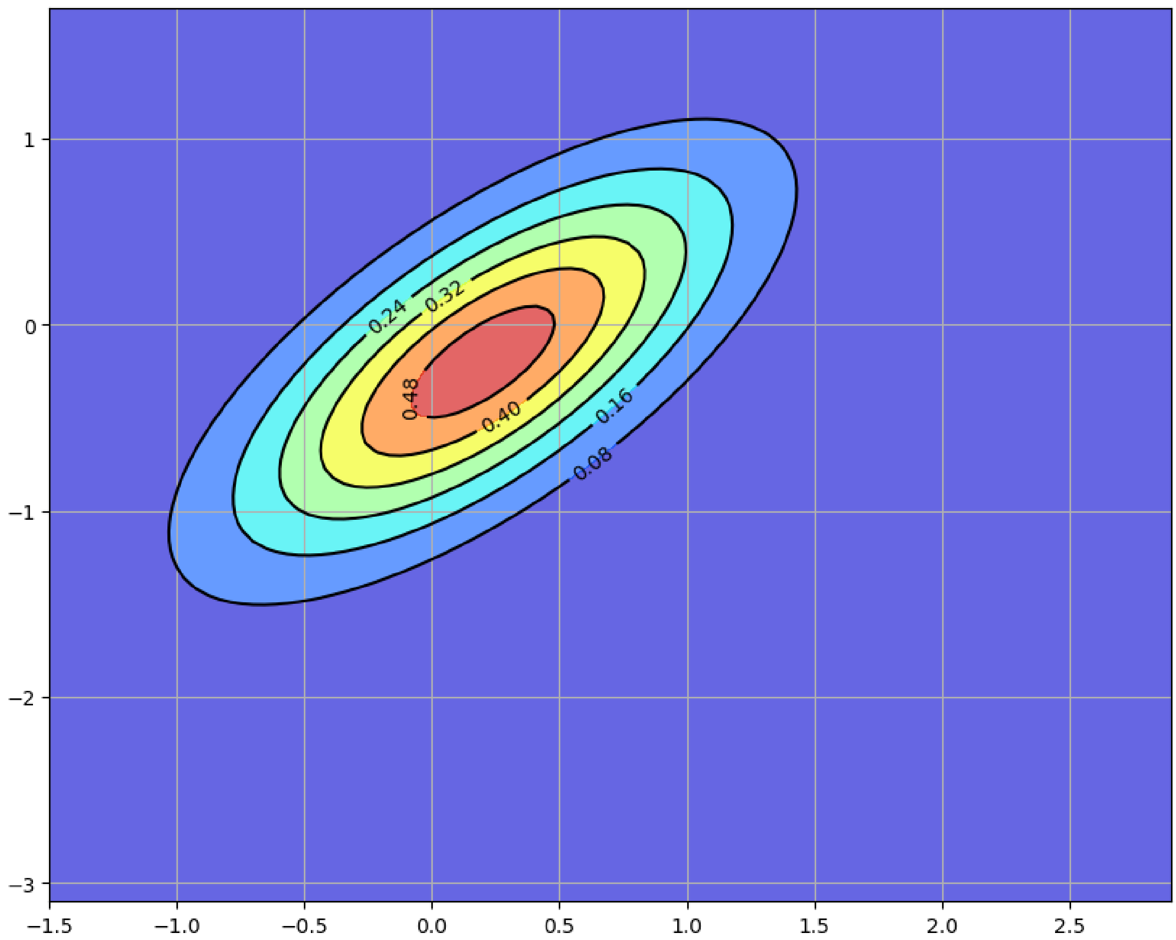
This density $p(x)$ is shown below as a contour map, with the center of the red ellipse being equal to $\hat{x}$.

```
# Set up the Gaussian prior density p
Σ = [[0.4, 0.3], [0.3, 0.45]]
Σ = np.matrix(Σ)
x_hat = np.matrix([0.2, -0.2]).T
# Define the matrices G and R from the equation y = G x + N(0, R)
G = [[1, 0], [0, 1]]
G = np.matrix(G)
R = 0.5 * Σ
# The matrices A and Q
A = [[1.2, 0], [0, -0.2]]
A = np.matrix(A)
Q = 0.3 * Σ
# The observed value of y
y = np.matrix([2.3, -1.9]).T

# Set up grid for plotting
x_grid = np.linspace(-1.5, 2.9, 100)
y_grid = np.linspace(-3.1, 1.7, 100)
X, Y = np.meshgrid(x_grid, y_grid)

def bivariate_normal(x, y, σ_x=1.0, σ_y=1.0, μ_x=0.0, μ_y=0.0, σ_xy=0.0):
```

(continues on next page)

(continued from previous page)

```
    """
    Compute and return the probability density function of bivariate normal
    distribution of normal random variables x and y

    Parameters
    ----------
    x : array_like(float)
        Random variable

    y : array_like(float)
        Random variable

    σ_x : array_like(float)
          Standard deviation of random variable x

    σ_y : array_like(float)
          Standard deviation of random variable y

    μ_x : scalar(float)
          Mean value of random variable x

    μ_y : scalar(float)
          Mean value of random variable y

    σ_xy : array_like(float)
           Covariance of random variables x and y

    """

    x_μ = x - μ_x
    y_μ = y - μ_y

    ρ = σ_xy / (σ_x * σ_y)
    z = x_μ**2 / σ_x**2 + y_μ**2 / σ_y**2 - 2 * ρ * x_μ * y_μ / (σ_x * σ_y)
    denom = 2 * np.pi * σ_x * σ_y * np.sqrt(1 - ρ**2)
    return np.exp(-z / (2 * (1 - ρ**2))) / denom

def gen_gaussian_plot_vals(μ, C):
    "Z values for plotting the bivariate Gaussian N(μ, C)"
    m_x, m_y = float(μ[0,0]), float(μ[1,0])
    s_x, s_y = np.sqrt(C[0, 0]), np.sqrt(C[1, 1])
    s_xy = C[0, 1]
    return bivariate_normal(X, Y, s_x, s_y, m_x, m_y, s_xy)

# Plot the figure

fig, ax = plt.subplots(figsize=(10, 8))
ax.grid()

Z = gen_gaussian_plot_vals(x_hat, Σ)
ax.contourf(X, Y, Z, 6, alpha=0.6, cmap=cm.jet)
cs = ax.contour(X, Y, Z, 6, colors="black")
ax.clabel(cs, inline=1, fontsize=10)

plt.show()
```



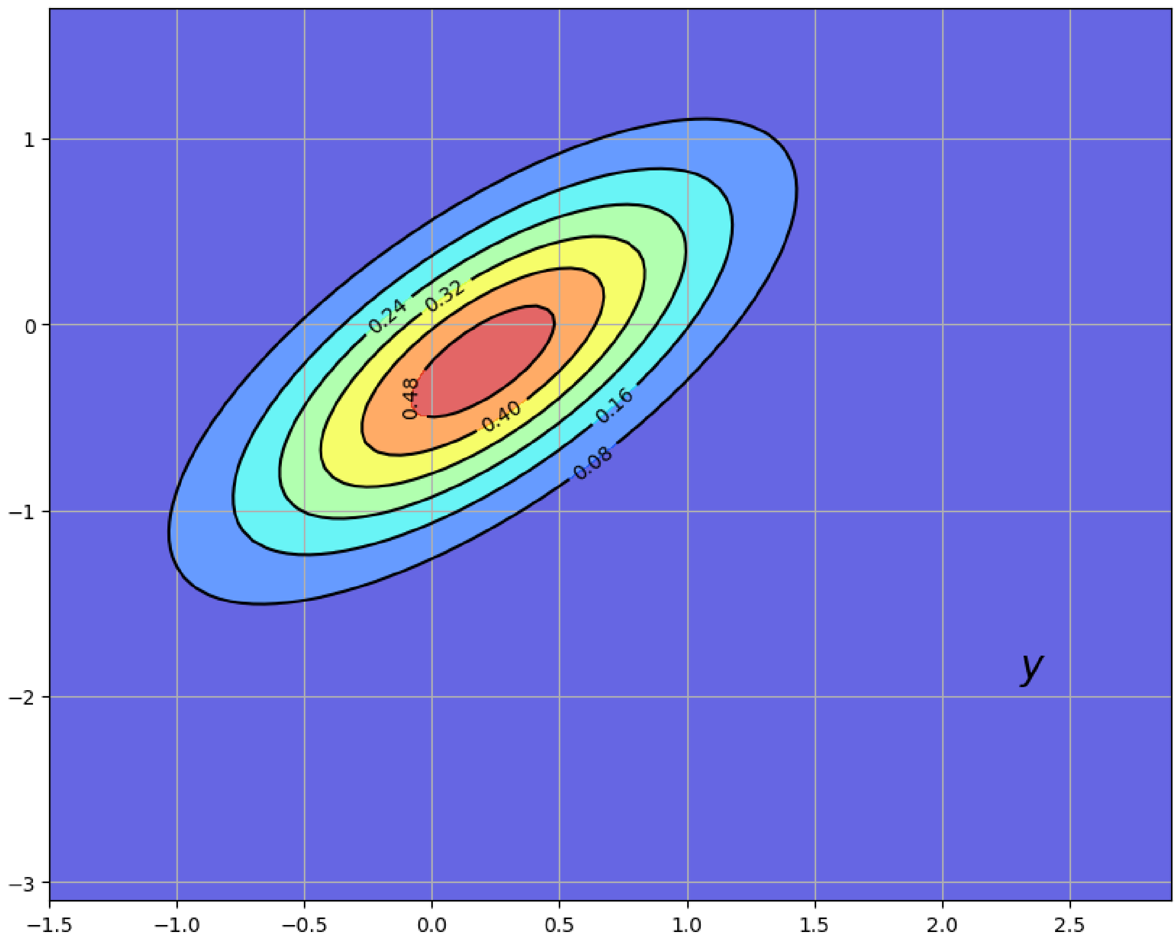
41.2.1 The Filtering Step

We are now presented with some good news and some bad news.

The good news is that the missile has been located by our sensors, which report that the current location is $y = (2.3, -1.9)$.

The next figure shows the original prior $p(x)$ and the new reported location y

```
fig, ax = plt.subplots(figsize=(10, 8))
ax.grid()

Z = gen_gaussian_plot_vals(x_hat, Σ)
ax.contourf(X, Y, Z, 6, alpha=0.6, cmap=cm.jet)
cs = ax.contour(X, Y, Z, 6, colors="black")
ax.clabel(cs, inline=1, fontsize=10)
ax.text(float(y[0].item()), float(y[1].item()), "$y$", fontsize=20, color="black")

plt.show()
```



The bad news is that our sensors are imprecise.

In particular, we should interpret the output of our sensor not as $y = x$, but rather as

$$y = Gx + v, \quad \text{where} \quad v \sim N(0, R) \tag{41.3}$$

Here G and R are 2×2 matrices with R positive definite. Both are assumed known, and the noise term v is assumed to be independent of x.

How then should we combine our prior $p(x) = N(\hat{x}, \Sigma)$ and this new information y to improve our understanding of the location of the missile?

As you may have guessed, the answer is to use Bayes' theorem, which tells us to update our prior $p(x)$ to $p(x \,|\, y)$ via

$$p(x \,|\, y) = \frac{p(y \,|\, x)\, p(x)}{p(y)}$$

where $p(y) = \int p(y \,|\, x)\, p(x) dx$.

In solving for $p(x \,|\, y)$, we observe that

- $p(x) = N(\hat{x}, \Sigma)$.
- In view of (41.3), the conditional density $p(y \,|\, x)$ is $N(Gx, R)$.
- $p(y)$ does not depend on x, and enters into the calculations only as a normalizing constant.

Because we are in a linear and Gaussian framework, the updated density can be computed by calculating population linear regressions.

In particular, the solution is known[1] to be

$$p(x \mid y) = N(\hat{x}^F, \Sigma^F)$$

where

$$\hat{x}^F := \hat{x} + \Sigma G' (G \Sigma G' + R)^{-1}(y - G\hat{x}) \quad \text{and} \quad \Sigma^F := \Sigma - \Sigma G' (G \Sigma G' + R)^{-1} G \Sigma \tag{41.4}$$

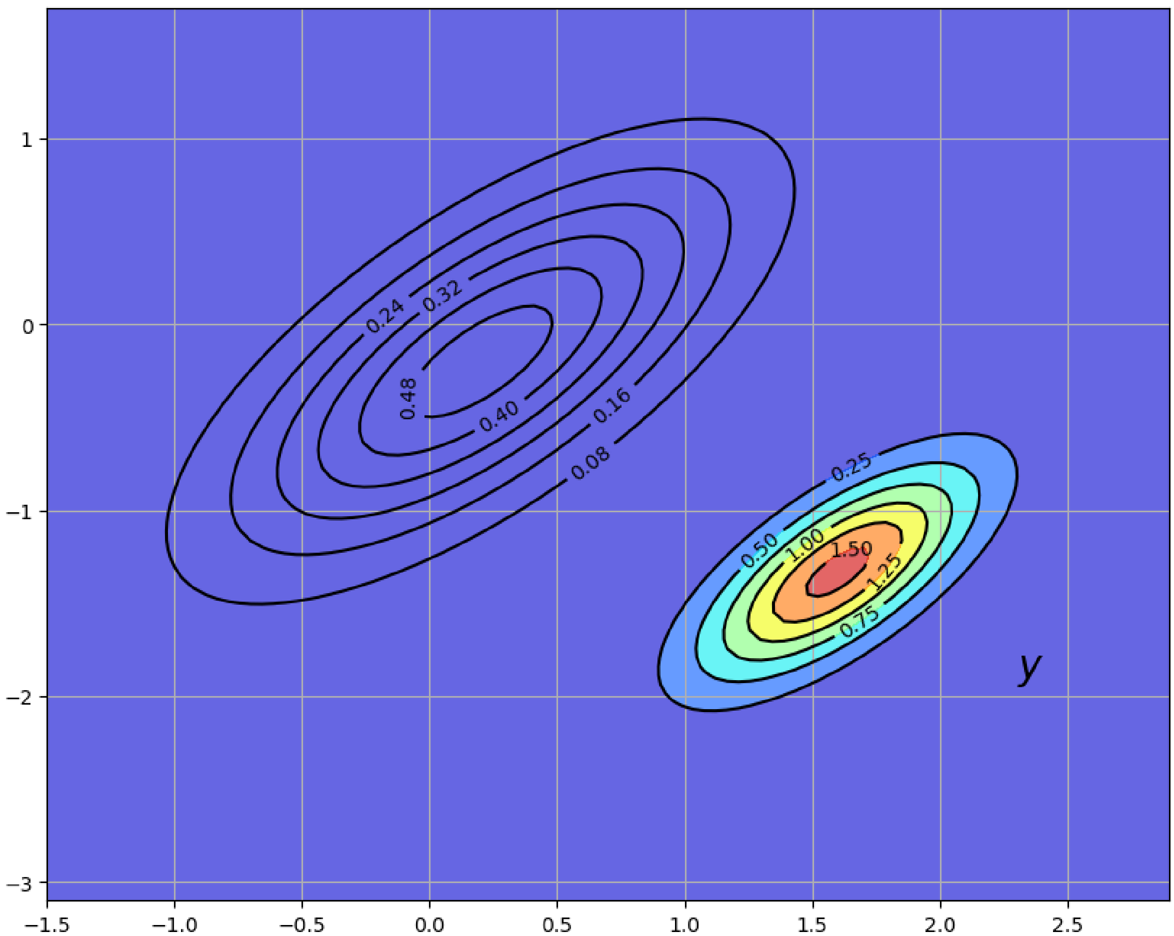
Here $\Sigma G' (G \Sigma G' + R)^{-1}$ is the matrix of population regression coefficients of the hidden object $x - \hat{x}$ on the surprise $y - G\hat{x}$.

This new density $p(x \mid y) = N(\hat{x}^F, \Sigma^F)$ is shown in the next figure via contour lines and the color map.

The original density is left in as contour lines for comparison

```
fig, ax = plt.subplots(figsize=(10, 8))
ax.grid()

Z = gen_gaussian_plot_vals(x_hat, Σ)
cs1 = ax.contour(X, Y, Z, 6, colors="black")
ax.clabel(cs1, inline=1, fontsize=10)
M = Σ * G.T * linalg.inv(G * Σ * G.T + R)
x_hat_F = x_hat + M * (y - G * x_hat)
Σ_F = Σ - M * G * Σ
new_Z = gen_gaussian_plot_vals(x_hat_F, Σ_F)
cs2 = ax.contour(X, Y, new_Z, 6, colors="black")
ax.clabel(cs2, inline=1, fontsize=10)
ax.contourf(X, Y, new_Z, 6, alpha=0.6, cmap=cm.jet)
ax.text(float(y[0].item()), float(y[1].item()), "$y$", fontsize=20, color="black")

plt.show()
```

[1] See, for example, page 93 of [Bishop, 2006]. To get from his expressions to the ones used above, you will also need to apply the Woodbury matrix identity.



Our new density twists the prior $p(x)$ in a direction determined by the new information $y - G\hat{x}$.

In generating the figure, we set G to the identity matrix and $R = 0.5\Sigma$ for Σ defined in (41.2).

41.2.2 The Forecast Step

What have we achieved so far?

We have obtained probabilities for the current location of the state (missile) given prior and current information.

This is called "filtering" rather than forecasting because we are filtering out noise rather than looking into the future.

- $p(x \mid y) = N(\hat{x}^F, \Sigma^F)$ is called the **filtering distribution**

But now let's suppose that we are given another task: to predict the location of the missile after one unit of time (whatever that may be) has elapsed.

To do this we need a model of how the state evolves.

Let's suppose that we have one, and that it's linear and Gaussian. In particular,

$$x_{t+1} = Ax_t + w_{t+1}, \quad \text{where} \quad w_t \sim N(0, Q) \tag{41.5}$$

Our aim is to combine this law of motion and our current distribution $p(x \mid y) = N(\hat{x}^F, \Sigma^F)$ to come up with a new **predictive** distribution for the location in one unit of time.

In view of (41.5), all we have to do is introduce a random vector $x^F \sim N(\hat{x}^F, \Sigma^F)$ and work out the distribution of $Ax^F + w$ where w is independent of x^F and has distribution $N(0, Q)$.

Since linear combinations of Gaussians are Gaussian, $Ax^F + w$ is Gaussian.

Elementary calculations and the expressions in (41.4) tell us that

$$\mathbb{E}[Ax^F + w] = A\mathbb{E}x^F + \mathbb{E}w = A\hat{x}^F = A\hat{x} + A\Sigma G'(G\Sigma G' + R)^{-1}(y - G\hat{x})$$

and

$$\operatorname{Var}[Ax^F + w] = A\operatorname{Var}[x^F]A' + Q = A\Sigma^F A' + Q = A\Sigma A' - A\Sigma G'(G\Sigma G' + R)^{-1}G\Sigma A' + Q$$

The matrix $A\Sigma G'(G\Sigma G' + R)^{-1}$ is often written as K_Σ and called the **Kalman gain**.

- The subscript Σ has been added to remind us that K_Σ depends on Σ, but not y or $\hat{x}$.

Using this notation, we can summarize our results as follows.

Our updated prediction is the density $N(\hat{x}_{new}, \Sigma_{new})$ where

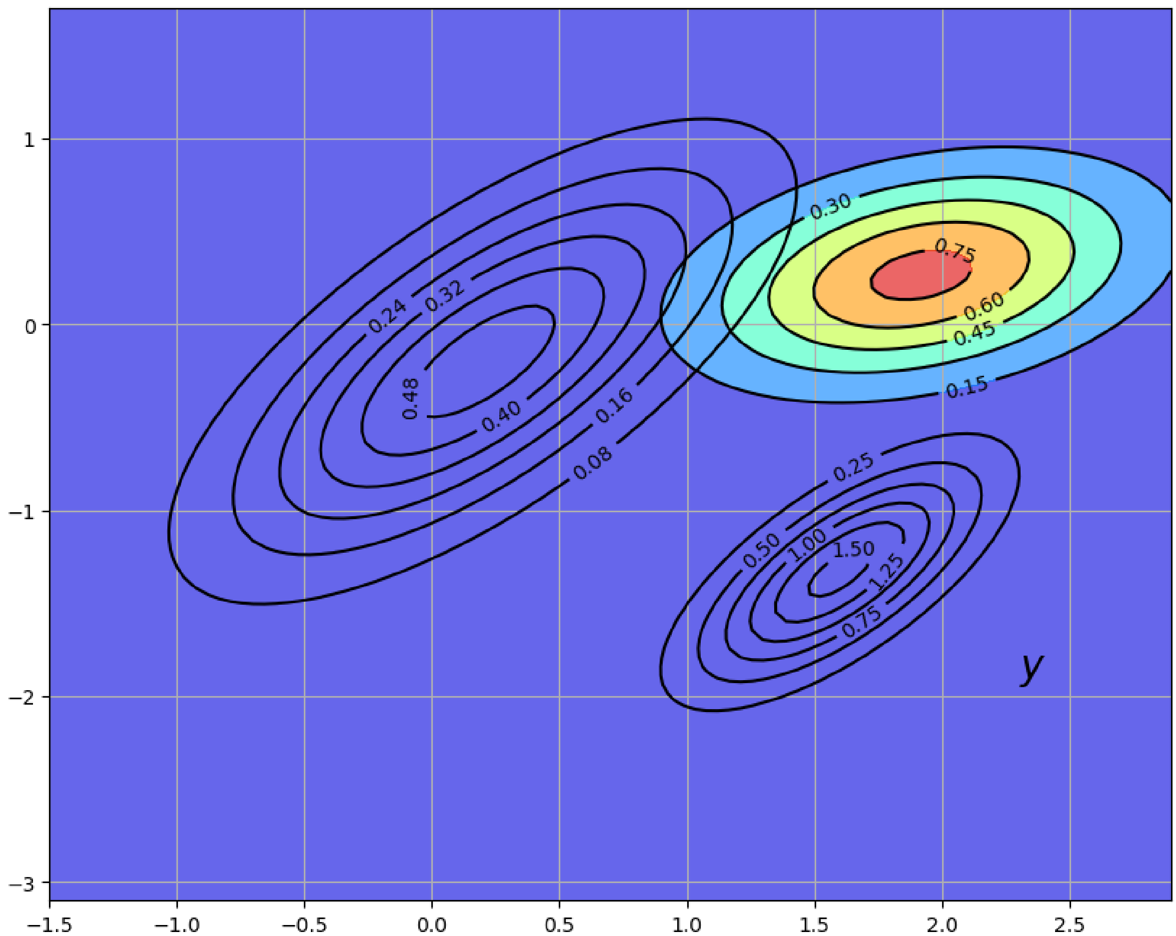
$$\begin{aligned} \hat{x}_{new} &:= A\hat{x} + K_\Sigma(y - G\hat{x}) \\ \Sigma_{new} &:= A\Sigma A' - K_\Sigma G\Sigma A' + Q \end{aligned}$$

- The density $p_{new}(x) = N(\hat{x}_{new}, \Sigma_{new})$ is called the **predictive distribution**

The predictive distribution is the new density shown in the following figure, where the update has used parameters.

$$A = \begin{pmatrix} 1.2 & 0.0 \\ 0.0 & -0.2 \end{pmatrix}, \qquad Q = 0.3 * \Sigma$$

```python
fig, ax = plt.subplots(figsize=(10, 8))
ax.grid()

# Density 1
Z = gen_gaussian_plot_vals(x_hat, Σ)
cs1 = ax.contour(X, Y, Z, 6, colors="black")
ax.clabel(cs1, inline=1, fontsize=10)

# Density 2
M = Σ * G.T * linalg.inv(G * Σ * G.T + R)
x_hat_F = x_hat + M * (y - G * x_hat)
Σ_F = Σ - M * G * Σ
Z_F = gen_gaussian_plot_vals(x_hat_F, Σ_F)
cs2 = ax.contour(X, Y, Z_F, 6, colors="black")
ax.clabel(cs2, inline=1, fontsize=10)

# Density 3
new_x_hat = A * x_hat_F
new_Σ = A * Σ_F * A.T + Q
new_Z = gen_gaussian_plot_vals(new_x_hat, new_Σ)
cs3 = ax.contour(X, Y, new_Z, 6, colors="black")
ax.clabel(cs3, inline=1, fontsize=10)
ax.contourf(X, Y, new_Z, 6, alpha=0.6, cmap=cm.jet)
ax.text(float(y[0].item()), float(y[1].item()), "$y$", fontsize=20, color="black")

plt.show()
```



41.2.3 The Recursive Procedure

Let's look back at what we've done.

We started the current period with a prior $p(x)$ for the location x of the missile.

We then used the current measurement y to update to $p(x \mid y)$.

Finally, we used the law of motion (41.5) for $\{x_t\}$ to update to $p_{new}(x)$.

If we now step into the next period, we are ready to go round again, taking $p_{new}(x)$ as the current prior.

Swapping notation $p_t(x)$ for $p(x)$ and $p_{t+1}(x)$ for $p_{new}(x)$, the full recursive procedure is:

1. Start the current period with prior $p_t(x) = N(\hat{x}_t, \Sigma_t)$.
2. Observe current measurement y_t.
3. Compute the filtering distribution $p_t(x \mid y) = N(\hat{x}_t^F, \Sigma_t^F)$ from $p_t(x)$ and y_t, applying Bayes rule and the conditional distribution (41.3).
4. Compute the predictive distribution $p_{t+1}(x) = N(\hat{x}_{t+1}, \Sigma_{t+1})$ from the filtering distribution and (41.5).
5. Increment t by one and go to step 1.

Repeating (41.6), the dynamics for $\hat{x}_t$ and Σ_t are as follows

$$\begin{aligned}\hat{x}_{t+1} &= A\hat{x}_t + K_{\Sigma_t}(y_t - G\hat{x}_t) \\ \Sigma_{t+1} &= A\Sigma_t A' - K_{\Sigma_t} G\Sigma_t A' + Q\end{aligned}$$

These are the standard dynamic equations for the Kalman filter (see, for example, [Ljungqvist and Sargent, 2018], page 58).

41.3 Convergence

The matrix Σ_t is a measure of the uncertainty of our prediction $\hat{x}_t$ of x_t.

Apart from special cases, this uncertainty will never be fully resolved, regardless of how much time elapses.

One reason is that our prediction $\hat{x}_t$ is made based on information available at $t-1$, not t.

Even if we know the precise value of x_{t-1} (which we don't), the transition equation (41.5) implies that $x_t = Ax_{t-1} + w_t$.

Since the shock w_t is not observable at $t-1$, any time $t-1$ prediction of x_t will incur some error (unless w_t is degenerate).

However, it is certainly possible that Σ_t converges to a constant matrix as $t \to \infty$.

To study this topic, let's expand the second equation in (41.6):

$$\Sigma_{t+1} = A\Sigma_t A' - A\Sigma_t G'(G\Sigma_t G' + R)^{-1} G\Sigma_t A' + Q \tag{41.6}$$

This is a nonlinear difference equation in Σ_t.

A fixed point of (41.6) is a constant matrix Σ such that

$$\Sigma = A\Sigma A' - A\Sigma G'(G\Sigma G' + R)^{-1} G\Sigma A' + Q \tag{41.7}$$

Equation (41.6) is known as a discrete-time Riccati difference equation.

Equation (41.7) is known as a discrete-time algebraic Riccati equation.

Conditions under which a fixed point exists and the sequence $\{\Sigma_t\}$ converges to it are discussed in [Anderson *et al.*, 1996] and [Anderson and Moore, 2005], chapter 4.

A sufficient (but not necessary) condition is that all the eigenvalues λ_i of A satisfy $|\lambda_i| < 1$ (cf. e.g., [Anderson and Moore, 2005], p. 77).

(This strong condition assures that the unconditional distribution of x_t converges as $t \to +\infty$.)

In this case, for any initial choice of Σ_0 that is both non-negative and symmetric, the sequence $\{\Sigma_t\}$ in (41.6) converges to a non-negative symmetric matrix Σ that solves (41.7).

41.4 Implementation

The class `Kalman` from the QuantEcon.py package implements the Kalman filter

- Instance data consists of:
  - the moments $(\hat{x}_t, \Sigma_t)$ of the current prior.
  - An instance of the LinearStateSpace class from QuantEcon.py.

The latter represents a linear state space model of the form

$$
\begin{aligned}
x_{t+1} &= Ax_t + Cw_{t+1} \\
y_t &= Gx_t + Hv_t
\end{aligned}
$$

where the shocks w_t and v_t are IID standard normals.

To connect this with the notation of this lecture we set

$$
Q := CC' \quad \text{and} \quad R := HH'
$$

- The class `Kalman` from the QuantEcon.py package has a number of methods, some that we will wait to use until we study more advanced applications in subsequent lectures.
- Methods pertinent for this lecture are:
  - `prior_to_filtered`, which updates $(\hat{x}_t, \Sigma_t)$ to $(\hat{x}_t^F, \Sigma_t^F)$
  - `filtered_to_forecast`, which updates the filtering distribution to the predictive distribution – which becomes the new prior $(\hat{x}_{t+1}, \Sigma_{t+1})$
  - `update`, which combines the last two methods
  - a `stationary_values`, which computes the solution to (41.7) and the corresponding (stationary) Kalman gain

You can view the program on GitHub.

41.5 Exercises

Exercise 41.5.1

Consider the following simple application of the Kalman filter, loosely based on [Ljungqvist and Sargent, 2018], section 2.9.2.

Suppose that

- all variables are scalars
- the hidden state $\{x_t\}$ is in fact constant, equal to some $\theta \in \mathbb{R}$ unknown to the modeler

State dynamics are therefore given by (41.5) with $A = 1$, $Q = 0$ and $x_0 = \theta$.

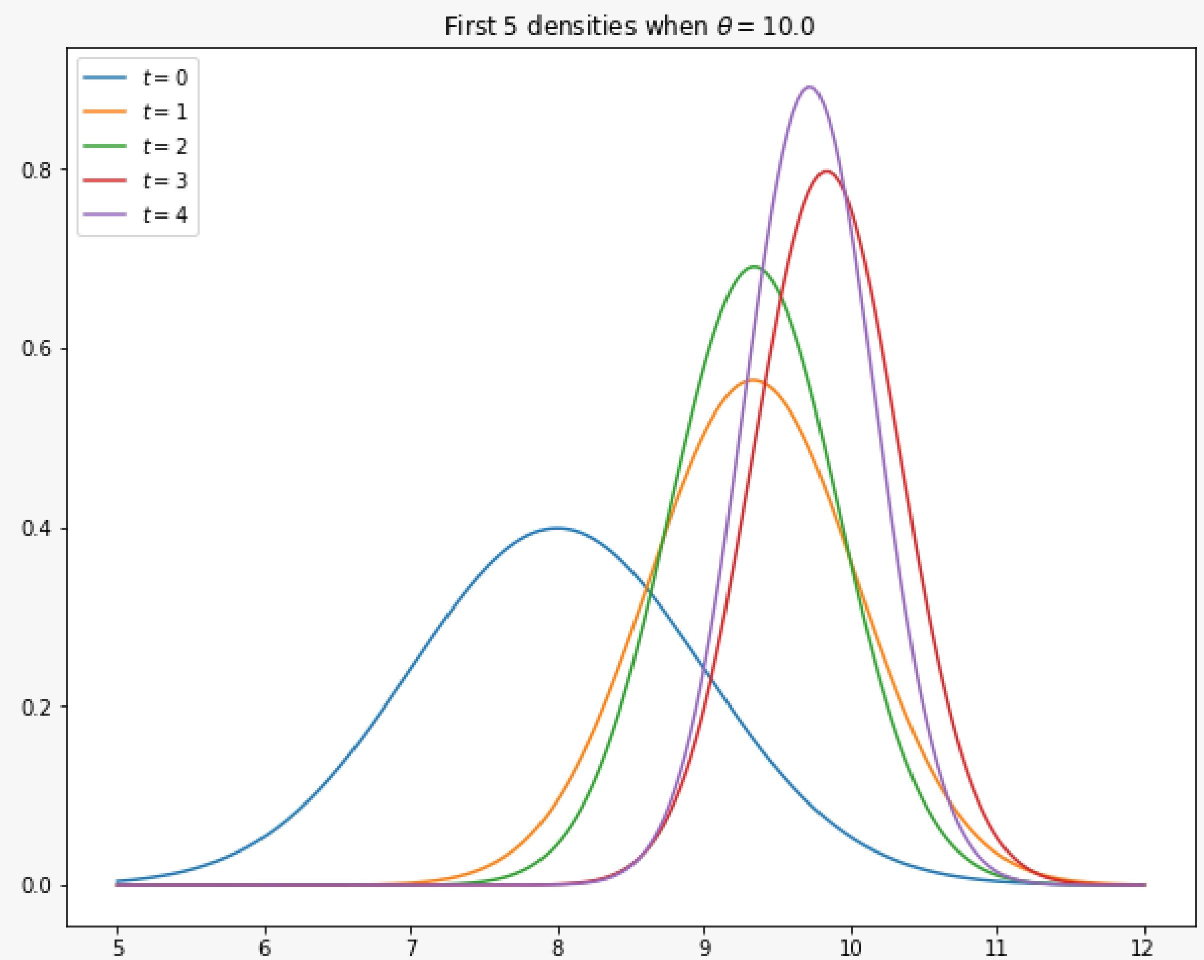
The measurement equation is $y_t = \theta + v_t$ where v_t is $N(0, 1)$ and IID.

The task of this exercise to simulate the model and, using the code from `kalman.py`, plot the first five predictive densities $p_t(x) = N(\hat{x}_t, \Sigma_t)$.

As shown in [Ljungqvist and Sargent, 2018], sections 2.9.1–2.9.2, these distributions asymptotically put all mass on the unknown value θ.

In the simulation, take $\theta = 10$, $\hat{x}_0 = 8$ and $\Sigma_0 = 1$.

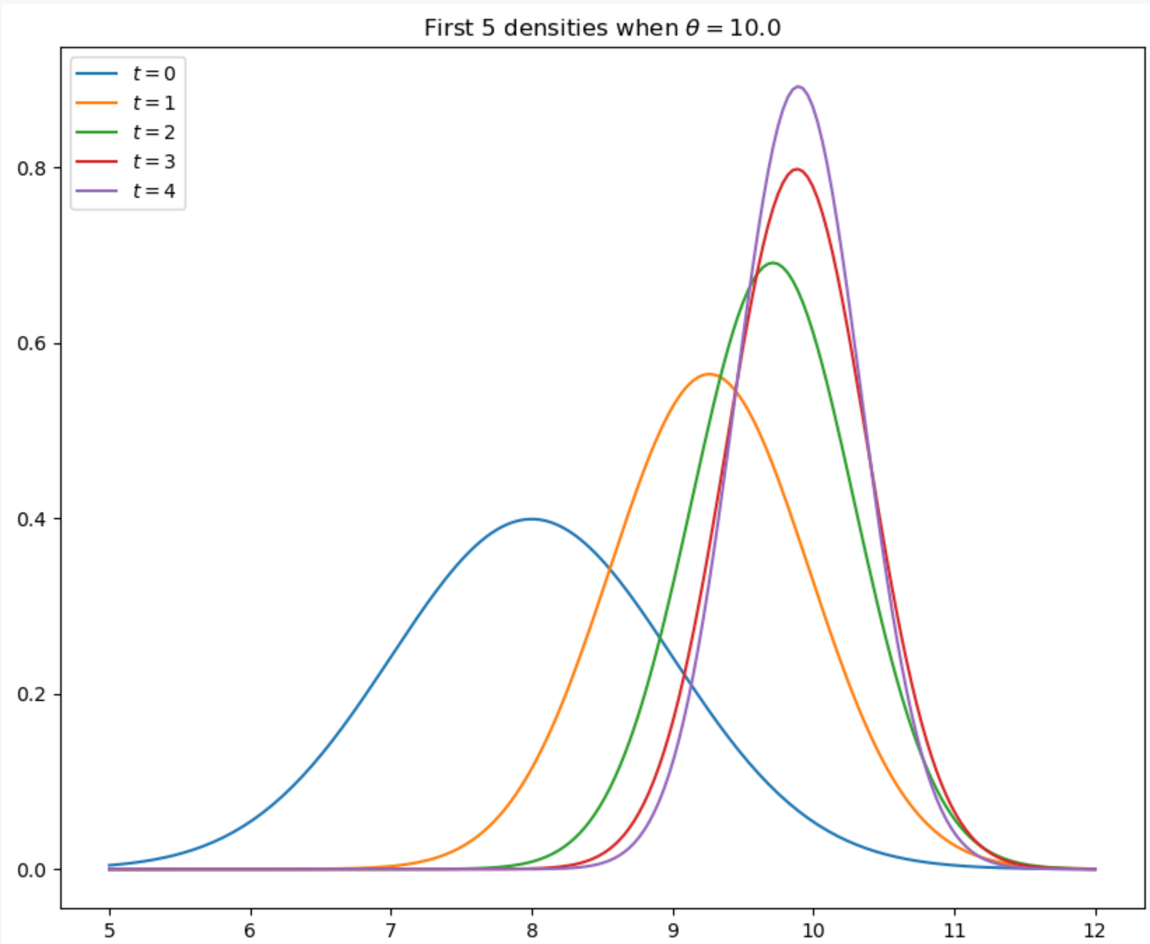
Your figure should – modulo randomness – look something like this



Solution

```python
# Parameters
θ = 10  # Constant value of state x_t
A, C, G, H = 1, 0, 1, 1
ss = LinearStateSpace(A, C, G, H, mu_0=θ)

# Set prior, initialize kalman filter
x_hat_0, Σ_0 = 8, 1
kalman = Kalman(ss, x_hat_0, Σ_0)

# Draw observations of y from state space model
N = 5
x, y = ss.simulate(N)
y = y.flatten()

# Set up plot
fig, ax = plt.subplots(figsize=(10,8))
xgrid = np.linspace(θ - 5, θ + 2, 200)

for i in range(N):
```

```
    # Record the current predicted mean and variance
    m, v = [float(z) for z in (kalman.x_hat.item(), kalman.Sigma.item())]
    # Plot, update filter
    ax.plot(xgrid, norm.pdf(xgrid, loc=m, scale=np.sqrt(v)), label=f'$t={i}$')
    kalman.update(y[i])

ax.set_title(f'First {N} densities when $\\theta = {θ:.1f}$')
ax.legend(loc='upper left')
plt.show()
```



Exercise 41.5.2

The preceding figure gives some support to the idea that probability mass converges to θ.

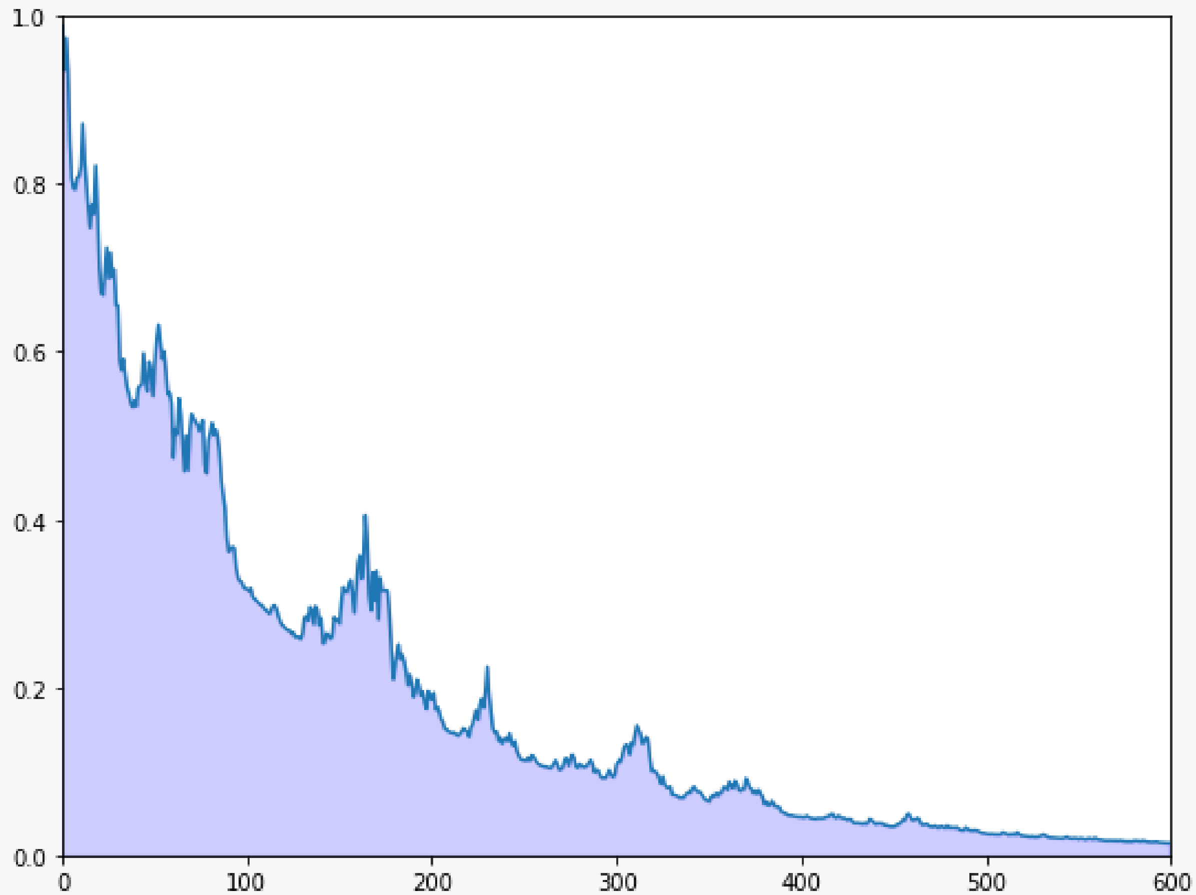
To get a better idea, choose a small $\epsilon > 0$ and calculate

$$z_t := 1 - \int_{\theta - \epsilon}^{\theta + \epsilon} p_t(x) dx$$

for $t = 0, 1, 2, \ldots, T$.

Plot z_t against T, setting $\epsilon = 0.1$ and $T = 600$.

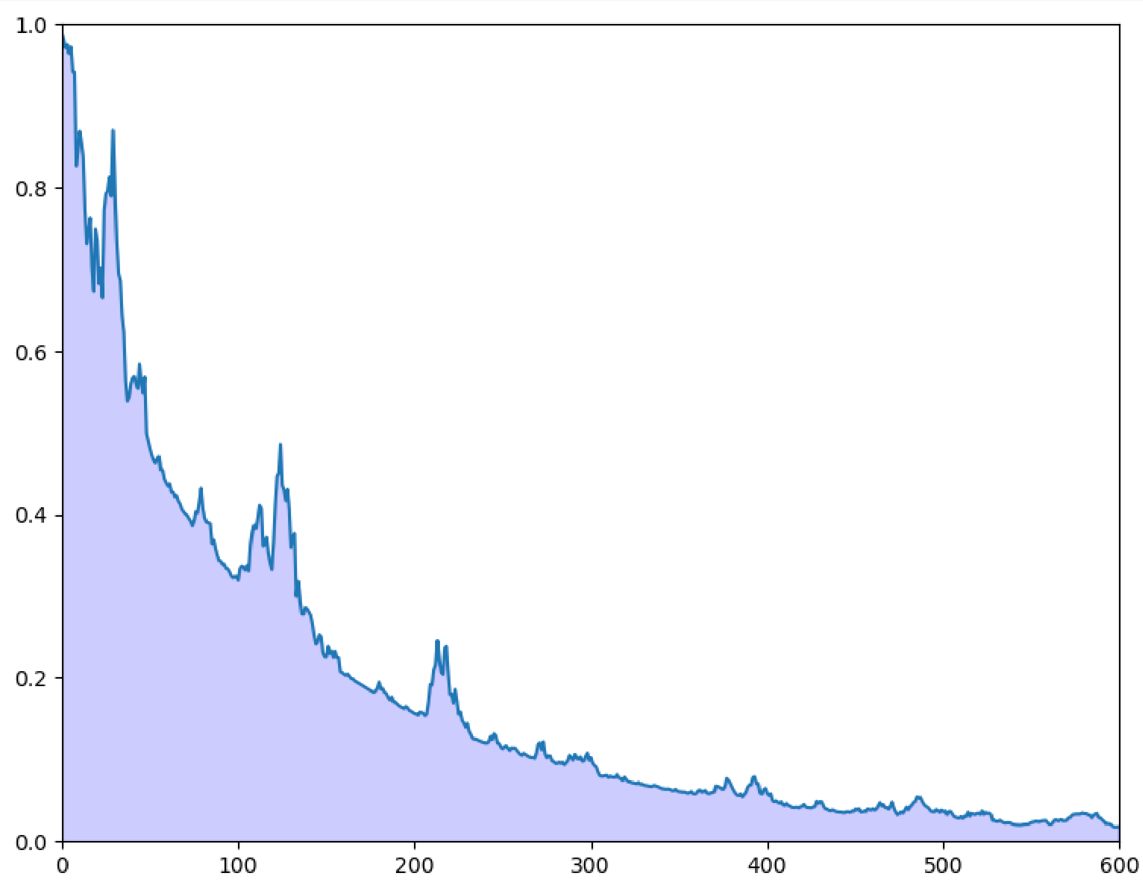
Your figure should show error erratically declining something like this



Solution

```
ϵ = 0.1
θ = 10  # Constant value of state x_t
A, C, G, H = 1, 0, 1, 1
ss = LinearStateSpace(A, C, G, H, mu_0=θ)

x_hat_0, Σ_0 = 8, 1
kalman = Kalman(ss, x_hat_0, Σ_0)

T = 600
z = np.empty(T)
x, y = ss.simulate(T)
y = y.flatten()

for t in range(T):
    # Record the current predicted mean and variance and plot their densities
    m, v = [float(temp) for temp in (kalman.x_hat.item(), kalman.Sigma.item())]

    f = lambda x: norm.pdf(x, loc=m, scale=np.sqrt(v))
    integral, error = quad(f, θ - ϵ, θ + ϵ)
    z[t] = 1 - integral
```

```
    kalman.update(y[t])

fig, ax = plt.subplots(figsize=(9, 7))
ax.set_ylim(0, 1)
ax.set_xlim(0, T)
ax.plot(range(T), z)
ax.fill_between(range(T), np.zeros(T), z, color="blue", alpha=0.2)
plt.show()
```



Exercise 41.5.3

As discussed *above*, if the shock sequence $\{w_t\}$ is not degenerate, then it is not in general possible to predict x_t without error at time $t-1$ (and this would be the case even if we could observe x_{t-1}).

Let's now compare the prediction $\hat{x}_t$ made by the Kalman filter against a competitor who **is** allowed to observe x_{t-1}.

This competitor will use the conditional expectation $\mathbb{E}[x_t \mid x_{t-1}]$, which in this case is Ax_{t-1}.

The conditional expectation is known to be the optimal prediction method in terms of minimizing mean squared error.

(More precisely, the minimizer of $\mathbb{E}\,\|x_t - g(x_{t-1})\|^2$ with respect to g is $g^*(x_{t-1}) := \mathbb{E}[x_t \mid x_{t-1}]$)

Thus we are comparing the Kalman filter against a competitor who has more information (in the sense of being able to observe the latent state) and behaves optimally in terms of minimizing squared error.

Our horse race will be assessed in terms of squared error.

In particular, your task is to generate a graph plotting observations of both $\|x_t - Ax_{t-1}\|^2$ and $\|x_t - \hat{x}_t\|^2$ against t for $t = 1, \ldots, 50$.

For the parameters, set $G = I, R = 0.5I$ and $Q = 0.3I$, where I is the 2×2 identity.

Set

$$A = \begin{pmatrix} 0.5 & 0.4 \\ 0.6 & 0.3 \end{pmatrix}$$

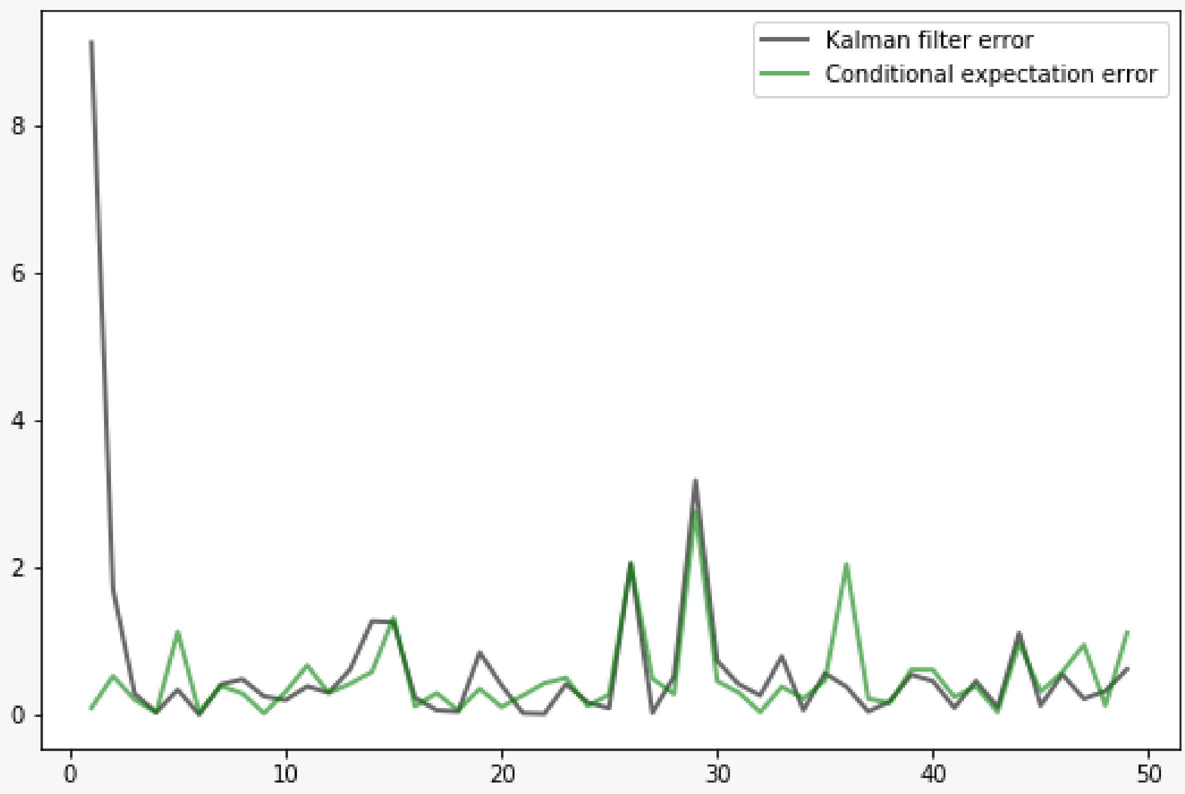
To initialize the prior density, set

$$\Sigma_0 = \begin{pmatrix} 0.9 & 0.3 \\ 0.3 & 0.9 \end{pmatrix}$$

and $\hat{x}_0 = (8, 8)$.

Finally, set $x_0 = (0, 0)$.

You should end up with a figure similar to the following (modulo randomness)



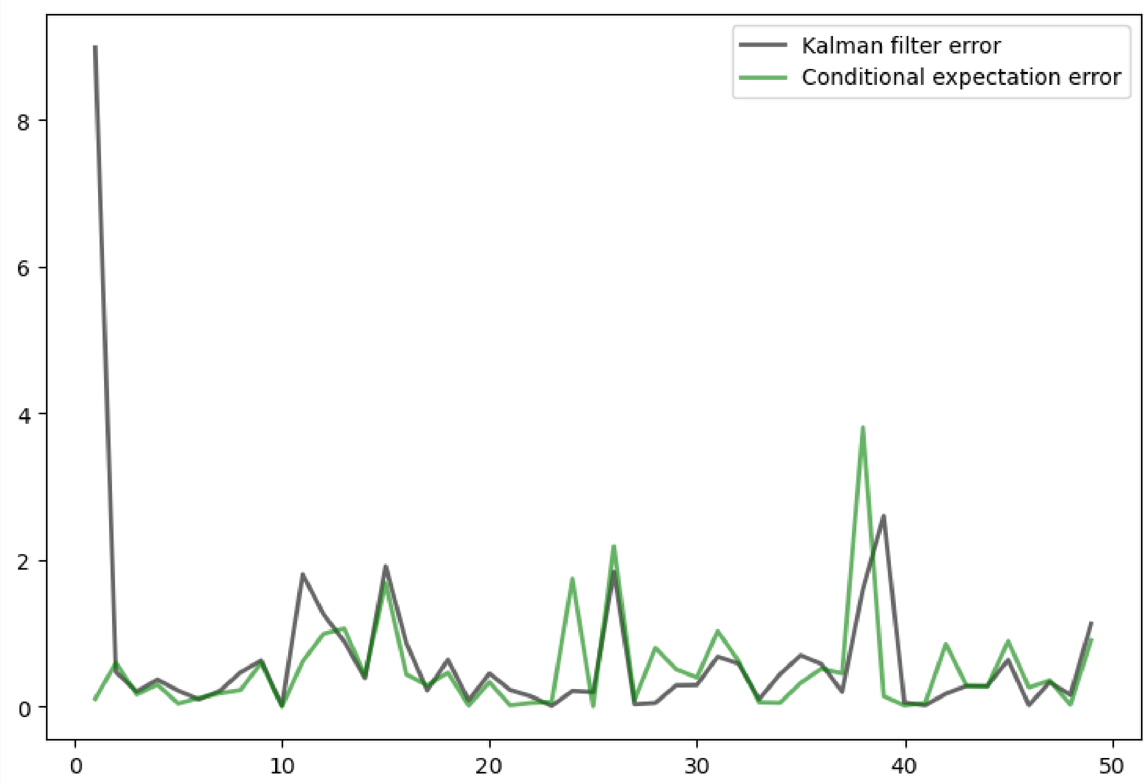
Observe how, after an initial learning period, the Kalman filter performs quite well, even relative to the competitor who predicts optimally with knowledge of the latent state.

Solution

```
# Define A, C, G, H
```

```
G = np.identity(2)
H = np.sqrt(0.5) * np.identity(2)

A = [[0.5, 0.4],
     [0.6, 0.3]]
C = np.sqrt(0.3) * np.identity(2)

# Set up state space mode, initial value x_0 set to zero
ss = LinearStateSpace(A, C, G, H, mu_0 = np.zeros(2))

# Define the prior density
Σ = [[0.9, 0.3],
     [0.3, 0.9]]
Σ = np.array(Σ)
x_hat = np.array([8, 8])

# Initialize the Kalman filter
kn = Kalman(ss, x_hat, Σ)

# Print eigenvalues of A
print("Eigenvalues of A:")
print(eigvals(A))

# Print stationary Σ
S, K = kn.stationary_values()
print("Stationary prediction error variance:")
print(S)

# Generate the plot
T = 50
x, y = ss.simulate(T)

e1 = np.empty(T-1)
e2 = np.empty(T-1)

for t in range(1, T):
    kn.update(y[:,t])
    diff1 = x[:, t] - kn.x_hat.flatten()
    diff2 = x[:, t] - A @ x[:, t-1]
    e1[t-1] = diff1 @ diff1
    e2[t-1] = diff2 @ diff2

fig, ax = plt.subplots(figsize=(9,6))
ax.plot(range(1, T), e1, 'k-', lw=2, alpha=0.6,
        label='Kalman filter error')
ax.plot(range(1, T), e2, 'g-', lw=2, alpha=0.6,
        label='Conditional expectation error')
ax.legend()
plt.show()
```

```
Eigenvalues of A:
[ 0.9+0.j -0.1+0.j]
Stationary prediction error variance:
[[0.40329108 0.1050718 ]
 [0.1050718  0.41061709]]
```



Exercise 41.5.4

Try varying the coefficient 0.3 in $Q = 0.3I$ up and down.

Observe how the diagonal values in the stationary solution Σ (see (41.7)) increase and decrease in line with this coefficient.

The interpretation is that more randomness in the law of motion for x_t causes more (permanent) uncertainty in prediction.

CHAPTER

FORTYTWO

ANOTHER LOOK AT THE KALMAN FILTER

Contents

In this quantecon lecture *A First Look at the Kalman filter*, we used a Kalman filter to estimate locations of a rocket.

In this lecture, we'll use the Kalman filter to infer a worker's human capital and the effort that the worker devotes to accumulating human capital, neither of which the firm observes directly.

The firm learns about those things only by observing a history of the output that the worker generates for the firm, and from understanding how that output depends on the worker's human capital and how human capital evolves as a function of the worker's effort.

We'll posit a rule that expresses how the much firm pays the worker each period as a function of the firm's information each period.

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
```

To conduct simulations, we bring in these imports, as in *A First Look at the Kalman filter*.

```
import matplotlib.pyplot as plt
import numpy as np
from quantecon import Kalman, LinearStateSpace
from collections import namedtuple
from scipy.stats import multivariate_normal
import matplotlib as mpl
mpl.rcParams['text.usetex'] = True
mpl.rcParams['text.latex.preamble'] = r'\usepackage{{amsmath}}'
```

42.1 A worker's output

A representative worker is permanently employed at a firm.

The workers' output is described by the following dynamic process:

$$
\begin{aligned}
h_{t+1} &= \alpha h_t + \beta u_t + c w_{t+1}, \quad c_{t+1} \sim \mathcal{N}(0,1) \\
u_{t+1} &= u_t \\
y_t &= g h_t + v_t, \quad v_t \sim \mathcal{N}(0,R)
\end{aligned}
\tag{42.1}
$$

Here

- h_t is the logarithm of human capital at time t
- u_t is the logarithm of the worker's effort at accumulating human capital at t
- y_t is the logarithm of the worker's output at time t
- $h_0 \sim \mathcal{N}(\hat{h}_0, \sigma_{h,0})$
- $u_0 \sim \mathcal{N}(\hat{u}_0, \sigma_{u,0})$

Parameters of the model are $\alpha, \beta, c, R, g, \hat{h}_0, \hat{u}_0, \sigma_h, \sigma_u$.

At time 0, a firm has hired the worker.

The worker is permanently attached to the firm and so works for the same firm at all dates $t = 0, 1, 2, \ldots$.

At the beginning of time 0, the firm observes neither the worker's innate initial human capital h_0 nor its hard-wired permanent effort level u_0.

The firm believes that u_0 for a particular worker is drawn from a Gaussian probability distribution, and so is described by $u_0 \sim \mathcal{N}(\hat{u}_0, \sigma_{u,0})$.

The h_t part of a worker's "type" moves over time, but the effort component of the worker's type is $u_t = u_0$.

This means that from the firm's point of view, the worker's effort is effectively an unknown fixed "parameter".

At time $t \geq 1$, for a particular worker the firm observed $y^{t-1} = [y_{t-1}, y_{t-2}, \ldots, y_0]$.

The firm does not observe the worker's "type" (h_0, u_0).

But the firm does observe the worker's output y_t at time t and remembers the worker's past outputs y^{t-1}.

42.2 A firm's wage-setting policy

Based on information about the worker that the firm has at time $t \geq 1$, the firm pays the worker log wage

$$
w_t = g E[h_t | y^{t-1}], \quad t \geq 1
$$

and at time 0 pays the worker a log wage equal to the unconditional mean of y_0:

$$
w_0 = g \hat{h}_0
$$

In using this payment rule, the firm is taking into account that the worker's log output today is partly due to the random component v_t that comes entirely from luck, and that is assumed to be independent of h_t and u_t.

42.3 A state-space representation

Write system *(42.1.1)* in the state-space form

$$\begin{bmatrix} h_{t+1} \\ u_{t+1} \end{bmatrix} = \begin{bmatrix} \alpha & \beta \\ 0 & 1 \end{bmatrix} \begin{bmatrix} h_t \\ u_t \end{bmatrix} + \begin{bmatrix} c \\ 0 \end{bmatrix} w_{t+1}$$
$$y_t = \begin{bmatrix} g & 0 \end{bmatrix} \begin{bmatrix} h_t \\ u_t \end{bmatrix} + v_t$$

which is equivalent with

$$\begin{aligned} x_{t+1} &= Ax_t + Cw_{t+1} \\ y_t &= Gx_t + v_t \\ x_0 &\sim \mathcal{N}(\hat{x}_0, \Sigma_0) \end{aligned} \tag{42.2}$$

where

$$x_t = \begin{bmatrix} h_t \\ u_t \end{bmatrix}, \quad \hat{x}_0 = \begin{bmatrix} \hat{h}_0 \\ \hat{u}_0 \end{bmatrix}, \quad \Sigma_0 = \begin{bmatrix} \sigma_{h,0} & 0 \\ 0 & \sigma_{u,0} \end{bmatrix}$$

To compute the firm's wage setting policy, we first we create a `namedtuple` to store the parameters of the model

```python
WorkerModel = namedtuple("WorkerModel",
                ('A', 'C', 'G', 'R', 'xhat_0', 'Σ_0'))

def create_worker(α=.8, β=.2, c=.2,
                  R=.5, g=1.0, hhat_0=4, uhat_0=4,
                  σ_h=4, σ_u=4):

    A = np.array([[α, β],
                  [0, 1]])
    C = np.array([[c],
                  [0]])
    G = np.array([g, 1])

    # Define initial state and covariance matrix
    xhat_0 = np.array([[hhat_0],
                       [uhat_0]])

    Σ_0 = np.array([[σ_h, 0],
                    [0, σ_u]])

    return WorkerModel(A=A, C=C, G=G, R=R, xhat_0=xhat_0, Σ_0=Σ_0)
```

Please note how the `WorkerModel` namedtuple creates all of the objects required to compute an associated state-space representation (42.2).

This is handy, because in order to simulate a history $\{y_t, h_t\}$ for a worker, we'll want to form state space system for him/her by using the `LinearStateSpace` class.

```python
# Define A, C, G, R, xhat_0, Σ_0
worker = create_worker()
A, C, G, R = worker.A, worker.C, worker.G, worker.R
xhat_0, Σ_0 = worker.xhat_0, worker.Σ_0

# Create a LinearStateSpace object
ss = LinearStateSpace(A, C, G, np.sqrt(R),
```

(continues on next page)

(continued from previous page)

```
        mu_0=xhat_0, Sigma_0=np.zeros((2,2)))

T = 100
x, y = ss.simulate(T)
y = y.flatten()

h_0, u_0 = x[0, 0], x[1, 0]
```

Next, to compute the firm's policy for setting the log wage based on the information it has about the worker, we use the Kalman filter described in this quantecon lecture *A First Look at the Kalman filter*.

In particular, we want to compute all of the objects in an "innovation representation".

42.4 An Innovations Representation

We have all the objects in hand required to form an innovations representation for the output process $\{y_t\}_{t=0}^T$ for a worker.

Let's code that up now.

$$\hat{x}_{t+1} = A\hat{x}_t + K_t a_t$$
$$y_t = G\hat{x}_t + a_t$$

where K_t is the Kalman gain matrix at time t.

We accomplish this in the following code that uses the `Kalman` class.

```
kalman = Kalman(ss, xhat_0, Σ_0)
Σ_t = np.zeros((*Σ_0.shape, T-1))
y_hat_t = np.zeros(T-1)
x_hat_t = np.zeros((2, T-1))

for t in range(1, T):
    kalman.update(y[t])
    x_hat, Σ = kalman.x_hat, kalman.Sigma
    Σ_t[:, :, t-1] = Σ
    x_hat_t[:, t-1] = x_hat.reshape(-1)
    [y_hat_t[t-1]] = worker.G @ x_hat

x_hat_t = np.concatenate((x[:, 1][:, np.newaxis],
                    x_hat_t), axis=1)
Σ_t = np.concatenate((worker.Σ_0[:, :, np.newaxis],
                    Σ_t), axis=2)
u_hat_t = x_hat_t[1, :]
```

For a draw of h_0, u_0, we plot $Ey_t = G\hat{x}_t$ where $\hat{x}_t = E[x_t|y^{t-1}]$.

We also plot $E[u_0|y^{t-1}]$, which is the firm inference about a worker's hard-wired "work ethic" u_0, conditioned on information y^{t-1} that it has about him or her coming into period t.

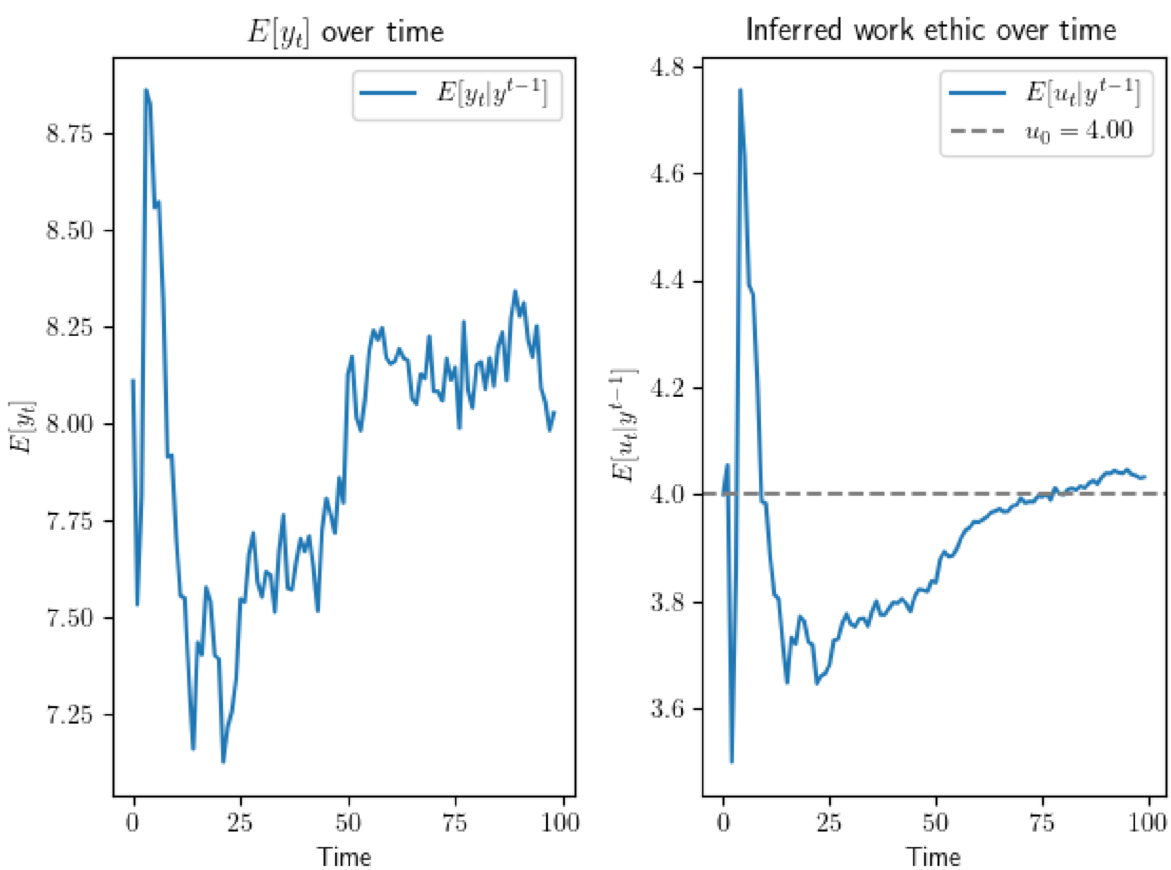
We can watch as the firm's inference $E[u_0|y^{t-1}]$ of the worker's work ethic converges toward the hidden u_0, which is not directly observed by the firm.

```
fig, ax = plt.subplots(1, 2)

ax[0].plot(y_hat_t, label=r'$E[y_t| y^{t-1}]$')
ax[0].set_xlabel('Time')
```

(continues on next page)

(continued from previous page)

```
ax[0].set_ylabel(r'$E[y_t]$')
ax[0].set_title(r'$E[y_t]$ over time')
ax[0].legend()

ax[1].plot(u_hat_t, label=r'$E[u_t|y^{t-1}]$')
ax[1].axhline(y=u_0, color='grey',
            linestyle='dashed', label=fr'$u_0={u_0:.2f}$')
ax[1].set_xlabel('Time')
ax[1].set_ylabel(r'$E[u_t|y^{t-1}]$')
ax[1].set_title('Inferred work ethic over time')
ax[1].legend()

fig.tight_layout()
plt.show()
```



42.5 Some Computational Experiments

Let's look at Σ_0 and Σ_T in order to see how much the firm learns about the hidden state during the horizon we have set.

```
print(Σ_t[:, :, 0])
```

```
[[4. 0.]
 [0. 4.]]
```

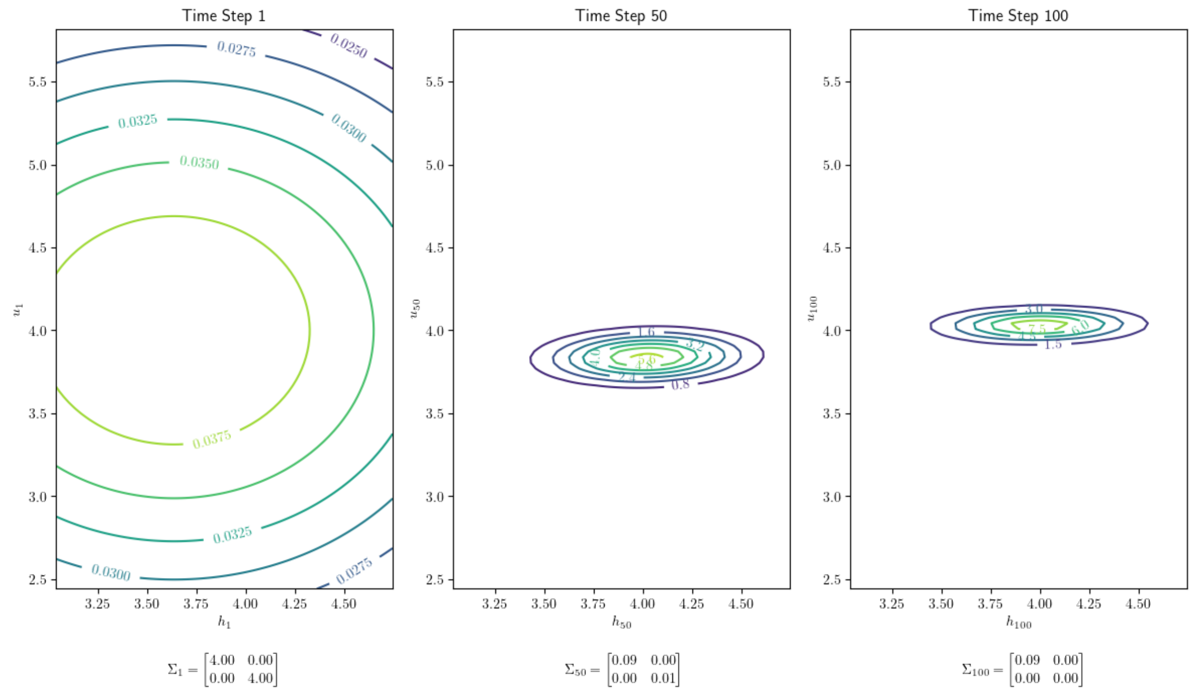
```
print(Σ_t[:, :, -1])
```

```
[[0.08805027 0.00100377]
 [0.00100377 0.00398351]]
```

Evidently, entries in the conditional covariance matrix become smaller over time.

It is enlightening to portray how conditional covariance matrices Σ_t evolve by plotting confidence ellipsoides around $E[x_t|y^{t-1}]$ at various t's.

```
# Create a grid of points for contour plotting
h_range = np.linspace(x_hat_t[0, :].min()-0.5*Σ_t[0, 0, 1],
                      x_hat_t[0, :].max()+0.5*Σ_t[0, 0, 1], 100)
u_range = np.linspace(x_hat_t[1, :].min()-0.5*Σ_t[1, 1, 1],
                      x_hat_t[1, :].max()+0.5*Σ_t[1, 1, 1], 100)
h, u = np.meshgrid(h_range, u_range)

# Create a figure with subplots for each time step
fig, axs = plt.subplots(1, 3, figsize=(12, 7))

# Iterate through each time step
for i, t in enumerate(np.linspace(0, T-1, 3, dtype=int)):
    # Create a multivariate normal distribution with x_hat and Σ at time step t
    mu = x_hat_t[:, t]
    cov = Σ_t[:, :, t]
    mvn = multivariate_normal(mean=mu, cov=cov)

    # Evaluate the multivariate normal PDF on the grid
    pdf_values = mvn.pdf(np.dstack((h, u)))

    # Create a contour plot for the PDF
    con = axs[i].contour(h, u, pdf_values, cmap='viridis')
    axs[i].clabel(con, inline=1, fontsize=10)
    axs[i].set_title(f'Time Step {t+1}')
    axs[i].set_xlabel(r'$h_{{{}}}$'.format(str(t+1)))
    axs[i].set_ylabel(r'$u_{{{}}}$'.format(str(t+1)))

    cov_latex = r'$\Sigma_{{{}}}= \begin{{bmatrix}} {:.2f} & {:.2f} \\ {:.2f} & {:.2f}
↪ \end{{bmatrix}}$'.format(
        t+1, cov[0, 0], cov[0, 1], cov[1, 0], cov[1, 1]
    )
    axs[i].text(0.33, -0.15, cov_latex, transform=axs[i].transAxes)


plt.tight_layout()
plt.show()
```



Note how the accumulation of evidence y^t affects the shape of the confidence ellipsoid as sample size t grows.

Now let's use our code to set the hidden state x_0 to a particular vector in order to watch how a firm learns starting from some x_0 we are interested in.

For example, let's say $h_0 = 0$ and $u_0 = 4$.

Here is one way to do this.

```
# For example, we might want h_0 = 0 and u_0 = 4
mu_0 = np.array([0.0, 4.0])

# Create a LinearStateSpace object with Sigma_0 as a matrix of zeros
ss_example = LinearStateSpace(A, C, G, np.sqrt(R), mu_0=mu_0,
                              # This line forces exact h_0=0 and u_0=4
                              Sigma_0=np.zeros((2, 2))
                             )

T = 100
x, y = ss_example.simulate(T)
y = y.flatten()

# Now h_0=0 and u_0=4
h_0, u_0 = x[0, 0], x[1, 0]
print('h_0 =', h_0)
print('u_0 =', u_0)
```

```
h_0 = 0.0
u_0 = 4.0
```

Another way to accomplish the same goal is to use the following code.

```
# If we want to set the initial
# h_0 = hhat_0 = 0 and u_0 = uhhat_0 = 4.0:
```

(continues on next page)

(continued from previous page)

```
worker = create_worker(hhat_0=0.0, uhat_0=4.0)

ss_example = LinearStateSpace(A, C, G, np.sqrt(R),
                              # This line takes h_0=hhat_0 and u_0=uhhat_0
                              mu_0=worker.xhat_0,
                              # This line forces exact h_0=hhat_0 and u_0=uhhat_0
                              Sigma_0=np.zeros((2, 2))
                             )

T = 100
x, y = ss_example.simulate(T)
y = y.flatten()

# Now h_0 and u_0 will be exactly hhat_0
h_0, u_0 = x[0, 0], x[1, 0]
print('h_0 =', h_0)
print('u_0 =', u_0)
```

```
h_0 = 0.0
u_0 = 4.0
```

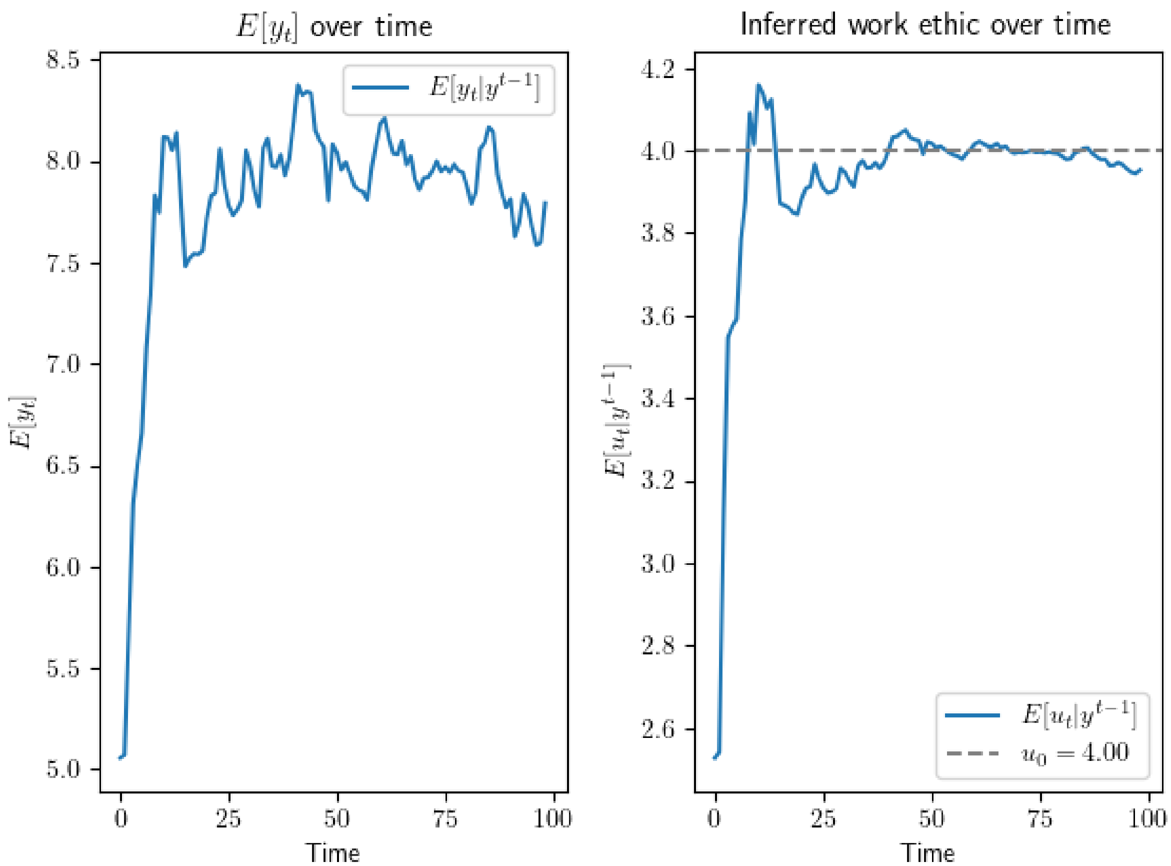
For this worker, let's generate a plot like the one above.

```
# First we compute the Kalman filter with initial xhat_0 and Σ_0
kalman = Kalman(ss, xhat_0, Σ_0)
Σ_t = []
y_hat_t = np.zeros(T-1)
u_hat_t = np.zeros(T-1)

# Then we iteratively update the Kalman filter class using
# observation y based on the linear state model above:
for t in range(1, T):
    kalman.update(y[t])
    x_hat, Σ = kalman.x_hat, kalman.Sigma
    Σ_t.append(Σ)
    [y_hat_t[t-1]] = worker.G @ x_hat
    [u_hat_t[t-1]] = x_hat[1]


# Generate plots for y_hat_t and u_hat_t
fig, ax = plt.subplots(1, 2)

ax[0].plot(y_hat_t, label=r'$E[y_t| y^{t-1}]$')
ax[0].set_xlabel('Time')
ax[0].set_ylabel(r'$E[y_t]$')
ax[0].set_title(r'$E[y_t]$ over time')
ax[0].legend()

ax[1].plot(u_hat_t, label=r'$E[u_t|y^{t-1}]$')
ax[1].axhline(y=u_0, color='grey',
            linestyle='dashed', label=fr'$u_0={u_0:.2f}$')
ax[1].set_xlabel('Time')
ax[1].set_ylabel(r'$E[u_t|y^{t-1}]$')
ax[1].set_title('Inferred work ethic over time')
ax[1].legend()
```

(continues on next page)

(continued from previous page)

```
fig.tight_layout()
plt.show()
```



More generally, we can change some or all of the parameters defining a worker in our `create_worker` namedtuple.

Here is an example.

```
# We can set these parameters when creating a worker -- just like classes!
hard_working_worker =  create_worker(α=.4, β=.8,
                        hhat_0=7.0, uhat_0=100, σ_h=2.5, σ_u=3.2)

print(hard_working_worker)
```

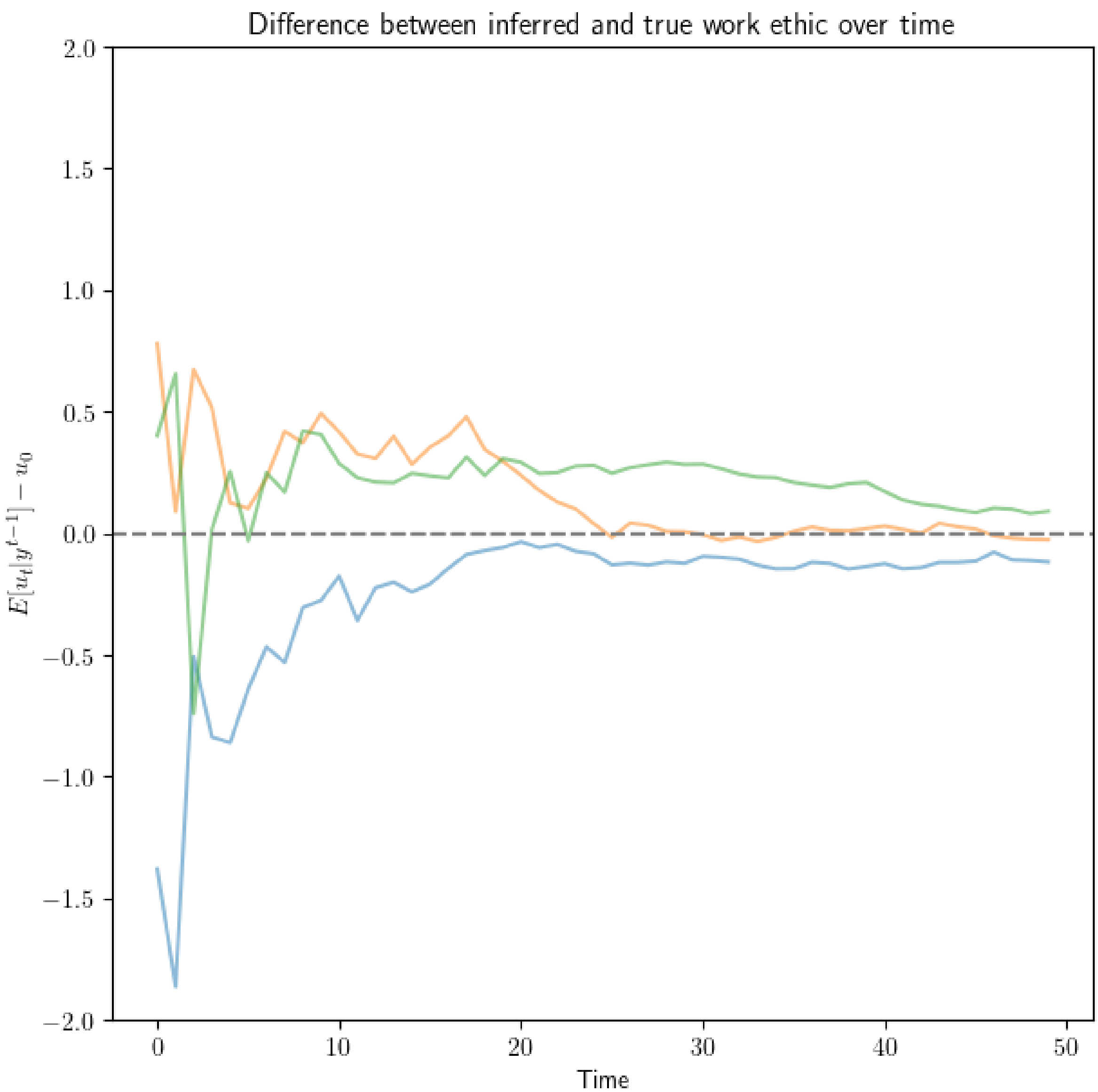
```
WorkerModel(A=array([[0.4, 0.8],
       [0. , 1. ]]), C=array([[0.2],
       [0. ]]), G=array([1., 1.]), R=0.5, xhat_0=array([[  7.],
       [100.]]), Σ_0=array([[2.5, 0. ],
       [0. , 3.2]]))
```

We can also simulate the system for $T = 50$ periods for different workers.

The difference between the inferred work ethics and true work ethics converges to 0 over time.

This shows that the filter is gradually teaching the worker and firm about the worker's effort.

```
num_workers = 3
T = 50
fig, ax = plt.subplots(figsize=(7, 7))

for i in range(num_workers):
    worker = create_worker(uhat_0=4+2*i)
    simulate_workers(worker, T, ax)
ax.set_ylim(ymin=-2, ymax=2)
plt.show()
```

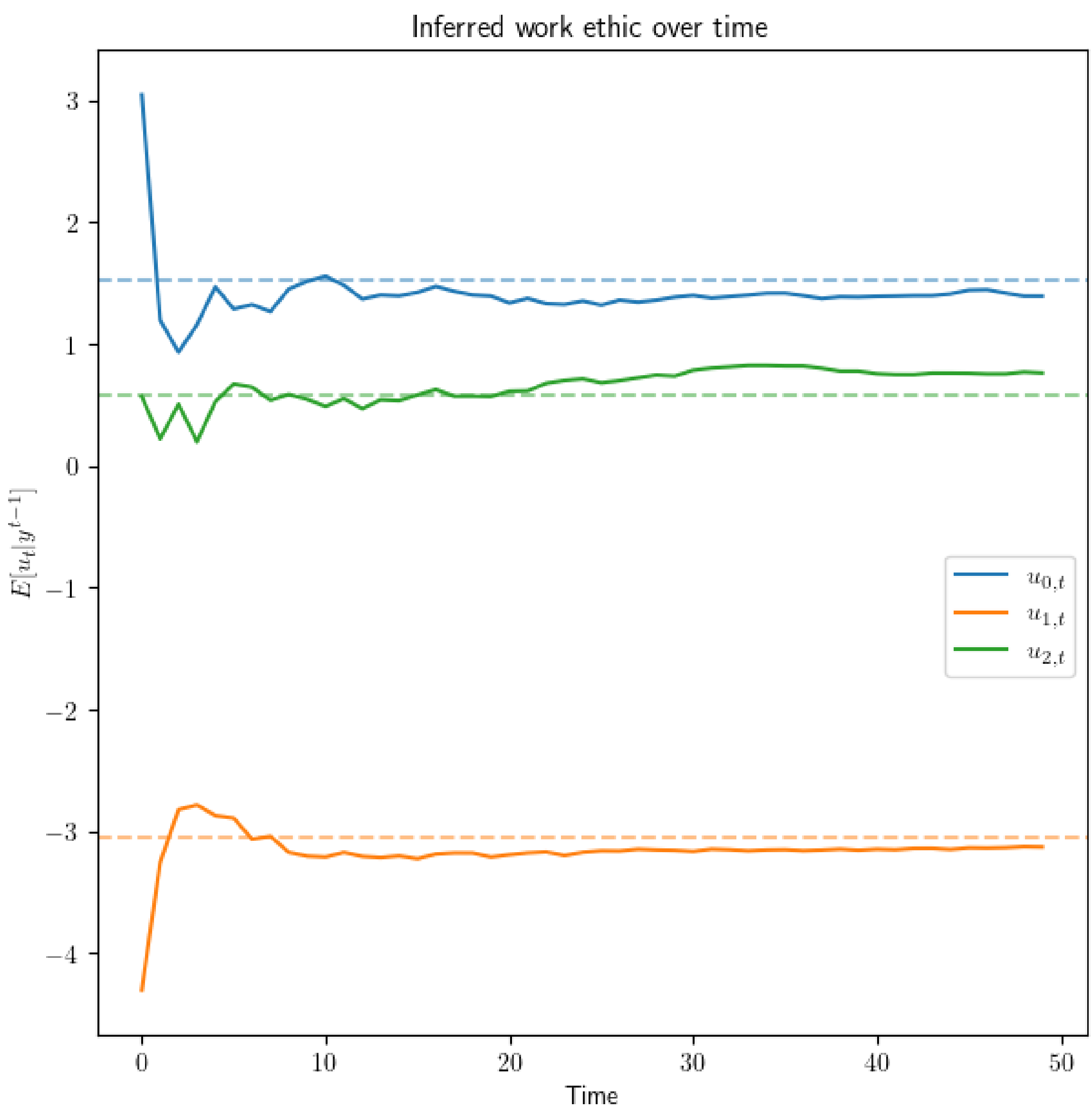



```
# We can also generate plots of u_t:

T = 50
fig, ax = plt.subplots(figsize=(7, 7))

uhat_0s = [2, -2, 1]
```

(continues on next page)

(continued from previous page)

```
αs = [0.2, 0.3, 0.5]
βs = [0.1, 0.9, 0.3]

for i, (uhat_0, α, β) in enumerate(zip(uhat_0s, αs, βs)):
    worker = create_worker(uhat_0=uhat_0, α=α, β=β)
    simulate_workers(worker, T, ax,
                     # By setting diff=False, it will give u_t
                     diff=False, name=r'$u_{{{}, t}}$'.format(i))

ax.legend(bbox_to_anchor=(1, 0.5))
plt.show()
```

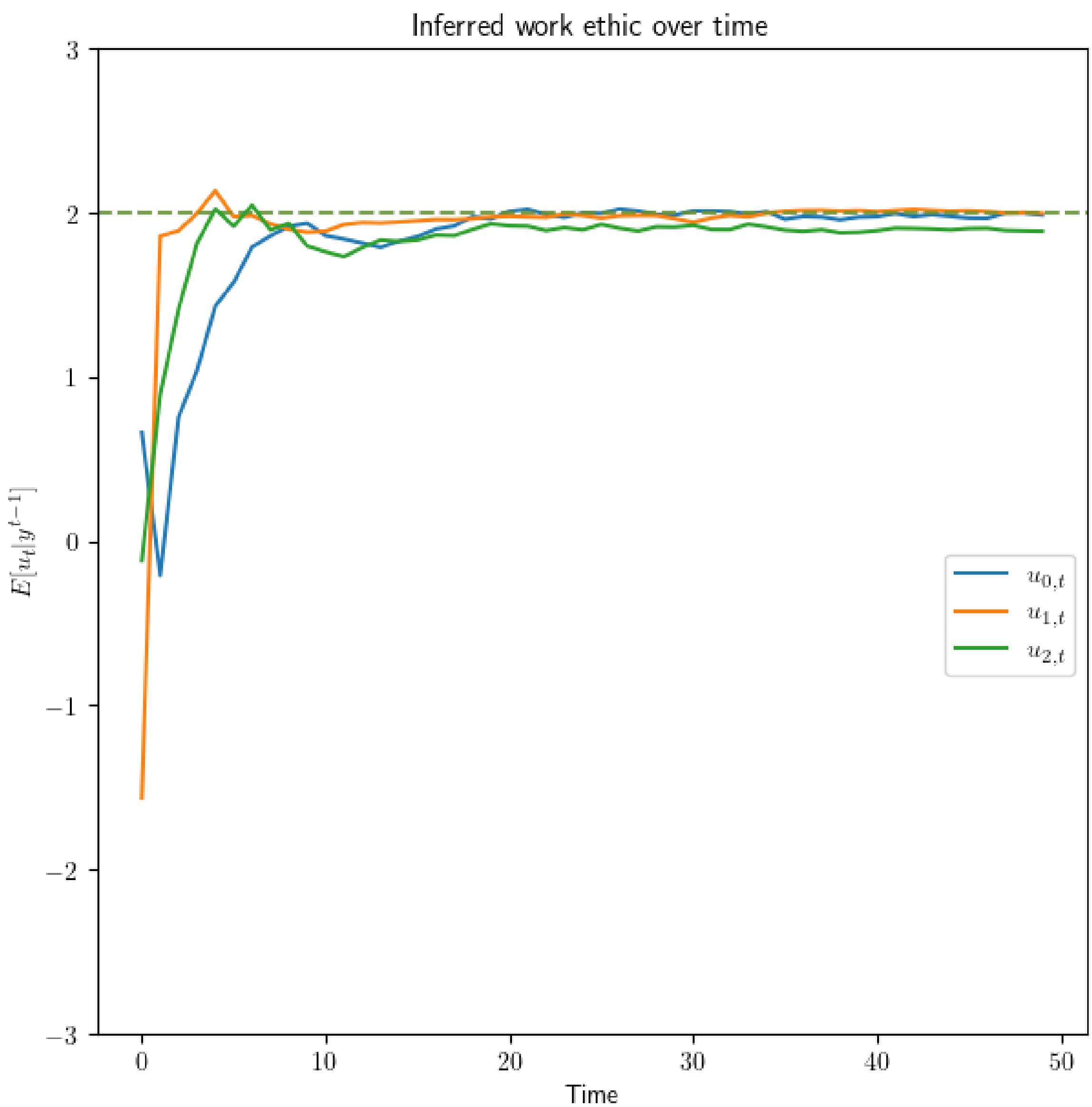



```
# We can also use exact u_0=1 and h_0=2 for all workers

T = 50
```

(continues on next page)

(continued from previous page)

```
fig, ax = plt.subplots(figsize=(7, 7))

# These two lines set u_0=1 and h_0=2 for all workers
mu_0 = np.array([[1],
                 [2]])
Sigma_0 = np.zeros((2,2))

uhat_0s = [2, -2, 1]
αs = [0.2, 0.3, 0.5]
βs = [0.1, 0.9, 0.3]

for i, (uhat_0, α, β) in enumerate(zip(uhat_0s, αs, βs)):
    worker = create_worker(uhat_0=uhat_0, α=α, β=β)
    simulate_workers(worker, T, ax, mu_0=mu_0, Sigma_0=Sigma_0,
                     diff=False, name=r'$u_{{{}, t}}$'.format(i))

# This controls the boundary of plots
ax.set_ylim(ymin=-3, ymax=3)
ax.legend(bbox_to_anchor=(1, 0.5))
plt.show()
```

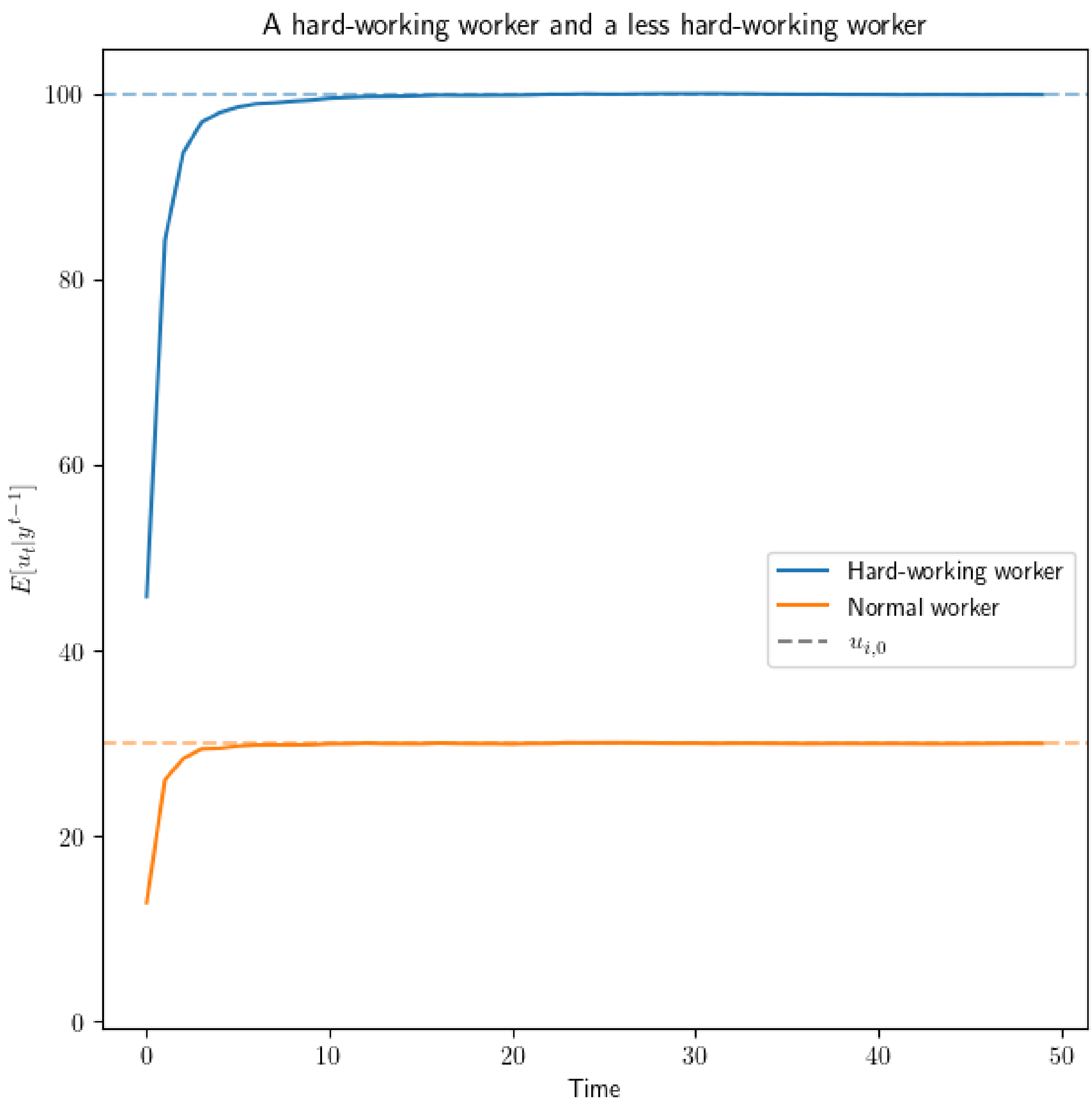



```
# We can generate a plot for only one of the workers:

T = 50
fig, ax = plt.subplots(figsize=(7, 7))

mu_0_1 = np.array([[1],
                   [100]])
mu_0_2 = np.array([[1],
                   [30]])
Sigma_0 = np.zeros((2,2))

uhat_0s = 100
αs = 0.5
βs = 0.3

worker = create_worker(uhat_0=uhat_0, α=α, β=β)
```

(continues on next page)

(continued from previous page)

```
simulate_workers(worker, T, ax, mu_0=mu_0_1, Sigma_0=Sigma_0,
                 diff=False, name=r'Hard-working worker')
simulate_workers(worker, T, ax, mu_0=mu_0_2, Sigma_0=Sigma_0,
                 diff=False,
                 title='A hard-working worker and a less hard-working worker',
                 name=r'Normal worker')
ax.axhline(y=u_0, xmin=0, xmax=0, color='grey',
           linestyle='dashed', label=r'$u_{i, 0}$')
ax.legend(bbox_to_anchor=(1, 0.5))
plt.show()
```



42.6 Future Extensions

We can do lots of enlightening experiments by creating new types of workers and letting the firm learn about their hidden (to the firm) states by observing just their output histories.

CHAPTER

FORTYTHREE

TWO MODELS OF MEASUREMENTS AND THE INVESTMENT ACCELERATOR

Contents

43.1 Overview

"Rational expectations econometrics" aims to interpret economic time series in terms of objects that are meaningful to economists, namely, parameters describing preferences, technologies, information sets, endowments, and equilibrium concepts.

When fully worked out, rational expectations models typically deliver a well-defined mapping from these economically interpretable parameters to the moments of the time series determined by the model.

If accurate observations on these time series are available, one can use that mapping to implement parameter estimation methods based either on the likelihood function or on the method of moments.

Note

This is why econometrics estimation is often called an "inverse" problem, while simulating a model for given parameter values is called a "direct problem". The direct problem refers to the mapping we have just described, while the inverse problem involves somehow applying an "inverse" of that mapping to a data set that is treated as if it were one draw from the joint probability distribution described by the mapping.

However, if only error-ridden data exist for the variables of interest, then more steps are needed to extract parameter estimates.

In effect, we require a model of the data reporting agency, one that is workable enough that we can determine the mapping induced jointly by the dynamic economic model and the measurement process to the probability law for the measured data.

The model chosen for the data collection agency is an aspect of an econometric specification that can make big differences in inferences about the economic structure.

Sargent [1989] describes two alternative models of data generation in a *permanent income* economy in which the investment accelerator, the mechanism studied in these two quantecon lectures – *Samuelson Multiplier-Accelerator* and *The Acceleration Principle and the Nature of Business Cycles* – shapes business cycle fluctuations.

- In Model 1, the data collecting agency simply reports the error-ridden data that it collects.
- In Model 2, the data collection agents first collects error-ridden data that satisfy a classical errors-in-variables model, then filters the data, and reports the filtered objects.

Although the two models have the same "deep parameters," they produce quite different sets of restrictions on the data.

In this lecture we follow Sargent [1989] and study how these alternative measurement schemes affect empirical implications.

We start with imports and helper functions to be used throughout this lecture to generate LaTeX output

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy import linalg
from IPython.display import Latex

np.set_printoptions(precision=3, suppress=True)

def df_to_latex_matrix(df, label=''):
    """Convert DataFrame to LaTeX matrix."""
    lines = [r'\begin{bmatrix}']

    for idx, row in df.iterrows():
        row_str = ' & '.join(
          [f'{v:.4f}' if isinstance(v, (int, float))
           else str(v) for v in row]) + r' \\'
        lines.append(row_str)

    lines.append(r'\end{bmatrix}')

    if label:
        return '$' + label + ' = ' + '\n'.join(lines) + '$'
    else:
        return '$' + '\n'.join(lines) + '$'

def df_to_latex_array(df):
    """Convert DataFrame to LaTeX array."""
    n_rows, n_cols = df.shape

    # Build column format (centered columns)
    col_format = 'c' * (n_cols + 1)  # +1 for index

    # Start array
    lines = [r'\begin{array}{' + col_format + '}']
```

(continues on next page)

(continued from previous page)

```
    # Header row
    header = ' & '.join([''] + [str(c) for c in df.columns]) + r' \\'
    lines.append(header)
    lines.append(r'\hline')

    # Data rows
    for idx, row in df.iterrows():
        row_str = str(idx) + ' & ' + ' & '.join(
          [f'{v:.3f}' if isinstance(v, (int, float)) else str(v)
          for v in row]) + r' \\'
        lines.append(row_str)

    lines.append(r'\end{array}')

    return '$' + '\n'.join(lines) + '$'
```

43.2 The economic model

The data are generated by a linear-quadratic version of a stochastic optimal growth model that is an instance of models described in this quantecon lecture: *The Permanent Income Model*.

A social planner chooses a stochastic process for $\{c_t, k_{t+1}\}_{t=0}^\infty$ that maximizes

$$E\sum_{t=0}^{\infty}\beta^t\left(u_0 + u_1 c_t - \frac{u_2}{2}c_t^2\right) \tag{43.1}$$

subject to the restrictions imposed by the technology

$$c_t + k_{t+1} = fk_t + \theta_t, \qquad \beta f^2 > 1. \tag{43.2}$$

Here c_t is consumption, k_t is the capital stock, f is the gross rate of return on capital, and θ_t is an endowment or technology shock following

$$a(L)\,\theta_t = \varepsilon_t, \tag{43.3}$$

where L is the backward shift (or 'lag') operator and $a(z) = 1 - a_1 z - a_2 z^2 - \cdots - a_r z^r$ having all its zeroes outside the unit circle.

43.2.1 Optimal decision rule

The optimal decision rule for c_t is

$$c_t = \frac{-\alpha}{f-1} + \left(1 - \frac{1}{\beta f^2}\right)\frac{L - f^{-1}a(f^{-1})^{-1}a(L)}{L - f^{-1}}\theta_t + fk_t, \qquad k_{t+1} = fk_t + \theta_t - c_t, \tag{43.4}$$

where $\alpha = u_1[1 - (\beta f)^{-1}]/u_2$.

Equations (43.3) and (43.4) exhibit the cross-equation restrictions characteristic of rational expectations models.

43.2.2 Net income and the accelerator

Define net output or national income as

$$y_{nt} = (f-1)k_t + \theta_t. \tag{43.5}$$

Note that (43.2) and (43.5) imply $(k_{t+1} - k_t) + c_t = y_{nt}$.

To obtain both a version of Friedman [1956]'s geometric distributed lag consumption function and a distributed lag accelerator, we impose two assumptions:

1. $a(L) = 1$, so that θ_t is white noise.
2. $\beta f = 1$, so the rate of return on capital equals the rate of time preference.

Assumption 1 is crucial for the strict form of the accelerator.

Relaxing it to allow serially correlated θ_t preserves an accelerator in a broad sense but loses the sharp geometric-lag form of (43.8).

Adding a second shock breaks the one-index structure entirely and can generate nontrivial Granger causality even without measurement error.

Assumption 2 is less important, affecting only various constants.

Under both assumptions, (43.4) simplifies to

$$c_t = (1 - f^{-1})\, \theta_t + (f-1)\, k_t. \tag{43.6}$$

When (43.6), (43.5), and (43.2) are combined, the optimal plan satisfies

$$c_t = \left(\frac{1-\beta}{1-\beta L}\right) y_{nt}, \tag{43.7}$$

$$k_{t+1} - k_t = f^{-1} \left(\frac{1-L}{1-\beta L}\right) y_{nt}, \tag{43.8}$$

$$y_{nt} = \theta_t + (1-\beta)(\theta_{t-1} + \theta_{t-2} + \cdots). \tag{43.9}$$

Equation (43.7) is Friedman's consumption model: consumption is a geometric distributed lag of income, with the decay coefficient β equal to the discount factor.

Equation (43.8) is the distributed lag accelerator: investment is a geometric distributed lag of the first difference of income.

This is the same mechanism that Chow [1968] documented empirically (see *The Acceleration Principle and the Nature of Business Cycles*).

Equation (43.9) states that the first difference of disposable income is a first-order moving average process with innovation equal to the innovation of the endowment shock θ_t.

As Muth [1960] showed, such a process is optimally forecast via a geometric distributed lag or "adaptive expectations" scheme.

43.2.3 The accelerator puzzle

When all variables are measured accurately and are driven by the single shock θ_t, the spectral density matrix of $(c_t,\ k_{t+1} - k_t,\ y_{nt})$ has rank one at all frequencies.

Each variable is an invertible one-sided distributed lag of the same white noise, so no variable Granger-causes any other.

Empirically, however, measures of output Granger-cause investment but not vice versa.

Sargent [1989] shows that measurement error can resolve this puzzle.

To illustrate, suppose first that output y_{nt} is measured perfectly while consumption and capital are each polluted by serially correlated measurement errors v_{ct} and v_{kt} orthogonal to θ_t.

Let $\bar{c}_t$ and $\bar{k}_{t+1} - \bar{k}_t$ denote the measured series. Then

$$\bar{c}_t = \left(\frac{1-\beta}{1-\beta L}\right) y_{nt} + v_{ct}, \tag{43.10}$$

$$\bar{k}_{t+1} - \bar{k}_t = \beta \left(\frac{1-L}{1-\beta L}\right) y_{nt} + (v_{k,t+1} - v_{kt}), \tag{43.11}$$

$$y_{nt} = \theta_t + (1-\beta)(\theta_{t-1} + \theta_{t-2} + \cdots). \tag{43.12}$$

In this case income Granger-causes consumption and investment but is not Granger-caused by them.

When each measured series is corrupted by measurement error, every measured variable will in general Granger-cause every other.

The strength of this Granger causality, as measured by decompositions of j-step-ahead prediction error variances, depends on the relative variances of the measurement errors.

In this case, each observed series mixes the common signal θ_t with idiosyncratic measurement noise.

A series with lower measurement error variance tracks θ_t more closely, so its innovations contain more information about future values of the other series.

Accordingly, in a forecast-error-variance decomposition, shocks to better-measured series account for a larger share of other variables' j-step-ahead prediction errors.

In a one-common-index model like this one (θ_t is the common index), better-measured variables extend more Granger causality to less well measured series than vice versa.

This asymmetry drives the numerical results we observe soon.

43.2.4 State-space formulation

Let's map the economic model and the measurement process into a linear state-space framework.

Set $f = 1.05$ and $\theta_t \sim \mathcal{N}(0, 1)$.

Define the state and observation vectors

$$x_t = \begin{bmatrix} k_t \\ \theta_t \end{bmatrix}, \qquad z_t = \begin{bmatrix} y_{nt} \\ c_t \\ \Delta k_t \end{bmatrix},$$

so that the error-free data are described by the state-space system

$$\begin{aligned} x_{t+1} &= A x_t + \varepsilon_t, \\ z_t &= C x_t. \end{aligned} \tag{43.13}$$

where $\varepsilon_t = \begin{bmatrix} 0 \\ \theta_t \end{bmatrix}$ has covariance $E\varepsilon_t\varepsilon_t^\top = Q$ and the matrices are

$$A = \begin{bmatrix} 1 & f^{-1} \\ 0 & 0 \end{bmatrix}, \qquad C = \begin{bmatrix} f-1 & 1 \\ f-1 & 1-f^{-1} \\ 0 & f^{-1} \end{bmatrix}, \qquad Q = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}.$$

Q is singular because there is only one source of randomness θ_t; the capital stock k_t evolves deterministically given θ_t.

```
# Baseline structural matrices for the true economy
f = 1.05
β = 1 / f

A = np.array([
    [1.0, 1.0 / f],
    [0.0, 0.0]
])

C = np.array([
    [f - 1.0, 1.0],
    [f - 1.0, 1.0 - 1.0 / f],
    [0.0, 1.0 / f]
])

Q = np.array([
    [0.0, 0.0],
    [0.0, 1.0]
])
```

43.2.5 True impulse responses

Before introducing measurement error, we compute the impulse response of the error-free variables to a unit shock $\theta_0 = 1$.

This benchmark clarifies what changes when we later switch from error-free variables to variables reported by the statistical agency.

The response shows the investment accelerator clearly: the full impact on net income y_n occurs at lag 0, while consumption adjusts by only $1 - f^{-1} \approx 0.048$ and investment absorbs the remainder.

From lag 1 onward the economy is in its new steady state

```
def table2_irf(A, C, n_lags=6):
    x = np.array([0.0, 1.0])  # k_0 = 0, theta_0 = 1
    rows = []
    for j in range(n_lags):
        y_n, c, d_k = C @ x
        rows.append([y_n, c, d_k])
        x = A @ x
    return pd.DataFrame(rows, columns=[r'y_n', r'c', r'\Delta k'],
                        index=pd.Index(range(n_lags), name='lag'))

table2 = table2_irf(A, C, n_lags=6)
display(Latex(df_to_latex_array(table2)))
```

	y_n	c	Δk
0	1.000	0.048	0.952
1	0.048	0.048	0.000
2	0.048	0.048	0.000
3	0.048	0.048	0.000
4	0.048	0.048	0.000
5	0.048	0.048	0.000

43.3 Measurement errors

Let's add the measurement layer that generates reported data.

The econometrician does not observe z_t directly but instead sees $\bar{z}_t = z_t + v_t$, where v_t is a vector of measurement errors.

Measurement errors follow an AR(1) process

$$v_{t+1} = Dv_t + \eta_t, \tag{43.14}$$

where η_t is a vector white noise with $E\eta_t\eta_t^\top = \Sigma_\eta$ and $E\varepsilon_t v_s^\top = 0$ for all t, s.

The parameters are

$$D = \text{diag}(0.6, 0.7, 0.3), \qquad \sigma_\eta = (0.05, 0.035, 0.65),$$

so the unconditional covariance of v_t is

$$R = \text{diag}\left(\frac{\sigma_{\eta,i}^2}{1-\rho_i^2}\right).$$

The innovation variances are smallest for consumption ($\sigma_\eta = 0.035$), next for income ($\sigma_\eta = 0.05$), and largest for investment ($\sigma_\eta = 0.65$).

As in Sargent [1989] and our discussion above, what matters for Granger-causality asymmetries is the overall measurement quality in the full system: output is relatively well measured while investment is relatively poorly measured.

```
ρ = np.array([0.6, 0.7, 0.3])
D = np.diag(ρ)

# Innovation std. devs of η_t
σ_η = np.array([0.05, 0.035, 0.65])
Σ_η = np.diag(σ_η**2)

# Unconditional covariance of measurement errors v_t
R = np.diag((σ_η / np.sqrt(1.0 - ρ**2))**2)

print(f"f = {f},  β = 1/f = {β:.6f}")
print()
display(Latex(df_to_latex_matrix(pd.DataFrame(A), 'A')))
display(Latex(df_to_latex_matrix(pd.DataFrame(C), 'C')))
display(Latex(df_to_latex_matrix(pd.DataFrame(D), 'D')))
```

```
f = 1.05,  β = 1/f = 0.952381
```

$$A = \begin{bmatrix} 1.0000 & 0.9524 \\ 0.0000 & 0.0000 \end{bmatrix}$$

$$C = \begin{bmatrix} 0.0500 & 1.0000 \\ 0.0500 & 0.0476 \\ 0.0000 & 0.9524 \end{bmatrix}$$

$$D = \begin{bmatrix} 0.6000 & 0.0000 & 0.0000 \\ 0.0000 & 0.7000 & 0.0000 \\ 0.0000 & 0.0000 & 0.3000 \end{bmatrix}$$

We will analyze the two reporting schemes separately, but first we need a solver for the steady-state Kalman gain and error covariances.

The function below iterates on the Riccati equation until convergence, returning the Kalman gain K, the state covariance S, and the innovation covariance V

```python
def steady_state_kalman(A, C_obs, Q, R, W=None, tol=1e-13, max_iter=200_000):
    """
    Solve steady-state Kalman equations for
        x_{t+1} = A x_t + w_{t+1}
        y_t     = C_obs x_t + v_t
    with cov(w)=Q, cov(v)=R, cov(w,v)=W.
    """
    n = A.shape[0]
    m = C_obs.shape[0]
    if W is None:
        W = np.zeros((n, m))

    S = Q.copy()
    for _ in range(max_iter):
        V = C_obs @ S @ C_obs.T + R
        K = (A @ S @ C_obs.T + W) @ np.linalg.inv(V)
        S_new = Q + A @ S @ A.T - K @ V @ K.T

        if np.max(np.abs(S_new - S)) < tol:
            S = S_new
            break
        S = S_new

    V = C_obs @ S @ C_obs.T + R
    K = (A @ S @ C_obs.T + W) @ np.linalg.inv(V)
    return K, S, V
```

With structural matrices and tools we need in place, we now follow Sargent [1989]'s two reporting schemes in sequence.

43.4 A classical model of measurements initially collected by an agency

A data collecting agency observes a noise-corrupted version of z_t, namely

$$\bar{z}_t = C x_t + v_t. \tag{43.15}$$

We refer to this as *Model 1*: the agency collects noisy data and reports them without filtering.

To represent the second moments of the $\bar{z}_t$ process, it is convenient to obtain its population vector autoregression.

The error vector in the vector autoregression is the innovation to $\bar{z}_t$ and can be taken to be the white noise in a Wold moving average representation, which can be obtained by "inverting" the autoregressive representation.

The population vector autoregression, and how it depends on the parameters of the state-space system and the measurement error process, carries insights about how to interpret estimated vector autoregressions for $\bar{z}_t$.

Constructing the vector autoregression is also useful as an intermediate step in computing the likelihood of a sample of $\bar{z}_t$'s as a function of the free parameters $\{A, C, D, Q, R\}$.

The particular method that will be used to construct the vector autoregressive representation also proves useful as an intermediate step in constructing a model of an optimal reporting agency.

We use recursive (Kalman filtering) methods to obtain the vector autoregression for $\bar{z}_t$.

43.4.1 Quasi-differencing

Because the measurement errors v_t are serially correlated, the standard Kalman filter with white-noise measurement error cannot be applied directly to $\bar{z}_t = Cx_t + v_t$.

An alternative is to augment the state vector with the measurement-error AR components (see Appendix B of Sargent [1989]).

Here we take the quasi-differencing route described in Sargent [1989], which reduces the system to one with serially uncorrelated observation noise.

Define

$$\tilde{z}_t = \bar{z}_{t+1} - D\bar{z}_t, \qquad \bar{\nu}_t = C\varepsilon_t + \eta_t, \qquad \bar{C} = CA - DC. \tag{43.16}$$

Then the state-space system (43.13), the measurement error process (43.14), and the observation equation (43.15) imply the state-space system

$$\begin{aligned} x_{t+1} &= Ax_t + \varepsilon_t, \\ \tilde{z}_t &= \bar{C}\,x_t + \bar{\nu}_t, \end{aligned} \tag{43.17}$$

where $(\varepsilon_t, \bar{\nu}_t)$ is a white noise process with

$$E\left[\varepsilon_t\right]\left[\varepsilon_t^\top \quad \bar{\nu}_t^\top\right] = \begin{bmatrix} Q & W_1 \\ W_1^\top & R_1 \end{bmatrix}, \qquad R_1 = CQC^\top + R, \quad W_1 = QC^\top. \tag{43.18}$$

System (43.17) with covariances (43.18) is characterized by the five matrices $[A, \bar{C}, Q, R_1, W_1]$.

43.4.2 Innovations representation

Associated with (43.17) and (43.18) is the **innovations representation** for $\tilde{z}_t$,

$$\begin{aligned} \hat{x}_{t+1} &= A\hat{x}_t + K_1 u_t, \\ \tilde{z}_t &= \bar{C}\hat{x}_t + u_t, \end{aligned} \tag{43.19}$$

where

$$\begin{aligned} \hat{x}_t &= E[x_t \mid \tilde{z}_{t-1}, \tilde{z}_{t-2}, \dots, \hat{x}_0] = E[x_t \mid \bar{z}_t, \bar{z}_{t-1}, \dots], \\ u_t &= \tilde{z}_t - E[\tilde{z}_t \mid \tilde{z}_{t-1}, \tilde{z}_{t-2}, \dots] = \bar{z}_{t+1} - E[\bar{z}_{t+1} \mid \bar{z}_t, \bar{z}_{t-1}, \dots], \end{aligned} \tag{43.20}$$

$[K_1, S_1]$ are computed from the steady-state Kalman filter applied to $[A, \bar{C}, Q, R_1, W_1]$, and

$$S_1 = E[(x_t - \hat{x}_t)(x_t - \hat{x}_t)^\top]. \tag{43.21}$$

From (43.20), u_t is the innovation process for the $\bar{z}_t$ process.

43.4.3 Wold representation

System (43.19) and definition (43.16) can be used to obtain a Wold vector moving average representation for the $\bar{z}_t$ process:

$$\bar{z}_{t+1} = (I - DL)^{-1}[\bar{C}(I - AL)^{-1}K_1 L + I]u_t, \tag{43.22}$$

where L is the lag operator.

From (43.17) and (43.19) the innovation covariance is

$$V_1 = E\, u_t u_t^\top = \bar{C}\, S_1\, \bar{C}^\top + R_1. \tag{43.23}$$

Below we compute K_1, S_1, and V_1 numerically

```
C_bar = C @ A - D @ C
R1 = C @ Q @ C.T + R
W1 = Q @ C.T

K1, S1, V1 = steady_state_kalman(A, C_bar, Q, R1, W1)
```

43.4.4 Computing coefficients in a Wold moving average representation

To compute the moving average coefficients in (43.22) numerically, define the augmented state

$$r_t = \begin{bmatrix} \hat{x}_{t-1} \\ \bar{z}_{t-1} \end{bmatrix},$$

with dynamics

$$r_{t+1} = F_1 r_t + G_1 u_t, \qquad \bar{z}_t = H_1 r_t + u_t,$$

where

$$F_1 = \begin{bmatrix} A & 0 \\ \bar{C} & D \end{bmatrix}, \quad G_1 = \begin{bmatrix} K_1 \\ I \end{bmatrix}, \quad H_1 = [\bar{C} \ \ D].$$

The moving average coefficients are then $\psi_0 = I$ and $\psi_j = H_1 F_1^{j-1} G_1$ for $j \geq 1$.

```
F1 = np.block([
    [A, np.zeros((2, 3))],
    [C_bar, D]
])
G1 = np.vstack([K1, np.eye(3)])
H1 = np.hstack([C_bar, D])


def measured_wold_coeffs(F, G, H, n_terms=25):
    psi = [np.eye(3)]
    Fpow = np.eye(F.shape[0])
    for _ in range(1, n_terms):
        psi.append(H @ Fpow @ G)
        Fpow = Fpow @ F
    return psi
```

(continues on next page)

(continued from previous page)

```
def fev_contributions(psi, V, n_horizons=20):
    """
    Returns contrib[var, shock, h-1] = contribution at horizon h.
    """
    P = linalg.cholesky(V, lower=True)
    out = np.zeros((3, 3, n_horizons))
    for h in range(1, n_horizons + 1):
        acc = np.zeros((3, 3))
        for j in range(h):
            T = psi[j] @ P
            acc += T**2
        out[:, :, h - 1] = acc
    return out


psi1 = measured_wold_coeffs(F1, G1, H1, n_terms=40)
resp1 = np.array(
      [psi1[j] @ linalg.cholesky(V1, lower=True) for j in range(14)])
decomp1 = fev_contributions(psi1, V1, n_horizons=20)
```

43.4.5 Gaussian likelihood

The Gaussian log-likelihood function for a sample $\{\bar{z}_t,\ t = 0, \ldots, T\}$, conditioned on an initial state estimate $\hat{x}_0$, can be represented as

$$\mathcal{L}^* = -T \ln 2\pi - \frac{1}{2} T \ln |V_1| - \frac{1}{2} \sum_{t=0}^{T-1} u_t^\top V_1^{-1} u_t, \tag{43.24}$$

where u_t is a function of $\{\bar{z}_t\}$ defined by (43.25) below.

To use (43.19) to compute $\{u_t\}$, it is useful to represent it as

$$\begin{aligned} \hat{x}_{t+1} &= (A - K_1 \bar{C})\, \hat{x}_t + K_1 \tilde{z}_t, \\ u_t &= -\bar{C}\, \hat{x}_t + \tilde{z}_t, \end{aligned} \tag{43.25}$$

where $\tilde{z}_t = \bar{z}_{t+1} - D\bar{z}_t$ is the quasi-differenced observation.

Given $\hat{x}_0$, equation (43.25) can be used recursively to compute a $\{u_t\}$ process.

Equations (43.24) and (43.25) give the likelihood function of a sample of error-corrupted data $\{\bar{z}_t\}$.

43.4.6 Forecast-error-variance decomposition

To measure the relative importance of each innovation, we decompose the j-step-ahead forecast-error variance of each measured variable.

Write $\bar{z}_{t+j} - E_t \bar{z}_{t+j} = \sum_{i=0}^{j-1} \psi_i u_{t+j-i}$.

Let P be the lower-triangular Cholesky factor of V_1 so that the orthogonalized innovations are $e_t = P^{-1} u_t$.

Then the contribution of orthogonalized innovation k to the j-step-ahead variance of variable m is $\sum_{i=0}^{j-1} (\psi_i P)^2_{mk}$.

The table below shows the cumulative contribution of each orthogonalized innovation to the forecast-error variance of y_n, c, and Δk at horizons 1 through 20.

Each panel fixes one orthogonalized innovation and reports its cumulative contribution to each variable's forecast-error variance.

Rows are forecast horizons and columns are forecasted variables.

```
horizons = np.arange(1, 21)
labels = [r'y_n', r'c', r'\Delta k']

def fev_table(decomp, shock_idx, horizons):
    return pd.DataFrame(
        np.round(decomp[:, shock_idx, :].T, 4),
        columns=labels,
        index=pd.Index(horizons, name='j')
    )
```

```
shock_titles = [r'\text{A. Innovation in } y_n',
                r'\text{B. Innovation in } c',
                r'\text{C. Innovation in } \Delta k']

parts = []
for i, title in enumerate(shock_titles):
    arr = df_to_latex_array(fev_table(decomp1, i, horizons)).strip('$')
    parts.append(
      r'\begin{array}{c} ' + title + r' \\ ' + arr + r' \end{array}')

display(Latex('$' + r' \quad '.join(parts) + '$'))
```

A. Innovation in y_n

	y_n	c	Δk
1	1.006	0.002	0.904
2	1.008	0.004	0.904
3	1.011	0.007	0.904
4	1.013	0.009	0.904
5	1.016	0.011	0.904
6	1.018	0.013	0.904
7	1.020	0.016	0.904
8	1.022	0.018	0.904
9	1.025	0.020	0.904
10	1.027	0.023	0.904
11	1.029	0.025	0.904
12	1.031	0.027	0.904
13	1.034	0.029	0.904
14	1.036	0.032	0.904
15	1.038	0.034	0.904
16	1.040	0.036	0.904
17	1.043	0.038	0.904
18	1.045	0.041	0.904
19	1.047	0.043	0.904
20	1.050	0.045	0.904

B. Innovation in c

	y_n	c	Δk
1	0.000	0.003	0.000
2	0.000	0.004	0.000
3	0.000	0.004	0.000
4	0.000	0.005	0.000
5	0.000	0.005	0.000
6	0.000	0.005	0.000
7	0.000	0.005	0.000
8	0.000	0.005	0.000
9	0.000	0.005	0.000
10	0.000	0.005	0.000
11	0.000	0.005	0.000
12	0.000	0.005	0.000
13	0.000	0.005	0.000
14	0.000	0.005	0.000
15	0.000	0.005	0.000
16	0.000	0.005	0.000
17	0.000	0.005	0.000
18	0.000	0.005	0.000
19	0.000	0.005	0.000
20	0.000	0.005	0.000

C. Innovation in Δk

	y_n	c	Δk
1	0.000	0.000	0.468
2	0.000	0.000	0.510
3	0.000	0.000	0.513
4	0.000	0.000	0.514
5	0.000	0.000	0.514
6	0.000	0.000	0.514
7	0.000	0.000	0.514
8	0.000	0.000	0.514
9	0.000	0.000	0.514
10	0.000	0.000	0.514
11	0.000	0.000	0.514
12	0.000	0.000	0.514
13	0.000	0.000	0.514
14	0.000	0.000	0.514
15	0.000	0.000	0.514
16	0.000	0.000	0.514
17	0.000	0.000	0.514
18	0.000	0.000	0.514
19	0.000	0.000	0.514
20	0.000	0.000	0.514

The income innovation accounts for substantial proportions of forecast-error variance in all three variables, while the consumption and investment innovations contribute mainly to their own variances.

This is a **Granger causality** pattern: income appears to Granger-cause consumption and investment, but not vice versa.

This matches the paper's message that, in a one-common-index model, the relatively best measured series has the strongest predictive content.

Let's look at the covariance matrix of the innovations

```
print('Covariance matrix of innovations:')
df_v1 = pd.DataFrame(np.round(V1, 4), index=labels, columns=labels)
display(Latex(df_to_latex_matrix(df_v1)))
```

```
Covariance matrix of innovations:
```

$$\begin{bmatrix} 1.0057 & 0.0476 & 0.9533 \\ 0.0476 & 0.0047 & 0.0453 \\ 0.9533 & 0.0453 & 1.3718 \end{bmatrix}$$

The covariance matrix of the innovations is not diagonal, but the eigenvalues are well separated as shown below

```
print('Eigenvalues of covariance matrix:')
print(np.sort(np.linalg.eigvalsh(V1))[::-1].round(4))
```

```
Eigenvalues of covariance matrix:
[2.161 0.218 0.002]
```

The first eigenvalue is much larger than the others, consistent with the presence of a dominant common shock θ_t

43.4.7 Wold impulse responses

Impulse responses in the Wold representation are reported using orthogonalized innovations (Cholesky factorization of V_1 with ordering y_n, c, Δk).

Under this method, lag-0 responses reflect both contemporaneous covariance and the Cholesky ordering.

We first define a helper function to format the response coefficients as a LaTeX array

```
lags = np.arange(14)

def wold_response_table(resp, shock_idx, lags):
    return pd.DataFrame(
        np.round(resp[:, :, shock_idx], 4),
        columns=labels,
        index=pd.Index(lags, name='j')
    )
```

Now we report the impulse responses to each orthogonalized innovation in a single table with three panels

```
wold_titles = [r'\text{A. Response to } y_n \text{ innovation}',
               r'\text{B. Response to } c \text{ innovation}',
               r'\text{C. Response to } \Delta k \text{ innovation}']

parts = []
for i, title in enumerate(wold_titles):
    arr = df_to_latex_array(wold_response_table(resp1, i, lags)).strip('$')
    parts.append(
      r'\begin{array}{c} ' + title + r' \\ ' + arr + r' \end{array}')
```

(continues on next page)

(continued from previous page)

```
display(Latex('$' + r' \quad '.join(parts) + '$'))
```

A. Response to y_n innovation

	y_n	c	Δk
0	1.003	0.048	0.951
1	0.051	0.048	0.000
2	0.050	0.048	0.000
3	0.049	0.048	0.000
4	0.048	0.048	0.000
5	0.048	0.048	0.000
6	0.048	0.048	0.000
7	0.048	0.048	0.000
8	0.048	0.048	0.000
9	0.048	0.048	0.000
10	0.048	0.048	0.000
11	0.048	0.048	0.000
12	0.048	0.048	0.000
13	0.048	0.048	0.000

B. Response to c innovation

	y_n	c	Δk
0	0.000	0.050	0.004
1	−0.002	0.035	−0.000
2	0.000	0.026	−0.000
3	0.001	0.019	−0.000
4	0.002	0.014	−0.000
5	0.002	0.011	−0.000
6	0.002	0.008	−0.000
7	0.003	0.007	−0.000
8	0.003	0.005	−0.000
9	0.003	0.005	−0.000
10	0.003	0.004	−0.000
11	0.003	0.004	−0.000
12	0.003	0.003	−0.000
13	0.003	0.003	−0.000

C. Response to Δk innovation

	y_n	c	Δk
0	0.000	0.000	0.684
1	−0.004	0.000	0.203
2	−0.002	0.000	0.061
3	−0.001	0.000	0.018
4	−0.001	0.000	0.005
5	−0.000	0.000	0.002
6	−0.000	0.000	0.001
7	−0.000	0.000	0.000
8	0.000	0.000	0.000
9	0.000	0.000	0.000
10	0.000	0.000	0.000
11	0.000	0.000	0.000
12	0.000	0.000	0.000
13	0.000	0.000	0.000

At impact, the first orthogonalized innovation loads on all three measured variables.

At subsequent lags the income innovation generates persistent responses in all three variables because, being the best-measured series, its innovation is dominated by the true permanent shock θ_t.

The consumption and investment innovations produce responses that decay according to the AR(1) structure of their respective measurement errors ($\rho_c = 0.7$, $\rho_{\Delta k} = 0.3$), with little spillover to other variables.

43.5 A model of optimal estimates reported by an agency

Suppose that instead of reporting the error-corrupted data $\bar{z}_t$, the data collecting agency reports linear least-squares projections of the true data on a history of the error-corrupted data.

This model provides a possible way of interpreting two features of the data-reporting process.

- *seasonal adjustment*: if the components of v_t have strong seasonals, the optimal filter will assume a shape that can be interpreted partly in terms of a seasonal adjustment filter, one that is one-sided in current and past $\bar{z}_t$'s.
- *data revisions*: if z_t contains current and lagged values of some variable of interest, then the model simultaneously determines "preliminary," "revised," and "final" estimates as successive conditional expectations based on progressively longer histories of error-ridden observations.

To make this operational, we impute to the reporting agency a model of the joint process generating the true data and the measurement errors.

We assume that the reporting agency has "rational expectations": it knows the economic and measurement structure leading to (43.17)–(43.18).

To prepare its estimates, the reporting agency itself computes the Kalman filter to obtain the innovations representation (43.19).

Rather than reporting the error-corrupted data $\bar{z}_t$, the agency reports $\tilde{z}_t = G\hat{x}_t$, where G is a "selection matrix," possibly equal to C, for the data reported by the agency.

The data $G\hat{x}_t = E[Gx_t \mid \bar{z}_t, \bar{z}_{t-1}, \dots, \hat{x}_0]$.

The state-space representation for the reported data is then

$$
\begin{aligned}
\hat{x}_{t+1} &= A\hat{x}_t + K_1 u_t, \\
\tilde{z}_t &= G\hat{x}_t,
\end{aligned}
\tag{43.26}
$$

where the first line of (43.26) is from the innovations representation (43.19).

Note that u_t is the innovation to $\bar{z}_{t+1}$ and is *not* the innovation to $\tilde{z}_t$.

To obtain a Wold representation for $\tilde{z}_t$ and the likelihood function for a sample of $\tilde{z}_t$ requires that we obtain an innovations representation for (43.26).

43.5.1 Innovations representation for filtered data

To add a little generality to (43.26) we amend it to the system

$$
\begin{aligned}
\hat{x}_{t+1} &= A\hat{x}_t + K_1 u_t, \\
\tilde{z}_t &= G\hat{x}_t + \eta_t,
\end{aligned}
\tag{43.27}
$$

where η_t is a type 2 white-noise measurement error process ("typos") with presumably very small covariance matrix R_2.

The covariance matrix of the joint noise is

$$
E \begin{bmatrix} K_1 u_t \\ \eta_t \end{bmatrix} \begin{bmatrix} K_1 u_t \\ \eta_t \end{bmatrix}^\top = \begin{bmatrix} Q_2 & 0 \\ 0 & R_2 \end{bmatrix},
\tag{43.28}
$$

where $Q_2 = K_1 V_1 K_1^\top$.

If R_2 is singular, it is necessary to adjust the Kalman filtering formulas by using transformations that induce a "reduced order observer."

In practice, we approximate a zero R_2 matrix with the matrix ϵI for a small $\epsilon > 0$ to keep the Kalman filter numerically well-conditioned.

For system (43.27) and (43.28), an innovations representation is

$$
\begin{aligned}
\check{x}_{t+1} &= A\check{x}_t + K_2 a_t, \\
\tilde{z}_t &= G\check{x}_t + a_t,
\end{aligned}
\tag{43.29}
$$

where

$$
\begin{aligned}
a_t &= \tilde{z}_t - E[\tilde{z}_t \mid \tilde{z}_{t-1}, \tilde{z}_{t-2}, \dots], \\
\check{x}_t &= E[\hat{x}_t \mid \tilde{z}_{t-1}, \tilde{z}_{t-2}, \dots, \check{x}_0], \\
S_2 &= E[(\hat{x}_t - \check{x}_t)(\hat{x}_t - \check{x}_t)^\top], \\
[K_2, S_2] &= \text{kalmanfilter}(A, G, Q_2, R_2, 0).
\end{aligned}
\tag{43.30}
$$

Thus $\{a_t\}$ is the innovation process for the reported data $\tilde{z}_t$, with innovation covariance

$$
V_2 = E\, a_t a_t^\top = G\, S_2\, G^\top + R_2.
\tag{43.31}
$$

43.5.2 Wold representation

A Wold moving average representation for $\tilde{z}_t$ is found from (43.29) to be

$$\tilde{z}_t = [G(I - AL)^{-1} K_2 L + I] a_t, \tag{43.32}$$

with coefficients $\psi_0 = I$ and $\psi_j = GA^{j-1} K_2$ for $j \geq 1$.

Note that this is simpler than the Model 1 Wold representation (43.22) because there is no quasi-differencing to undo.

43.5.3 Gaussian likelihood

When a method analogous to Model 1 is used, a Gaussian log-likelihood for $\tilde{z}_t$ can be computed by first computing an $\{a_t\}$ sequence from observations on $\tilde{z}_t$ by using

$$\begin{aligned} \check{x}_{t+1} &= (A - K_2 G)\, \check{x}_t + K_2 \tilde{z}_t, \\ a_t &= -G\, \check{x}_t + \tilde{z}_t. \end{aligned} \tag{43.33}$$

The likelihood function for a sample of T observations $\{\tilde{z}_t\}$ is then

$$\mathcal{L}^{**} = -T \ln 2\pi - \tfrac{1}{2} T \ln |V_2| - \tfrac{1}{2} \sum_{t=0}^{T-1} a_t^\top V_2^{-1} a_t. \tag{43.34}$$

Note that relative to computing the likelihood function (43.24) for the error-corrupted data, computing the likelihood function for the optimally filtered data requires more calculations.

Both likelihood functions require that the Kalman filter (43.20) be computed, while the likelihood function for the filtered data requires that the Kalman filter (43.30) also be computed.

In effect, in order to interpret and use the filtered data reported by the agency, it is necessary to retrace the steps that the agency used to synthesize those data.

The Kalman filter (43.20) is supposed to be formed by the agency.

The agency need not use Kalman filter (43.30) because it does not need the Wold representation for the filtered data.

In our parameterization $G = C$.

```
Q2 = K1 @ V1 @ K1.T
ε = 1e-6

K2, S2, V2 = steady_state_kalman(A, C, Q2, ε * np.eye(3))


def filtered_wold_coeffs(A, C, K, n_terms=25):
    psi = [np.eye(3)]
    Apow = np.eye(2)
    for _ in range(1, n_terms):
        psi.append(C @ Apow @ K)
        Apow = Apow @ A
    return psi


psi2 = filtered_wold_coeffs(A, C, K2, n_terms=40)
resp2 = np.array(
  [psi2[j] @ linalg.cholesky(V2, lower=True) for j in range(14)])
decomp2 = fev_contributions(psi2, V2, n_horizons=20)
```

43.5.4 Forecast-error-variance decomposition

Because the filtered data are nearly noiseless, the innovation covariance V_2 is close to singular with one dominant eigenvalue.

This means the filtered economy is driven by essentially one shock, just like the true economy

```
parts = []
for i, title in enumerate(shock_titles):
    arr = df_to_latex_array(fev_table(decomp2, i, horizons)).strip('$')
    parts.append(
      r'\begin{array}{c} ' + title + r' \\ ' + arr + r' \end{array}')

display(Latex('$' + r' \quad '.join(parts) + '$'))
```

A. Innovation in y_n

	y_n	c	Δk
1	0.995	0.002	0.902
2	0.997	0.004	0.902
3	0.999	0.007	0.902
4	1.001	0.009	0.902
5	1.004	0.011	0.902
6	1.006	0.014	0.902
7	1.008	0.016	0.902
8	1.010	0.018	0.902
9	1.013	0.020	0.902
10	1.015	0.023	0.902
11	1.017	0.025	0.902
12	1.019	0.027	0.902
13	1.022	0.029	0.902
14	1.024	0.032	0.902
15	1.026	0.034	0.902
16	1.028	0.036	0.902
17	1.031	0.038	0.902
18	1.033	0.041	0.902
19	1.035	0.043	0.902
20	1.038	0.045	0.902

B. Innovation in c

	y_n	c	Δk
1	0.000	0.000	0.000
2	0.000	0.000	0.000
3	0.000	0.000	0.000
4	0.000	0.000	0.000
5	0.000	0.000	0.000
6	0.000	0.000	0.000
7	0.000	0.000	0.000
8	0.000	0.000	0.000
9	0.000	0.000	0.000
10	0.000	0.000	0.000
11	0.000	0.000	0.000
12	0.000	0.000	0.000
13	0.000	0.000	0.000
14	0.000	0.000	0.000
15	0.000	0.000	0.000
16	0.000	0.000	0.000
17	0.000	0.000	0.000
18	0.000	0.000	0.000
19	0.000	0.000	0.000
20	0.000	0.000	0.000

C. Innovation in Δk

	y_n	c	Δk
1	0.000	0.000	0.000
2	0.000	0.000	0.000
3	0.000	0.000	0.000
4	0.000	0.000	0.000
5	0.000	0.000	0.000
6	0.000	0.000	0.000
7	0.000	0.000	0.000
8	0.000	0.000	0.000
9	0.000	0.000	0.000
10	0.000	0.000	0.000
11	0.000	0.000	0.000
12	0.000	0.000	0.000
13	0.000	0.000	0.000
14	0.000	0.000	0.000
15	0.000	0.000	0.000
16	0.000	0.000	0.000
17	0.000	0.000	0.000
18	0.000	0.000	0.000
19	0.000	0.000	0.000
20	0.000	0.000	0.000

In Model 2, the first innovation accounts for virtually all forecast-error variance, just as in the true economy where the single structural shock θ_t drives everything.

The second and third innovations contribute negligibly.

This confirms that filtering strips away the measurement noise that created the appearance of multiple independent sources of variation in Model 1.

The covariance matrix and eigenvalues of the Model 2 innovations are

```
print('Covariance matrix of innovations:')
df_v2 = pd.DataFrame(np.round(V2, 4), index=labels, columns=labels)
display(Latex(df_to_latex_matrix(df_v2)))
```

```
Covariance matrix of innovations:
```

$$\begin{bmatrix} 0.9945 & 0.0474 & 0.9471 \\ 0.0474 & 0.0023 & 0.0452 \\ 0.9471 & 0.0452 & 0.9019 \end{bmatrix}$$

```
print('Eigenvalues of covariance matrix:')
print(np.sort(np.linalg.eigvalsh(V2))[::-1].round(4))
```

```
Eigenvalues of covariance matrix:
[1.899 0.    0.   ]
```

As Sargent [1989] emphasizes, the two models of measurement produce quite different inferences about the economy's dynamics despite sharing identical underlying parameters.

43.5.5 Wold impulse responses

We again use orthogonalized Wold representation impulse responses with a Cholesky decomposition of V_2 ordered as y_n, c, Δk.

```
parts = []
for i, title in enumerate(wold_titles):
    arr = df_to_latex_array(
      wold_response_table(resp2, i, lags)).strip('$')
    parts.append(
      r'\begin{array}{c} ' + title + r' \\ ' + arr + r' \end{array}')

display(Latex('$' + r' \quad '.join(parts) + '$'))
```

A. Response to y_n innovation

	y_n	c	Δk
0	0.997	0.048	0.950
1	0.048	0.048	0.000
2	0.048	0.048	0.000
3	0.048	0.048	0.000
4	0.048	0.048	0.000
5	0.048	0.048	0.000
6	0.048	0.048	0.000
7	0.048	0.048	0.000
8	0.048	0.048	0.000
9	0.048	0.048	0.000
10	0.048	0.048	0.000
11	0.048	0.048	0.000
12	0.048	0.048	0.000
13	0.048	0.048	0.000

B. Response to c innovation

	y_n	c	Δk
0	0.000	0.003	−0.002
1	0.002	0.002	0.000
2	0.002	0.002	0.000
3	0.002	0.002	0.000
4	0.002	0.002	0.000
5	0.002	0.002	0.000
6	0.002	0.002	0.000
7	0.002	0.002	0.000
8	0.002	0.002	0.000
9	0.002	0.002	0.000
10	0.002	0.002	0.000
11	0.002	0.002	0.000
12	0.002	0.002	0.000
13	0.002	0.002	0.000

C. Response to Δk innovation

	y_n	c	Δk
0	0.000	0.000	0.002
1	−0.001	−0.001	0.000
2	−0.001	−0.001	0.000
3	−0.001	−0.001	0.000
4	−0.001	−0.001	0.000
5	−0.001	−0.001	0.000
6	−0.001	−0.001	0.000
7	−0.001	−0.001	0.000
8	−0.001	−0.001	0.000
9	−0.001	−0.001	0.000
10	−0.001	−0.001	0.000
11	−0.001	−0.001	0.000
12	−0.001	−0.001	0.000
13	−0.001	−0.001	0.000

The income innovation in Model 2 produces responses that closely approximate the true impulse response function from the structural shock θ_t.

Readers can compare the left table with the table in the *True impulse responses* section above.

The numbers are essentially the same.

The consumption and investment innovations produce responses that are orders of magnitude smaller, confirming that the filtered data are driven by essentially one shock.

Unlike Model 1, the filtered data from Model 2 *cannot* reproduce the apparent Granger causality pattern that the accelerator literature has documented empirically.

Hence, at the population level, the two measurement models imply different empirical stories even though they share the same structural economy.

- In Model 1 (raw data), measurement noise creates multiple innovations and an apparent Granger-causality pattern.
- In Model 2 (filtered data), innovations collapse back to essentially one dominant shock, mirroring the true one-index economy.

Let's verify these implications in a finite sample simulation.

43.6 Simulation

The tables above characterize population moments of the two models.

Let's simulate 80 periods of true, measured, and filtered data to compare population implications with finite-sample behavior.

First, we define a function to simulate the true economy, generate measured data with AR(1) measurement errors, and apply the Model 1 Kalman filter to produce filtered estimates

```
def simulate_series(seed=7909, T=80, k0=10.0):
    """
    Simulate true, measured, and filtered series.
    """
    rng = np.random.default_rng(seed)

    # True state/observables
    θ = rng.normal(0.0, 1.0, size=T)
    k = np.empty(T + 1)
    k[0] = k0

    y = np.empty(T)
    c = np.empty(T)
    dk = np.empty(T)

    for t in range(T):
        x_t = np.array([k[t], θ[t]])
        y[t], c[t], dk[t] = C @ x_t
        k[t + 1] = k[t] + (1.0 / f) * θ[t]

    # Measured data with AR(1) errors
    v_prev = np.zeros(3)
    v = np.empty((T, 3))
    for t in range(T):
        η_t = rng.multivariate_normal(np.zeros(3), Σ_η)
        v_prev = D @ v_prev + η_t
        v[t] = v_prev

    z_meas = np.column_stack([y, c, dk]) + v

    # Filtered data via Model 1 transformed filter
    xhat_prev = np.array([k0, 0.0])
    z_prev = np.zeros(3)
    z_filt = np.empty((T, 3))
```

(continues on next page)

(continued from previous page)

```
    k_filt = np.empty(T)

    for t in range(T):
        z_bar_t = z_meas[t] - D @ z_prev
        u_t = z_bar_t - C_bar @ xhat_prev
        xhat_t = A @ xhat_prev + K1 @ u_t

        z_filt[t] = C @ xhat_t
        k_filt[t] = xhat_t[0]

        xhat_prev = xhat_t
        z_prev = z_meas[t]

    out = {
        "y_true": y, "c_true": c, "dk_true": dk, "k_true": k[:-1],
        "y_meas": z_meas[:, 0], "c_meas": z_meas[:, 1],
        "dk_meas": z_meas[:, 2],
        "y_filt": z_filt[:, 0], "c_filt": z_filt[:, 1],
        "dk_filt": z_filt[:, 2], "k_filt": k_filt
    }
    return out


sim = simulate_series(seed=7909, T=80, k0=10.0)
```

We use the following helper function to plot the true series against either the measured or filtered series

```
def plot_true_vs_other(t, true_series, other_series,
                                  other_label, ylabel=""):
    fig, ax = plt.subplots(figsize=(8, 4))
    ax.plot(t, true_series, lw=2, label="true")
    ax.plot(t, other_series, lw=2, ls="--", label=other_label)
    ax.set_xlabel("time")
    ax.set_ylabel(ylabel)
    ax.legend()
    plt.tight_layout()
    plt.show()


t = np.arange(1, 81)
```

Let's first compare the true series with the measured series to see how measurement errors distort the data

```
plot_true_vs_other(t, sim["c_true"], sim["c_meas"],
                                    "measured", ylabel="consumption")
```

```
plot_true_vs_other(t, sim["dk_true"], sim["dk_meas"],
                                    "measured", ylabel="investment")
```

```
plot_true_vs_other(t, sim["y_true"], sim["y_meas"],
                                    "measured", ylabel="income")
```

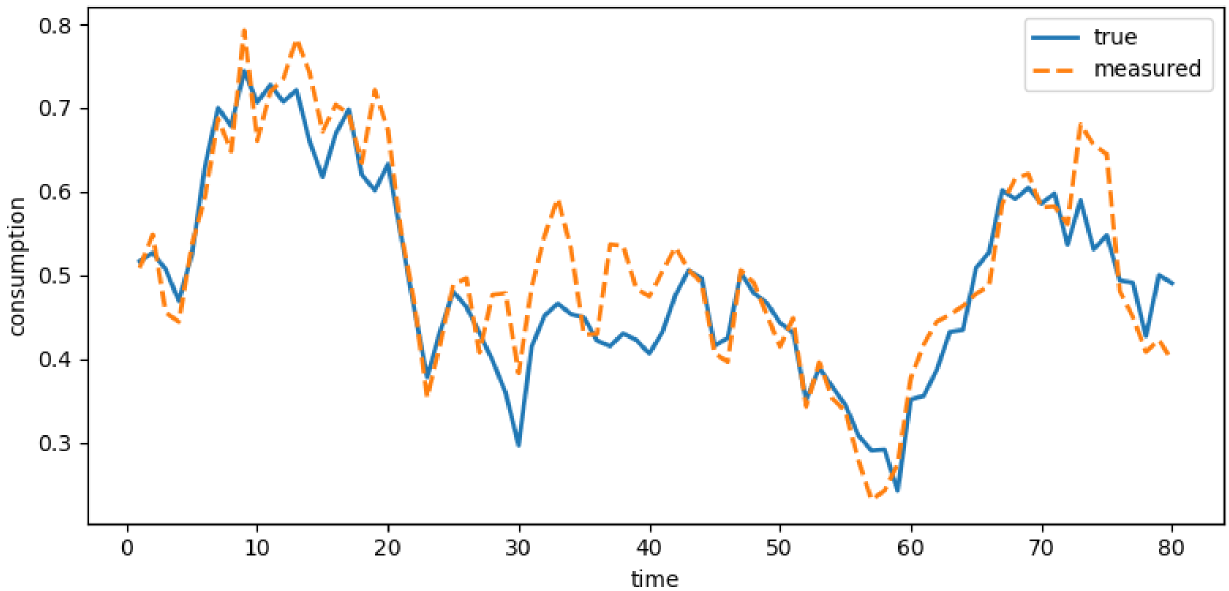


Fig. 43.1: True and measured consumption

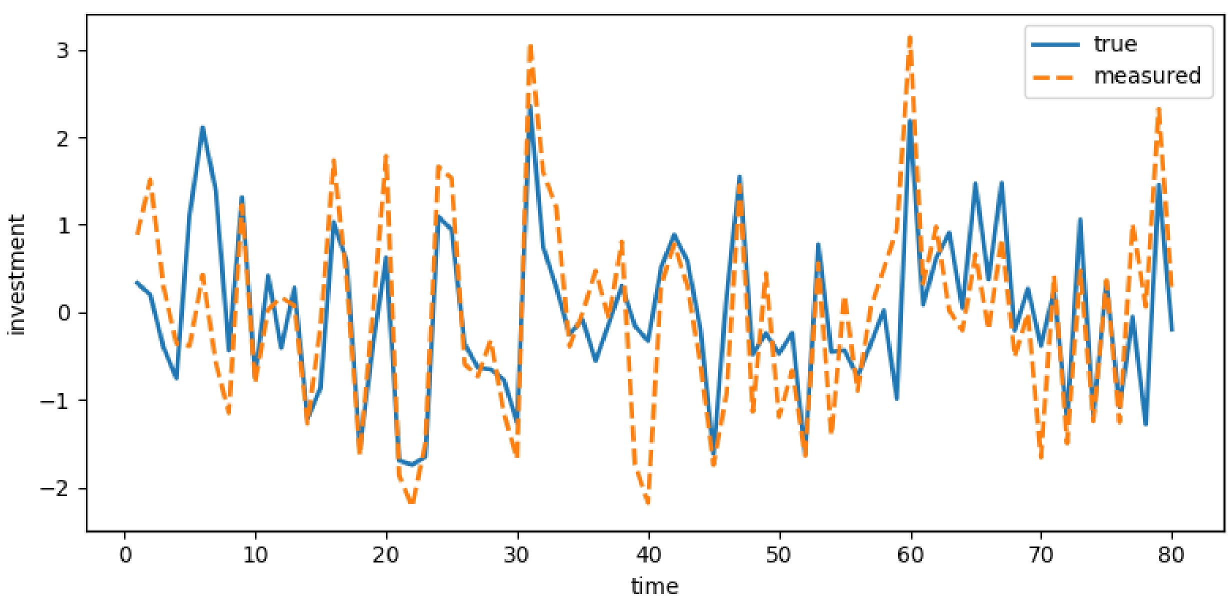


Fig. 43.2: True and measured investment

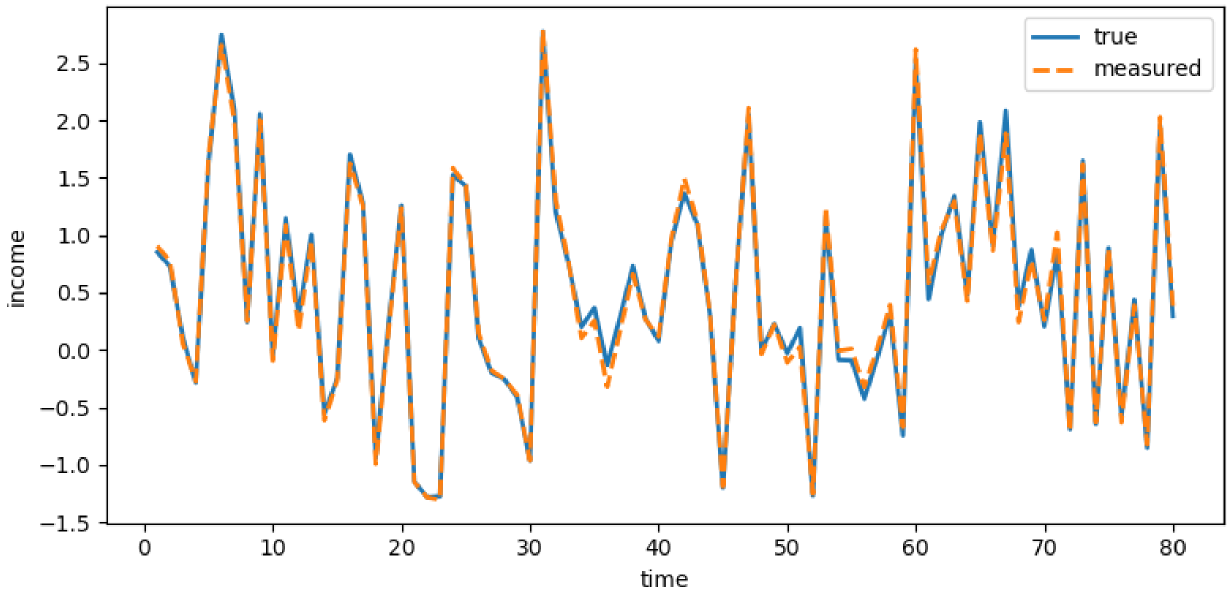


Fig. 43.3: True and measured income

Investment is distorted the most because its measurement error has the largest innovation variance ($\sigma_\eta = 0.65$), while income is distorted the least ($\sigma_\eta = 0.05$).

For the filtered series, we expect the Kalman filter to recover the true series more closely by stripping away measurement noise

```
plot_true_vs_other(t, sim["c_true"], sim["c_filt"],
                   "filtered", ylabel="consumption")
```

```
plot_true_vs_other(t, sim["dk_true"], sim["dk_filt"],
                   "filtered", ylabel="investment")
```

```
plot_true_vs_other(t, sim["y_true"], sim["y_filt"],
                   "filtered", ylabel="income")
```

```
plot_true_vs_other(t, sim["k_true"], sim["k_filt"],
                   "filtered", ylabel="capital stock")
```

Indeed, Kalman-filtered estimates from Model 1 remove much of the measurement noise and track the truth closely.

In the true model the national income identity $c_t + \Delta k_t = y_{n,t}$ holds exactly.

Independent measurement errors break this accounting identity in the measured data.

The Kalman filter approximately restores it.

The following figure confirms this by showing the residual $c_t + \Delta k_t - y_{n,t}$ for both measured and filtered data

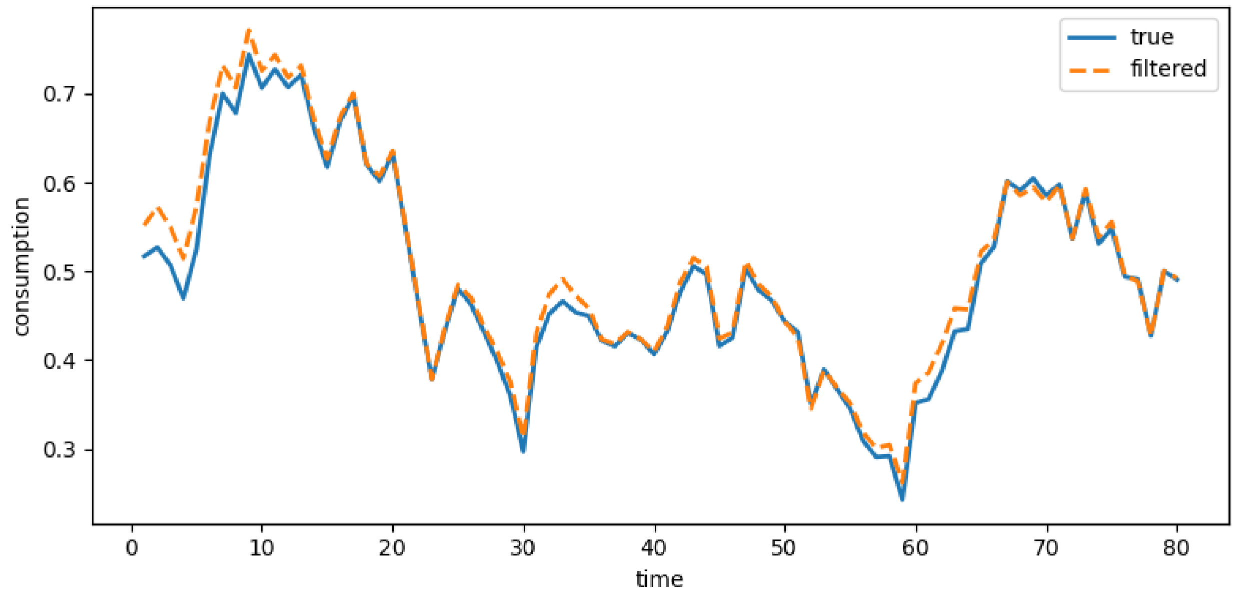


Fig. 43.4: True and filtered consumption

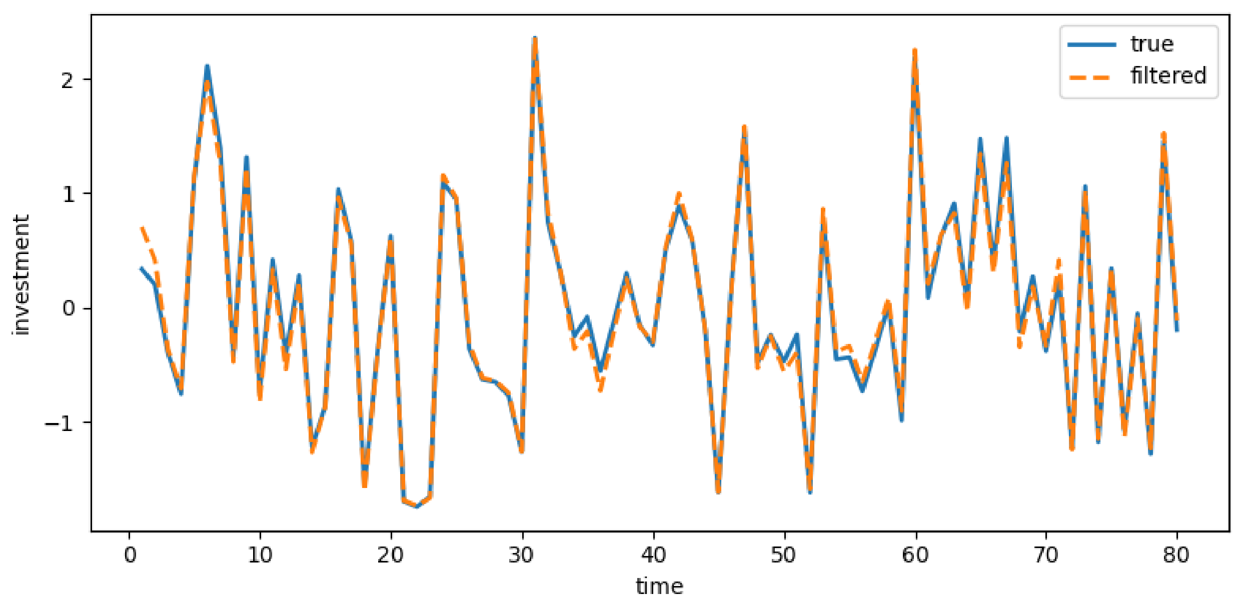


Fig. 43.5: True and filtered investment

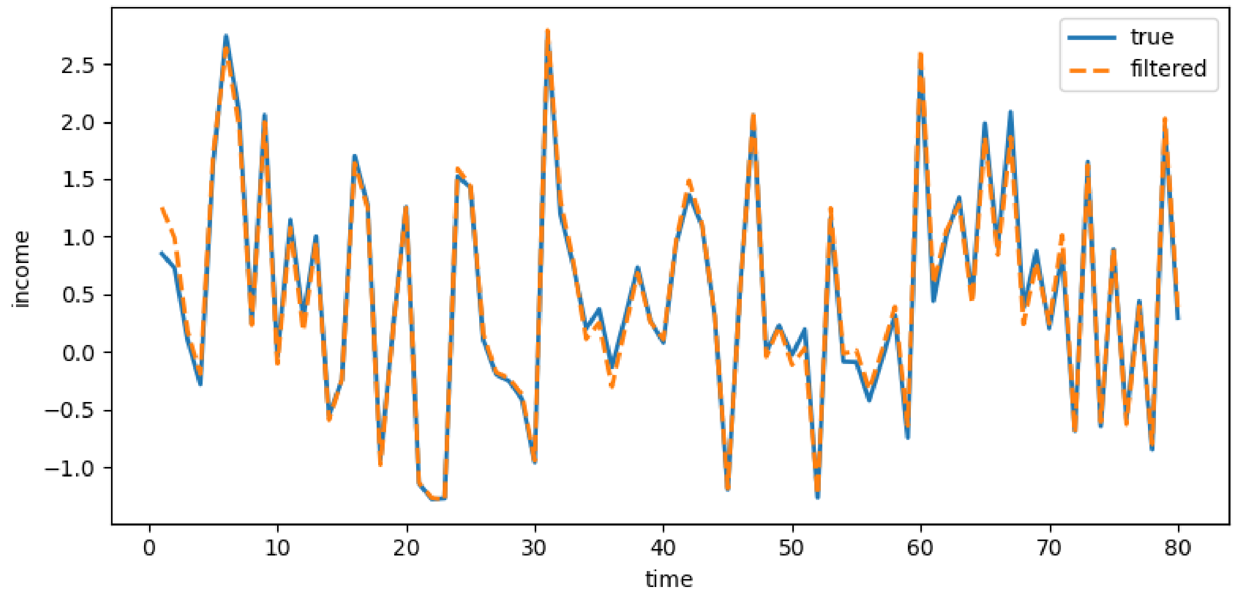


Fig. 43.6: True and filtered income

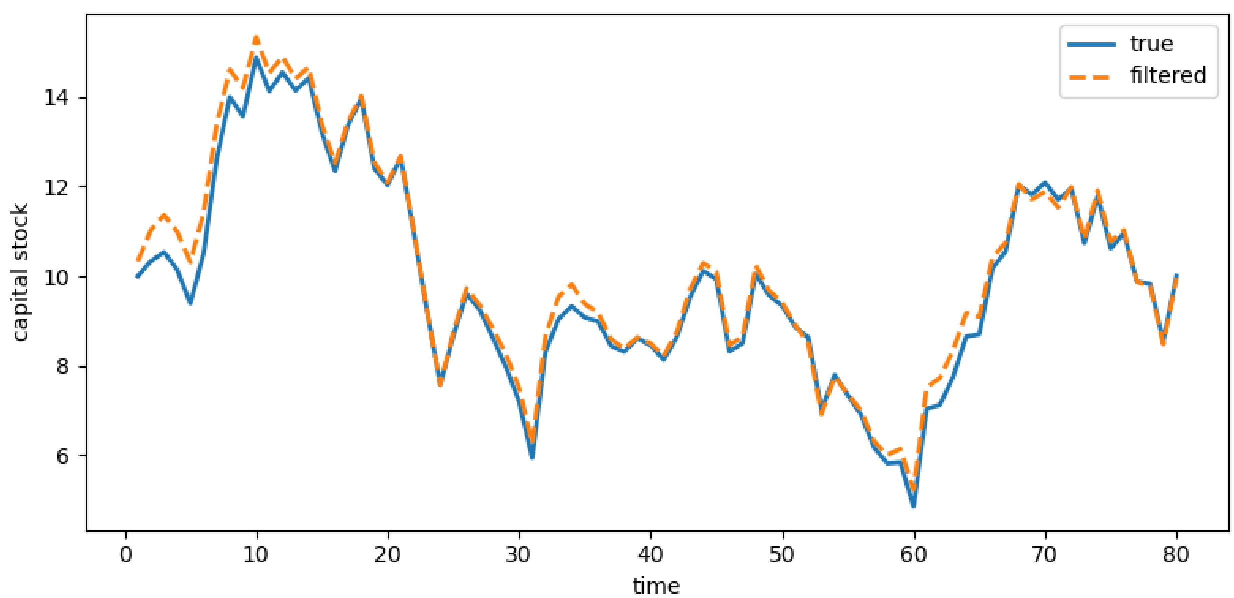


Fig. 43.7: True and filtered capital stock

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 4))

ax1.plot(t, sim["c_meas"] + sim["dk_meas"] - sim["y_meas"], lw=2)
ax1.axhline(0, color='black', lw=0.8, ls='--', alpha=0.5)
ax1.set_xlabel("time")
ax1.set_ylabel("measured residual")

ax2.plot(t, sim["c_filt"] + sim["dk_filt"] - sim["y_filt"], lw=2)
ax2.axhline(0, color='black', lw=0.8, ls='--', alpha=0.5)
ax2.set_xlabel("time")
ax2.set_ylabel("filtered residual")

plt.tight_layout()
plt.show()
```

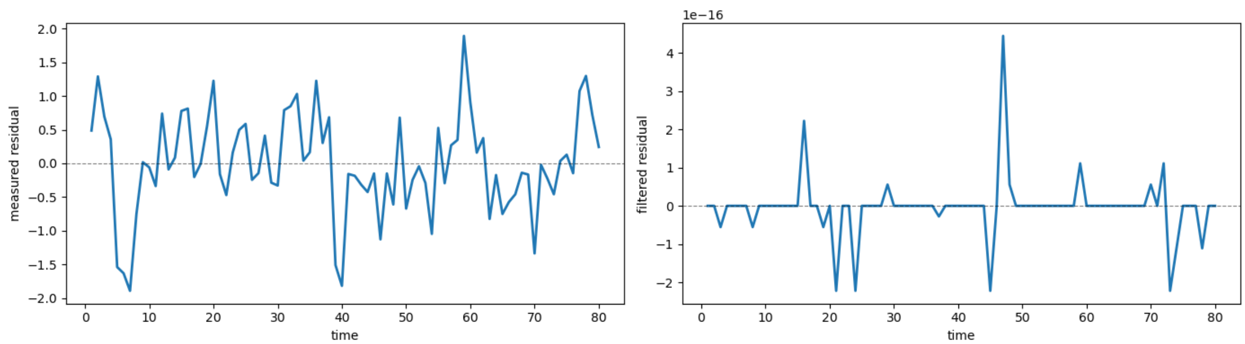


Fig. 43.8: National income identity residual

As we have predicted, the residual for the measured data is large and volatile, while the residual for the filtered data is numerically 0.

43.7 Summary

Sargent [1989] shows how measurement error alters an econometrician's view of a permanent income economy driven by the investment accelerator.

The Wold representations and variance decompositions of Model 1 (raw measurements) and Model 2 (filtered measurements) differ substantially, even though the underlying economy is the same.

Measurement error can reshape inferences about which shocks drive which variables.

Model 1 reproduces the **Granger causality** pattern documented in the empirical accelerator literature: income appears to Granger-cause consumption and investment, a result Sargent [1989] attributes to measurement error and signal extraction in raw reported data.

Model 2, working with filtered data, attributes nearly all variance to the single structural shock θ_t and *cannot* reproduce the Granger causality pattern.

The *Kalman filter* effectively strips measurement noise from the data, so the filtered series track the truth closely.

Raw measurement error breaks the national income accounting identity, but the near-zero residual shows that the filter approximately restores it.

Part VII

Search

CHAPTER

FORTYFOUR

JOB SEARCH I: THE MCCALL SEARCH MODEL

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the “play” icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

Contents

> “Questioning a McCall worker is like having a conversation with an out-of-work friend: ‘Maybe you are setting your sights too high’, or ‘Why did you quit your old job before you had a new one lined up?’ This is real social science: an attempt to model, to understand, human behavior by visualizing the situation people find themselves in, the options they face and the pros and cons as they themselves see them.” – Robert E. Lucas, Jr.

In addition to what’s in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon jax
```

44.1 Overview

The McCall search model [McCall, 1970] helped transform economists' way of thinking about labor markets.

To clarify notions such as "involuntary" unemployment, McCall modeled the decision problem of an unemployed worker in terms of factors including

- current and likely future wages
- impatience
- unemployment compensation

To solve the decision problem McCall used dynamic programming.

Here we set up McCall's model and use dynamic programming to analyze it.

As we'll see, McCall's model is not only interesting in its own right but also an excellent vehicle for learning dynamic programming.

Let's start with some imports:

```
import matplotlib.pyplot as plt
import numpy as np
import numba
import jax
import jax.numpy as jnp
from typing import NamedTuple
from functools import partial
import quantecon as qe
from quantecon.distributions import BetaBinomial
```

44.2 The McCall Model

An unemployed agent receives in each period a job offer at wage W_t.

In this lecture, we adopt the following simple environment:

- The offer sequence $\{W_t\}_{t\geq 0}$ is IID, with $q(w)$ being the probability of observing wage w in finite set $\mathbb{W}$.
- The agent observes W_t at the start of t.
- The agent knows that $\{W_t\}$ is IID with common distribution q and can use this when computing expectations.

(In later lectures, we will relax these assumptions.)

At time t, our agent has two choices:

1. Accept the offer and work permanently at constant wage W_t.
2. Reject the offer, receive unemployment compensation c, and reconsider next period.

The agent is infinitely lived and aims to maximize the expected discounted sum of earnings

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t y_t \tag{44.1}$$

The constant β lies in $(0, 1)$ and is called a **discount factor**.

The smaller is β, the more the agent discounts future earnings relative to current earnings.

The variable y_t is income, equal to

- his/her wage W_t when employed
- unemployment compensation c when unemployed

44.2.1 A Trade-Off

The worker faces a trade-off:

- Waiting too long for a good offer is costly, since the future is discounted.
- Accepting too early is costly, since better offers might arrive in the future.

To decide the optimal wait time in the face of this trade-off, we use dynamic programming.

Dynamic programming can be thought of as a two-step procedure that

1. first assigns values to "states" and
2. then deduces optimal actions given those values

We'll go through these steps in turn.

44.2.2 The Value Function

In order to optimally trade-off current and future rewards, we need to think about two things:

1. the current payoffs we get from different choices
2. the different states that those choices will lead to in next period

To weigh these two aspects of the decision problem, we need to assign *values* to states.

To this end, let $v^*(w)$ be the total lifetime value accruing to an unemployed worker who enters the current period unemployed when the wage is $w \in \mathbb{W}$.

(In particular, the agent has wage offer w in hand and can accept or reject it.)

More precisely, $v^*(w)$ denotes the total sum of expected discounted earnings when an agent always behaves in an optimal way. points in time.

Of course $v^*(w)$ is not trivial to calculate because we don't yet know what decisions are optimal and what aren't!

If we don't know what opimal choices are, it feels imposible to calculate $v^*(w)$.

But let's put this aside for now and think of v^* as a function that assigns to each possible wage w the maximal lifetime value $v^*(w)$ that can be obtained with that offer in hand.

A crucial observation is that this function v^* must satisfy

$$v^*(w) = \max \left\{ \frac{w}{1-\beta}, \, c + \beta \sum_{w' \in \mathbb{W}} v^*(w') q(w') \right\} \tag{44.2}$$

for every possible w in $\mathbb{W}$.

This is a version of the **Bellman equation**, which is ubiquitous in economic dynamics and other fields involving planning over time.

The intuition behind it is as follows:

- the first term inside the max operation is the lifetime payoff from accepting current offer, since such a worker works forever at w and values this income stream as

$$\frac{w}{1-\beta} = w + \beta w + \beta^2 w + \cdots$$

- the second term inside the max operation is the continuation value, which is the lifetime payoff from rejecting the current offer and then behaving optimally in all subsequent periods

If we optimize and pick the best of these two options, we obtain maximal lifetime value from today, given current offer w.

But this is precisely $v^*(w)$, which is the left-hand side of (44.2).

Putting this all together, we see that (44.2) is valid for all w.

44.2.3 The Optimal Policy

We still don't know how to compute v^* (although (44.2) gives us hints we'll return to below).

But suppose for now that we do know v^*.

Once we have this function in hand we can easily make optimal choices (i.e., make the right choice between accept and reject given any w).

All we have to do is select the maximal choice on the right-hand side of (44.2).

In other words, we make the best choice between stopping and continuing, given the information provided to us by v^*.

The optimal action is best thought of as a **policy**, which is, in general, a map from states to actions.

Given any w, we can read off the corresponding best choice (accept or reject) by picking the max on the right-hand side of (44.2).

Thus, we have a map from $\mathbb{W}$ to $\{0, 1\}$, with 1 meaning accept and 0 meaning reject.

We can write the policy as follows

$$\sigma(w) := \mathbf{1}\left\{\frac{w}{1-\beta} \geq c + \beta \sum_{w' \in \mathbb{W}} v^*(w') q(w')\right\}$$

Here $\mathbf{1}\{P\} = 1$ if statement P is true and equals 0 otherwise.

We can also write this as

$$\sigma(w) := \mathbf{1}\{w \geq \bar{w}\}$$

where

$$\bar{w} := (1-\beta)\left\{c + \beta \sum_{w'} v^*(w') q(w')\right\} \tag{44.3}$$

Here $\bar{w}$ (called the **reservation wage**) is a constant depending on β, c and the wage distribution.

The agent should accept if and only if the current wage offer exceeds the reservation wage.

In view of (44.3), we can compute this reservation wage if we can compute the value function.

44.3 Computing the Optimal Policy: Take 1

To put the above ideas into action, we need to compute the value function at each $w \in \mathbb{W}$.

To simplify notation, let's set

$$\mathbb{W} := \{w_1, \dots, w_n\} \quad \text{and} \quad v^*(i) := v^*(w_i)$$

The value function is then represented by the vector $v^* = (v^*(i))_{i=1}^n$.

In view of (44.2), this vector satisfies the nonlinear system of equations

$$v^*(i) = \max \left\{ \frac{w(i)}{1-\beta}, \, c + \beta \sum_{j=1}^n v^*(j) q(j) \right\} \quad \text{for } i = 1, \dots, n \tag{44.4}$$

44.3.1 The Algorithm

To compute this vector, we use successive approximations:

Step 1: pick an arbitrary initial guess $v \in \mathbb{R}^n$.

Step 2: compute a new vector $v' \in \mathbb{R}^n$ via

$$v'(i) = \max \left\{ \frac{w(i)}{1-\beta}, \, c + \beta \sum_{j=1}^n v(j) q(j) \right\} \quad \text{for } i = 1, \dots, n \tag{44.5}$$

Step 3: calculate a measure of a discrepancy between v and v', such as $\max_i |v(i) - v'(i)|$.

Step 4: if the deviation is larger than some fixed tolerance, set $v = v'$ and go to step 2, else continue.

Step 5: return v.

For a small tolerance, the returned function v is a close approximation to the value function v^*.

The theory below elaborates on this point.

44.3.2 Fixed Point Theory

What's the mathematics behind these ideas?

First, one defines a mapping T from $\mathbb{R}^n$ to itself via

$$(Tv)(i) = \max \left\{ \frac{w(i)}{1-\beta}, \, c + \beta \sum_{j=1}^n v(j) q(j) \right\} \quad \text{for } i = 1, \dots, n \tag{44.6}$$

(A new vector Tv is obtained from given vector v by evaluating the r.h.s. at each i.)

The element v_k in the sequence $\{v_k\}$ of successive approximations corresponds to $T^k v$.

- This is T applied k times, starting at the initial guess v

One can show that the conditions of the Banach fixed point theorem are satisfied by T on $\mathbb{R}^n$.

One implication is that T has a unique fixed point in $\mathbb{R}^n$.

- That is, a unique vector $\bar{v}$ such that $T\bar{v} = \bar{v}$.

Moreover, it's immediate from the definition of T that this fixed point is v^*.

A second implication of the Banach contraction mapping theorem is that $\{T^k v\}$ converges to the fixed point v^* regardless of v.

44.3.3 Implementation

Our default for q, the wage offer distribution, will be Beta-binomial.

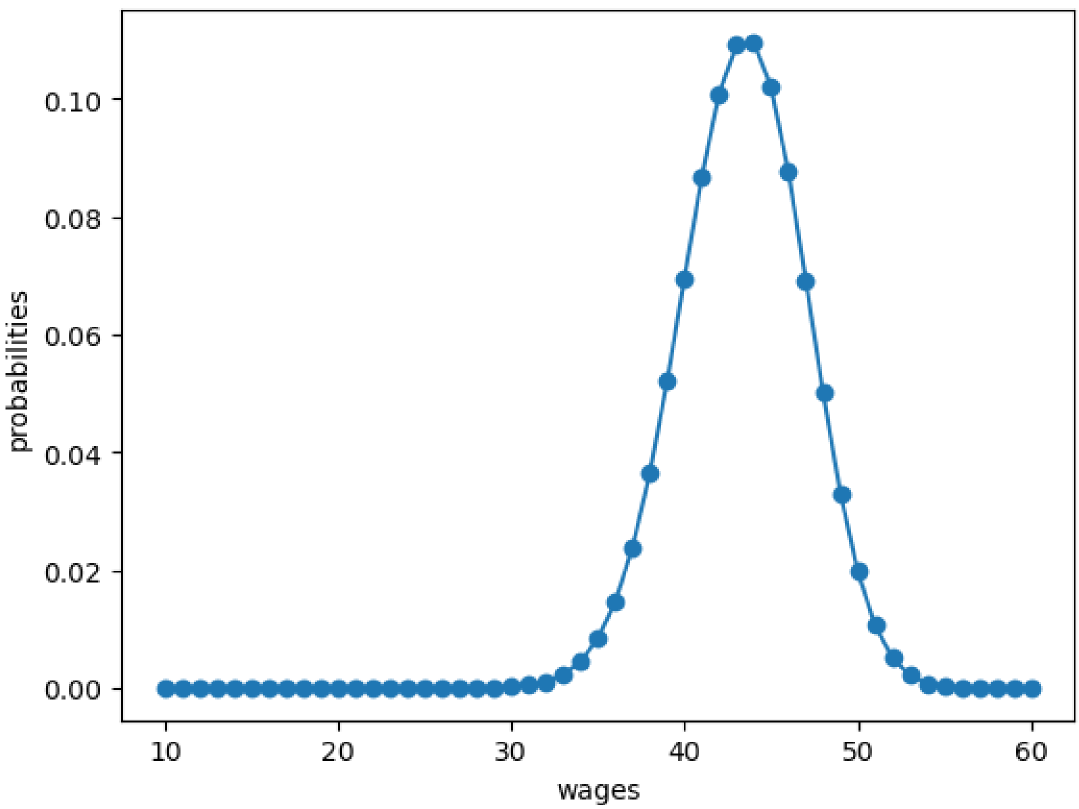
```
n, a, b = 50, 200, 100                        # default parameters
q_default = jnp.array(BetaBinomial(n, a, b).pdf())
```

Our default set of values for wages will be

```
w_min, w_max = 10, 60
w_default = jnp.linspace(w_min, w_max, n+1)
```

Here's a plot of the probabilities of different wage outcomes:

```
fig, ax = plt.subplots()
ax.plot(w_default, q_default, '-o', label='$q(w(i))$')
ax.set_xlabel('wages')
ax.set_ylabel('probabilities')

plt.show()
```



We will use JAX to write our code.

We'll use `NamedTuple` for our model class to maintain immutability, which works well with JAX's functional programming paradigm.

Here's a class that stores the model parameters with default values.

```
class McCallModel(NamedTuple):
    c: float = 25                 # unemployment compensation
    β: float = 0.99               # discount factor
    w: jnp.ndarray = w_default  # array of wage values, w[i] = wage at state i
    q: jnp.ndarray = q_default  # array of probabilities
```

We implement the Bellman operator T from (44.6), which we can write in terms of array operations as

$$Tv = \max\left\{\frac{w}{1-\beta}, \, c + \beta \sum_{j=1}^{n} v(j)q(j)\right\} \tag{44.7}$$

(The first term inside the max is an array and the second is just a number – here we mean that the max comparison against this number is done element-by-element for all elements in the array.)

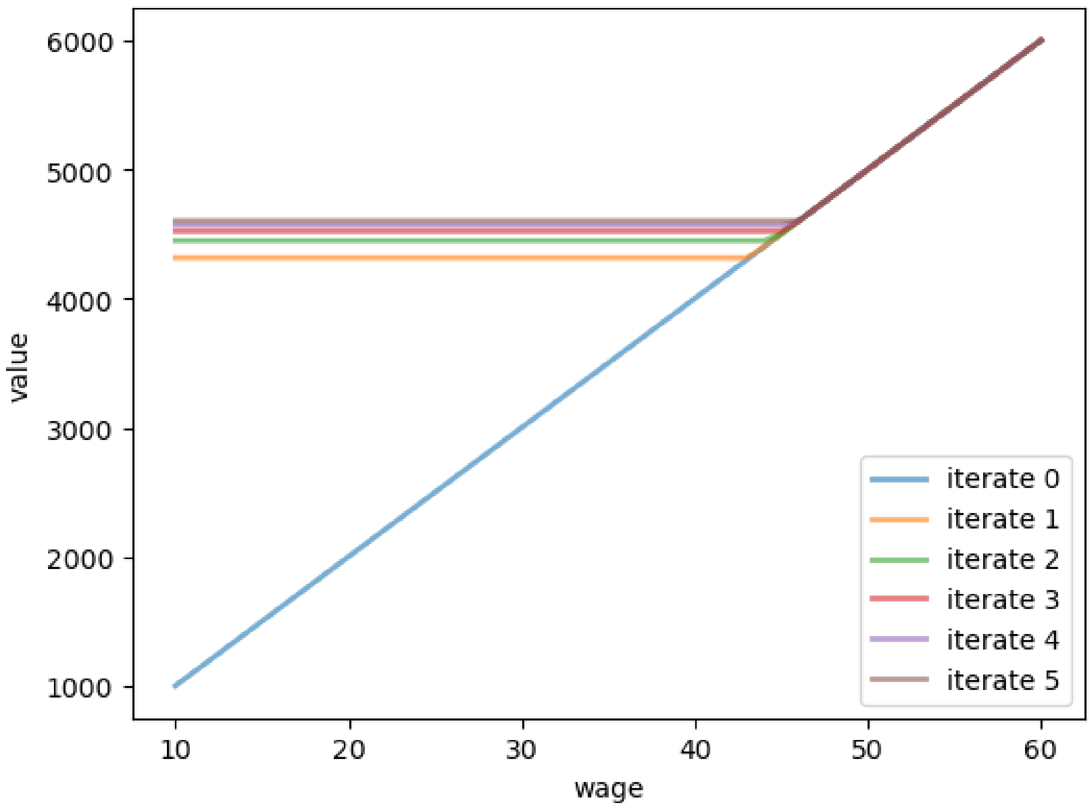
We can code T up as follows.

```
def T(model: McCallModel, v: jnp.ndarray):
    c, β, w, q = model
    accept = w / (1 - β)
    reject = c + β * v @ q
    return jnp.maximum(accept, reject)
```

Based on these defaults, let's try plotting the first few approximate value functions in the sequence $\{T^k v\}$.

We will start from guess v given by $v(i) = w(i)/(1-\beta)$, which is the value of accepting at every given wage.

```
model = McCallModel()
c, β, w, q = model
v = w / (1 - β)    # Initial condition
fig, ax = plt.subplots()

num_plots = 6
for i in range(num_plots):
    ax.plot(w, v, '-', alpha=0.6, lw=2, label=f"iterate {i}")
    v = T(model, v)

ax.legend(loc='lower right')
ax.set_xlabel('wage')
ax.set_ylabel('value')
plt.show()
```



You can see that convergence is occurring: successive iterates are getting closer together.

Here's a more serious iteration effort to compute the limit, which continues until measured deviation between successive iterates is below `tol`.

Once we obtain a good approximation to the limit, we will use it to calculate the reservation wage.

```
def compute_reservation_wage(
        model: McCallModel,    # instance containing default parameters
        v_init: jnp.ndarray,   # initial condition for iteration
        tol: float=1e-6,       # error tolerance
        max_iter: int=500,     # maximum number of iterations for loop
    ):
    "Computes the reservation wage in the McCall job search model."
    c, β, w, q = model
    i = 0
    error = tol + 1
    v = v_init

    while i < max_iter and error > tol:
        v_next = T(model, v)
        error = jnp.max(jnp.abs(v_next - v))
        v = v_next
        i += 1

    w_bar = (1 - β) * (c + β * v @ q)
    return v, w_bar
```

The cell computes the reservation wage at the default parameters

```
model = McCallModel()
c, β, w, q = model
v_init = w / (1 - β)  # initial guess
v, w_bar = compute_reservation_wage(model, v_init)
print(w_bar)
```

```
47.316475
```

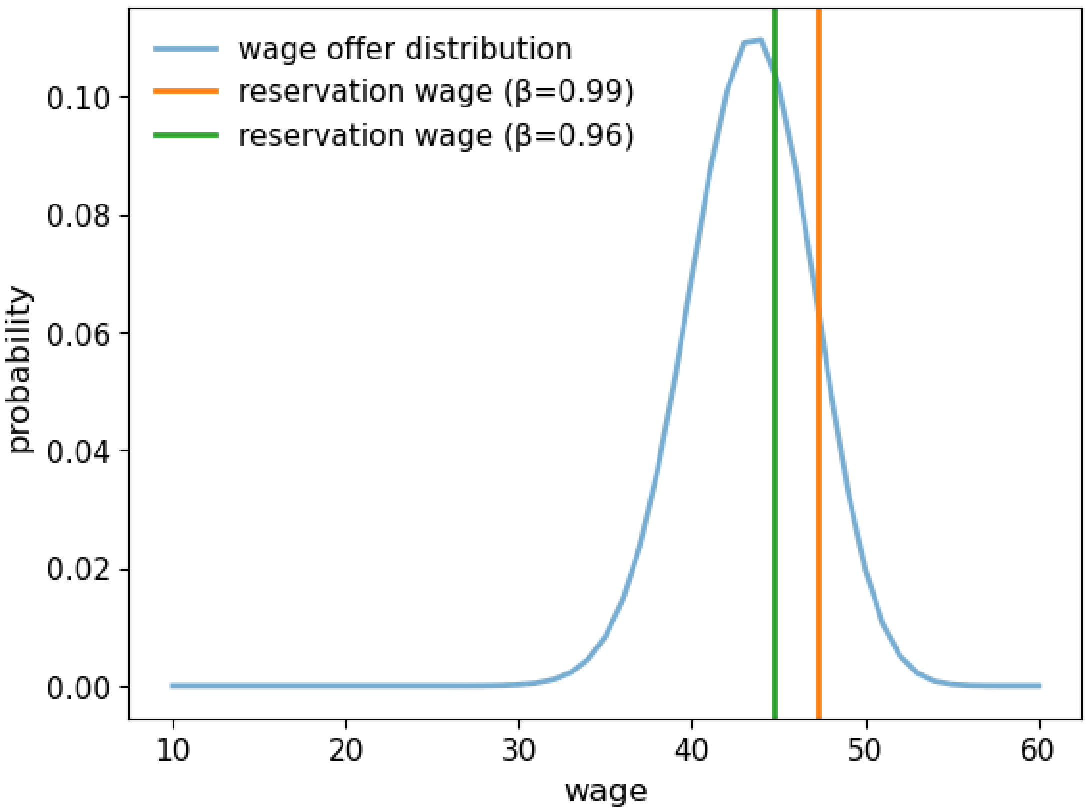
44.3.4 Comparative Statics

Now that we know how to compute the reservation wage, let's see how it varies with parameters.

Here we compare the reservation wage at two values of β.

The reservation wages will be plotted alongside the wage offer distribution, so that we can get a sense of what fraction of offers will be accepted.

```
fig, ax = plt.subplots()

# Get the default color cycle
prop_cycle = plt.rcParams['axes.prop_cycle']
colors = prop_cycle.by_key()['color']

# Plot the wage offer distribution
ax.plot(w, q, '-', alpha=0.6, lw=2,
        label='wage offer distribution',
        color=colors[0])

# Compute reservation wage with default beta
model_default = McCallModel()
c, β, w, q = model_default
v_init = w / (1 - β)
v_default, res_wage_default = compute_reservation_wage(
    model_default, v_init
)

# Compute reservation wage with lower beta
β_new = 0.96
model_low_beta = McCallModel(β=β_new)
c, β_low, w, q = model_low_beta
v_init_low = w / (1 - β_low)
v_low, res_wage_low = compute_reservation_wage(
    model_low_beta, v_init_low
)

# Plot vertical lines for reservation wages
ax.axvline(x=res_wage_default, color=colors[1], lw=2,
           label=f'reservation wage (β={β})')
ax.axvline(x=res_wage_low, color=colors[2], lw=2,
           label=f'reservation wage (β={β_new})')

ax.set_xlabel('wage', fontsize=12)
ax.set_ylabel('probability', fontsize=12)
ax.tick_params(axis='both', which='major', labelsize=11)
ax.legend(loc='upper left', frameon=False, fontsize=11)
plt.show()
```



We see that the reservation wage is higher when β is higher.

This is not surprising, since higher β is associated with more patience.

Now let's look more systematically at what happens when we change β and c.

As a first step, given that we'll use it many times, let's create a more efficient, jit-complied version of the function that computes the reservation wage:

```
@jax.jit
def compute_res_wage_jitted(
        model: McCallModel,    # instance containing default parameters
        v_init: jnp.ndarray,  # initial condition for iteration
        tol: float=1e-6,       # error tolerance
        max_iter: int=500,     # maximum number of iterations for loop
    ):
    c, β, w, q = model
    i = 0
    error = tol + 1
    initial_state = v_init, i, error

    def cond(loop_state):
        v, i, error = loop_state
        return jnp.logical_and(i < max_iter, error > tol)

    def update(loop_state):
        v, i, error = loop_state
        v_next = T(model, v)
        error = jnp.max(jnp.abs(v_next - v))
        i += 1
```

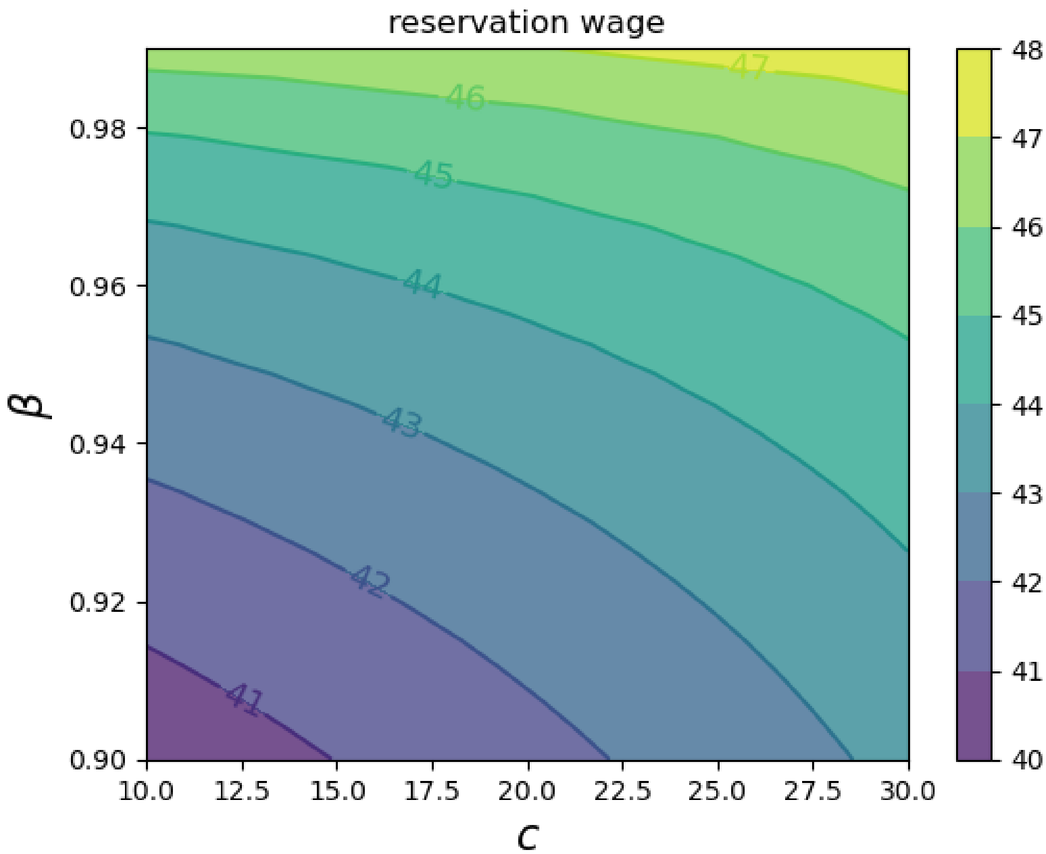
(continues on next page)

(continued from previous page)

```
        new_loop_state = v_next, i, error
        return new_loop_state

    final_state = jax.lax.while_loop(cond, update, initial_state)
    v, i, error = final_state

    w_bar = (1 - β) * (c + β * v @ q)
    return v, w_bar
```

Now we compute the reservation wage at each c, β pair.

```
grid_size = 25
c_vals = jnp.linspace(10.0, 30.0, grid_size)
β_vals = jnp.linspace(0.9, 0.99, grid_size)

res_wage_matrix = np.empty((grid_size, grid_size))
model = McCallModel()
v_init = model.w / (1 - model.β)

for i, c in enumerate(c_vals):
    for j, β in enumerate(β_vals):
        model = McCallModel(c=c, β=β)
        v, w_bar = compute_res_wage_jitted(model, v_init)
        v_init = v
        res_wage_matrix[i, j] = w_bar

fig, ax = plt.subplots()
cs1 = ax.contourf(c_vals, β_vals, res_wage_matrix.T, alpha=0.75)
ctr1 = ax.contour(c_vals, β_vals, res_wage_matrix.T)
plt.clabel(ctr1, inline=1, fontsize=13)
plt.colorbar(cs1, ax=ax)
ax.set_title("reservation wage")
ax.set_xlabel("$c$", fontsize=16)
ax.set_ylabel("$β$", fontsize=16)
ax.ticklabel_format(useOffset=False)
plt.show()
```



As expected, the reservation wage increases with both patience and unemployment compensation.

44.4 Computing an Optimal Policy: Take 2

The approach to dynamic programming just described is standard and broadly applicable.

But for our McCall search model there's also an easier way that circumvents the need to compute the value function.

Let h denote the continuation value:

$$h = c + \beta \sum_{w'} v^*(w') q(w') \tag{44.8}$$

The Bellman equation can now be written as

$$v^*(w') = \max \left\{ \frac{w'}{1-\beta}, \, h \right\} \tag{44.9}$$

Now let's derive a nonlinear equation for h alone.

Starting from (44.9), we multiply both sides by $q(w')$ to get

$$v^*(w') q(w') = \max \left\{ \frac{w'}{1-\beta}, h \right\} q(w')$$

Next, we sum both sides over $w' \in \mathbb{W}$:

$$\sum_{w' \in \mathbb{W}} v^*(w') q(w') = \sum_{w' \in \mathbb{W}} \max \left\{ \frac{w'}{1-\beta}, h \right\} q(w')$$

Now multiply both sides by β:

$$\beta \sum_{w' \in \mathbb{W}} v^*(w') q(w') = \beta \sum_{w' \in \mathbb{W}} \max \left\{ \frac{w'}{1-\beta}, h \right\} q(w')$$

Add c to both sides:

$$c + \beta \sum_{w' \in \mathbb{W}} v^*(w') q(w') = c + \beta \sum_{w' \in \mathbb{W}} \max \left\{ \frac{w'}{1-\beta}, h \right\} q(w')$$

Finally, using the definition of h from (44.8), the left-hand side is just h, giving us

$$h = c + \beta \sum_{w' \in \mathbb{W}} \max \left\{ \frac{w'}{1-\beta}, h \right\} q(w') \tag{44.10}$$

This is a nonlinear equation in the single scalar h that we can solve for h.

As before, we will use successive approximations:

Step 1: pick an initial guess h.

Step 2: compute the update h' via

$$h' = c + \beta \sum_{w' \in \mathbb{W}} \max \left\{ \frac{w'}{1-\beta}, h \right\} q(w') \tag{44.11}$$

Step 3: calculate the deviation $|h - h'|$.

Step 4: if the deviation is larger than some fixed tolerance, set $h = h'$ and go to step 2, else return h.

One can again use the Banach contraction mapping theorem to show that this process always converges.

The big difference here, however, is that we're iterating on a scalar h, rather than an n-vector, $v(i), i = 1, \ldots, n$.

Here's an implementation:

```python
def compute_reservation_wage_two(
        model: McCallModel,    # instance containing default parameters
        tol: float=1e-5,       # error tolerance
        max_iter: int=500,     # maximum number of iterations for loop
    ):
    c, β, w, q = model
    h = (w @ q) / (1 - β)   # initial condition
    i = 0
    error = tol + 1
    initial_loop_state = i, h, error

    def cond(loop_state):
        i, h, error = loop_state
        return jnp.logical_and(i < max_iter, error > tol)

    def update(loop_state):
        i, h, error = loop_state
        s = jnp.maximum(w / (1 - β), h)
        h_next = c + β * (s @ q)
        error = jnp.abs(h_next - h)
        i_next = i + 1
        new_loop_state = i_next, h_next, error
        return new_loop_state
```

(continues on next page)

(continued from previous page)

```
    final_state = jax.lax.while_loop(cond, update, initial_loop_state)
    i, h, error = final_state

    # Compute and return the reservation wage
    return (1 - β) * h
```

You can use this code to solve the exercise below.

44.5 Continuous Offer Distribution

The discrete wage offer distribution used above is convenient for theory and computation, but many realistic distributions are continuous (i.e., have a density).

Fortunately, the theory changes little in our simple model when we shift to a continuous offer distribution.

Recall that h in (44.8) denotes the value of not accepting a job in this period but then behaving optimally in all subsequent periods.

To shift to a continuous offer distribution, we can replace (44.8) by

$$h = c + \beta \int v^*(s')q(s')ds'. \tag{44.12}$$

Equation (44.10) becomes

$$h = c + \beta \int \max\left\{\frac{w(s')}{1-\beta}, h\right\} q(s')ds' \tag{44.13}$$

The aim is to solve this nonlinear equation by iteration, and from it obtain the reservation wage.

44.5.1 Implementation with Lognormal Wages

Let's implement this for the case where

- the state sequence $\{s_t\}$ is IID and standard normal and
- the wage function is $w(s) = \exp(\mu + \sigma s)$.

This gives us a lognormal wage distribution.

We use Monte Carlo integration to evaluate the integral, averaging over a large number of wage draws.

For default parameters, we use `c=25, β=0.99, σ=0.5, μ=2.5`.

```
class McCallModelContinuous(NamedTuple):
    c: float               # unemployment compensation
    β: float               # discount factor
    σ: float               # scale parameter in lognormal distribution
    μ: float               # location parameter in lognormal distribution
    w_draws: jnp.ndarray   # draws of wages for Monte Carlo


def create_mccall_continuous(
        c=25, β=0.99, σ=0.5, μ=2.5, mc_size=1000, seed=1234
    ):
    key = jax.random.PRNGKey(seed)
```

(continues on next page)

(continued from previous page)

```
    s = jax.random.normal(key, (mc_size,))
    w_draws = jnp.exp(μ + σ * s)
    return McCallModelContinuous(c, β, σ, μ, w_draws)


@jax.jit
def compute_reservation_wage_continuous(model, max_iter=500, tol=1e-5):
    c, β, σ, μ, w_draws = model

    h = jnp.mean(w_draws) / (1 - β)  # initial guess

    def update(state):
        h, i, error = state
        integral = jnp.mean(jnp.maximum(w_draws / (1 - β), h))
        h_next = c + β * integral
        error = jnp.abs(h_next - h)
        return h_next, i + 1, error

    def cond(state):
        h, i, error = state
        return jnp.logical_and(i < max_iter, error > tol)

    initial_state = (h, 0, tol + 1)
    final_state = jax.lax.while_loop(cond, update, initial_state)
    h_final, _, _ = final_state

    # Now compute the reservation wage
    return (1 - β) * h_final
```

Now let's investigate how the reservation wage changes with c and β using a contour plot.

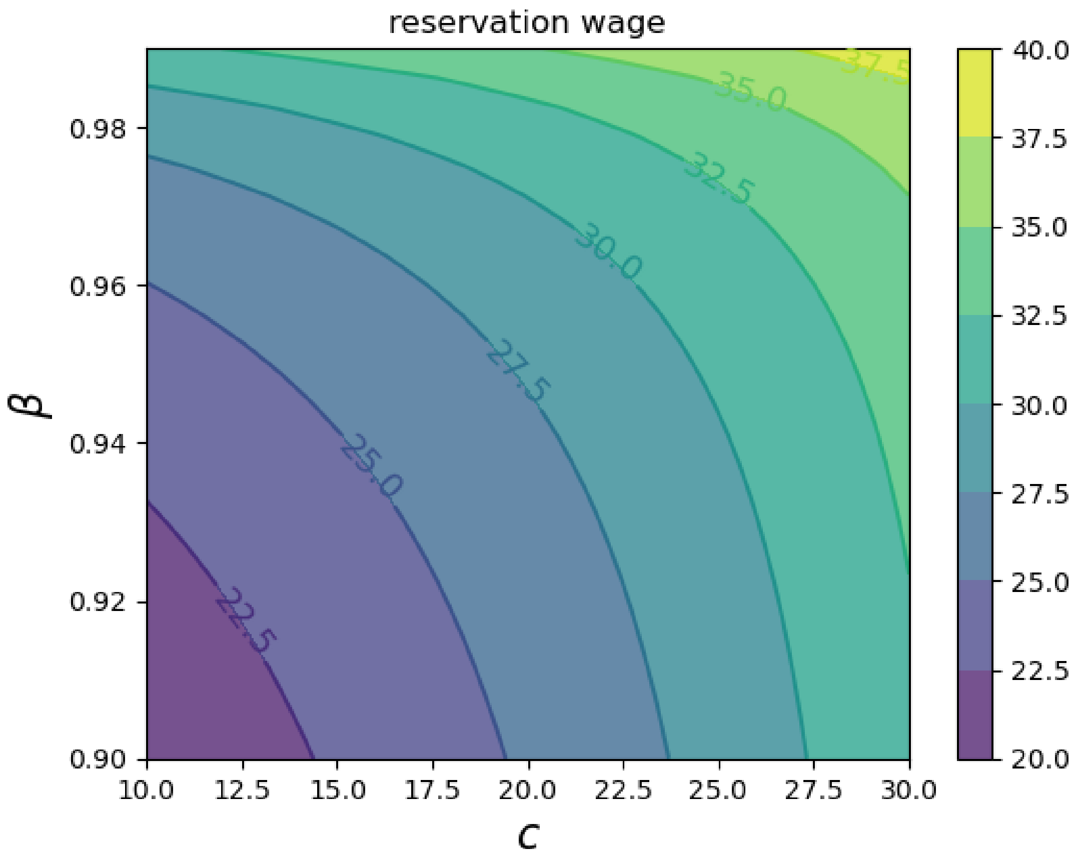
```
grid_size = 25
c_vals = jnp.linspace(10.0, 30.0, grid_size)
β_vals = jnp.linspace(0.9, 0.99, grid_size)

def compute_R_element(c, β):
    model = create_mccall_continuous(c=c, β=β)
    return compute_reservation_wage_continuous(model)

# First, vectorize over β (holding c fixed)
compute_R_over_β = jax.vmap(compute_R_element, in_axes=(None, 0))

# Next, vectorize over c (applying the above function to each c)
compute_R_vectorized = jax.vmap(compute_R_over_β, in_axes=(0, None))

# Apply to compute the full grid
R = compute_R_vectorized(c_vals, β_vals)
```

```
fig, ax = plt.subplots()

cs1 = ax.contourf(c_vals, β_vals, R.T, alpha=0.75)
ctr1 = ax.contour(c_vals, β_vals, R.T)

plt.clabel(ctr1, inline=1, fontsize=13)
plt.colorbar(cs1, ax=ax)
```

(continues on next page)

(continued from previous page)

```
ax.set_title("reservation wage")
ax.set_xlabel("$c$", fontsize=16)
ax.set_ylabel("$β$", fontsize=16)

ax.ticklabel_format(useOffset=False)

plt.show()
```



As with the discrete case, the reservation wage increases with both patience and unemployment compensation.

44.6 Volatility

An interesting feature of the McCall model is that increased volatility in wage offers tends to increase the reservation wage.

The intuition is that volatility is attractive to the worker because they can enjoy the upside (high wage offers) while rejecting the downside (low wage offers).

Hence, with more volatility, workers are more willing to continue searching rather than accept a given offer, which means the reservation wage rises.

To illustrate this phenomenon, we use a mean-preserving spread of the wage distribution.

In particular, we vary the scale parameter σ in the lognormal wage distribution $w(s) = \exp(\mu + \sigma s)$ while adjusting μ to keep the mean constant.

Recall that for a lognormal distribution with parameters μ and σ, the mean is $\exp(\mu + \sigma^2/2)$.

To keep the mean constant at some value m, we need:

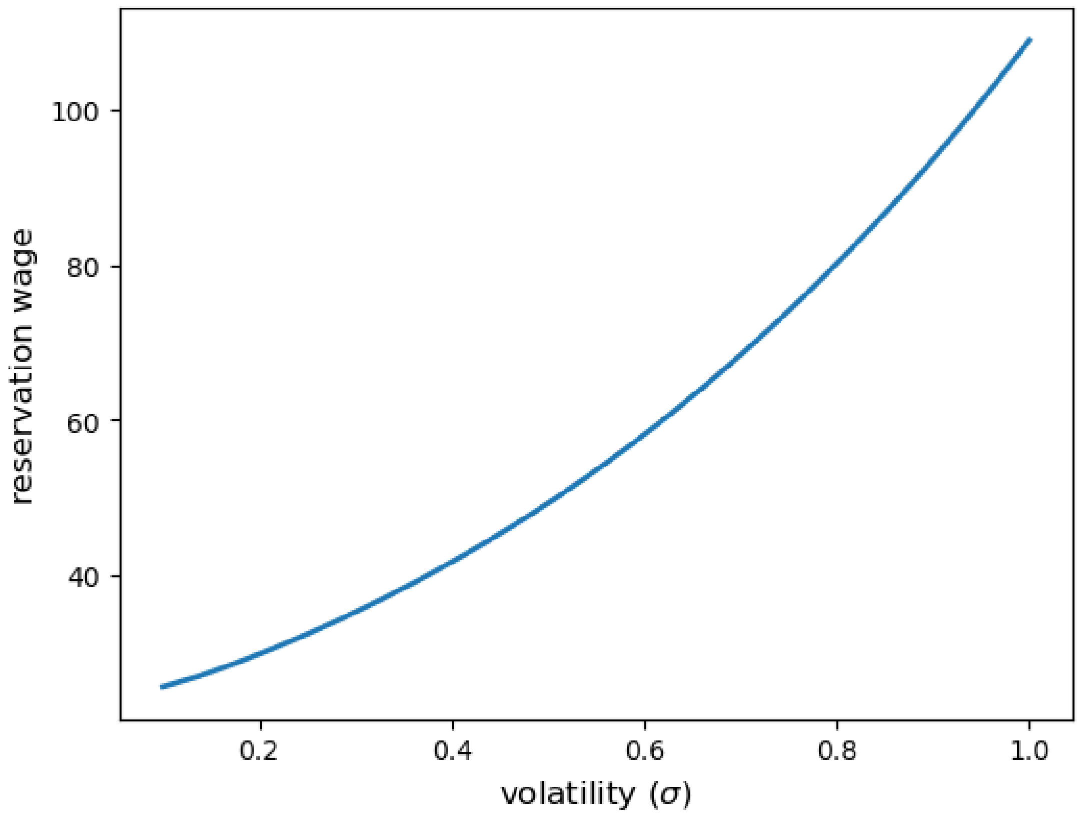
$$\mu = \ln(m) - \frac{\sigma^2}{2}$$

Let's implement this and compute the reservation wage for different values of σ:

```
# Fix the mean wage
mean_wage = 20.0

# Create a range of volatility values
σ_vals = jnp.linspace(0.1, 1.0, 25)

# Given σ, compute μ to maintain constant mean
def compute_μ_for_mean(σ, mean_wage):
    return jnp.log(mean_wage) - (σ**2) / 2

# Compute reservation wage for each volatility level
res_wages_volatility = []

for σ in σ_vals:
    μ = compute_μ_for_mean(σ, mean_wage)
    model = create_mccall_continuous(σ=float(σ), μ=float(μ))
    w_bar = compute_reservation_wage_continuous(model)
    res_wages_volatility.append(w_bar)

res_wages_volatility = jnp.array(res_wages_volatility)
```

Now let's plot the reservation wage as a function of volatility:

```
fig, ax = plt.subplots()
ax.plot(σ_vals, res_wages_volatility, linewidth=2)
ax.set_xlabel(r'volatility ($\sigma$)', fontsize=12)
ax.set_ylabel('reservation wage', fontsize=12)
plt.show()
```



As expected, the reservation wage is increasing in σ.

44.6.1 Lifetime Value and Volatility

We've seen that the reservation wage increases with volatility.

It's also the case that maximal lifetime value increases with volatility.

Higher volatility provides more upside potential, while at the same time workers can protect themselves against downside risk by rejecting low offers.

This option value translates into higher expected lifetime utility.

To demonstrate this, we will:

1. Compute the reservation wage for each volatility level
2. Calculate the expected discounted value of the lifetime income stream associated with that reservation wage, using Monte Carlo.

The simulation works as follows:

1. Compute the present discounted value of one lifetime earnings path, from a given wage path.
2. Average over a large number of such calculations to approximate expected discounted value.

We truncate each path at $T = 100$, which provides sufficient resolution for our purposes.

```
@jax.jit
def simulate_lifetime_value(key, model, w_bar, n_periods=100):
    """
```

(continues on next page)

(continued from previous page)

```
    Simulate one realization of the wage path and compute lifetime value.

    Parameters:
    -----------
    key : jax.random.PRNGKey
        Random key for JAX
    model : McCallModelContinuous
        The model containing parameters
    w_bar : float
        The reservation wage
    n_periods : int
        Number of periods to simulate

    Returns:
    --------
    lifetime_value : float
        Discounted sum of income over n_periods
    """
    c, β, σ, μ, w_draws = model

    # Draw all wage offers upfront
    key, subkey = jax.random.split(key)
    s_vals = jax.random.normal(subkey, (n_periods,))
    wage_offers = jnp.exp(μ + σ * s_vals)

    # Determine which offers are acceptable
    accept = wage_offers >= w_bar

    # Track employment status: employed from first acceptance onward
    employed = jnp.cumsum(accept) > 0

    # Get the accepted wage (first wage where accept is True)
    first_accept_idx = jnp.argmax(accept)
    accepted_wage = wage_offers[first_accept_idx]

    # Earnings at each period: accepted_wage if employed, c if unemployed
    earnings = jnp.where(employed, accepted_wage, c)

    # Compute discounted sum
    periods = jnp.arange(n_periods)
    discount_factors = β ** periods
    lifetime_value = jnp.sum(discount_factors * earnings)

    return lifetime_value


@jax.jit
def compute_mean_lifetime_value(model, w_bar, num_reps=10000, seed=1234):
    """
    Compute mean lifetime value across many simulations.

    """
    key = jax.random.PRNGKey(seed)
    keys = jax.random.split(key, num_reps)

    # Vectorize the simulation across all replications
    simulate_fn = jax.vmap(simulate_lifetime_value, in_axes=(0, None, None))
```

(continues on next page)

(continued from previous page)

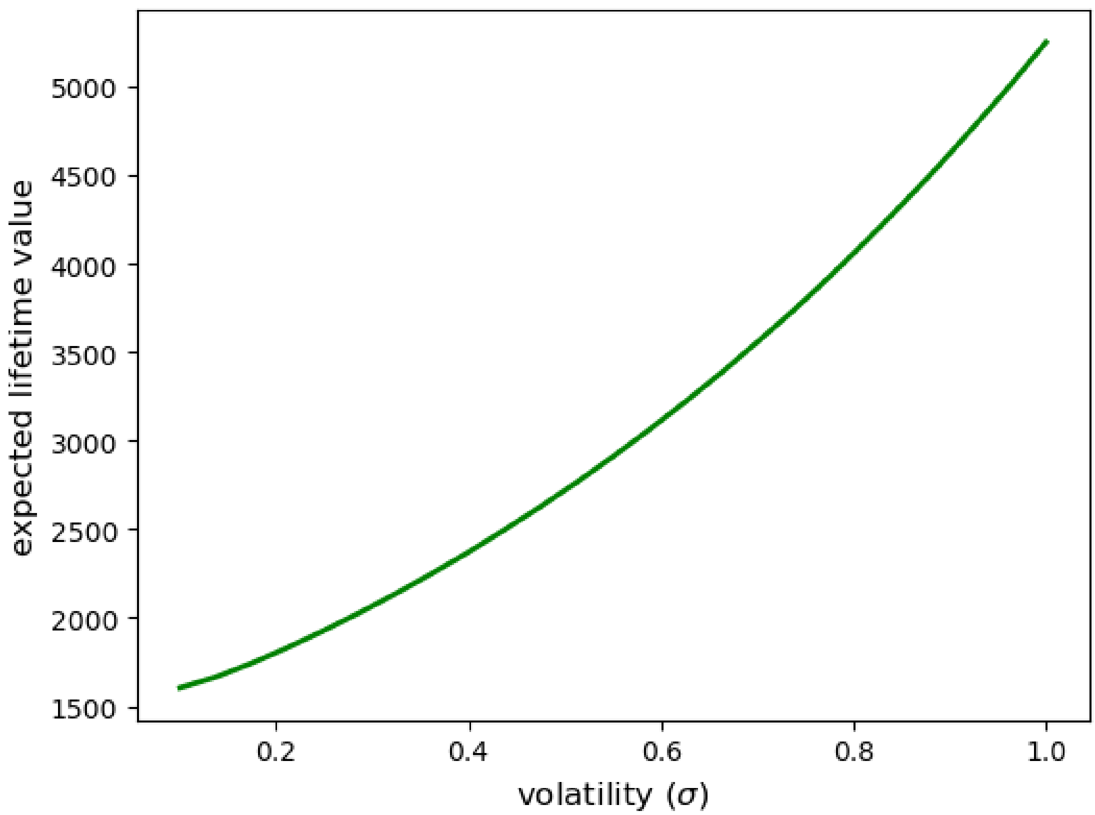
```
    lifetime_values = simulate_fn(keys, model, w_bar)
    return jnp.mean(lifetime_values)
```

Now let's compute the expected lifetime value for each volatility level:

```
# Use the same volatility range and mean wage
σ_vals = jnp.linspace(0.1, 1.0, 25)
mean_wage = 20.0

lifetime_vals = []
for σ in σ_vals:
    μ = compute_μ_for_mean(σ, mean_wage)
    model = create_mccall_continuous(σ=σ, μ=μ)
    w_bar = compute_reservation_wage_continuous(model)
    lv = compute_mean_lifetime_value(model, w_bar)
    lifetime_vals.append(lv)
```

Let's visualize the expected lifetime value as a function of volatility:

```
fig, ax = plt.subplots()
ax.plot(σ_vals, lifetime_vals, linewidth=2, color='green')
ax.set_xlabel(r'volatility ($\sigma$)', fontsize=12)
ax.set_ylabel('expected lifetime value', fontsize=12)
plt.show()
```



The plot confirms that despite workers setting higher reservation wages when facing more volatile wage offers (as shown above), they achieve higher expected lifetime values due to the option value of search.

44.7 Exercises

Exercise 44.7.1

Compute the average duration of unemployment when $\beta = 0.99$ and c takes the following values

```
c_vals = np.linspace(10, 40, 4)
```

That is, start the agent off as unemployed, compute their reservation wage given the parameters, and then simulate to see how long it takes to accept.

Repeat a large number of times and take the average.

Plot mean unemployment duration as a function of c in `c_vals`.

Try to explain what you see.

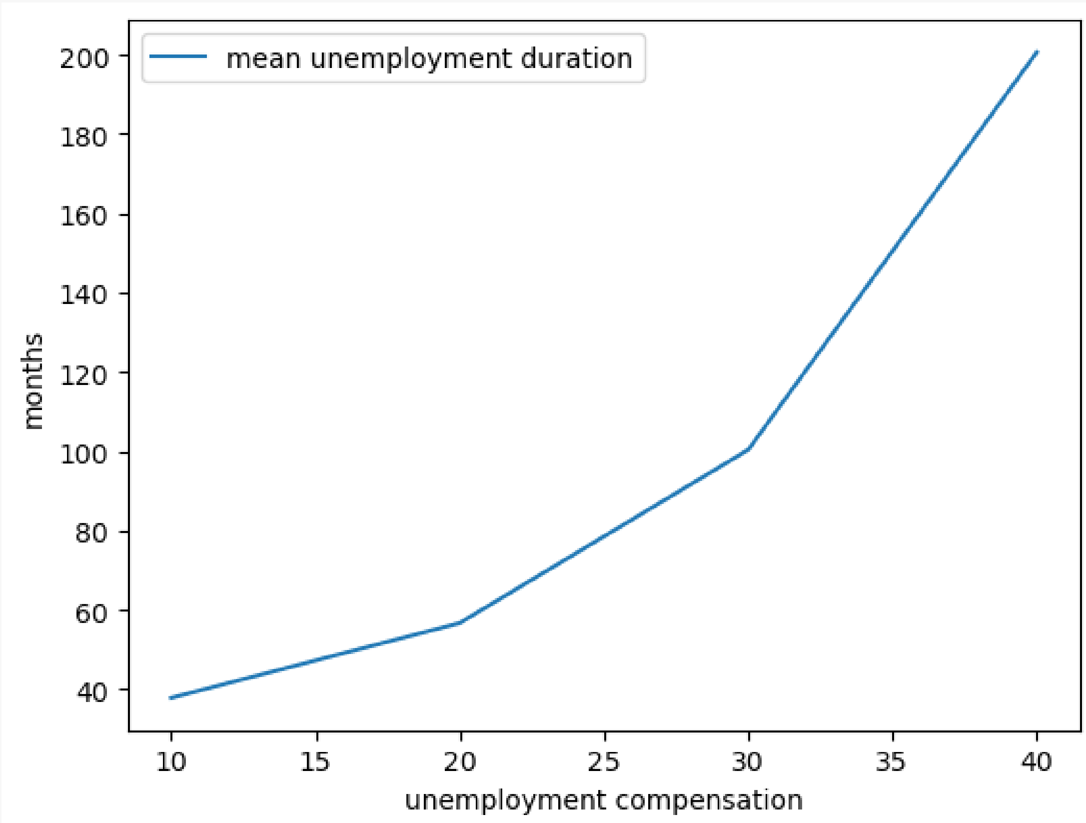
Solution

Here's a solution using the continuous wage offer distribution with JAX.

```
def compute_stopping_time_continuous(w_bar, key, model):
    """
    Compute stopping time by drawing wages from the continuous distribution
    until one exceeds `w_bar`.

    Parameters:
    ------------
    w_bar : float
        The reservation wage
    key : jax.random.PRNGKey
        Random key for JAX
    model : McCallModelContinuous
        The model containing wage draws

    Returns:
    --------
    t_final : int
        The stopping time (number of periods until acceptance)
    """
    c, β, σ, μ, w_draws = model

    def update(loop_state):
        t, key, accept = loop_state
        key, subkey = jax.random.split(key)
        # Draw a standard normal and transform to wage
        s = jax.random.normal(subkey)
        w = jnp.exp(μ + σ * s)
        accept = w >= w_bar
        t = t + 1
        return t, key, accept

    def cond(loop_state):
        _, _, accept = loop_state
        return jnp.logical_not(accept)

    initial_loop_state = (0, key, False)
```

```
    t_final, _, _ = jax.lax.while_loop(cond, update, initial_loop_state)
    return t_final


def compute_mean_stopping_time_continuous(w_bar, model, num_reps=100000,␣
↪seed=1234):
    """
    Generate a mean stopping time over `num_reps` repetitions.

    Parameters:
    ------------
    w_bar : float
        The reservation wage
    model : McCallModelContinuous
        The model containing parameters
    num_reps : int
        Number of simulation replications
    seed : int
        Random seed

    Returns:
    ---------
    mean_time : float
        Average stopping time across all replications
    """
    # Generate a key for each MC replication
    key = jax.random.PRNGKey(seed)
    keys = jax.random.split(key, num_reps)

    # Vectorize compute_stopping_time_continuous and evaluate across keys
    compute_fn = jax.vmap(compute_stopping_time_continuous, in_axes=(None, 0,␣
↪None))
    obs = compute_fn(w_bar, keys, model)

    # Return mean stopping time
    return jnp.mean(obs)


# Compute mean stopping time for different values of c
c_vals = jnp.linspace(10, 40, 4)

@jax.jit
def compute_stop_time_for_c_continuous(c):
    """Compute mean stopping time for a given compensation value c."""
    model = create_mccall_continuous(c=c)
    w_bar = compute_reservation_wage_continuous(model)
    return compute_mean_stopping_time_continuous(w_bar, model)

# Vectorize across all c values
compute_stop_time_vectorized = jax.vmap(compute_stop_time_for_c_continuous)
stop_times = compute_stop_time_vectorized(c_vals)

fig, ax = plt.subplots()

ax.plot(c_vals, stop_times, label="mean unemployment duration")
ax.set(xlabel="unemployment compensation", ylabel="months")
ax.legend()
```

```
plt.show()
```


mean unemployment duration
months
200
180
160
140
120
100
80
60
40
10
15
20
25
30
35
40
unemployment compensation

CHAPTER

FORTYFIVE

JOB SEARCH II: SEARCH AND SEPARATION

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

Contents

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon jax myst-nb
```

45.1 Overview

Previously *we looked* at the McCall job search model [McCall, 1970] as a way of understanding unemployment and worker decisions.

One unrealistic feature of that version of the model was that every job is permanent.

In this lecture, we extend the model by introducing job separation.

Once separation enters the picture, the agent comes to view

- the loss of a job as a capital loss, and
- a spell of unemployment as an *investment* in searching for an acceptable job

The other minor addition is that a utility function will be included to make worker preferences slightly more sophisticated.

We'll need the following imports

```
import matplotlib.pyplot as plt
import numpy as np
import jax
import jax.numpy as jnp
from typing import NamedTuple
from quantecon.distributions import BetaBinomial
from myst_nb import glue
```

45.2 The model

The model is similar to the *baseline McCall job search model*.

It concerns the life of an infinitely lived worker and

- the opportunities he or she (let's say he to save one character) has to work at different wages
- exogenous events that destroy his current job
- his decision making process while unemployed

The worker can be in one of two states: employed or unemployed.

He wants to maximize

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t u(y_t) \tag{45.1}$$

At this stage the only difference from the *baseline model* is that we've added some flexibility to preferences by introducing a utility function u.

It satisfies $u' > 0$ and $u'' < 0$.

Wage offers $\{W_t\}$ are IID with common distribution q.

The set of possible wage values is denoted by $\mathbb{W}$.

45.2.1 Timing and decisions

At the start of each period, the agent can be either

- unemployed or
- employed at some existing wage level w.

If currently employed at wage w, the worker

1. receives utility $u(w)$ from their current wage and
2. is fired with some (small) probability α, becoming unemployed next period.

If currently unemployed, the worker receives random wage offer W_t and either accepts or rejects.

If he accepts, then he begins work immediately at wage W_t.

If he rejects, then he receives unemployment compensation c.

The process then repeats.

> **Note**
>
> We do not allow for job search while employed—this topic is taken up in a *later lecture*.

45.3 Solving the model

We drop time subscripts in what follows and primes denote next period values.

Let

- $v_e(w)$ be maximum lifetime value for a worker who enters the current period employed with wage w
- $v_u(w)$ be maximum lifetime for a worker who who enters the current period unemployed and receives wage offer w.

Here, **maximum lifetime value** means the value of (45.1) when the worker makes optimal decisions at all future points in time.

As we now show, obtaining these functions is key to solving the model.

45.3.1 The Bellman equations

We recall that, in *the original job search model*, the value function (the value of being unemployed with a given wage offer) satisfied a Bellman equation.

Here this function again satisfies a Bellman equation that looks very similar.

$$v_u(w) = \max \left\{ v_e(w),\, u(c) + \beta \sum_{w' \in \mathbb{W}} v_u(w') q(w') \right\} \tag{45.2}$$

The difference is that the value of accepting is $v_e(w)$ rather than $w/(1-\beta)$.

We have to make this change because jobs are not permanent.

Accepting transitions the worker to employment and hence yields reward $v_e(w)$, which we discuss below.

Rejecting leads to unemployment compensation and unemployment tomorrow.

Equation (45.2) expresses the value of being unemployed with offer w in hand as a maximum over the value of two options: accept or reject the current offer.

The function v_e also satisfies a Bellman equation:

$$v_e(w) = u(w) + \beta \left[(1-\alpha) v_e(w) + \alpha \sum_{w' \in \mathbb{W}} v_u(w') q(w') \right] \tag{45.3}$$

Note

This equation differs from a traditional Bellman equation because there is no max.

There is no max because an employed agent has no choices.

Nonetheless, in keeping with most of the literature, we also refer to it as a Bellman equation.

Equation (45.3) expresses the value of being employed at wage w in terms of

- current reward $u(w)$ plus
- discounted expected reward tomorrow, given the α probability of being fired

As we will see, equations (45.3) and (45.2) provide enough information to solve for both v_e and v_u.

Once we have them in hand, we will be able to make optimal choices.

45.3.2 The reservation wage

Let

$$h := u(c) + \beta \sum_{w' \in \mathbb{W}} v_u(w') q(w') \tag{45.4}$$

This is the **continuation value** for an unemployed agent – the value of rejecting the current offer and then making optimal choices.

From (45.2), we see that an unemployed agent accepts current offer w if $v_e(w) \geq h$.

This means precisely that the value of accepting is higher than the value of rejecting.

The function v_e is increasing in w, since an employed agent is never made worse off by a higher current wage.

Hence, we can express the optimal choice as accepting wage offer w if and only if $w \geq \bar{w}$, where the **reservation wage** $\bar{w}$ is the first wage level $w \in \mathbb{W}$ such that

$$v_e(w) \geq h$$

45.4 Code

Let's now implement a solution method based on the two Bellman equations (45.2) and (45.3).

45.4.1 Set up

The default utility function is a CRRA utility function

```
def u(x, γ):
    return (x**(1 - γ) - 1) / (1 - γ)
```

Also, here's a default wage distribution, based around the BetaBinomial distribution:

```
n = 60                                  # n possible outcomes for w
w_default = jnp.linspace(10, 20, n)     # wages between 10 and 20
a, b = 600, 400                         # shape parameters
dist = BetaBinomial(n-1, a, b)          # distribution
q_default = jnp.array(dist.pdf())       # probabilities as a JAX array
```

Here's our model class for the McCall model with separation.

```
class Model(NamedTuple):
    α: float = 0.2                 # job separation rate
    β: float = 0.98                # discount factor
    γ: float = 2.0                 # utility parameter (CRRA)
    c: float = 6.0                 # unemployment compensation
    w: jnp.ndarray = w_default     # wage outcome space
    q: jnp.ndarray = q_default     # probabilities over wage offers
```

45.4.2 Operators

We'll use a similar iterative approach to solving the Bellman equations that we adopted in the *first job search lecture*.

As a first step, to iterate on the Bellman equations, we define two operators, one for each value function.

These operators take the current value functions as inputs and return updated versions.

```
def T_u(model, v_u, v_e):
    """
    Apply the unemployment Bellman update rule and return new guess of v_u.

    """
    α, β, γ, c, w, q = model
    h = u(c, γ) + β * (v_u @ q)
    v_u_new = jnp.maximum(v_e, h)
    return v_u_new
```

```
def T_e(model, v_u, v_e):
    """
    Apply the employment Bellman update rule and return new guess of v_e.

    """
    α, β, γ, c, w, q = model
    v_e_new = u(w, γ) + β * ((1 - α) * v_e + α * (v_u @ q))
    return v_e_new
```

45.4.3 Iteration

Now we write an iteration routine, which updates the pair of arrays v_u, v_e until convergence.

More precisely, we iterate until successive realizations are closer together than some small tolerance level.

```
def solve_full_model(
        model,
        tol: float = 1e-6,
        max_iter: int = 1_000,
    ):
    """
    Solves for both value functions v_u and v_e iteratively.

    """
    α, β, γ, c, w, q = model
    i = 0
    error = tol + 1
    v_e = v_u = w / (1 - β)

    while i < max_iter and error > tol:
        v_u_next = T_u(model, v_u, v_e)
        v_e_next = T_e(model, v_u, v_e)
        error_u = jnp.max(jnp.abs(v_u_next - v_u))
        error_e = jnp.max(jnp.abs(v_e_next - v_e))
        error = jnp.max(jnp.array([error_u, error_e]))
        v_u = v_u_next
        v_e = v_e_next
        i += 1

    return v_u, v_e
```

45.4.4 Computing the reservation wage

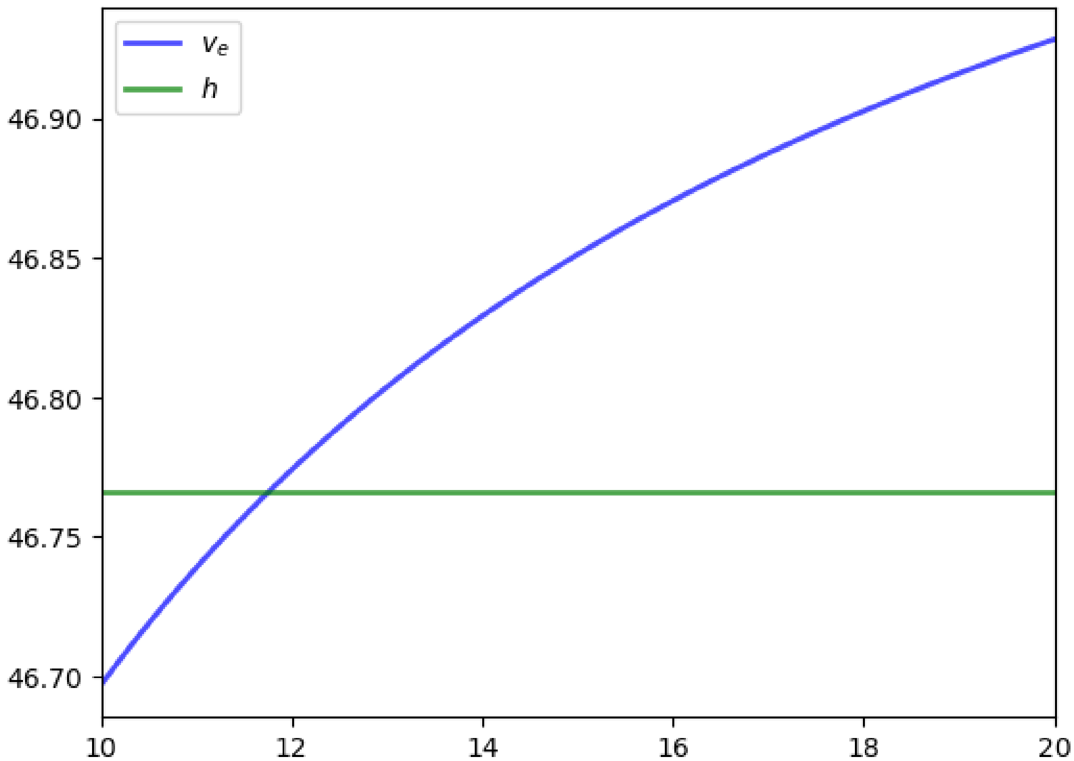
Now that we can solve for both value functions, let's investigate the reservation wage.

Recall from above that the reservation wage $\bar{w}$ is the first $w \in \mathbb{W}$ satisfying $v_e(w) \geq h$, where h is the continuation value defined in (45.4).

Let's compare v_e and h to see what they look like.

We'll use the default parameterizations found in the code above.

```
model = Model()
α, β, γ, c, w, q = model
v_u, v_e = solve_full_model(model)
h = u(c, γ) + β * (v_u @ q)

fig, ax = plt.subplots()
ax.plot(w, v_e, 'b-', lw=2, alpha=0.7, label='$v_e$')
ax.plot(w, [h] * len(w), 'g-', lw=2, alpha=0.7, label='$h$')
ax.set_xlim(min(w), max(w))
ax.legend()
plt.show()
```



The value v_e is increasing because higher w generates a higher wage flow conditional on staying employed.

The reservation wage is the w where these lines meet.

Let's compute this reservation wage explicitly:

```
def compute_reservation_wage_full(model):
    """
    Computes the reservation wage using the full model solution.
    """
    α, β, γ, c, w, q = model
    v_u, v_e = solve_full_model(model)
    h = u(c, γ) + β * (v_u @ q)
    # Find the first w such that v_e(w) >= h, or +inf if none exist
    accept = v_e >= h
    i = jnp.argmax(accept)  # returns first accept index
    w_bar = jnp.where(jnp.any(accept), w[i], jnp.inf)
    return w_bar

w_bar_full = compute_reservation_wage_full(model)
print(f"Reservation wage (full model): {w_bar_full:.4f}")
```

```
Reservation wage (full model): 11.8644
```

This value seems close to where the two lines meet.

45.5 A simplifying transformation

The approach above works, but iterating over two vector-valued functions is computationally expensive.

With some mathematics and some brain power, we can form a solution method that is far more efficient.

(This process will be analogous to our *second pass* at the plain vanilla McCall model, where we reduced the Bellman equation to an equation in an unknown scalar value, rather than an unknown vector.)

First, we use the continuation value h, as defined in (45.4), to write (45.2) as

$$v_u(w) = \max \{v_e(w), \, h\}$$

Taking the expectation of both sides and then discounting, this becomes

$$\beta \sum_{w'} v_u(w') q(w') = \beta \sum_{w'} \max \{v_e(w'), \, h\} \, q(w')$$

Adding $u(c)$ to both sides and using (45.4) again gives

$$h = u(c) + \beta \sum_{w'} \max \{v_e(w'), \, h\} \, q(w') \tag{45.5}$$

This is a nice scalar equation in the continuation value, which is already useful.

But we can go further, but eliminating v_e from the above equation.

45.5.1 Simplifying to a single equation

As a first step, we rearrange the expression defining h (see (45.4)) to obtain

$$\sum_{w'} v_u(w') q(w') = \frac{h - u(c)}{\beta}$$

Using this, the Bellman equation for v_e, as given in (45.3), can now be rewritten as

$$v_e(w) = u(w) + \beta \left[(1 - \alpha) v_e(w) + \alpha \frac{h - u(c)}{\beta} \right] \tag{45.6}$$

Our next step is to solve (45.6) for v_e as a function of h.

Rearranging (45.6) gives

$$v_e(w) = u(w) + \beta(1 - \alpha) v_e(w) + \alpha(h - u(c))$$

or

$$v_e(w) - \beta(1 - \alpha) v_e(w) = u(w) + \alpha(h - u(c))$$

Solving for $v_e(w)$ gives

$$v_e(w) = \frac{u(w) + \alpha(h - u(c))}{1 - \beta(1 - \alpha)} \tag{45.7}$$

Substituting this into (45.5) yields

$$h = u(c) + \beta \sum_{w' \in \mathbb{W}} \max \left\{ \frac{u(w') + \alpha(h - u(c))}{1 - \beta(1 - \alpha)}, \, h \right\} q(w') \tag{45.8}$$

Finally we have a single scalar equation in h!

If we can solve this for h, we can easily recover v_e using (45.7).

Then we have enough information to compute the reservation wage.

45.5.2 Solving the Bellman equations

To solve (45.8), we use the iteration rule

$$h_{n+1} = u(c) + \beta \sum_{w' \in \mathbb{W}} \max \left\{ \frac{u(w') + \alpha(h_n - u(c))}{1 - \beta(1 - \alpha)}, h_n \right\} q(w') \tag{45.9}$$

starting from some initial condition h_0.

(It is possible to prove that (45.9) converges via the Banach contraction mapping theorem.)

45.6 Implementation

To implement iteration on h, we provide a function that provides one update, from h_n to h_{n+1}

```
def update_h(model, h):
    " One update of the scalar h. "
    α, β, γ, c, w, q = model
    v_e = compute_v_e(model, h)
    h_new = u(c, γ) + β * (jnp.maximum(v_e, h) @ q)
    return h_new
```

Also, we provide a function to compute v_e from (45.7).

```
def compute_v_e(model, h):
    " Compute v_e from h using the closed-form expression. "
    α, β, γ, c, w, q = model
    return (u(w, γ) + α * (h - u(c, γ))) / (1 - β * (1 - α))
```

This function will be applied once convergence is achieved.

Now we can write our model solver.

```
@jax.jit
def solve_model(model, tol=1e-5, max_iter=2000):
    " Iterates to convergence on the Bellman equations. "

    def cond(loop_state):
        h, i, error = loop_state
        return jnp.logical_and(error > tol, i < max_iter)

    def update(loop_state):
        h, i, error = loop_state
        h_new = update_h(model, h)
        error_new = jnp.abs(h_new - h)
        return h_new, i + 1, error_new

    # Initialize
    h_init = u(model.c, model.γ) / (1 - model.β)
    i_init = 0
    error_init = tol + 1
    init_state = (h_init, i_init, error_init)

    final_state = jax.lax.while_loop(cond, update, init_state)
    h_final, _, _ = final_state
```

(continues on next page)

(continued from previous page)

```
    # Compute v_e from the converged h
    v_e_final = compute_v_e(model, h_final)

    return v_e_final, h_final
```

Finally, here's a function `compute_reservation_wage` that uses all the logic above, taking an instance of `Model` and returning the associated reservation wage.

```
def compute_reservation_wage(model):
    """
    Computes the reservation wage of an instance of the McCall model
    by finding the smallest w such that v_e(w) >= h.

    """
    # Find the first i such that v_e(w_i) >= h and return w[i]
    # If no such w exists, then w_bar is set to np.inf
    v_e, h = solve_model(model)
    accept = v_e >= h
    i = jnp.argmax(accept)   # take first accept index
    w_bar = jnp.where(jnp.any(accept), model.w[i], jnp.inf)
    return w_bar
```

Let's verify that this simplified approach gives the same answer as the full model:

```
w_bar_simplified = compute_reservation_wage(model)
print(f"Reservation wage (simplified): {w_bar_simplified:.4f}")
print(f"Reservation wage (full model): {w_bar_full:.4f}")
print(f"Difference: {abs(w_bar_simplified - w_bar_full):.6f}")
```

```
Reservation wage (simplified): 11.8644
Reservation wage (full model): 11.8644
Difference: 0.000000
```

As we can see, both methods produce essentially the same reservation wage.

However, the simplified method is far more efficient.

Next we will investigate how the reservation wage varies with parameters.

45.7 Impact of parameters

In each instance below, we'll show you a figure and then ask you to reproduce it in the exercises.

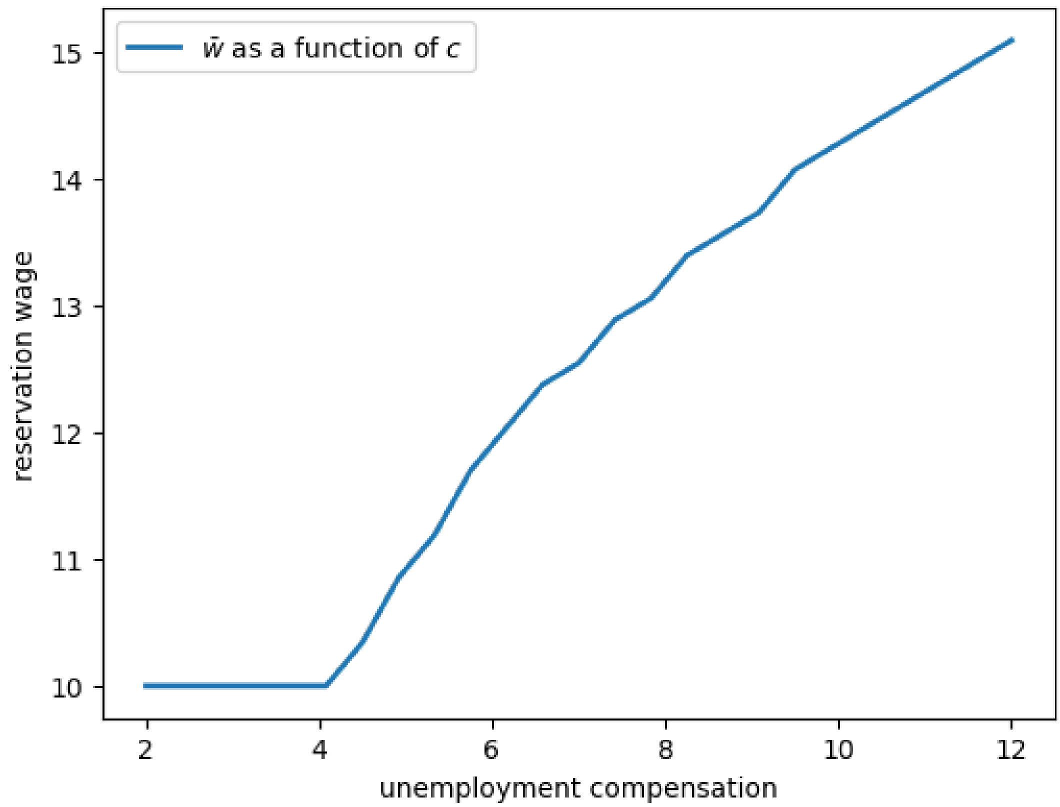
45.7.1 The reservation wage and unemployment compensation

First, let's look at how $\bar{w}$ varies with unemployment compensation.

In the figure below, we use the default parameters in the `Model` class, apart from c (which takes the values given on the horizontal axis)

As expected, higher unemployment compensation causes the worker to hold out for higher wages.

In effect, the cost of continuing job search is reduced.



45.7.2 The reservation wage and discounting

Next, let's investigate how $\bar{w}$ varies with the discount factor.

The next figure plots the reservation wage associated with different values of β

Again, the results are intuitive: More patient workers will hold out for higher wages.

45.7.3 The reservation wage and job destruction

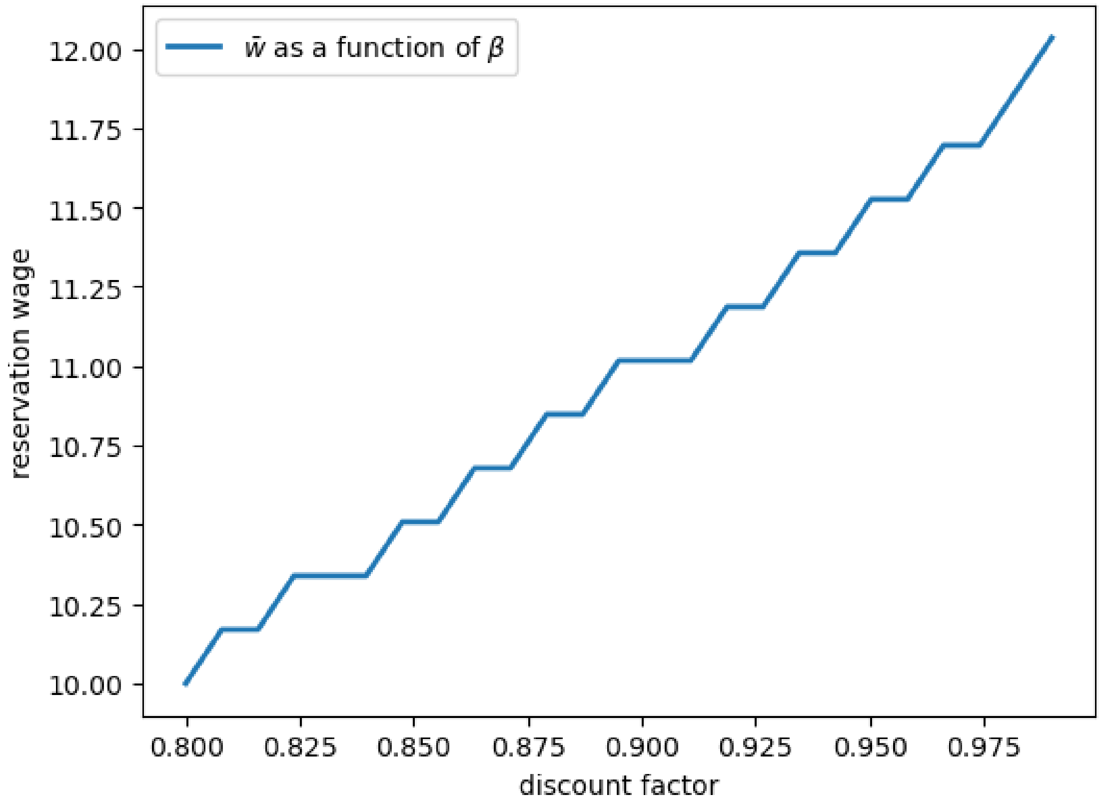
Finally, let's look at how $\bar{w}$ varies with the job separation rate α.

Higher α translates to a greater chance that a worker will face termination in each period once employed.

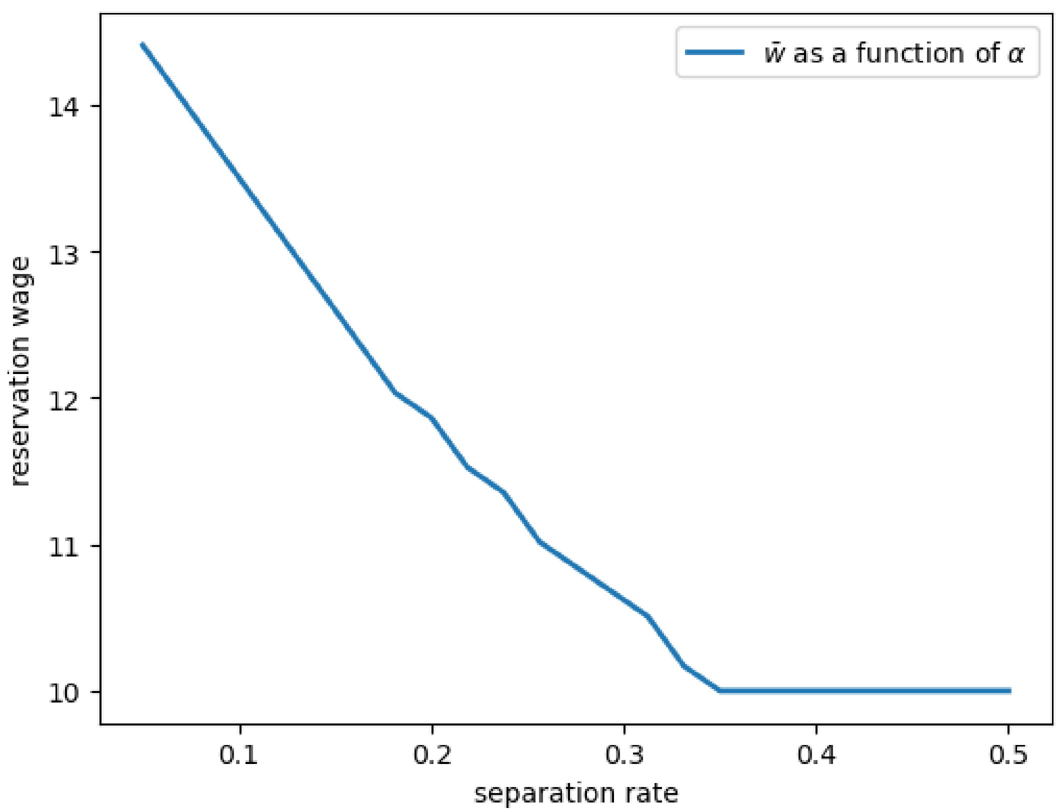
Once more, the results are in line with our intuition.

If the separation rate is high, then the benefit of holding out for a higher wage falls.

Hence the reservation wage is lower.



$\bar{w}$ as a function of β
reservation wage
discount factor
12.00
11.75
11.50
11.25
11.00
10.75
10.50
10.25
10.00
0.800
0.825
0.850
0.875
0.900
0.925
0.950
0.975



$\bar{w}$ as a function of α
reservation wage
separation rate
14
13
12
11
10
0.1
0.2
0.3
0.4
0.5

45.8 Exercises

Exercise 45.8.1

Reproduce all the reservation wage figures shown above.

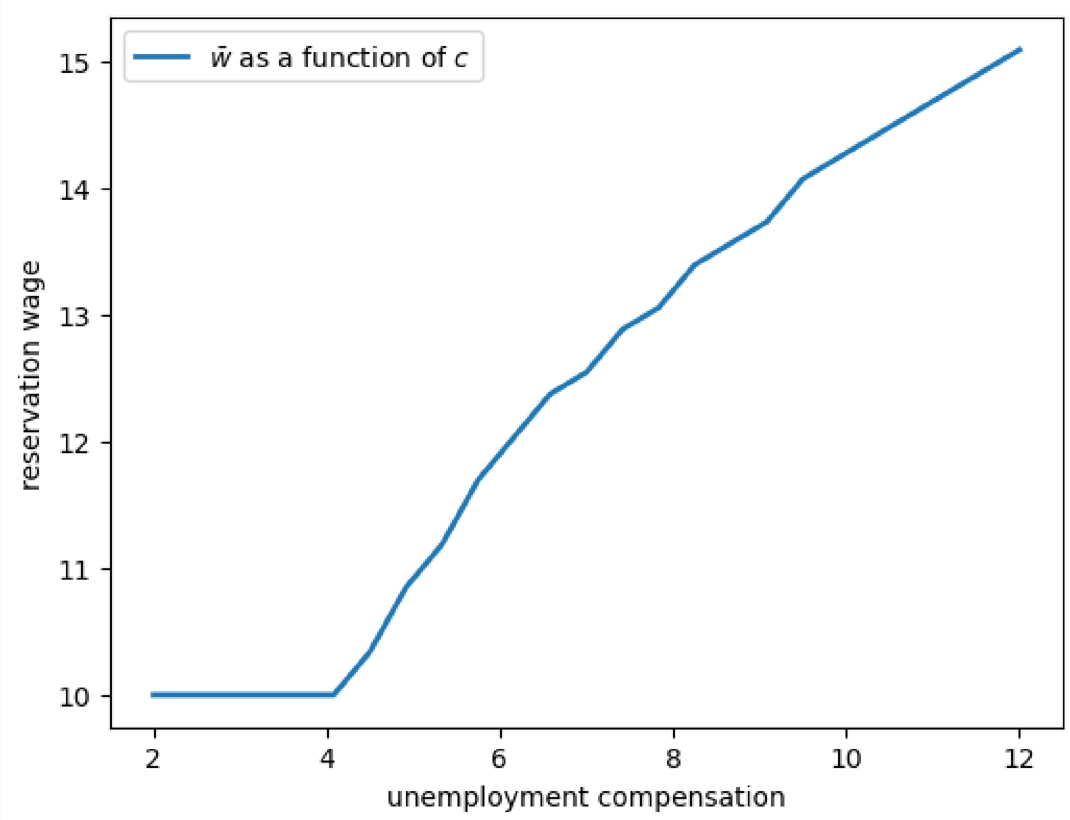
Regarding the values on the horizontal axis, use

```
grid_size = 25
c_vals = jnp.linspace(2, 12, grid_size)         # unemployment compensation
β_vals = jnp.linspace(0.8, 0.99, grid_size)     # discount factors
α_vals = jnp.linspace(0.05, 0.5, grid_size)     # separation rate
```

Solution

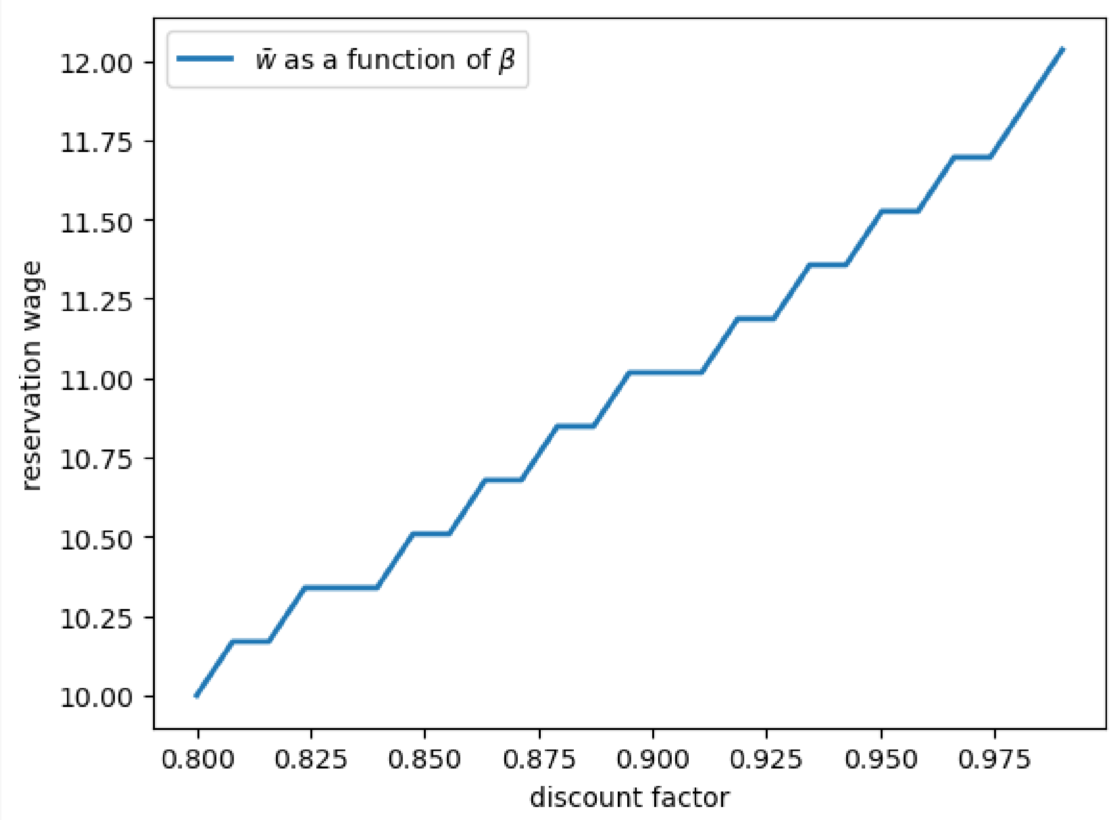
Here's the first figure.

```
def compute_res_wage_given_c(c):
    model = Model(c=c)
    w_bar = compute_reservation_wage(model)
    return w_bar

w_bar_vals = jax.vmap(compute_res_wage_given_c)(c_vals)

fig, ax = plt.subplots()
ax.set(xlabel='unemployment compensation', ylabel='reservation wage')
ax.plot(c_vals, w_bar_vals, lw=2, label=r'$\bar w$ as a function of $c$')
ax.legend()
glue("mccall_resw_c", fig, display=False)
plt.show()
```



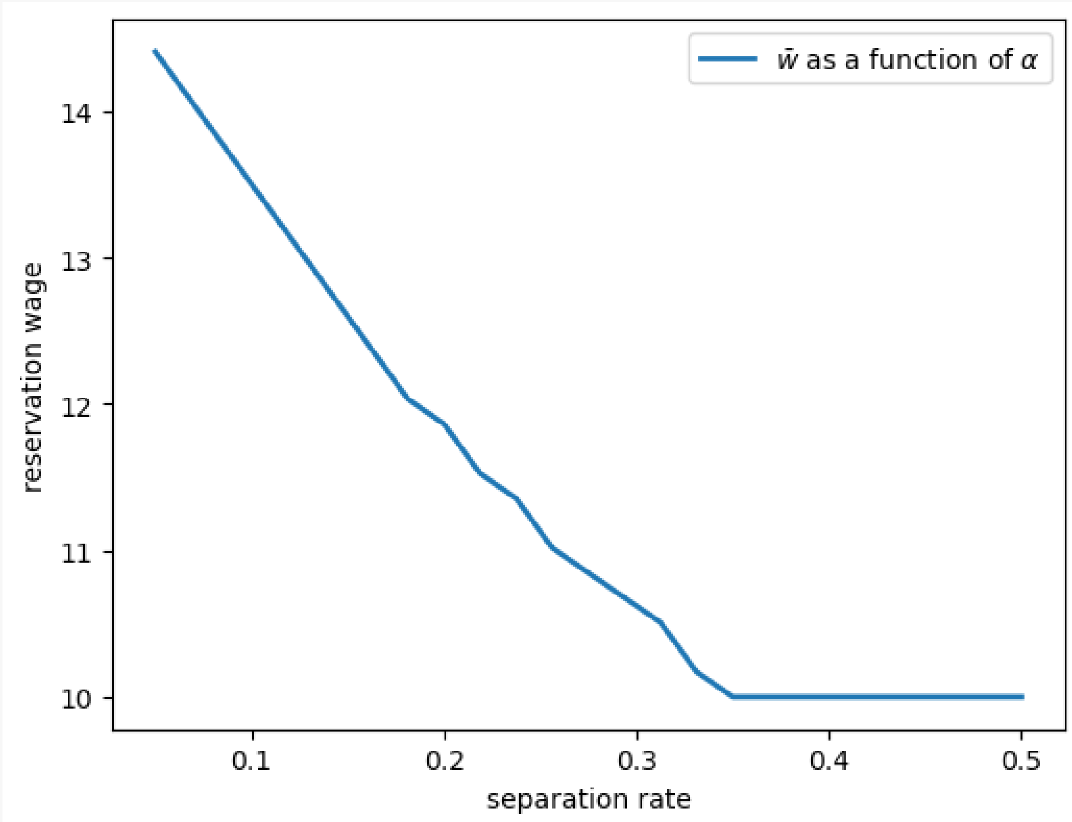
Here's the second one.

```
def compute_res_wage_given_beta(β):
    model = Model(β=β)
    w_bar = compute_reservation_wage(model)
    return w_bar

w_bar_vals = jax.vmap(compute_res_wage_given_beta)(β_vals)

fig, ax = plt.subplots()
ax.set(xlabel='discount factor', ylabel='reservation wage')
ax.plot(β_vals, w_bar_vals, lw=2, label=r'$\bar w$ as a function of $\beta$')
ax.legend()
glue("mccall_resw_beta", fig, display=False)
plt.show()
```



Here's the third.

```
def compute_res_wage_given_alpha(α):
    model = Model(α=α)
    w_bar = compute_reservation_wage(model)
    return w_bar

w_bar_vals = jax.vmap(compute_res_wage_given_alpha)(α_vals)

fig, ax = plt.subplots()
ax.set(xlabel='separation rate', ylabel='reservation wage')
ax.plot(α_vals, w_bar_vals, lw=2, label=r'$\bar w$ as a function of $\alpha$')
ax.legend()
glue("mccall_resw_alpha", fig, display=False)
plt.show()
```



$\bar{w}$ as a function of α
reservation wage
separation rate
14
13
12
11
10
0.1
0.2
0.3
0.4
0.5

CHAPTER

FORTYSIX

JOB SEARCH III: SEARCH WITH SEPARATION AND MARKOV WAGES

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

Contents

This lecture builds on the job search model with separation presented in the *previous lecture*.

The key difference is that wage offers now follow a *Markov chain* rather than being IID.

This modification adds persistence to the wage offer process, meaning that today's wage offer provides information about tomorrow's offer.

This feature makes the model more realistic, as labor market conditions tend to exhibit serial correlation over time.

In addition to what's in Anaconda, this lecture will need the following libraries

```
!pip install quantecon jax
```

We use the following imports:

```
from quantecon.markov import tauchen
import jax.numpy as jnp
import jax
from jax import lax
from typing import NamedTuple
import matplotlib.pyplot as plt
from functools import partial
```

46.1 Model setup

The setting is as follows:

- Each unemployed agent receives a wage offer w from a finite set $\mathbb{W}$
- Wage offers follow a Markov chain with transition matrix P
- Jobs terminate with probability α each period (separation rate)
- Unemployed workers receive compensation c per period
- Future payoffs are discounted by factor $\beta \in (0, 1)$

46.1.1 Decision problem

When unemployed and receiving wage offer w, the agent chooses between:

1. Accept offer w: Become employed at wage w
2. Reject offer: Remain unemployed, receive c, get new offer next period

The wage updates are as follows:

- If an unemployed agent rejects offer w, then their next offer is drawn from $P(w, \cdot)$
- If an employed agent loses a job in which they were paid wage w, then their next offer is drawn from $P(w, \cdot)$

46.1.2 The wage offer process

To construct the wage offer process we start with an AR1 process.

$$X_{t+1} = \rho X_t + \nu Z_{t+1}$$

where $\{Z_t\}$ is IID and standard normal.

Below we will always choose $\rho \in (0, 1)$.

This means that the wage process will be positively correlated: the higher the current wage offer, the more likely we are to get a high offer tomorrow.

To go from the AR1 process to the wage offer process, we set $W_t = \exp(X_t)$.

Actually, in practice, we approximate this wage process as follows:

- discretize the AR1 process using *Tauchen's method* and

- take the exponential of the resulting wage offer values.

46.1.3 Value functions

We let

- $v_u(w)$ be the value of being unemployed when current wage offer is w
- $v_e(w)$ be the value of being employed at wage w

The Bellman equations are obvious modifications of the *IID case*.

The only change is that expectations for next period are computed using the transition matrix P conditioned on current wage w, instead of being drawn independently from q.

The unemployed worker's value function satisfies the Bellman equation

$$v_u(w) = \max \left\{ v_e(w), u(c) + \beta \sum_{w'} v_u(w') P(w, w') \right\}$$

The employed worker's value function satisfies the Bellman equation

$$v_e(w) = u(w) + \beta \left[\alpha \sum_{w'} v_u(w') P(w, w') + (1 - \alpha) v_e(w) \right]$$

As a matter of notation, given a function h assigning values to wages, it is common to set

$$(Ph)(w) = \sum_{w'} h(w') P(w, w')$$

(To understand this expression, think of P as a matrix, h as a column vector, and w as a row index.)

With this notation, the Bellman equations become

$$v_u(w) = \max\{v_e(w), u(c) + \beta(Pv_u)(w)\}$$

and

$$v_e(w) = u(w) + \beta \left[\alpha(Pv_u)(w) + (1 - \alpha)v_e(w)\right]$$

46.1.4 Optimal policy

Once we have the solutions v_e and v_u to these Bellman equations, we can compute the optimal policy: accept at current wage offer w if

$$v_e(w) \geq u(c) + \beta(Pv_u)(w)$$

The optimal policy turns out to be a reservation wage strategy: accept all wages above some threshold.

46.2 Code

Let's now implement the model.

46.2.1 Set up

The default utility function is a CRRA utility function

```
def u(x, γ):
    return (x**(1 - γ) - 1) / (1 - γ)
```

Let's set up a `Model` class to store information needed to solve the model.

We include `P_cumsum`, the row-wise cumulative sum of the transition matrix, to optimize simulation – the details are explained below.

```
class Model(NamedTuple):
    n: int
    w_vals: jnp.ndarray
    P: jnp.ndarray
    P_cumsum: jnp.ndarray
    β: float
    c: float
    α: float
    γ: float
```

The next function holds default values and creates a `Model` instance:

```
def create_js_with_sep_model(
        n: int = 200,          # wage grid size
        ρ: float = 0.9,        # wage persistence
        ν: float = 0.2,        # wage volatility
        β: float = 0.96,       # discount factor
        α: float = 0.05,       # separation rate
        c: float = 1.0,        # unemployment compensation
        γ: float = 1.5         # utility parameter
    ) -> Model:
    """
    Creates an instance of the job search model with separation.

    """
    mc = tauchen(n, ρ, ν)
    w_vals, P = jnp.exp(jnp.array(mc.state_values)), jnp.array(mc.P)
    P_cumsum = jnp.cumsum(P, axis=1)
    return Model(n, w_vals, P, P_cumsum, β, c, α, γ)
```

46.2.2 Solution: first pass

Let's put together a (not very efficient) routine for calculating the reservation wage.

(We will think carefully about efficiency below.)

It works by starting with guesses for v_e and v_u and iterating to convergence.

Here are Bellman operators that update v_u and v_e respectively.

```
def T_u(model, v_u, v_e):
    """
    Apply the unemployment Bellman update rule and return new guess of v_u.

    """
```

(continues on next page)

(continued from previous page)

```
    n, w_vals, P, P_cumsum, β, c, α, γ = model
    h = u(c, γ) + β * P @ v_u
    v_u_new = jnp.maximum(v_e, h)
    return v_u_new
```

```
def T_e(model, v_u, v_e):
    """
    Apply the employment Bellman update rule and return new guess of v_e.

    """
    n, w_vals, P, P_cumsum, β, c, α, γ = model
    v_e_new = u(w_vals, γ) + β * ((1 - α) * v_e + α * P @ v_u)
    return v_e_new
```

Here's a routine to iterate to convergence and then compute the reservation wage.

```
def solve_model_first_pass(
        model: Model,            # instance containing default parameters
        v_u_init: jnp.ndarray,   # initial condition for v_u
        v_e_init: jnp.ndarray,   # initial condition for v_e
        tol: float=1e-6,         # error tolerance
        max_iter: int=1_000,     # maximum number of iterations for loop
    ):
    n, w_vals, P, P_cumsum, β, c, α, γ = model
    i = 0
    error = tol + 1
    v_u = v_u_init
    v_e = v_e_init

    while i < max_iter and error > tol:
        v_u_next = T_u(model, v_u, v_e)
        v_e_next = T_e(model, v_u, v_e)
        error_u = jnp.max(jnp.abs(v_u_next - v_u))
        error_e = jnp.max(jnp.abs(v_e_next - v_e))
        error = jnp.maximum(error_u, error_e)
        v_u = v_u_next
        v_e = v_e_next
        i += 1

    # Compute accept and reject values
    continuation_values = u(c, γ) + β * P @ v_u

    # Find where acceptance becomes optimal
    accept_indices = v_e >= continuation_values
    first_accept_idx = jnp.argmax(accept_indices)   # index of first True

    # If no acceptance (all False), return infinity
    # Otherwise return the wage at the first acceptance index
    w_bar = jnp.where(
        jnp.any(accept_indices), w_vals[first_accept_idx], jnp.inf
    )
    return v_u, v_e, w_bar
```

46.2.3 Road test

Let's solve the model:

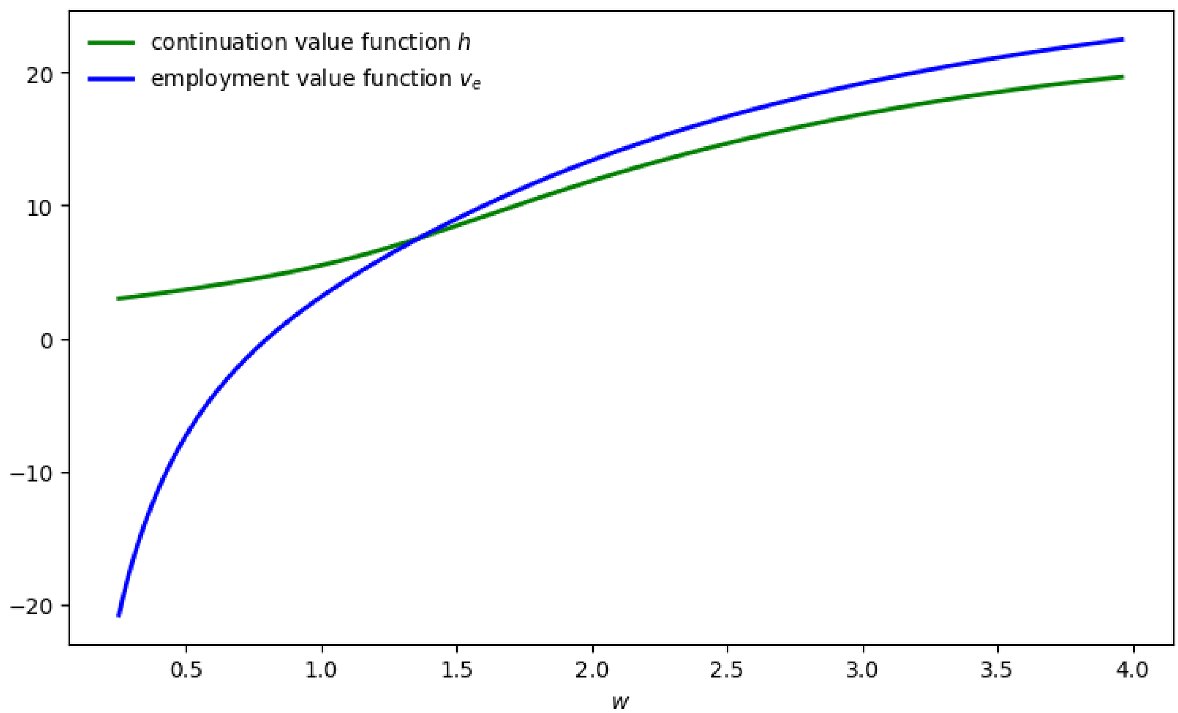
```
model = create_js_with_sep_model()
n, w_vals, P, P_cumsum, β, c, α, γ = model
v_u_init = jnp.zeros(n)
v_e_init = jnp.zeros(n)
v_u, v_e, w_bar_first = solve_model_first_pass(model, v_u_init, v_e_init)
```

Next we compute the continuation values.

```
h = u(c, γ) + β * P @ v_u
```

Let's plot our results.

```
fig, ax = plt.subplots(figsize=(9, 5.2))
ax.plot(w_vals, h, 'g-', linewidth=2,
        label="continuation value function $h$")
ax.plot(w_vals, v_e, 'b-', linewidth=2,
        label="employment value function $v_e$")
ax.legend(frameon=False)
ax.set_xlabel(r"$w$")
plt.show()
```



The reservation wage is at the intersection of v_e, and the continuation value function, which is the value of rejecting.

46.3 Improving efficiency

The solution method described above works fine but we can do much better.

First, we use the employed worker's Bellman equation to express v_e in terms of Pv_u

$$v_e(w) = \frac{1}{1 - \beta(1 - \alpha)} \cdot (u(w) + \alpha\beta(Pv_u)(w))$$

Next we substitute into the unemployed agent's Bellman equation to get

$$v_u(w) = \max\left\{\frac{1}{1 - \beta(1 - \alpha)} \cdot (u(w) + \alpha\beta(Pv_u)(w)), u(c) + \beta(Pv_u)(w)\right\}$$

Then we use value function iteration to solve for v_u.

With v_u in hand, we can recover v_e through the equations above and then compute the reservation wage.

Here's the new Bellman operator for the unemployed worker's value function:

```
def T(v: jnp.ndarray, model: Model) -> jnp.ndarray:
    """
    The Bellman operator for v_u.

    """
    n, w_vals, P, P_cumsum, β, c, α, γ = model
    d = 1 / (1 - β * (1 - α))
    v_e = d * (u(w_vals, γ) + α * β * P @ v)
    h = u(c, γ) + β * P @ v
    return jnp.maximum(v_e, h)
```

Here's a routine for value function iteration.

```
@jax.jit
def vfi(
        model: Model,
        tolerance: float = 1e-6,    # Error tolerance
        max_iter: int = 100_000,    # Max iteration bound
    ):

    v_init = jnp.zeros(model.w_vals.shape)

    def cond(loop_state):
        v, error, i = loop_state
        return (error > tolerance) & (i <= max_iter)

    def update(loop_state):
        v, error, i = loop_state
        v_new = T(v, model)
        error = jnp.max(jnp.abs(v_new - v))
        new_loop_state = v_new, error, i + 1
        return new_loop_state

    initial_state = (v_init, tolerance + 1, 1)
    final_loop_state = lax.while_loop(cond, update, initial_state)
    v_final, error, i = final_loop_state

    return v_final
```

Here is a routine that computes the reservation wage from the value function.

```
@jax.jit
def get_reservation_wage(v: jnp.ndarray, model: Model) -> float:
    """
    Calculate the reservation wage from the unemployed agents
    value function v := v_u.

    The reservation wage is the lowest wage w where accepting (v_e(w))
    is at least as good as rejecting (u(c) + β(Pv_u)(w)).

    """
    n, w_vals, P, P_cumsum, β, c, α, γ = model

    # Compute accept and reject values
    d = 1 / (1 - β * (1 - α))
    v_e = d * (u(w_vals, γ) + α * β * P @ v)
    continuation_values = u(c, γ) + β * P @ v

    # Find where acceptance becomes optimal
    accept_indices = v_e >= continuation_values
    first_accept_idx = jnp.argmax(accept_indices)  # index of first True

    # If no acceptance (all False), return infinity
    # Otherwise return the wage at the first acceptance index
    return jnp.where(jnp.any(accept_indices), w_vals[first_accept_idx], jnp.inf)
```

Let's solve the model using our new method:

```
model = create_js_with_sep_model()
n, w_vals, P, P_cumsum, β, c, α, γ = model
v_u = vfi(model)
w_bar = get_reservation_wage(v_u, model)
```

Let's verify that both methods produce the same reservation wage:

```
print(f"Reservation wage (first method):  {w_bar_first:.6f}")
print(f"Reservation wage (second method): {w_bar:.6f}")
print(f"Difference: {abs(w_bar - w_bar_first):.2e}")
```

```
Reservation wage (first method):  1.365155
Reservation wage (second method): 1.365155
Difference: 0.00e+00
```

Next we compute some related quantities for plotting.

```
d = 1 / (1 - β * (1 - α))
v_e = d * (u(w_vals, γ) + α * β * P @ v_u)
h = u(c, γ) + β * P @ v_u
```

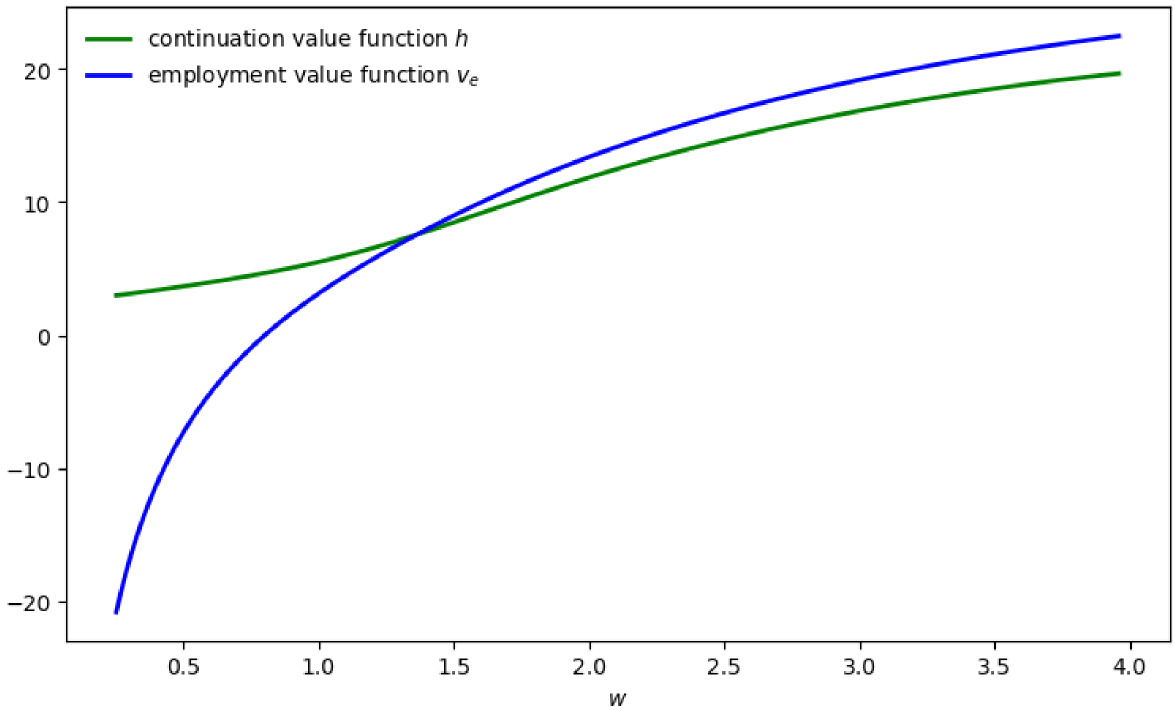
Let's plot our results.

```
fig, ax = plt.subplots(figsize=(9, 5.2))
ax.plot(w_vals, h, 'g-', linewidth=2,
        label="continuation value function $h$")
ax.plot(w_vals, v_e, 'b-', linewidth=2,
        label="employment value function $v_e$")
ax.legend(frameon=False)
```

(continues on next page)

(continued from previous page)

```
ax.set_xlabel(r"$w$")
plt.show()
```

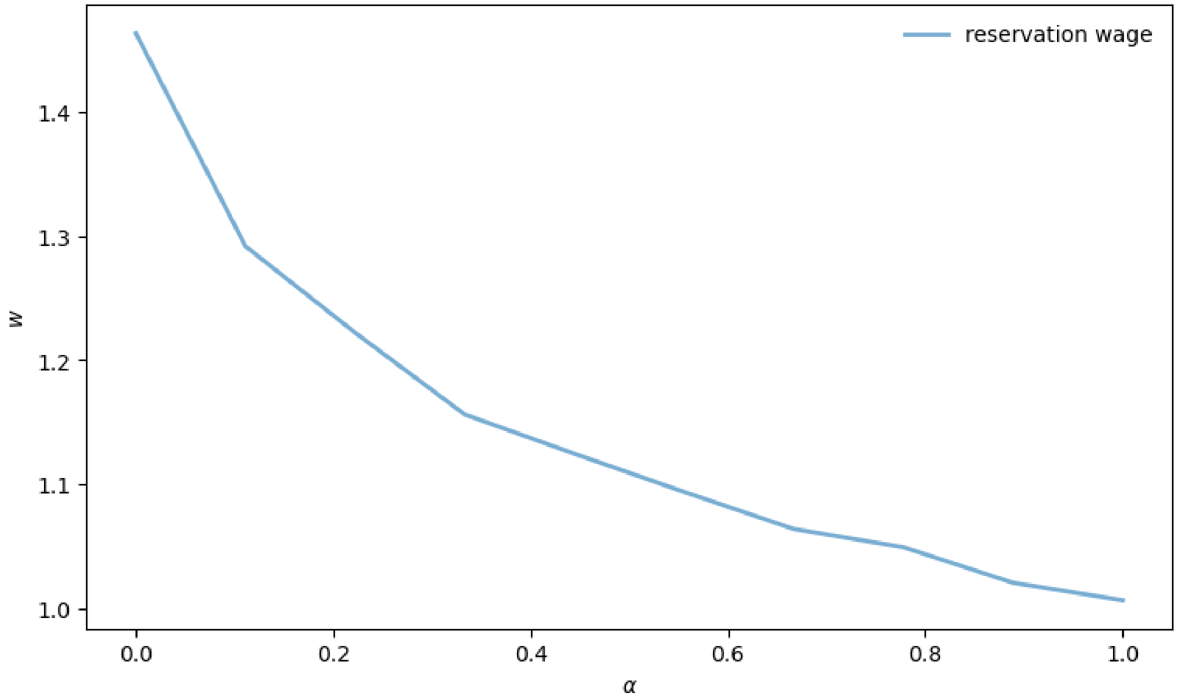


The result is the same as before but we only iterate on one array — and also our JAX code is more efficient.

46.4 Sensitivity analysis

Let's examine how reservation wages change with the separation rate.

```
α_vals: jnp.ndarray = jnp.linspace(0.0, 1.0, 10)

w_bar_vec = []
for α in α_vals:
    model = create_js_with_sep_model(α=α)
    v_u = vfi(model)
    w_bar = get_reservation_wage(v_u, model)
    w_bar_vec.append(w_bar)

fig, ax = plt.subplots(figsize=(9, 5.2))
ax.plot(
    α_vals, w_bar_vec, linewidth=2, alpha=0.6, label="reservation wage"
)
ax.legend(frameon=False)
ax.set_xlabel(r"$\alpha$")
ax.set_ylabel(r"$w$")
plt.show()
```



Can you provide an intuitive economic story behind the outcome that you see in this figure?

46.5 Employment simulation

Now let's simulate the employment dynamics of a single agent under the optimal policy.

Note that, when simulating the Markov chain for wage offers, we need to draw from the distribution in each row of P many times.

To do this, we use the inverse transform method: draw a uniform random variable and find where it falls in the cumulative distribution.

This is implemented via `jnp.searchsorted` on the precomputed cumulative sum `P_cumsum`, which is much faster than recomputing the cumulative sum each time.

The function `update_agent` advances the agent's state by one period.

The agent's state is a pair (S_t, W_t), where S_t is employment status (0 if unemployed, 1 if employed) and W_t is

- their current wage offer, if unemployed, or
- their current wage, if employed.

```
def update_agent(key, status, wage_idx, model, w_bar):
    """
    Updates an agent's employment status and current wage.

    Parameters:
    - key: JAX random key
    - status: Current employment status (0 or 1)
    - wage_idx: Current wage, recorded as an array index
    - model: Model instance
    - w_bar: Reservation wage
```

(continues on next page)

(continued from previous page)

```
    """
    n, w_vals, P, P_cumsum, β, c, α, γ = model

    key1, key2 = jax.random.split(key)
    # Use precomputed cumulative sum for efficient sampling
    # via the inverse transform method.
    new_wage_idx = jnp.searchsorted(
        P_cumsum[wage_idx, :], jax.random.uniform(key1)
    )
    separation_occurs = jax.random.uniform(key2) < α
    # Accept if current wage meets or exceeds reservation wage
    accepts = w_vals[wage_idx] >= w_bar

    # If employed: status = 1 if no separation, 0 if separation
    # If unemployed: status = 1 if accepts, 0 if rejects
    next_status = jnp.where(
        status,
        1 - separation_occurs.astype(jnp.int32),   # employed path
        accepts.astype(jnp.int32)                  # unemployed path
    )

    # If employed: wage = current if no separation, new if separation
    # If unemployed: wage = current if accepts, new if rejects
    next_wage = jnp.where(
        status,
        jnp.where(separation_occurs, new_wage_idx, wage_idx),  # employed path
        jnp.where(accepts, wage_idx, new_wage_idx)             # unemployed path
    )

    return next_status, next_wage
```

Here's a function to simulate the employment path of a single agent.

```
def simulate_employment_path(
        model: Model,     # Model details
        w_bar: float,    # Reservation wage
        T: int = 2_000,   # Simulation length
        seed: int = 42    # Set seed for simulation
    ):
    """
    Simulate employment path for T periods starting from unemployment.

    """
    key = jax.random.PRNGKey(seed)
    # Unpack model
    n, w_vals, P, P_cumsum, β, c, α, γ = model

    # Initial conditions
    status = 0
    wage_idx = 0

    wage_path = []
    status_path = []

    for t in range(T):
        wage_path.append(w_vals[wage_idx])
```

(continues on next page)

(continued from previous page)

```
        status_path.append(status)

        key, subkey = jax.random.split(key)
        status, wage_idx = update_agent(
            subkey, status, wage_idx, model, w_bar
        )

    return jnp.array(wage_path), jnp.array(status_path)
```

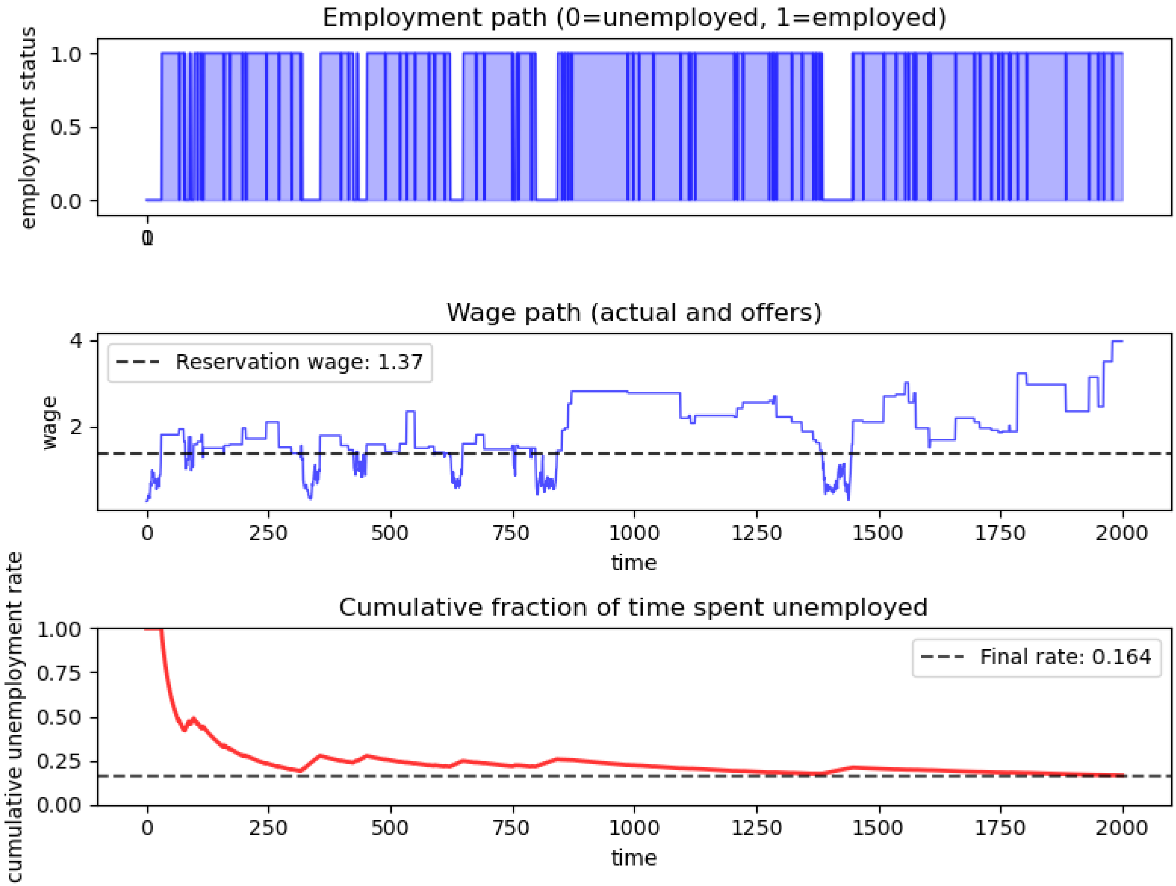
Let's create a comprehensive plot of the employment simulation:

```
model = create_js_with_sep_model()

# Calculate reservation wage for plotting
v_u = vfi(model)
w_bar = get_reservation_wage(v_u, model)

wage_path, employment_status = simulate_employment_path(model, w_bar)

fig, (ax1, ax2, ax3) = plt.subplots(3, 1, figsize=(8, 6))

# Plot employment status
ax1.plot(employment_status, 'b-', alpha=0.7, linewidth=1)
ax1.fill_between(
    range(len(employment_status)), employment_status, alpha=0.3, color='blue'
)
ax1.set_ylabel('employment status')
ax1.set_title('Employment path (0=unemployed, 1=employed)')
ax1.set_xticks((0, 1))
ax1.set_ylim(-0.1, 1.1)

# Plot wage path with employment status coloring
ax2.plot(wage_path, 'b-', alpha=0.7, linewidth=1)
ax2.axhline(y=w_bar, color='black', linestyle='--', alpha=0.8,
           label=f'Reservation wage: {w_bar:.2f}')
ax2.set_xlabel('time')
ax2.set_ylabel('wage')
ax2.set_title('Wage path (actual and offers)')
ax2.legend()

# Plot cumulative fraction of time unemployed
unemployed_indicator = (employment_status == 0).astype(int)
cumulative_unemployment = (
    jnp.cumsum(unemployed_indicator) /
    jnp.arange(1, len(employment_status) + 1)
)

ax3.plot(cumulative_unemployment, 'r-', alpha=0.8, linewidth=2)
ax3.axhline(y=jnp.mean(unemployed_indicator), color='black',
            linestyle='--', alpha=0.7,
            label=f'Final rate: {jnp.mean(unemployed_indicator):.3f}')
ax3.set_xlabel('time')
ax3.set_ylabel('cumulative unemployment rate')
ax3.set_title('Cumulative fraction of time spent unemployed')
ax3.legend()
ax3.set_ylim(0, 1)
```

(continues on next page)

(continued from previous page)

```
plt.tight_layout()
plt.show()
```



The simulation helps to visualize outcomes associated with this model.

The agent follows a reservation wage strategy.

Often the agent loses her job and immediately takes another job at a different wage.

This is because she uses the wage w from her last job to draw a new wage offer via $P(w, \cdot)$, and positive correlation means that a high current w is often leads a high new draw.

46.6 Ergodic property

Below we examine cross-sectional unemployment.

In particular, we will look at the unemployment rate in a cross-sectional simulation and compare it to the time-average unemployment rate, which is the fraction of time an agent spends unemployed over a long time series.

We will see that these two values are approximately equal – in fact they are exactly equal in the limit.

The reason is that the process (S_t, W_t), where

- S_t is the employment status and
- W_t is the wage

is Markovian, since the next pair depends only on the current pair and iid randomness, and ergodic.

Ergodicity holds as a result of irreducibility.

Indeed, from any (status, wage) pair, an agent can eventually reach any other (status, wage) pair.

This holds because:

- Unemployed agents can become employed by accepting offers
- Employed agents can become unemployed through separation (probability α)
- The wage process can transition between all wage states (because P is itself irreducible)

These properties ensure the chain is ergodic with a unique stationary distribution π over states (s, w).

For an ergodic Markov chain, the ergodic theorem guarantees that time averages = cross-sectional averages.

In particular, the fraction of time a single agent spends unemployed (across all wage states) converges to the cross-sectional unemployment rate:

$$\lim_{T\to\infty} \frac{1}{T} \sum_{t=1}^{T} \mathbb{1}\{S_t = \text{unemployed}\} = \sum_{w=1}^{n} \pi(\text{unemployed}, w)$$

This holds regardless of initial conditions – provided that we burn in the cross-sectional distribution (run it forward in time from a given initial cross section in order to remove the influence of that initial condition).

As a result, we can study steady-state unemployment either by:

- Following one agent for a long time (time average), or
- Observing many agents at a single point in time (cross-sectional average)

Often the second approach is better for our purposes, since it's easier to parallelize.

46.7 Cross-sectional analysis

Now let's simulate many agents simultaneously to examine the cross-sectional unemployment rate.

To do this efficiently, we need a different approach than `simulate_employment_path` defined above.

The key differences are:

- `simulate_employment_path` records the entire history (all T periods) for a single agent, which is useful for visualization but memory-intensive
- The new function `sim_agent` below only tracks and returns the final state, which is all we need for cross-sectional statistics
- `sim_agent` uses `lax.fori_loop` instead of a Python loop, making it JIT-compilable and suitable for vectorization across many agents

We first define a function that simulates a single agent forward T time steps:

```
@jax.jit
def sim_agent(key, initial_status, initial_wage_idx, model, w_bar, T):
    """
    Simulate a single agent forward T time steps using lax.fori_loop.

    Uses fold_in to generate a new key at each time step.

    Parameters:
```

(continues on next page)

(continued from previous page)

```
    - key: JAX random key for this agent
    - initial_status: Initial employment status (0 or 1)
    - initial_wage_idx: Initial wage index
    - model: Model instance
    - w_bar: Reservation wage
    - T: Number of time periods to simulate

    Returns:
    - final_status: Employment status after T periods
    - final_wage_idx: Wage index after T periods
    """
    def update(t, loop_state):
        status, wage_idx = loop_state
        step_key = jax.random.fold_in(key, t)
        status, wage_idx = update_agent(step_key, status, wage_idx, model, w_bar)
        return status, wage_idx

    initial_loop_state = (initial_status, initial_wage_idx)
    final_loop_state = lax.fori_loop(0, T, update, initial_loop_state)
    final_status, final_wage_idx = final_loop_state
    return final_status, final_wage_idx


# Create vectorized version of sim_agent to process multiple agents in parallel
sim_agents_vmap = jax.vmap(sim_agent, in_axes=(0, 0, 0, None, None, None))


def simulate_cross_section(
        model: Model,               # Model instance with parameters
        n_agents: int = 100_000,    # Number of agents to simulate
        T: int = 200,               # Length of burn-in
        seed: int = 42              # For reproducibility
    ) -> float:
    """
    Simulate cross-section of agents and return unemployment rate.

    This approach:
    1. Generates n_agents random keys
    2. Calls sim_agent for each agent (vectorized via vmap)
    3. Collects the final states to produce the cross-section

    Returns the cross-sectional unemployment rate.
    """
    key = jax.random.PRNGKey(seed)

    # Solve for optimal reservation wage
    v_u = vfi(model)
    w_bar = get_reservation_wage(v_u, model)

    # Initialize arrays
    initial_wage_indices = jnp.zeros(n_agents, dtype=jnp.int32)
    initial_status_vec = jnp.zeros(n_agents, dtype=jnp.int32)

    # Generate n_agents random keys
    agent_keys = jax.random.split(key, n_agents)

    # Simulate each agent forward T steps (vectorized)
```

(continues on next page)

(continued from previous page)

```
    final_status, final_wage_idx = sim_agents_vmap(
        agent_keys, initial_status_vec, initial_wage_indices, model, w_bar, T
    )

    unemployment_rate = 1 - jnp.mean(final_status)
    return unemployment_rate
```

This function generates a histogram showing the distribution of employment status across many agents:

```
def plot_cross_sectional_unemployment(
        model: Model,
        t_snapshot: int = 200,     # Time of cross-sectional snapshot
        n_agents: int = 20_000     # Number of agents to simulate
    ):
    """
    Generate histogram of cross-sectional unemployment at a specific time.

    """
    # Get final employment state directly
    key = jax.random.PRNGKey(42)
    v_u = vfi(model)
    w_bar = get_reservation_wage(v_u, model)

    # Initialize arrays
    initial_wage_indices = jnp.zeros(n_agents, dtype=jnp.int32)
    initial_status_vec = jnp.zeros(n_agents, dtype=jnp.int32)

    # Generate n_agents random keys
    agent_keys = jax.random.split(key, n_agents)

    # Simulate each agent forward T steps (vectorized)
    final_status, _ = sim_agents_vmap(
        agent_keys, initial_status_vec, initial_wage_indices, model, w_bar, t_snapshot
    )

    # Calculate unemployment rate
    unemployment_rate = 1 - jnp.mean(final_status)

    fig, ax = plt.subplots(figsize=(8, 5))

    # Plot histogram as density (bars sum to 1)
    weights = jnp.ones_like(final_status) / len(final_status)
    ax.hist(final_status, bins=[-0.5, 0.5, 1.5],
            alpha=0.7, color='blue', edgecolor='black',
            density=True, weights=weights)

    ax.set_xlabel('employment status (0=unemployed, 1=employed)')
    ax.set_ylabel('density')
    ax.set_title(f'Cross-sectional distribution at t={t_snapshot}, ' +
                 f'unemployment rate = {unemployment_rate:.3f}')
    ax.set_xticks([0, 1])

    plt.tight_layout()
    plt.show()
```

Now let's compare the time-average unemployment rate (from a single agent's long simulation) with the cross-sectional unemployment rate (from many agents at a single point in time).

We claimed above that these numbers will be approximately equal in large samples, due to ergodicity.

Let's see if that's true.

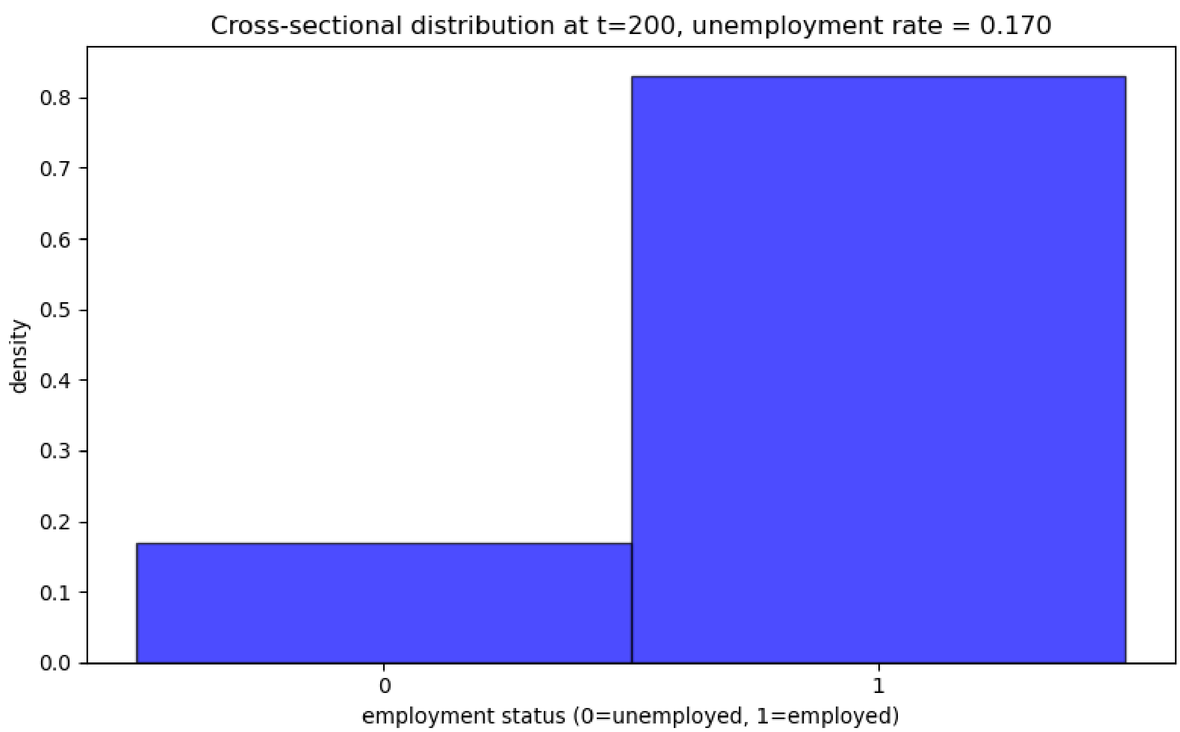
```
model = create_js_with_sep_model()
cross_sectional_unemp = simulate_cross_section(
    model, n_agents=20_000, T=200
)

time_avg_unemp = jnp.mean(unemployed_indicator)
print(f"Time-average unemployment rate (single agent): "
      f"{time_avg_unemp:.4f}")
print(f"Cross-sectional unemployment rate (at t=200): "
      f"{cross_sectional_unemp:.4f}")
print(f"Difference: {abs(time_avg_unemp - cross_sectional_unemp):.4f}")
```

```
Time-average unemployment rate (single agent): 0.1640
Cross-sectional unemployment rate (at t=200): 0.1696
Difference: 0.0056
```

Indeed, they are very close.

Now let's visualize the cross-sectional distribution:

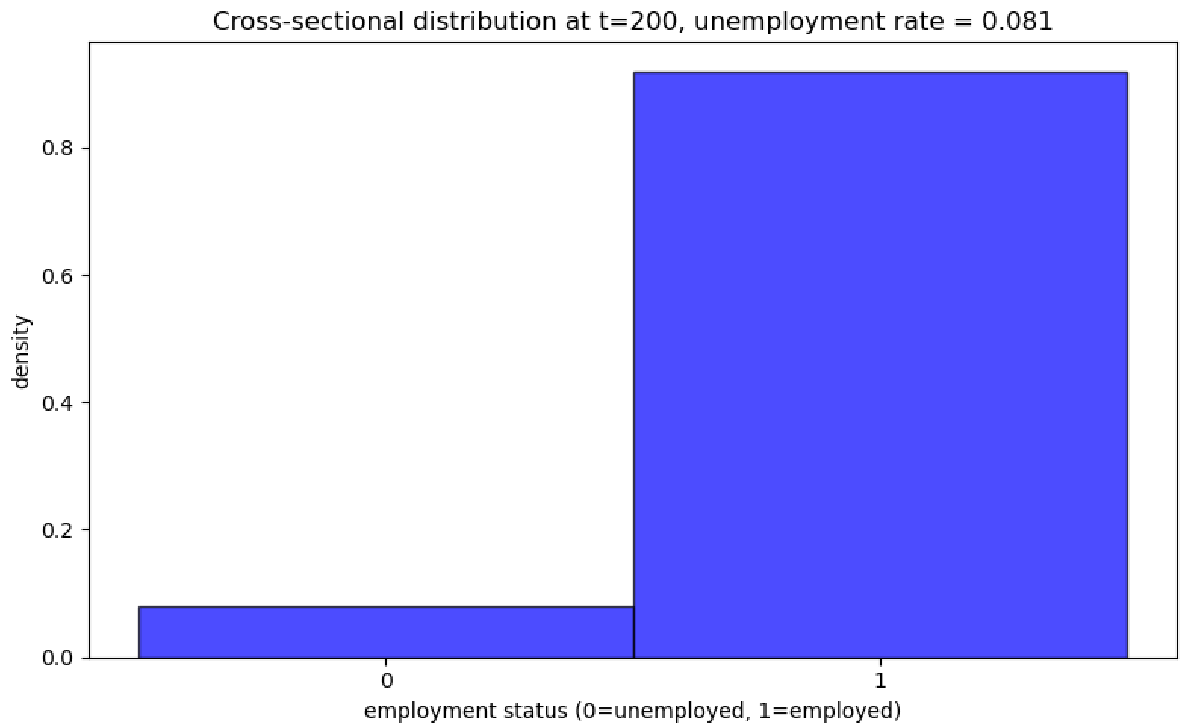
```
plot_cross_sectional_unemployment(model)
```



46.8 Lower unemployment compensation (c=0.5)

What happens to the cross-sectional unemployment rate with lower unemployment compensation?

```
model_low_c = create_js_with_sep_model(c=0.5)
plot_cross_sectional_unemployment(model_low_c)
```



46.9 Exercises

Exercise 46.9.1

Create a plot that investigates more carefully how the steady state cross-sectional unemployment rate changes with unemployment compensation.

Try a range of values for unemployment compensation `c`, such as `c = 0.2, 0.4, 0.6, 0.8, 1.0`. For each value, compute the steady-state cross-sectional unemployment rate and plot it against `c`.

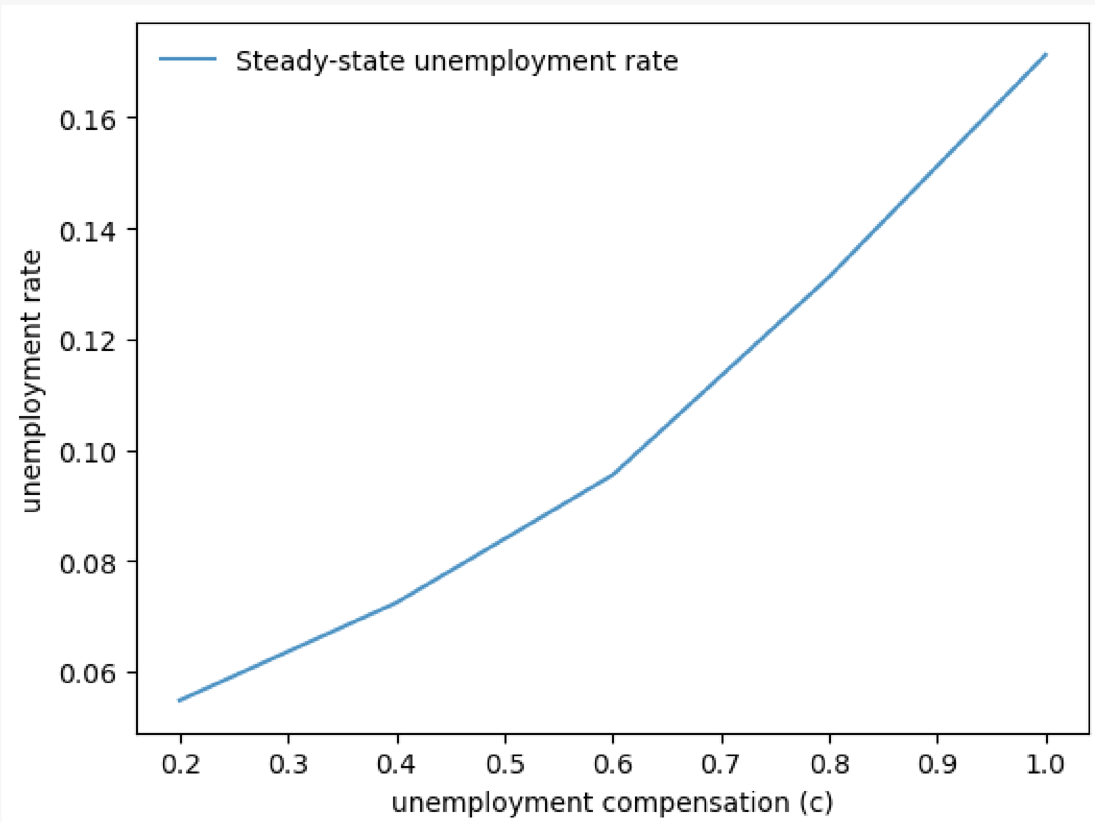
What relationship do you observe between unemployment compensation and the unemployment rate?

Solution

We compute the steady-state unemployment rate for different values of unemployment compensation:

```
c_values = 1.0, 0.8, 0.6, 0.4, 0.2
rates = []
for c in c_values:
    model = create_js_with_sep_model(c=c)
```

```
    unemployment_rate = simulate_cross_section(model)
    rates.append(unemployment_rate)

fig, ax = plt.subplots()
ax.plot(
    c_values, rates, alpha=0.8,
    linewidth=1.5, label='Steady-state unemployment rate'
)
ax.set_xlabel('unemployment compensation (c)')
ax.set_ylabel('unemployment rate')
ax.legend(frameon=False)
plt.show()
```



CHAPTER

FORTYSEVEN

JOB SEARCH IV: FITTED VALUE FUNCTION ITERATION

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

Contents

47.1 Overview

This lecture follows on from the job search model with separation presented in the *previous lecture*.

That lecture combined exogenous job separation events and a Markov wage offer process.

In this lecture we continue with this set and, in addition, allow the wage offer process to be continuous rather than discrete.

In particular,

$$W_t = \exp(X_t) \quad \text{where} \quad X_{t+1} = \rho X_t + \nu Z_{t+1}$$

and $\{Z_t\}$ is IID and standard normal.

While we already considered continuous wage distributions briefly in *Job Search I: The McCall Search Model*, the change was relatively trivial in that case.

The reason is that we were able to reduce the problem to solving for a single scalar value (the continuation value).

Here, in our Markov setting, the change is less trivial, since a continuous wage distribution leads to an uncountably infinite state space.

The infinite state space leads to additional challenges, particularly when it comes to applying value function iteration (VFI).

These challenges will lead us to modify VFI by adding an interpolation step.

The combination of VFI and this interpolation step is called **fitted value function iteration** (fitted VFI).

Fitted VFI is very common in practice, so we will take some time to work through the details.

In addition to what's in Anaconda, this lecture will need the following libraries

```
!pip install quantecon jax
```

We will use the following imports:

```
import matplotlib.pyplot as plt
import jax
import jax.numpy as jnp
from jax import lax
from typing import NamedTuple
from functools import partial
import quantecon as qe
```

47.2 Model

Assuming that readers are familiar with the content of *Job Search III: Search with Separation and Markov Wages*, the model can be summarized as follows.

- Wage offers follow a continuous Markov process: $W_t = \exp(X_t)$ where $X_{t+1} = \rho X_t + \nu Z_{t+1}$
- $\{Z_t\}$ is IID and standard normal
- Jobs terminate with probability α each period (separation rate)
- Unemployed workers receive compensation c per period
- Workers have CRRA utility $u(x) = \frac{x^{1-\gamma}-1}{1-\gamma}$
- Future payoffs are discounted by factor $\beta \in (0,1)$

47.3 Solution method

Let's discuss how we can solve this model.

The only real change from *Job Search III: Search with Separation and Markov Wages* is that we replace sums with integrals.

47.3.1 Value function iteration

In the *discrete case*, we ended up iterating on the Bellman operator

$$(Tv_u)(w) = \max\left\{\frac{1}{1-\beta(1-\alpha)} \cdot (u(w) + \alpha\beta(Pv_u)(w)), u(c) + \beta(Pv_u)(w)\right\} \tag{47.1}$$

where

$$(Pv_u)(w) := \sum_{w'} v_u(w')P(w,w')$$

Here we iterate on the same law after changing the definition of the P operator to

$$(Pv_u)(w) := \int v_u(w')p(w,w')dw'$$

where $p(w,\cdot)$ is the conditional density of w' given w.

Here we are thinking of v_u as a function on all of $\mathbb{R}_+$.

After taking ψ to be the standard normal density, we can write the expression above more explicitly as

$$(Pv_u)(w) := \int v_u(w^\rho \exp(\nu z))\psi(z)dz,$$

To understand this expression, recall that $W_t = \exp(X_t)$ where $X_{t+1} = \rho X_t + \nu Z_{t+1}$.

As a result $W_{t+1} = \exp(X_{t+1}) = \exp(\rho \log(W_t) + \nu Z_{t+1}) = W_t^\rho \exp(\nu Z_{t+1})$.

The integral above regards the current wage W_t as fixed at w and takes the expectation of $v_u(w^\rho \exp(\nu Z_{t+1}))$.

47.3.2 Fitting

In theory, we should now proceed as follows:

1. Begin with a guess v
2. Applying T to obtain the update $v' = Tv$
3. Unless some stopping condition is satisfied, set $v = v'$ and go to step 2.

However, there is a problem we must confront before we implement this procedure: The iterates of the value function can neither be calculated exactly nor stored on a computer.

To see the issue, consider (47.1).

Even if v is a known function, the only way to store its update v' is to record its value $v'(w)$ for every $w \in \mathbb{R}_+$.

Clearly, this is impossible.

47.3.3 Fitted value function iteration

What we will do instead is use **fitted value function iteration**.

The procedure is as follows:

Let a current guess v be given.

Now we record the value of the function v' at only finitely many "grid" points $w_1 < w_2 < \cdots < w_I$ and then reconstruct v' from this information when required.

More precisely, the algorithm will be

1. Begin with an array **v** representing the values of an initial guess of the value function on some grid points $\{w_i\}$.
2. Build a function v on the state space $\mathbb{R}_+$ by interpolation or approximation, based on **v** and $\{w_i\}$.
3. Obtain and record the samples of the updated function $v'(w_i)$ on each grid point w_i.
4. Unless some stopping condition is satisfied, take this as the new array and go to step 1.

How should we go about step 2?

This is a problem of function approximation, and there are many ways to approach it.

What's important here is that the function approximation scheme must not only produce a good approximation to each v, but also that it combines well with the broader iteration algorithm described above.

One good choice from both respects is continuous piecewise linear interpolation.

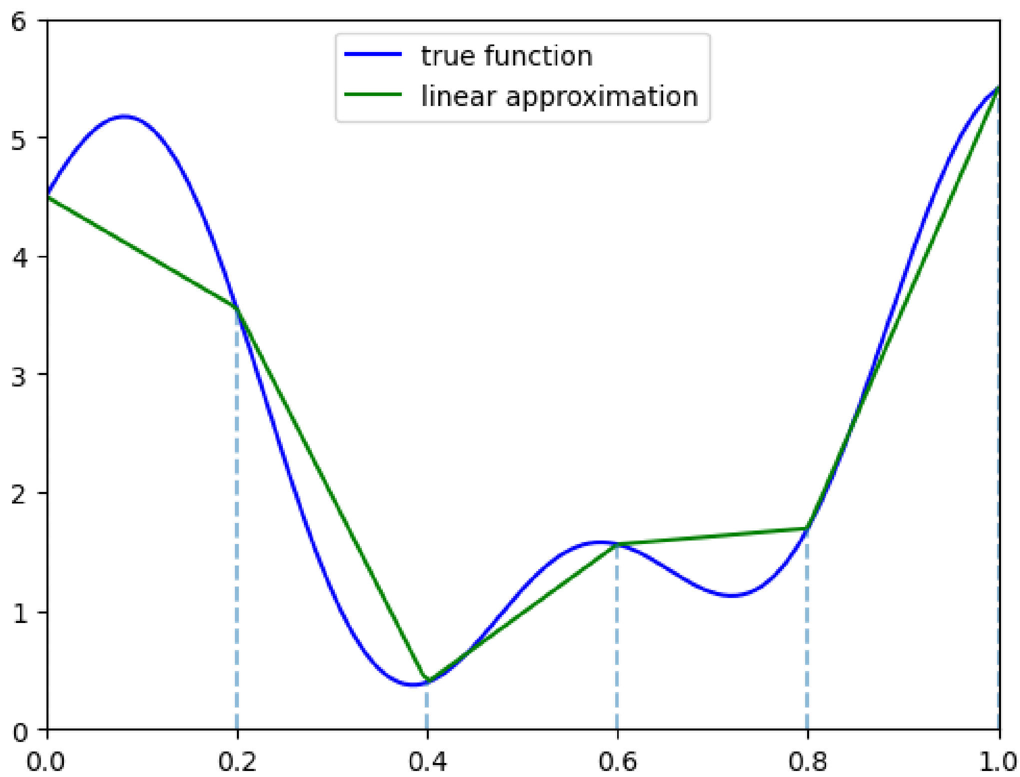
This method

1. combines well with value function iteration (see, e.g., [Gordon, 1995] or [Stachurski, 2008]) and
2. preserves useful shape properties such as monotonicity and concavity/convexity.

Linear interpolation will be implemented using JAX's interpolation function `jnp.interp`.

The next figure illustrates piecewise linear interpolation of an arbitrary function on grid points $0, 0.2, 0.4, 0.6, 0.8, 1$.

```
def f(x):
    y1 = 2 * jnp.cos(6 * x) + jnp.sin(14 * x)
    return y1 + 2.5

c_grid = jnp.linspace(0, 1, 6)
f_grid = jnp.linspace(0, 1, 150)

def Af(x):
    return jnp.interp(x, c_grid, f(c_grid))

fig, ax = plt.subplots()

ax.plot(f_grid, f(f_grid), 'b-', label='true function')
ax.plot(f_grid, Af(f_grid), 'g-', label='linear approximation')
ax.vlines(c_grid, c_grid * 0, f(c_grid), linestyle='dashed', alpha=0.5)

ax.legend(loc="upper center")

ax.set(xlim=(0, 1), ylim=(0, 6))
plt.show()
```



47.4 Implementation

Let's code up and solve the model.

47.4.1 Setup

The first step is to build a JAX-compatible structure for the McCall model with separation and a continuous wage offer distribution.

The key computational challenge is evaluating the conditional expectation $(Pv_u)(w) = \int v_u(w')p(w,w')dw'$ at each wage grid point.

Recall that we have:

$$(Pv_u)(w) = \int v_u(w^\rho \exp(\nu z))\psi(z)dz$$

where ψ is the standard normal density.

We will approximate this integral using Monte Carlo integration with draws $\{Z_i\}$ from the standard normal distribution:

$$(Pv_u)(w) \approx \frac{1}{N}\sum_{i=1}^{N} v_u(w^\rho \exp(\nu Z_i))$$

For this reason, our data structure will include a fixed set of IID $N(0,1)$ draws $\{Z_i\}$.

```
class Model(NamedTuple):
    c: float                # unemployment compensation
    α: float                # job separation rate
    β: float                # discount factor
    ρ: float                # wage persistence
    ν: float                # wage volatility
    γ: float                # utility parameter
    w_grid: jnp.ndarray     # grid of points for fitted VFI
    z_draws: jnp.ndarray    # draws from the standard normal distribution

def create_mccall_model(
        c: float = 1.0,
        α: float = 0.05,
        β: float = 0.96,
        ρ: float = 0.9,
        ν: float = 0.2,
        γ: float = 1.5,
        grid_size: int = 100,
        mc_size: int = 1000,
        seed: int = 1234
    ):
    """Factory function to create a McCall model instance."""

    key = jax.random.PRNGKey(seed)
    z_draws = jax.random.normal(key, (mc_size,))

    # Discretize just to get a suitable wage grid for interpolation
    mc = qe.markov.tauchen(grid_size, ρ, ν)
    w_grid = jnp.exp(jnp.array(mc.state_values))

    return Model(c, α, β, ρ, ν, γ, w_grid, z_draws)
```

We use the same CRRA utility function as in the discrete case:

```
def u(x, γ):
    return (x**(1 - γ) - 1) / (1 - γ)
```

47.4.2 Iteration

Here is the Bellman operator, where we use Monte Carlo integration to evaluate the expectation.

```
def T(model, v):
    """Update the value function."""

    # Unpack model parameters
    c, α, β, ρ, ν, γ, w_grid, z_draws = model

    # Interpolate array represented value function
    vf = lambda x: jnp.interp(x, w_grid, v)

    def compute_expectation(w):
        # Use Monte Carlo to evaluate integral (P v)(w) = E[v(W' | w)]
        # where W' = w^ρ * exp(ν * Z)
        w_next = w**ρ * jnp.exp(ν * z_draws)
        return jnp.mean(vf(w_next))

```

(continues on next page)

(continued from previous page)

```
    compute_exp_on_grid = jax.vmap(compute_expectation)
    Pv = compute_exp_on_grid(w_grid)

    d = 1 / (1 - β * (1 - α))
    v_e = d * (u(w_grid, γ) + α * β * Pv)
    continuation_values = u(c, γ) + β * Pv
    return jnp.maximum(v_e, continuation_values)
```

Here's the solver, which computes an approximate fixed point v_u of T.

```
@jax.jit
def vfi(
        model: Model,
        tolerance: float = 1e-6,    # Error tolerance
        max_iter: int = 100_000,    # Max iteration bound
    ):
    """
    Compute the fixed point v_u of T.

    """

    v_init = jnp.zeros(model.w_grid.shape)

    def cond(loop_state):
        v, error, i = loop_state
        return (error > tolerance) & (i <= max_iter)

    def update(loop_state):
        v, error, i = loop_state
        v_new = T(model, v)
        error = jnp.max(jnp.abs(v_new - v))
        new_loop_state = v_new, error, i + 1
        return new_loop_state

    initial_state = (v_init, tolerance + 1, 1)
    final_loop_state = lax.while_loop(cond, update, initial_state)
    v_final, error, i = final_loop_state

    return v_final
```

Here's a function that uses a solution v_u to compute the remaining functions of interest: v_e, and the continuation value function h.

We use the same expressions as we did in the *discrete case*, after replacing sums with integrals.

```
def compute_solution_functions(model, v_u):

    # Unpack model parameters
    c, α, β, ρ, ν, γ, w_grid, z_draws = model

    # Interpolate v_u on the wage grid
    vf = lambda x: jnp.interp(x, w_grid, v_u)

    def compute_expectation(w):
        # Use Monte Carlo to evaluate integral (P v)(w)
        # Compute E[v(w' | w)] where w' = w^ρ * exp(ν * z)
        w_next = w**ρ * jnp.exp(ν * z_draws)
```

(continues on next page)

(continued from previous page)

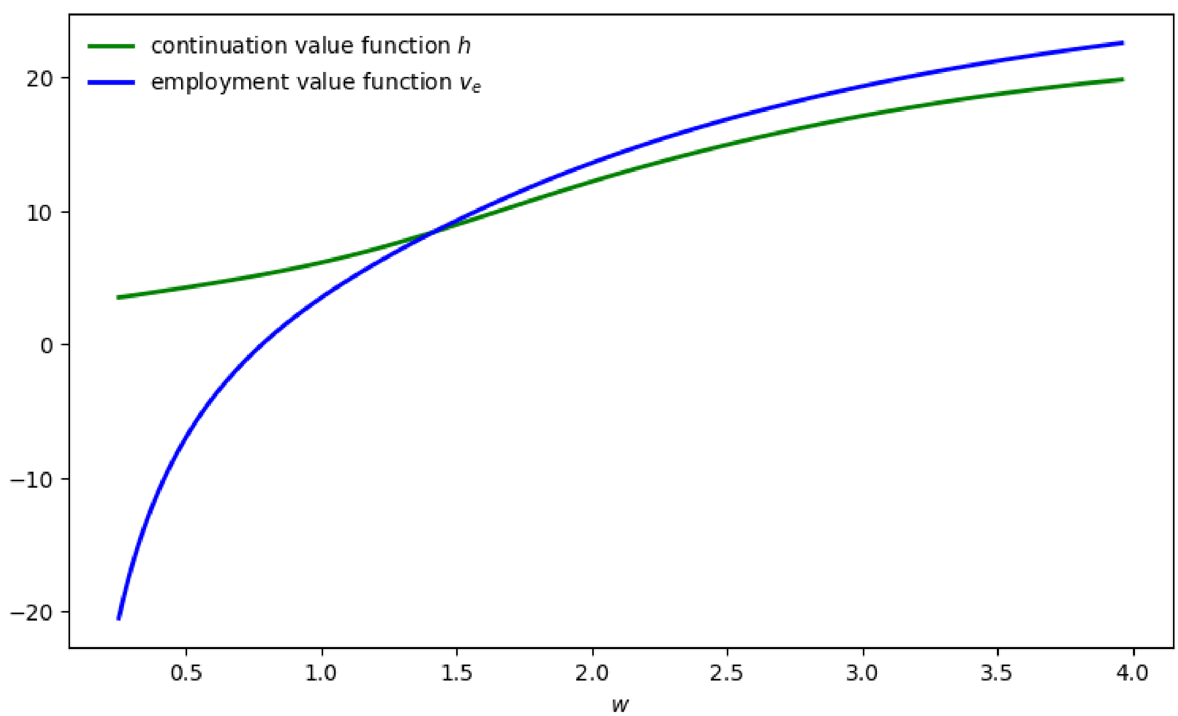
```
        return jnp.mean(vf(w_next))

    compute_exp_on_grid = jax.vmap(compute_expectation)
    Pv = compute_exp_on_grid(w_grid)

    d = 1 / (1 - β * (1 - α))
    v_e = d * (u(w_grid, γ) + α * β * Pv)
    h = u(c, γ) + β * Pv

    return v_e, h
```

Let's try solving the model:

```
model = create_mccall_model()
c, α, β, ρ, ν, γ, w_grid, z_draws = model
v_u = vfi(model)
v_e, h = compute_solution_functions(model, v_u)
```

Let's plot our results.

```
fig, ax = plt.subplots(figsize=(9, 5.2))
ax.plot(w_grid, h, 'g-', linewidth=2,
        label="continuation value function $h$")
ax.plot(w_grid, v_e, 'b-', linewidth=2,
        label="employment value function $v_e$")
ax.legend(frameon=False)
ax.set_xlabel(r"$w$")
plt.show()
```



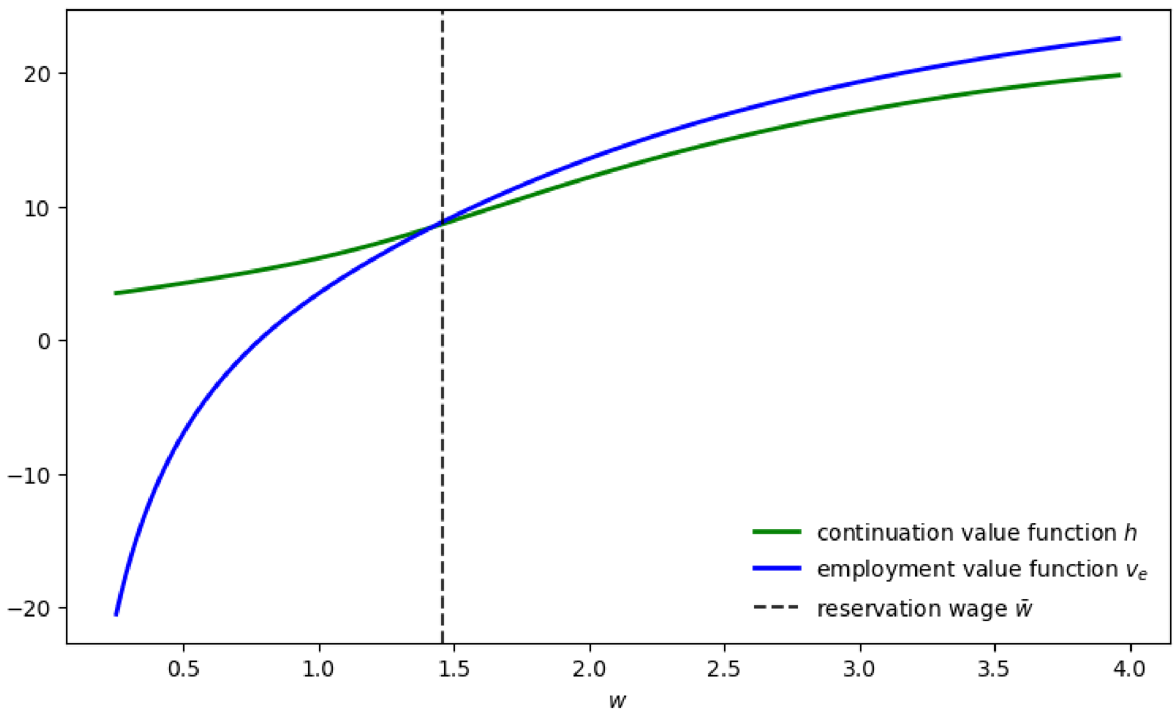
The reservation wage is at the intersection of the employment value function v_e and the continuation value function h.

Here's a function to compute it explicitly.

```python
@jax.jit
def get_reservation_wage(model: Model) -> float:
    """
    Calculate the reservation wage for a given model.

    """
    c, α, β, ρ, ν, γ, w_grid, z_draws = model

    v_u = vfi(model)
    v_e, h = compute_solution_functions(model, v_u)

    # Compute optimal policy (acceptance indices)
    σ = v_e >= h

    # Find first index where policy indicates acceptance
    first_accept_idx = jnp.argmax(σ) # returns first True value

    # If no acceptance (all False), return infinity
    # Otherwise return the wage at the first acceptance index
    return jnp.where(jnp.any(σ), w_grid[first_accept_idx], jnp.inf)
```

Let's repeat our plot, but now inserting the reservation wage.

```python
w_bar = get_reservation_wage(model)

fig, ax = plt.subplots(figsize=(9, 5.2))
ax.plot(w_grid, h, 'g-', linewidth=2,
        label="continuation value function $h$")
ax.plot(w_grid, v_e, 'b-', linewidth=2,
        label="employment value function $v_e$")
ax.axvline(x=w_bar, color='black', linestyle='--', alpha=0.8,
           label=f'reservation wage $\\bar{{w}}$')
ax.legend(frameon=False)
ax.set_xlabel(r"$w$")
plt.show()
```



47.5 Simulation

Now we run some simulations with a focus on unemployment rate.

47.5.1 Single agent dynamics

Let's simulate the employment path of a single agent under the optimal policy.

We need a function to update the agent's state by one period.

```
def update_agent(key, status, wage, model, w_bar):
    """
    Updates an agent's employment status and current wage by one period.

    Parameters:
    - key: JAX random key
    - status: Current employment status (0 or 1)
    - wage: Current wage if employed, current offer if unemployed
    - model: Model instance
    - w_bar: Reservation wage

    """
    c, α, β, ρ, ν, γ, w_grid, z_draws = model

    # Draw new wage offer based on current wage
    key1, key2 = jax.random.split(key)
    z = jax.random.normal(key1)
    new_wage = wage**ρ * jnp.exp(ν * z)
```

(continues on next page)

(continued from previous page)

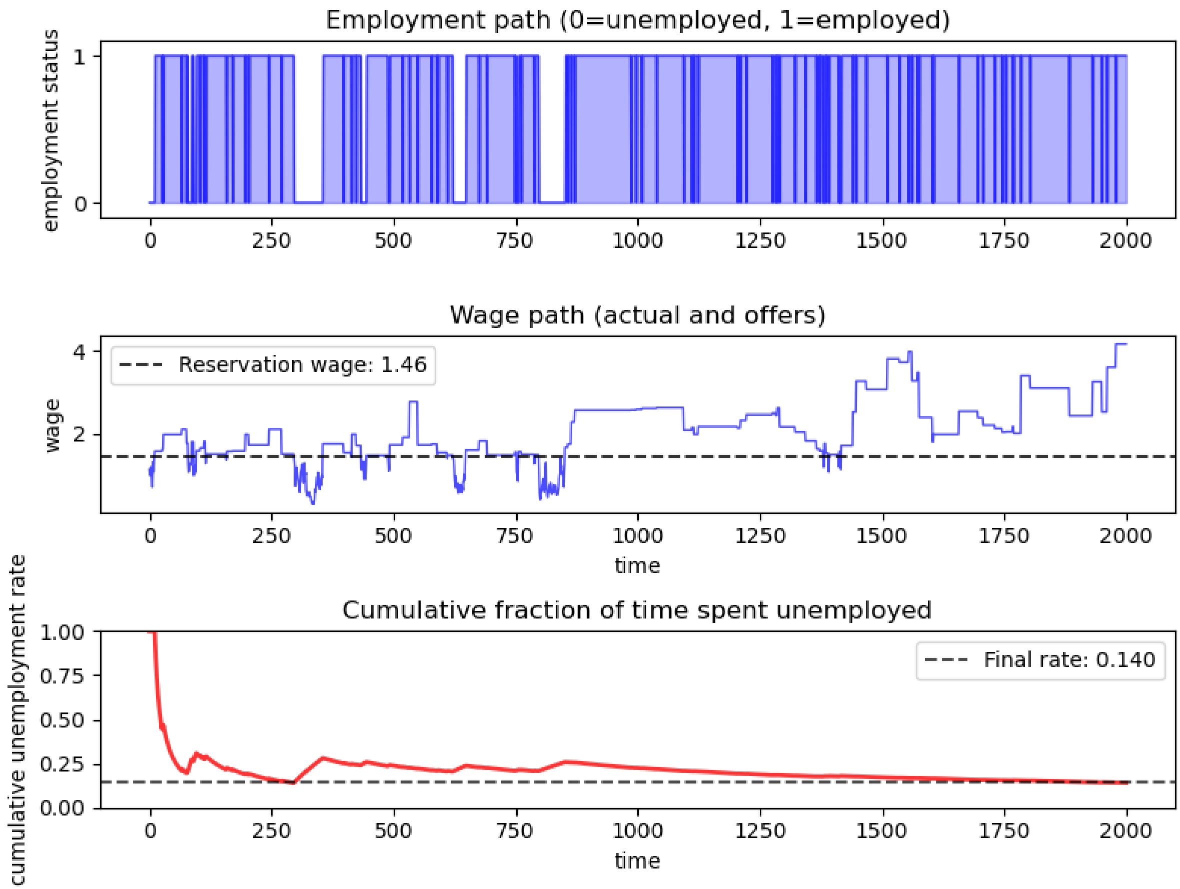
```
    # Check if separation occurs (for employed workers)
    separation_occurs = jax.random.uniform(key2) < α

    # Accept if current wage meets or exceeds reservation wage
    accepts = wage >= w_bar

    # If employed: status = 1 if no separation, 0 if separation
    # If unemployed: status = 1 if accepts, 0 if rejects
    next_status = jnp.where(
        status,
        1 - separation_occurs.astype(jnp.int32),  # employed path
        accepts.astype(jnp.int32)                  # unemployed path
    )

    # If employed: wage = current if no separation, new if separation
    # If unemployed: wage = current if accepts, new if rejects
    next_wage = jnp.where(
        status,
        jnp.where(separation_occurs, new_wage, wage),  # employed path
        jnp.where(accepts, wage, new_wage)             # unemployed path
    )

    return next_status, next_wage
```

Here's a function to simulate the employment path of a single agent.

```
def simulate_employment_path(
        model: Model,      # Model details
        w_bar: float,      # Reservation wage
        T: int = 2_000,    # Simulation length
        seed: int = 42     # Set seed for simulation
    ):
    """
    Simulate employment path for T periods starting from unemployment.

    """
    key = jax.random.PRNGKey(seed)
    c, α, β, ρ, ν, γ, w_grid, z_draws = model

    # Initial conditions: start unemployed with initial wage draw
    status = 0
    key, subkey = jax.random.split(key)
    wage = jnp.exp(jax.random.normal(subkey) * ν)

    wage_path = []
    status_path = []

    for t in range(T):
        wage_path.append(wage)
        status_path.append(status)

        key, subkey = jax.random.split(key)
        status, wage = update_agent(
            subkey, status, wage, model, w_bar
        )

    return jnp.array(wage_path), jnp.array(status_path)
```

Let's create a comprehensive plot of the employment simulation:

```
model = create_mccall_model()
w_bar = get_reservation_wage(model)

wage_path, employment_status = simulate_employment_path(model, w_bar)

fig, (ax1, ax2, ax3) = plt.subplots(3, 1, figsize=(8, 6))

# Plot employment status
ax1.plot(employment_status, 'b-', alpha=0.7, linewidth=1)
ax1.fill_between(
    range(len(employment_status)), employment_status, alpha=0.3, color='blue'
)
ax1.set_ylabel('employment status')
ax1.set_title('Employment path (0=unemployed, 1=employed)')
ax1.set_yticks((0, 1))
ax1.set_ylim(-0.1, 1.1)

# Plot wage path with reservation wage
ax2.plot(wage_path, 'b-', alpha=0.7, linewidth=1)
ax2.axhline(y=w_bar, color='black', linestyle='--', alpha=0.8,
           label=f'Reservation wage: {w_bar:.2f}')
ax2.set_xlabel('time')
ax2.set_ylabel('wage')
ax2.set_title('Wage path (actual and offers)')
ax2.legend()

# Plot cumulative fraction of time unemployed
unemployed_indicator = (employment_status == 0).astype(int)
cumulative_unemployment = (
    jnp.cumsum(unemployed_indicator) /
    jnp.arange(1, len(employment_status) + 1)
)

ax3.plot(cumulative_unemployment, 'r-', alpha=0.8, linewidth=2)
ax3.axhline(y=jnp.mean(unemployed_indicator), color='black',
            linestyle='--', alpha=0.7,
            label=f'Final rate: {jnp.mean(unemployed_indicator):.3f}')
ax3.set_xlabel('time')
ax3.set_ylabel('cumulative unemployment rate')
ax3.set_title('Cumulative fraction of time spent unemployed')
ax3.legend()
ax3.set_ylim(0, 1)

plt.tight_layout()
plt.show()
```



The simulation shows the agent cycling between employment and unemployment.

The agent starts unemployed and receives wage offers according to the Markov process.

When unemployed, the agent accepts offers that exceed the reservation wage.

When employed, the agent faces job separation with probability α each period.

47.5.2 Cross-sectional analysis

Now let's simulate many agents simultaneously to examine the cross-sectional unemployment rate.

To do this efficiently, we need a different approach than `simulate_employment_path` defined above.

The key differences are:

- `simulate_employment_path` records the entire history (all T periods) for a single agent, which is useful for visualization but memory-intensive
- The new function `sim_agent` below only tracks and returns the final state, which is all we need for cross-sectional statistics
- `sim_agent` uses `lax.fori_loop` instead of a Python loop, making it JIT-compilable and suitable for vectorization across many agents

We first define a function that simulates a single agent forward T time steps:

```
@jax.jit
def sim_agent(key, initial_status, initial_wage, model, w_bar, T):
    """
    Simulate a single agent forward T time steps using lax.fori_loop.

    Uses fold_in to generate a new key at each time step.

    Parameters:
    - key: JAX random key for this agent
    - initial_status: Initial employment status (0 or 1)
    - initial_wage: Initial wage
    - model: Model instance
    - w_bar: Reservation wage
    - T: Number of time periods to simulate

    Returns:
    - final_status: Employment status after T periods
    - final_wage: Wage after T periods
    """
    def update(t, loop_state):
        status, wage = loop_state
        step_key = jax.random.fold_in(key, t)
        status, wage = update_agent(step_key, status, wage, model, w_bar)
        return status, wage

    initial_loop_state = (initial_status, initial_wage)
    final_loop_state = lax.fori_loop(0, T, update, initial_loop_state)
    final_status, final_wage = final_loop_state
    return final_status, final_wage


# Create vectorized version of sim_agent to process multiple agents in parallel
sim_agents_vmap = jax.vmap(sim_agent, in_axes=(0, 0, 0, None, None, None))


def simulate_cross_section(
        model: Model,
        n_agents: int = 100_000,
        T: int = 200,
        seed: int = 42
    ) -> float:
    """
    Simulate cross-section of agents and return unemployment rate.

    This approach:
    1. Generates n_agents random keys
    2. Calls sim_agent for each agent (vectorized via vmap)
    3. Collects the final states to produce the cross-section

    Returns the cross-sectional unemployment rate.
    """
    c, α, β, ρ, ν, γ, w_grid, z_draws = model

    key = jax.random.PRNGKey(seed)

    # Solve for optimal reservation wage
    w_bar = get_reservation_wage(model)
```

(continues on next page)

(continued from previous page)

```
    # Initialize arrays
    init_key, subkey = jax.random.split(key)
    initial_wages = jnp.exp(jax.random.normal(subkey, (n_agents,)) * ν)
    initial_status_vec = jnp.zeros(n_agents, dtype=jnp.int32)

    # Generate n_agents random keys
    agent_keys = jax.random.split(init_key, n_agents)

    # Simulate each agent forward T steps (vectorized)
    final_status, final_wages = sim_agents_vmap(
        agent_keys, initial_status_vec, initial_wages, model, w_bar, T
    )

    unemployment_rate = 1 - jnp.mean(final_status)
    return unemployment_rate
```

Now let's compare the time-average unemployment rate (from a single agent's long simulation) with the cross-sectional unemployment rate (from many agents at a single point in time).

```
model = create_mccall_model()
cross_sectional_unemp = simulate_cross_section(
    model, n_agents=20_000, T=200
)

time_avg_unemp = jnp.mean(unemployed_indicator)
print(f"Time-average unemployment rate (single agent, T=2000): "
      f"{time_avg_unemp:.4f}")
print(f"Cross-sectional unemployment rate (at t=200): "
      f"{cross_sectional_unemp:.4f}")
print(f"Difference: {abs(time_avg_unemp - cross_sectional_unemp):.4f}")
```

```
Time-average unemployment rate (single agent, T=2000): 0.1395
Cross-sectional unemployment rate (at t=200): 0.2029
Difference: 0.0634
```

The difference above can be further reduced by increasing the simulation length for the single agent.

```
wage_path_long, employment_status_long = simulate_employment_path(model, w_bar, T=10_
↪000)
unemployed_indicator_long = (employment_status_long == 0).astype(int)
time_avg_unemp_long = jnp.mean(unemployed_indicator_long)

print(f"Time-average unemployment rate (single agent, T=10000): "
      f"{time_avg_unemp_long:.4f}")
print(f"Cross-sectional unemployment rate (at t=200): "
      f"{cross_sectional_unemp:.4f}")
print(f"Difference: {abs(time_avg_unemp_long - cross_sectional_unemp):.4f}")
```

```
Time-average unemployment rate (single agent, T=10000): 0.2071
Cross-sectional unemployment rate (at t=200): 0.2029
Difference: 0.0042
```

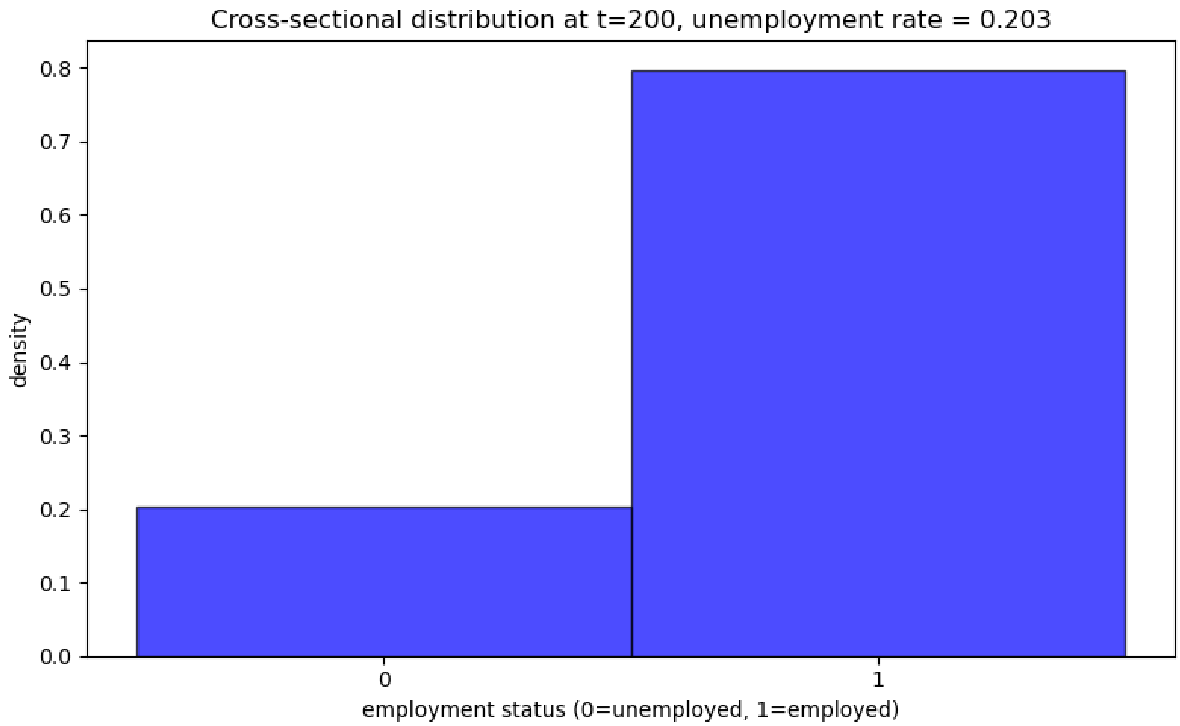
47.5.3 Visualization

This function generates a histogram showing the distribution of employment status across many agents:

```
def plot_cross_sectional_unemployment(
        model: Model,              # Model instance with parameters
        t_snapshot: int = 200,     # Time for cross-sectional snapshot
        n_agents: int = 20_000     # Number of agents to simulate
    ):
    """
    Generate histogram of cross-sectional unemployment at a specific time.

    """
    c, α, β, ρ, ν, γ, w_grid, z_draws = model

    # Get final employment state directly
    key = jax.random.PRNGKey(42)
    w_bar = get_reservation_wage(model)

    # Initialize arrays
    init_key, subkey = jax.random.split(key)
    initial_wages = jnp.exp(jax.random.normal(subkey, (n_agents,)) * ν)
    initial_status_vec = jnp.zeros(n_agents, dtype=jnp.int32)

    # Generate n_agents random keys
    agent_keys = jax.random.split(init_key, n_agents)

    # Simulate each agent forward T steps (vectorized)
    final_status, _ = sim_agents_vmap(
        agent_keys, initial_status_vec, initial_wages, model, w_bar, t_snapshot
    )

    # Calculate unemployment rate
    unemployment_rate = 1 - jnp.mean(final_status)

    fig, ax = plt.subplots(figsize=(8, 5))

    # Plot histogram as density (bars sum to 1)
    weights = jnp.ones_like(final_status) / len(final_status)
    ax.hist(final_status, bins=[-0.5, 0.5, 1.5],
            alpha=0.7, color='blue', edgecolor='black',
            density=True, weights=weights)

    ax.set_xlabel('employment status (0=unemployed, 1=employed)')
    ax.set_ylabel('density')
    ax.set_title(f'Cross-sectional distribution at t={t_snapshot}, ' +
                 f'unemployment rate = {unemployment_rate:.3f}')
    ax.set_xticks([0, 1])

    plt.tight_layout()
    plt.show()
```

Let’s plot the cross-sectional distribution:

```
plot_cross_sectional_unemployment(model)
```



47.6 Exercises

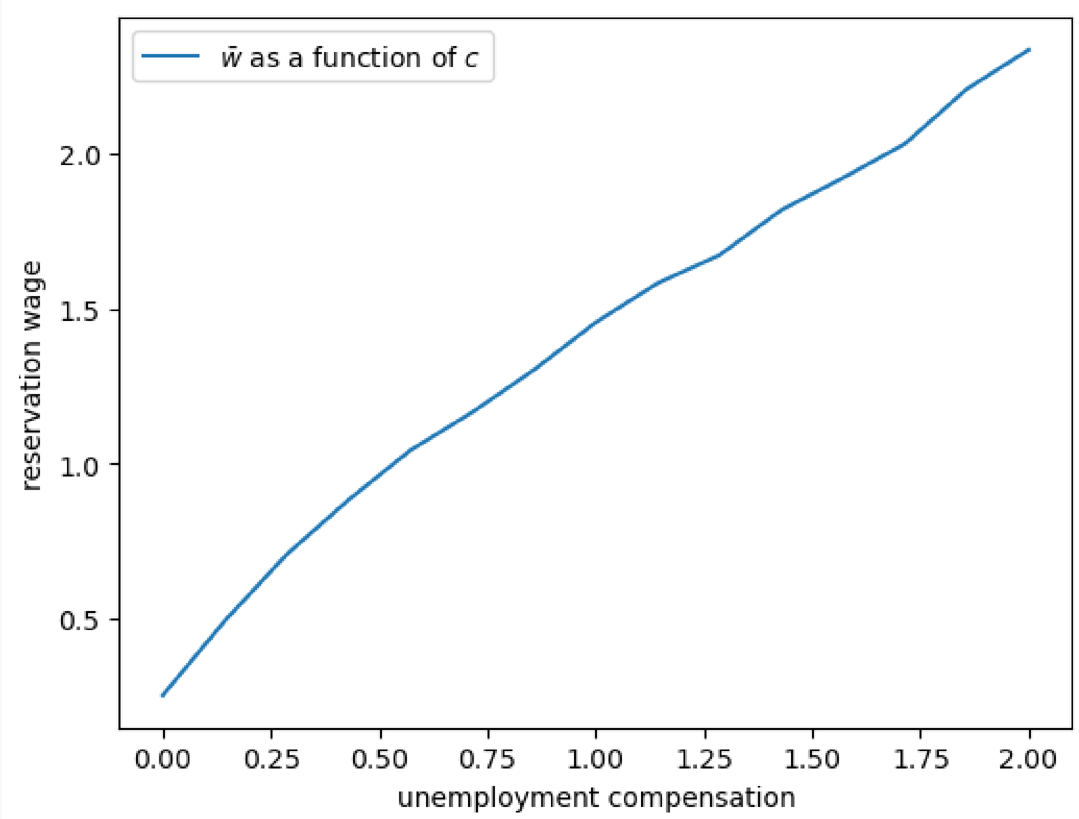
Exercise 47.6.1

Use the code above to explore what happens to the reservation wage when c changes.

Solution

Here is one solution

```
def compute_res_wage_given_c(c):
    model = create_mccall_model(c=c)
    w_bar = get_reservation_wage(model)
    return w_bar

c_vals = jnp.linspace(0.0, 2.0, 15)
w_bar_vals = jax.vmap(compute_res_wage_given_c)(c_vals)

fig, ax = plt.subplots()
ax.set(xlabel='unemployment compensation', ylabel='reservation wage')
ax.plot(c_vals, w_bar_vals, label=r'$\bar w$ as a function of $c$')
ax.legend()
plt.show()
```



As unemployment compensation increases, the reservation wage also increases.

This makes economic sense: when the value of being unemployed rises (through higher c), workers become more selective about which job offers to accept.

Exercise 47.6.2

Create a plot that shows how the reservation wage changes with the risk aversion parameter γ.

Use `γ_vals = jnp.linspace(1.2, 2.5, 15)` and keep all other parameters at their default values.

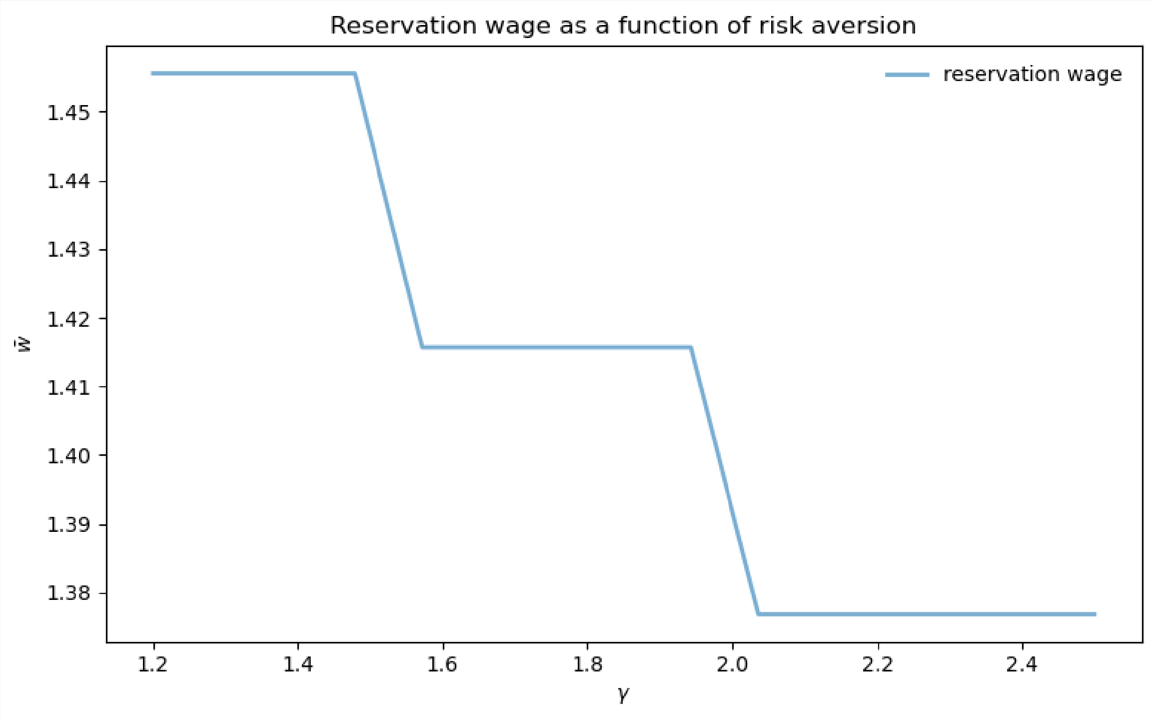
How do you expect the reservation wage to vary with γ? Why?

Solution

We compute the reservation wage for different values of the risk aversion parameter:

```
γ_vals = jnp.linspace(1.2, 2.5, 15)
w_bar_vec = jnp.empty_like(γ_vals)

for i, γ in enumerate(γ_vals):
    model = create_mccall_model(γ=γ)
    w_bar = get_reservation_wage(model)
    w_bar_vec = w_bar_vec.at[i].set(w_bar)

fig, ax = plt.subplots(figsize=(9, 5.2))
```

```
ax.plot(γ_vals, w_bar_vec, linewidth=2, alpha=0.6,
        label='reservation wage')
ax.legend(frameon=False)
ax.set_xlabel(r'$\gamma$')
ax.set_ylabel(r'$\bar{w}$')
ax.set_title('Reservation wage as a function of risk aversion')
plt.show()
```



As risk aversion (γ) increases, the reservation wage decreases.

This occurs because more risk-averse workers place higher value on the security of employment relative to the uncertainty of continued search.

With higher γ, the utility cost of unemployment (foregone consumption) becomes more severe, making workers more willing to accept lower wages rather than continue searching.

CHAPTER

FORTYEIGHT

JOB SEARCH V: PERSISTENT AND TRANSITORY WAGE SHOCKS

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

Contents

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon jax
```

48.1 Overview

In this lecture we extend the *McCall job search model* by decomposing wage offers into **persistent** and **transitory** components.

In the *baseline model*, wage offers are IID over time, which is unrealistic.

In *Job Search III*, we introduced correlated wage draws using a Markov chain, but we also added job separation.

Here we take a different approach: we model wage dynamics through an AR(1) process for the persistent component plus a transitory shock, while returning to the assumption that jobs are permanent (as in the *baseline model*).

This persistent-transitory decomposition is:

- More realistic for modeling actual wage processes
- Commonly used in labor economics (see, e.g., [MaCurdy, 1982], [Meghir and Pistaferri, 2004])
- Simple enough to analyze while capturing key features of wage dynamics

By keeping jobs permanent, we can focus on understanding how persistent and transitory wage shocks affect search behavior and reservation wages.

We will solve the model using fitted value function iteration with linear interpolation, as introduced in *Job Search IV*.

We will use the following imports:

```
import matplotlib.pyplot as plt
import jax
import jax.numpy as jnp
import jax.random
import quantecon as qe
from typing import NamedTuple
```

48.2 The model

Wages at each point in time are given by

$$W_t = \exp(Z_t) + Y_t$$

where

$$Y_t \sim \exp(\mu + s\zeta_t) \quad \text{and} \quad Z_{t+1} = d + \rho Z_t + \sigma \epsilon_{t+1}$$

Here $\{\zeta_t\}$ and $\{\epsilon_t\}$ are both IID and standard normal.

Here $\{Y_t\}$ is a transitory component and $\{Z_t\}$ is persistent.

As before, the worker can either

1. accept an offer and work permanently at that wage, or
2. take unemployment compensation c and wait till next period.

The value function satisfies the Bellman equation

$$v^*(w, z) = \max\left\{\frac{u(w)}{1-\beta}, u(c) + \beta\,\mathbb{E}_z v^*(w', z')\right\}$$

In this expression, u is a utility function and $\mathbb{E}_z$ is the expectation of next period variables given current z.

The variable z enters as a state in the Bellman equation because its current value helps predict future wages.

48.2.1 A simplification

There is a way that we can reduce dimensionality in this problem, which greatly accelerates computation.

To start, let f^* be the continuation value function, defined by

$$f^*(z) := u(c) + \beta \, \mathbb{E}_z v^*(w', z')$$

The Bellman equation can now be written

$$v^*(w, z) = \max \left\{ \frac{u(w)}{1-\beta}, \, f^*(z) \right\}$$

Combining the last two expressions, we see that the continuation value function satisfies

$$f^*(z) = u(c) + \beta \, \mathbb{E}_z \max \left\{ \frac{u(w')}{1-\beta}, f^*(z') \right\}$$

We'll solve this functional equation for f^* by introducing the operator

$$Qf(z) = u(c) + \beta \, \mathbb{E}_z \max \left\{ \frac{u(w')}{1-\beta}, f(z') \right\}$$

By construction, f^* is a fixed point of Q, in the sense that $Qf^* = f^*$.

Under mild assumptions, it can be shown that Q is a contraction mapping over a suitable space of continuous functions on $\mathbb{R}$.

By Banach's contraction mapping theorem, this means that f^* is the unique fixed point and we can calculate it by iterating with Q from any reasonable initial condition.

Once we have f^*, we can solve the search problem by stopping when the reward for accepting exceeds the continuation value, or

$$\frac{u(w)}{1-\beta} \geq f^*(z)$$

For utility, we take $u(x) = \ln(x)$.

The reservation wage is the wage where equality holds in the last expression.

That is,

$$\bar{w}(z) := \exp(f^*(z)(1-\beta)) \tag{48.1}$$

Our main aim is to solve for the reservation rule and study its properties and implications.

48.3 Implementation

Let f be our initial guess of f^*.

When we iterate, we use the *fitted value function iteration* algorithm.

In particular, f and all subsequent iterates are stored as a vector of values on a grid.

These points are interpolated into a function as required, using piecewise linear interpolation.

The integral in the definition of Qf is calculated by Monte Carlo.

Here's a `NamedTuple` that stores the model parameters and data.

Default parameter values are embedded in the model.

```
class Model(NamedTuple):
    μ: float      # transient shock log mean
    s: float      # transient shock log variance
    d: float      # shift coefficient of persistent state
    ρ: float      # correlation coefficient of persistent state
    σ: float      # state volatility
    β: float      # discount factor
    c: float      # unemployment compensation
    z_grid: jnp.ndarray
    e_draws: jnp.ndarray

def create_job_search_model(μ=0.0, s=1.0, d=0.0, ρ=0.9, σ=0.1, β=0.98, c=5.0,
                            mc_size=1000, grid_size=100, key=jax.random.PRNGKey(1234)):
    """
    Create a Model with computed grid and draws.
    """
    # Set up grid
    z_mean = d / (1 - ρ)
    z_sd = σ / jnp.sqrt(1 - ρ**2)
    k = 3  # std devs from mean
    a, b = z_mean - k * z_sd, z_mean + k * z_sd
    z_grid = jnp.linspace(a, b, grid_size)

    # Draw and store shocks
    e_draws = jax.random.normal(key, (2, mc_size))

    return Model(μ, s, d, ρ, σ, β, c, z_grid, e_draws)
```

Next, we implement the Q operator.

```
def Q(model, f_in):
    """
    Apply the operator Q.

        * model is an instance of Model
        * f_in is an array that represents f
        * returns Qf

    """
    μ, s, d = model.μ, model.s, model.d
    ρ, σ, β, c = model.ρ, model.σ, model.β, model.c
    z_grid, e_draws = model.z_grid, model.e_draws
    M = e_draws.shape[1]

    def compute_expectation(z):
        def evaluate_shock(e):
            e1, e2 = e[0], e[1]
            z_next = d + ρ * z + σ * e1
            go_val = jnp.interp(z_next, z_grid, f_in)  # f(z')
            y_next = jnp.exp(μ + s * e2)               # y' draw
            w_next = jnp.exp(z_next) + y_next          # w' draw
            stop_val = jnp.log(w_next) / (1 - β)
            return jnp.maximum(stop_val, go_val)

        expectations = jax.vmap(evaluate_shock)(e_draws.T)
        return jnp.mean(expectations)
```

(continues on next page)

(continued from previous page)

```
    expectations = jax.vmap(compute_expectation)(z_grid)
    f_out = jnp.log(c) + β * expectations
    return f_out
```

Here's a function to compute an approximation to the fixed point of Q.

```
@jax.jit
def compute_fixed_point(model, tol=1e-4, max_iter=1000):
    """
    Compute an approximation to the fixed point of Q.
    """

    def cond_fun(loop_state):
        f, i, error = loop_state
        return jnp.logical_and(error > tol, i < max_iter)

    def body_fun(loop_state):
        f, i, error = loop_state
        f_new = Q(model, f)
        error_new = jnp.max(jnp.abs(f_new - f))
        return f_new, i + 1, error_new

    # Initial state
    f_init = jnp.full(len(model.z_grid), jnp.log(model.c))
    init_state = (f_init, 0, tol + 1)

    # Run iteration
    f_final, iterations, final_error = jax.lax.while_loop(
        cond_fun, body_fun, init_state
    )

    return f_final
```

Let's try generating an instance and solving the model.

```
model = create_job_search_model()

with qe.Timer():
    f_star = compute_fixed_point(model).block_until_ready()
```

```
0.53 seconds elapsed
```

Next, we will compute and plot the reservation wage function defined in (48.1).

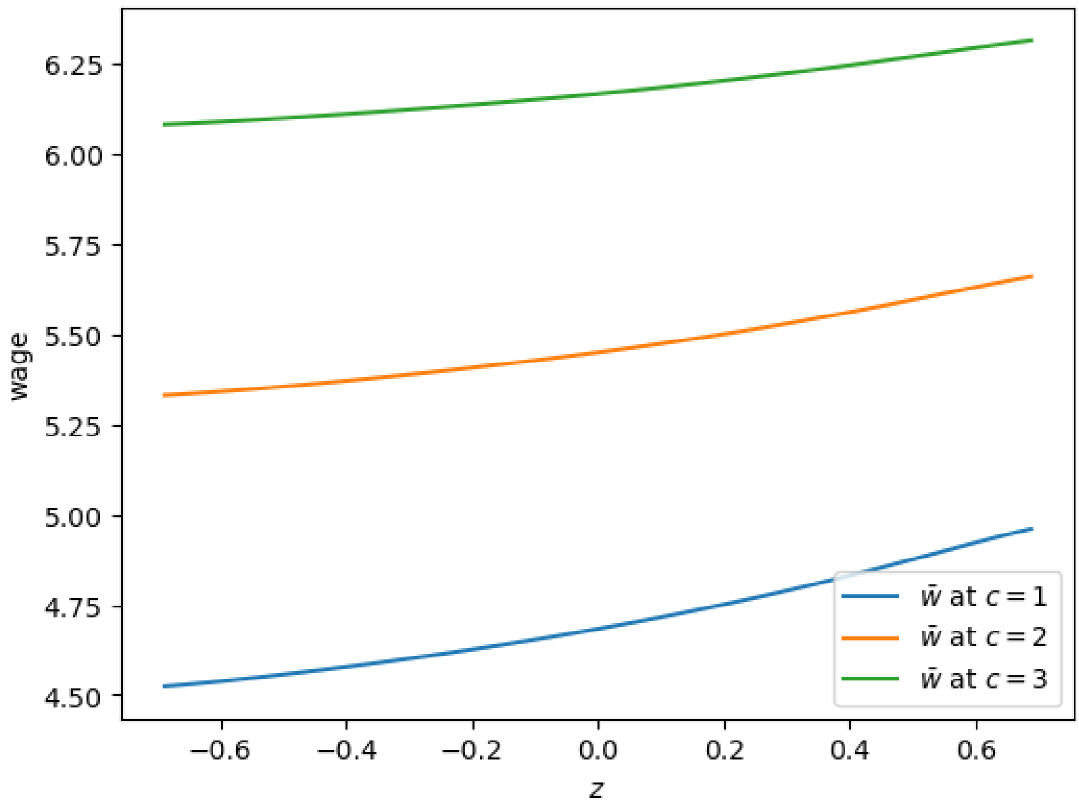
```
res_wage_function = jnp.exp(f_star * (1 - model.β))

fig, ax = plt.subplots()
ax.plot(
    model.z_grid, res_wage_function, label="reservation wage given $z$"
)
ax.set(xlabel="$z$", ylabel="wage")
ax.legend()
plt.show()
```

Notice that the reservation wage is increasing in the current state z.

This is because a higher state leads the agent to predict higher future wages, increasing the option value of waiting.

Let's try changing unemployment compensation and looking at its impact on the reservation wage:

```
c_vals = 1, 2, 3

fig, ax = plt.subplots()

for c in c_vals:
    model = create_job_search_model(c=c)
    f_star = compute_fixed_point(model)
    res_wage_function = jnp.exp(f_star * (1 - model.β))
    ax.plot(model.z_grid, res_wage_function,
                label=rf"$\bar w$ at $c = {c}$")

ax.set(xlabel="$z$", ylabel="wage")
ax.legend()
plt.show()
```



As expected, higher unemployment compensation shifts the reservation wage up at all state values.

48.4 Unemployment duration

Next, we study how mean unemployment duration varies with unemployment compensation.

For simplicity, we'll fix the initial state at $Z_0 = 0$.

```
@jax.jit
def draw_duration(key, μ, s, d, ρ, σ, β, z_grid, f_star, t_max=10_000):
    """
    Draw unemployment duration for a single simulation.

    """
    def f_star_function(z):
        return jnp.interp(z, z_grid, f_star)

    def cond_fun(loop_state):
        z, t, unemployed, key = loop_state
        return jnp.logical_and(unemployed, t < t_max)

    def body_fun(loop_state):
        z, t, unemployed, key = loop_state
        key1, key2, key = jax.random.split(key, 3)

        # Draw current wage
        y = jnp.exp(μ + s * jax.random.normal(key1))
```

(continues on next page)

(continued from previous page)

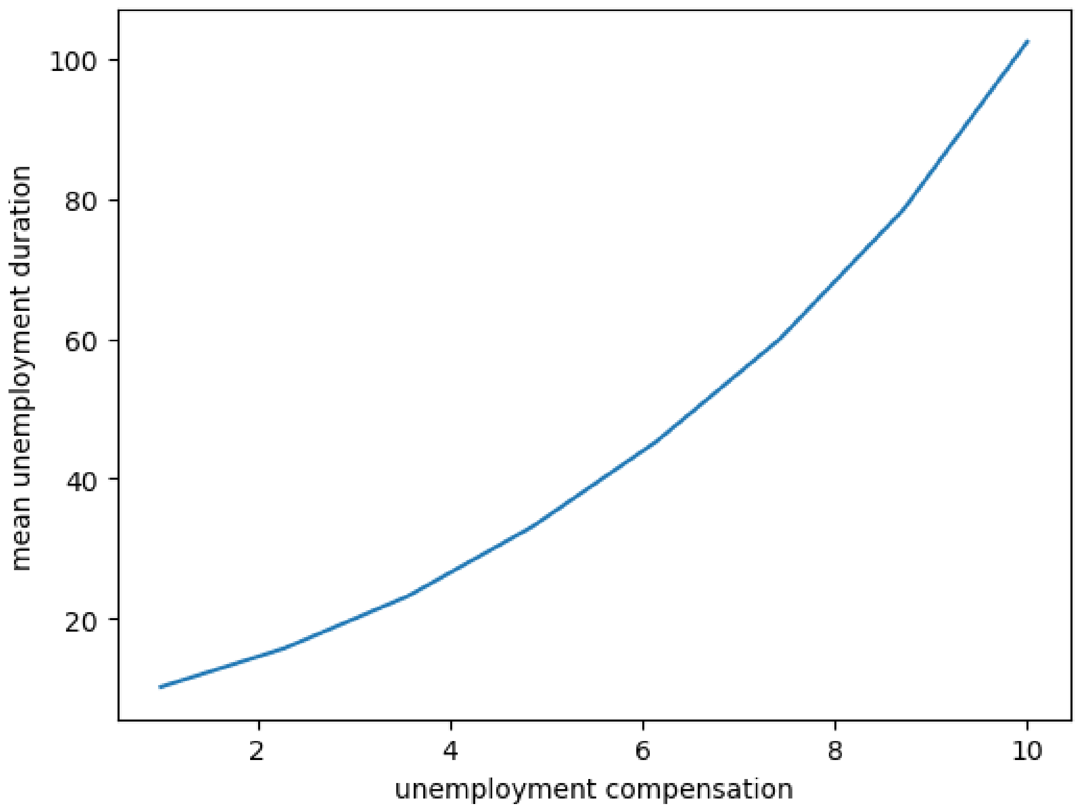
```
        w = jnp.exp(z) + y
        res_wage = jnp.exp(f_star_function(z) * (1 - β))

        # Check if optimal to stop
        accept = w >= res_wage
        τ = jnp.where(accept, t, t_max)

        # Update state if not accepting
        z_new = jnp.where(accept, z,
                          ρ * z + d + σ * jax.random.normal(key2))
        t_new = t + 1
        unemployed_new = jnp.logical_not(accept)

        return z_new, t_new, unemployed_new, key

    # Initial loop_state: (z, t, unemployed, key)
    init_state = (0.0, 0, True, key)
    z_final, t_final, unemployed_final, _ = jax.lax.while_loop(
        cond_fun, body_fun, init_state)

    # Return final time if job found, otherwise t_max
    return jnp.where(unemployed_final, t_max, t_final)


def compute_unemployment_duration(
        model, key=jax.random.PRNGKey(1234), num_reps=100_000
    ):
    """
    Compute expected unemployment duration.

    """
    f_star = compute_fixed_point(model)
    μ, s, d = model.μ, model.s, model.d
    ρ, σ, β = model.ρ, model.σ, model.β
    z_grid = model.z_grid

    # Generate keys for all simulations
    keys = jax.random.split(key, num_reps)

    # Vectorize over simulations
    τ_vals = jax.vmap(
        lambda k: draw_duration(k, μ, s, d, ρ, σ, β, z_grid, f_star)
    )(keys)

    return jnp.mean(τ_vals)
```

Let's test this out with some possible values for unemployment compensation.

```
c_vals = jnp.linspace(1.0, 10.0, 8)
durations = []
for i, c in enumerate(c_vals):
    model = create_job_search_model(c=c)
    τ = compute_unemployment_duration(model, num_reps=10_000)
    durations.append(τ)
durations = jnp.array(durations)
```

Here is a plot of the results.

```
fig, ax = plt.subplots()
ax.plot(c_vals, durations)
ax.set_xlabel("unemployment compensation")
ax.set_ylabel("mean unemployment duration")
plt.show()
```



Not surprisingly, unemployment duration increases when unemployment compensation is higher.

This is because the value of waiting increases with unemployment compensation.

48.5 Exercises

Exercise 48.5.1

Investigate how mean unemployment duration varies with the discount factor β.

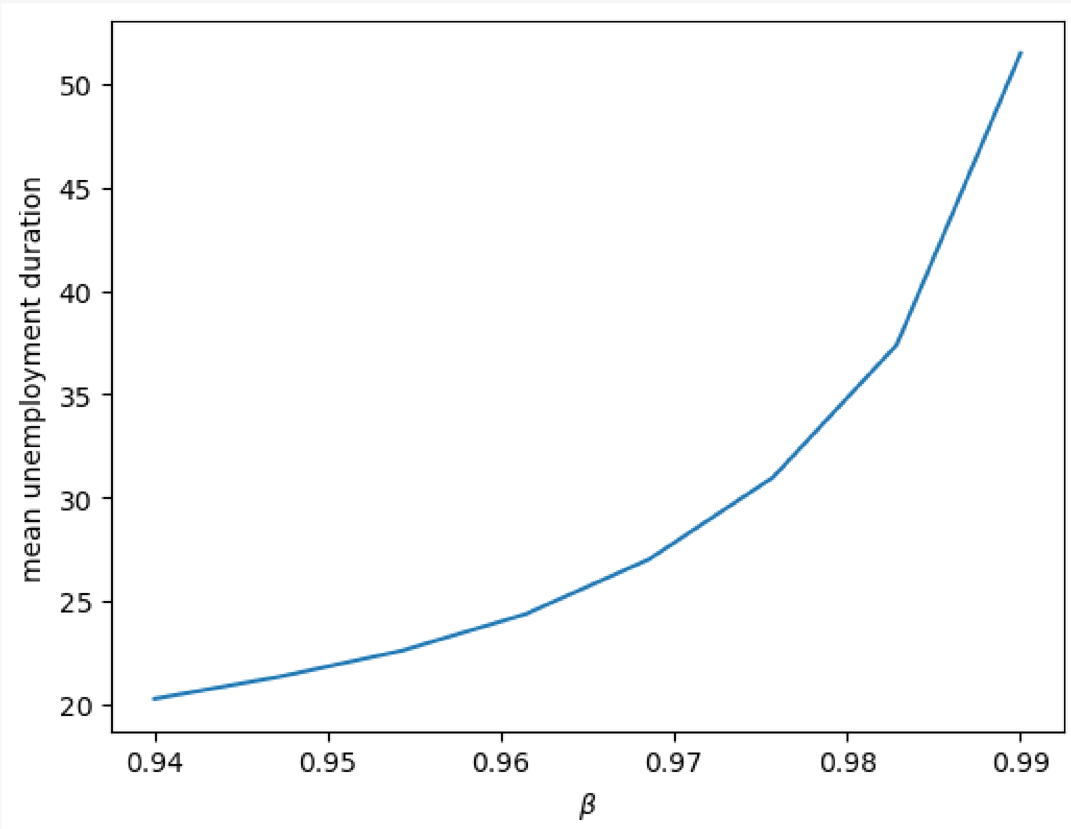
- What is your prior expectation?
- Do your results match up?

Solution

Here is one solution:

```
beta_vals = jnp.linspace(0.94, 0.99, 8)
durations = []
for i, β in enumerate(beta_vals):
    model = create_job_search_model(β=β)
    τ = compute_unemployment_duration(model, num_reps=10_000)
    durations.append(τ)
durations = jnp.array(durations)
```

```
fig, ax = plt.subplots()
ax.plot(beta_vals, durations)
ax.set_xlabel(r"$\beta$")
ax.set_ylabel("mean unemployment duration")
plt.show()
```



The figure shows that more patient individuals tend to wait longer before accepting an offer.

CHAPTER

FORTYNINE

JOB SEARCH VI: MODELING CAREER CHOICE

Contents

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
```

49.1 Overview

Next, we study a computational problem concerning career and job choices.

The model is originally due to Derek Neal [Neal, 1999].

This exposition draws on the presentation in [Ljungqvist and Sargent, 2018], section 6.5.

We begin with some imports:

```
import matplotlib.pyplot as plt
import numpy as np
import quantecon as qe
from numba import jit, prange
from quantecon.distributions import BetaBinomial
from scipy.special import binom, beta
from mpl_toolkits.mplot3d.axes3d import Axes3D
from matplotlib import cm
```

49.1.1 Model Features

- Career and job within career both chosen to maximize expected discounted wage flow.
- Infinite horizon dynamic programming with two state variables.

49.2 Model

In what follows we distinguish between a career and a job, where

- a **career** is understood to be a general field encompassing many possible jobs, and
- a **job** is understood to be a position with a particular firm

For workers, wages can be decomposed into the contribution of job and career

- $w_t = \theta_t + \epsilon_t$, where
  - θ_t is the contribution of career at time t
  - ϵ_t is the contribution of the job at time t

At the start of time t, a worker has the following options

- retain a current (career, job) pair (θ_t, ϵ_t) — referred to hereafter as "stay put"
- retain a current career θ_t but redraw a job ϵ_t — referred to hereafter as "new job"
- redraw both a career θ_t and a job ϵ_t — referred to hereafter as "new life"

Draws of θ and ϵ are independent of each other and past values, with

- $\theta_t \sim F$
- $\epsilon_t \sim G$

Notice that the worker does not have the option to retain a job but redraw a career — starting a new career always requires starting a new job.

A young worker aims to maximize the expected sum of discounted wages

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t w_t \tag{49.1}$$

subject to the choice restrictions specified above.

Let $v(\theta, \epsilon)$ denote the value function, which is the maximum of (49.1) overall feasible (career, job) policies, given the initial state (θ, ϵ).

The value function obeys

$$v(\theta, \epsilon) = \max\{I, II, III\}$$

where

$$\begin{aligned}
I &= \theta + \epsilon + \beta v(\theta, \epsilon) \\
II &= \theta + \int \epsilon' G(d\epsilon') + \beta \int v(\theta, \epsilon') G(d\epsilon') \\
III &= \int \theta' F(d\theta') + \int \epsilon' G(d\epsilon') + \beta \int \int v(\theta', \epsilon') G(d\epsilon') F(d\theta')
\end{aligned}$$

Evidently I, II and III correspond to "stay put", "new job" and "new life", respectively.

49.2.1 Parameterization

As in [Ljungqvist and Sargent, 2018], section 6.5, we will focus on a discrete version of the model, parameterized as follows:

- both θ and ϵ take values in the set `np.linspace(0, B, grid_size)` — an even grid of points between 0 and B inclusive
- `grid_size = 50`
- `B = 5`
- `β = 0.95`

The distributions F and G are discrete distributions generating draws from the grid points `np.linspace(0, B, grid_size)`.

A very useful family of discrete distributions is the Beta-binomial family, with probability mass function

$$p(k \,|\, n, a, b) = \binom{n}{k} \frac{B(k+a, n-k+b)}{B(a,b)}, \qquad k = 0, \ldots, n$$

Interpretation:

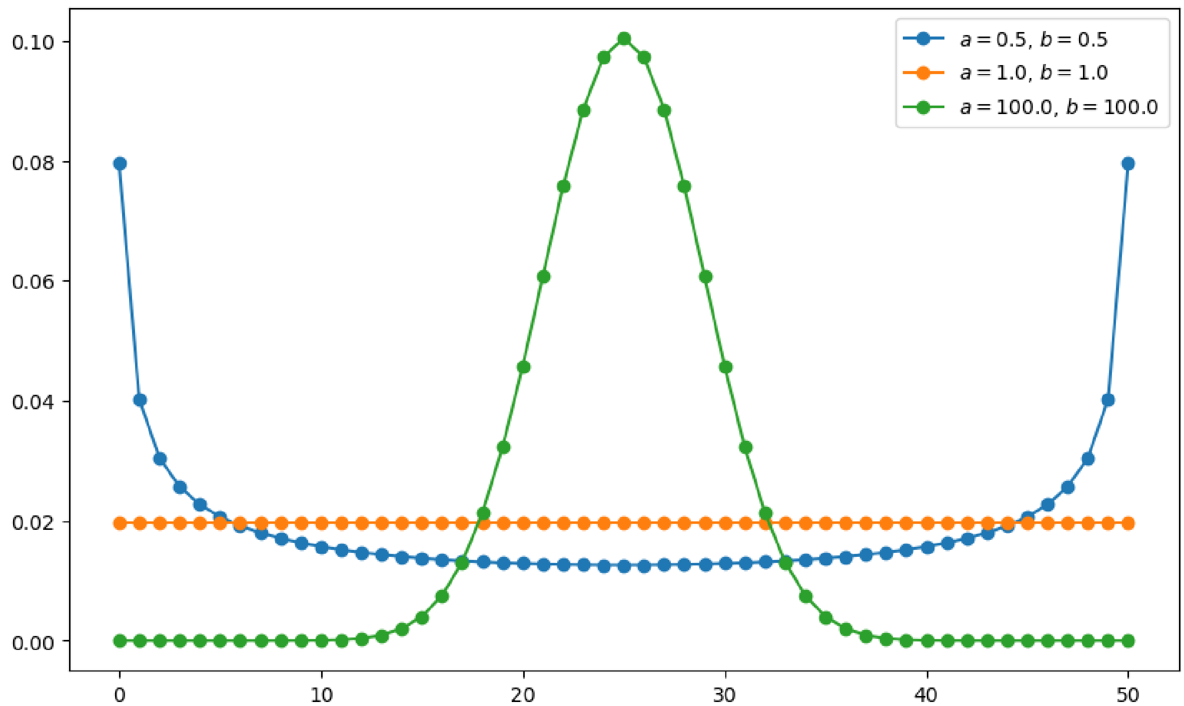
- draw q from a Beta distribution with shape parameters (a, b)
- run n independent binary trials, each with success probability q
- $p(k \,|\, n, a, b)$ is the probability of k successes in these n trials

Nice properties:

- very flexible class of distributions, including uniform, symmetric unimodal, etc.
- only three parameters

Here's a figure showing the effect on the pmf of different shape parameters when $n = 50$.

```
def gen_probs(n, a, b):
    probs = np.zeros(n+1)
    for k in range(n+1):
        probs[k] = binom(n, k) * beta(k + a, n - k + b) / beta(a, b)
    return probs

n = 50
a_vals = [0.5, 1, 100]
b_vals = [0.5, 1, 100]
fig, ax = plt.subplots(figsize=(10, 6))
for a, b in zip(a_vals, b_vals):
    ab_label = f'$a = {a:.1f}$, $b = {b:.1f}$'
    ax.plot(list(range(0, n+1)), gen_probs(n, a, b), '-o', label=ab_label)
ax.legend()
plt.show()
```



49.3 Implementation

We will first create a class `CareerWorkerProblem` which will hold the default parameterizations of the model and an initial guess for the value function.

```
class CareerWorkerProblem:

    def __init__(self,
                 B=5.0,           # Upper bound
                 β=0.95,          # Discount factor
                 grid_size=50,    # Grid size
                 F_a=1,
                 F_b=1,
                 G_a=1,
                 G_b=1):

        self.β, self.grid_size, self.B = β, grid_size, B

        self.θ = np.linspace(0, B, grid_size)     # Set of θ values
        self.ϵ = np.linspace(0, B, grid_size)     # Set of ϵ values

        self.F_probs = BetaBinomial(grid_size - 1, F_a, F_b).pdf()
        self.G_probs = BetaBinomial(grid_size - 1, G_a, G_b).pdf()
        self.F_mean = self.θ @ self.F_probs
        self.G_mean = self.ϵ @ self.G_probs

        # Store these parameters for str and repr methods
        self._F_a, self._F_b = F_a, F_b
        self._G_a, self._G_b = G_a, G_b
```

The following function takes an instance of `CareerWorkerProblem` and returns the corresponding Bellman operator

T and the greedy policy function.

In this model, T is defined by $Tv(\theta, \epsilon) = \max\{I, II, III\}$, where I, II and III are as given in (49.2).

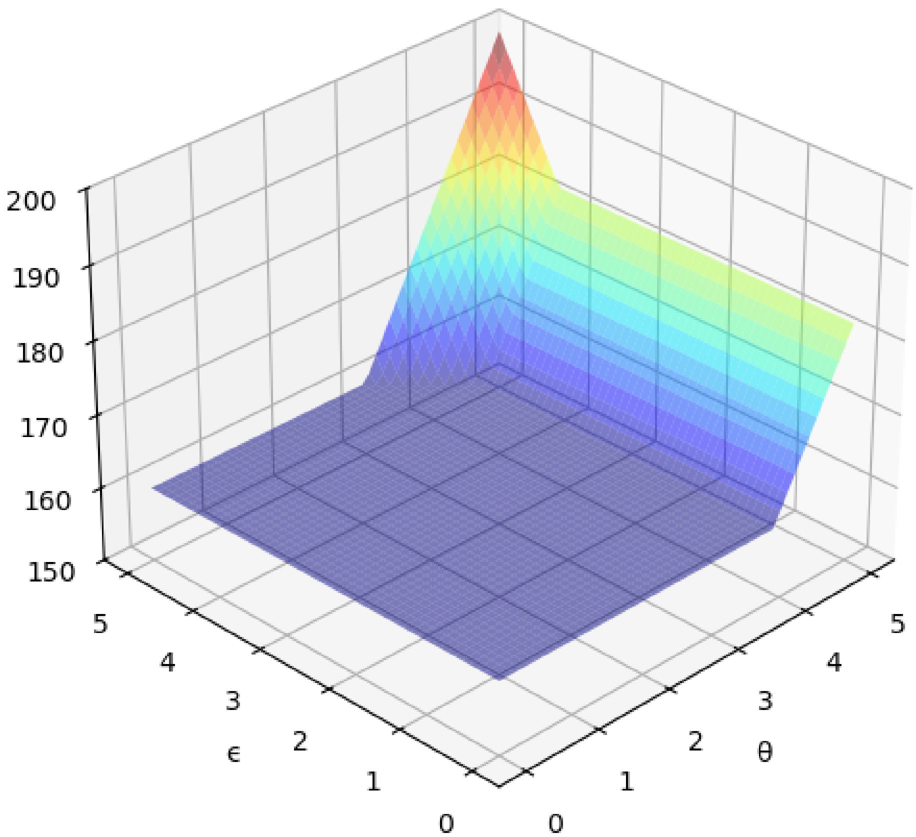
```python
def operator_factory(cw, parallel_flag=True):

    """
    Returns jitted versions of the Bellman operator and the
    greedy policy function

    cw is an instance of ``CareerWorkerProblem``
    """

    θ, ϵ, β = cw.θ, cw.ϵ, cw.β
    F_probs, G_probs = cw.F_probs, cw.G_probs
    F_mean, G_mean = cw.F_mean, cw.G_mean

    @jit(parallel=parallel_flag)
    def T(v):
        "The Bellman operator"

        v_new = np.empty_like(v)

        for i in prange(len(v)):
            for j in prange(len(v)):
                v1 = θ[i] + ϵ[j] + β * v[i, j]                    # Stay put
                v2 = θ[i] + G_mean + β * v[i, :] @ G_probs        # New job
                v3 = G_mean + F_mean + β * F_probs @ v @ G_probs  # New life
                v_new[i, j] = max(v1, v2, v3)

        return v_new

    @jit
    def get_greedy(v):
        "Computes the v-greedy policy"

        σ = np.empty(v.shape)

        for i in range(len(v)):
            for j in range(len(v)):
                v1 = θ[i] + ϵ[j] + β * v[i, j]
                v2 = θ[i] + G_mean + β * v[i, :] @ G_probs
                v3 = G_mean + F_mean + β * F_probs @ v @ G_probs
                if v1 > max(v2, v3):
                    action = 1
                elif v2 > max(v1, v3):
                    action = 2
                else:
                    action = 3
                σ[i, j] = action

        return σ

    return T, get_greedy
```

Lastly, `solve_model` will take an instance of `CareerWorkerProblem` and iterate using the Bellman operator to find the fixed point of the Bellman equation.

```
def solve_model(cw,
                use_parallel=True,
                tol=1e-4,
                max_iter=1000,
                verbose=True,
                print_skip=25):

    T, _ = operator_factory(cw, parallel_flag=use_parallel)

    # Set up loop
    v = np.full((cw.grid_size, cw.grid_size), 100.)  # Initial guess
    i = 0
    error = tol + 1

    while i < max_iter and error > tol:
        v_new = T(v)
        error = np.max(np.abs(v - v_new))
        i += 1
        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")
        v = v_new

    if error > tol:
        print("Failed to converge!")

    elif verbose:
        print(f"\nConverged in {i} iterations.")

    return v_new
```

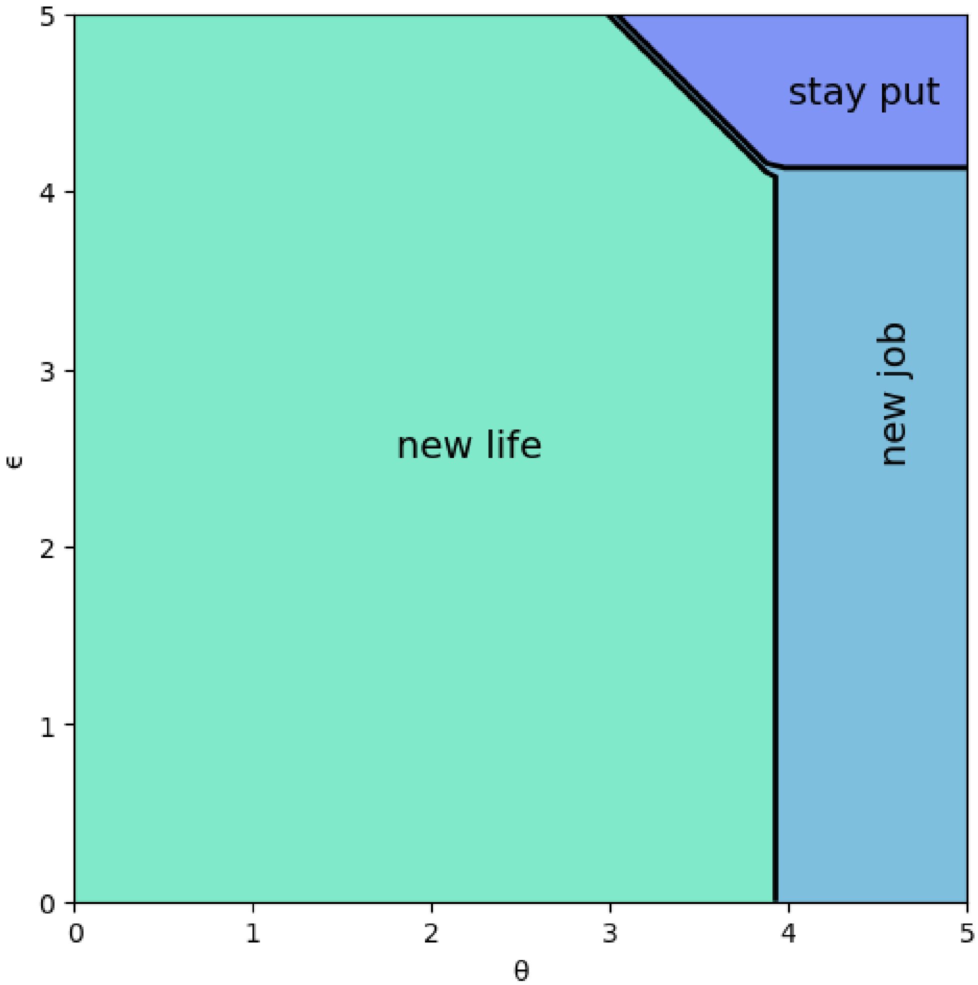
Here's the solution to the model – an approximate value function

```
cw = CareerWorkerProblem()
T, get_greedy = operator_factory(cw)
v_star = solve_model(cw, verbose=False)
greedy_star = get_greedy(v_star)

fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')
tg, eg = np.meshgrid(cw.θ, cw.ϵ)
ax.plot_surface(tg,
                eg,
                v_star.T,
                cmap=cm.jet,
                alpha=0.5,
                linewidth=0.25)
ax.set(xlabel='θ', ylabel='ϵ', zlim=(150, 200))
ax.view_init(ax.elev, 225)
plt.show()
```



And here is the optimal policy

```
fig, ax = plt.subplots(figsize=(6, 6))
tg, eg = np.meshgrid(cw.θ, cw.ϵ)
lvls = (0.5, 1.5, 2.5, 3.5)
ax.contourf(tg, eg, greedy_star.T, levels=lvls, cmap=cm.winter, alpha=0.5)
ax.contour(tg, eg, greedy_star.T, colors='k', levels=lvls, linewidths=2)
ax.set(xlabel='θ', ylabel='ϵ')
ax.text(1.8, 2.5, 'new life', fontsize=14)
ax.text(4.5, 2.5, 'new job', fontsize=14, rotation='vertical')
ax.text(4.0, 4.5, 'stay put', fontsize=14)
plt.show()
```



Interpretation:

- If both job and career are poor or mediocre, the worker will experiment with a new job and new career.
- If career is sufficiently good, the worker will hold it and experiment with new jobs until a sufficiently good one is found.
- If both job and career are good, the worker will stay put.

Notice that the worker will always hold on to a sufficiently good career, but not necessarily hold on to even the best paying job.

The reason is that high lifetime wages require both variables to be large, and the worker cannot change careers without changing jobs.

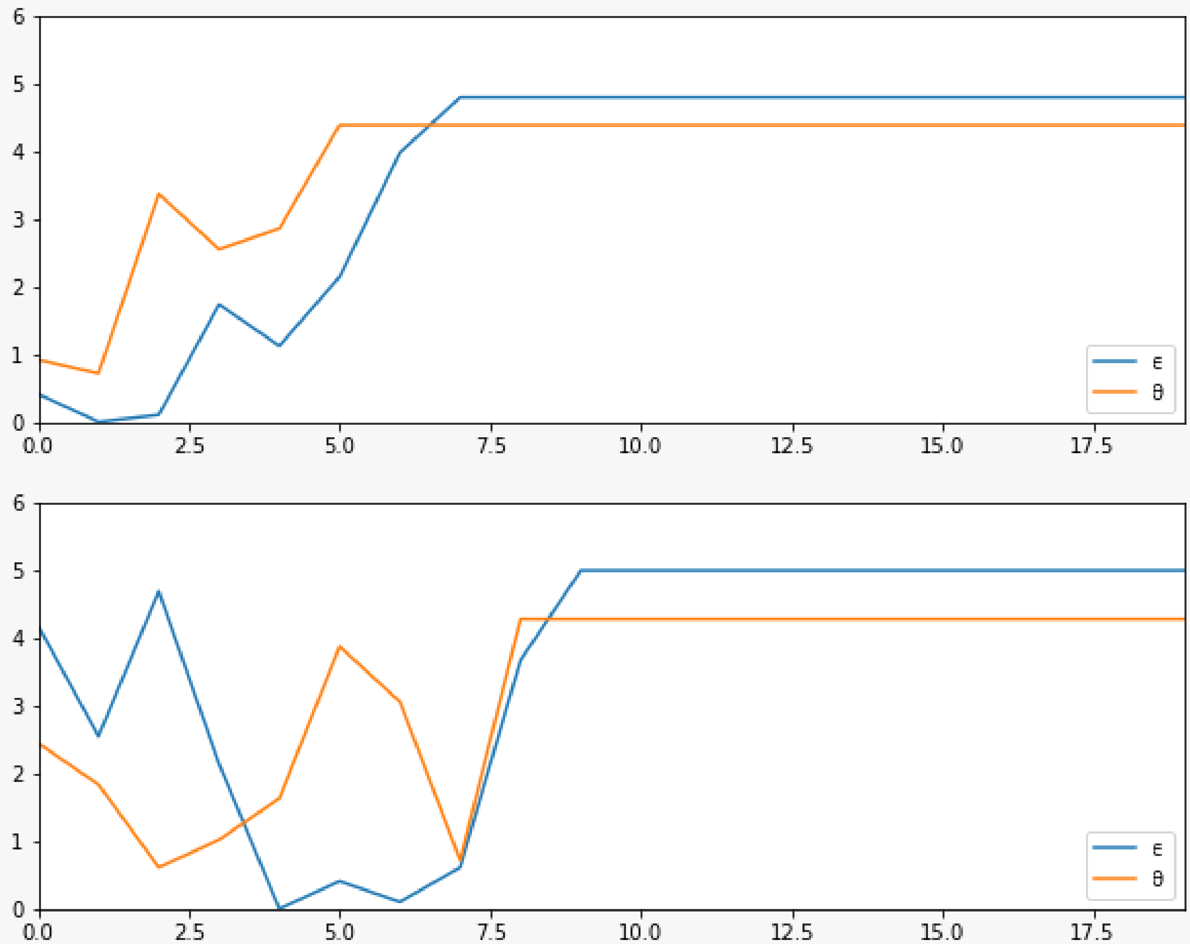
- Sometimes a good job must be sacrificed in order to change to a better career.

49.4 Exercises

Exercise 49.4.1

Using the default parameterization in the class `CareerWorkerProblem`, generate and plot typical sample paths for θ and ϵ when the worker follows the optimal policy.

In particular, modulo randomness, reproduce the following figure (where the horizontal axis represents time)



Hint

To generate the draws from the distributions F and G, use `quantecon.random.draw()`.

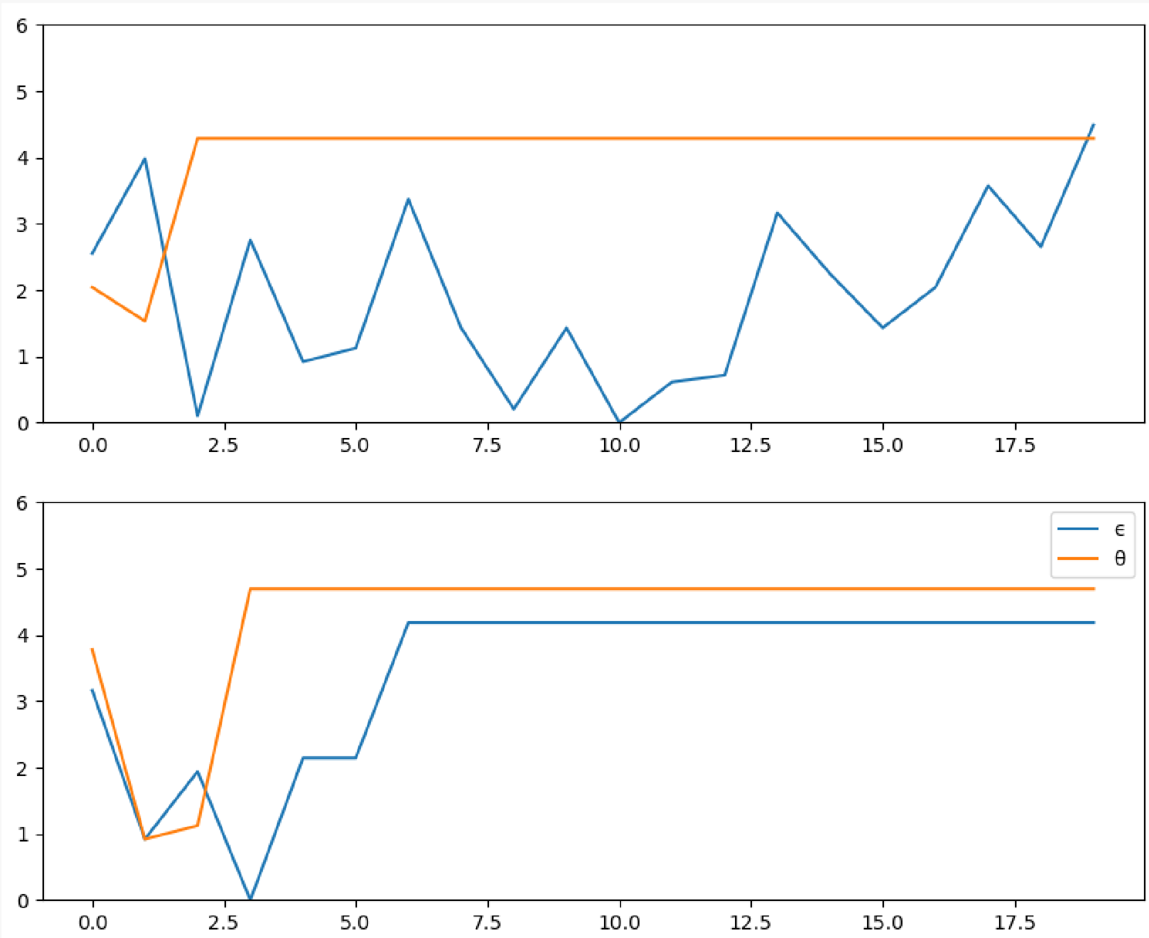
Solution

Simulate job/career paths.

In reading the code, recall that `optimal_policy[i, j]` = policy at (θ_i, ϵ_j) = either 1, 2 or 3; meaning 'stay put', 'new job' and 'new life'.

```
F = np.cumsum(cw.F_probs)
```

```
G = np.cumsum(cw.G_probs)
v_star = solve_model(cw, verbose=False)
T, get_greedy = operator_factory(cw)
greedy_star = get_greedy(v_star)

def gen_path(optimal_policy, F, G, t=20):
    i = j = 0
    θ_index = []
    ϵ_index = []
    for t in range(t):
        if optimal_policy[i, j] == 1:       # Stay put
            pass

        elif greedy_star[i, j] == 2:     # New job
            j = qe.random.draw(G)

        else:                            # New life
            i, j = qe.random.draw(F), qe.random.draw(G)
        θ_index.append(i)
        ϵ_index.append(j)
    return cw.θ[θ_index], cw.ϵ[ϵ_index]


fig, axes = plt.subplots(2, 1, figsize=(10, 8))
for ax in axes:
    θ_path, ϵ_path = gen_path(greedy_star, F, G)
    ax.plot(ϵ_path, label='ϵ')
    ax.plot(θ_path, label='θ')
    ax.set_ylim(0, 6)

plt.legend()
plt.show()
```



Exercise 49.4.2

Let's now consider how long it takes for the worker to settle down to a permanent job, given a starting point of $(\theta, \epsilon) = (0, 0)$.

In other words, we want to study the distribution of the random variable

$$T^* := \text{the first point in time from which the worker's job no longer changes}$$

Evidently, the worker's job becomes permanent if and only if (θ_t, ϵ_t) enters the "stay put" region of (θ, ϵ) space.

Letting S denote this region, T^* can be expressed as the first passage time to S under the optimal policy:

$$T^* := \inf\{t \geq 0 \,|\, (\theta_t, \epsilon_t) \in S\}$$

Collect 25,000 draws of this random variable and compute the median (which should be about 7).

Repeat the exercise with $\beta = 0.99$ and interpret the change.

Solution

The median for the original parameterization can be computed as follows

```
cw = CareerWorkerProblem()
F = np.cumsum(cw.F_probs)
G = np.cumsum(cw.G_probs)
T, get_greedy = operator_factory(cw)
v_star = solve_model(cw, verbose=False)
greedy_star = get_greedy(v_star)

@jit
def passage_time(optimal_policy, F, G):
    t = 0
    i = j = 0
    while True:
        if optimal_policy[i, j] == 1:     # Stay put
            return t
        elif optimal_policy[i, j] == 2:  # New job
            j = qe.random.draw(G)
        else:                            # New life
            i, j  = qe.random.draw(F), qe.random.draw(G)
        t += 1

@jit(parallel=True)
def median_time(optimal_policy, F, G, M=25000):
    samples = np.empty(M)
    for i in prange(M):
        samples[i] = passage_time(optimal_policy, F, G)
    return np.median(samples)

median_time(greedy_star, F, G)
```

```
7.0
```

To compute the median with $\beta = 0.99$ instead of the default value $\beta = 0.95$, replace `cw = CareerWorkerProblem()` with `cw = CareerWorkerProblem(β=0.99)`.

The medians are subject to randomness but should be about 7 and 14 respectively.

Not surprisingly, more patient workers will wait longer to settle down to their final job.

Exercise 49.4.3

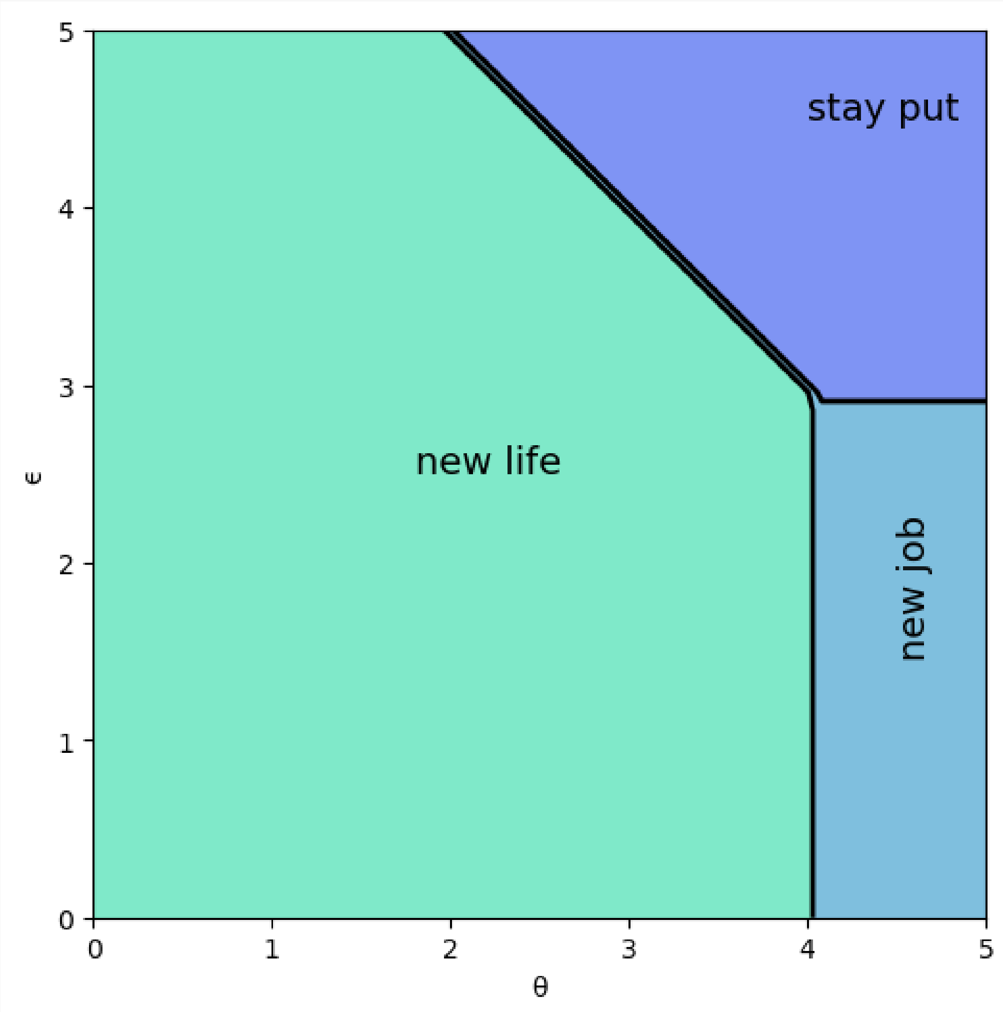
Set the parameterization to `G_a = G_b = 100` and generate a new optimal policy figure – interpret.

Solution

Here is one solution

```
cw = CareerWorkerProblem(G_a=100, G_b=100)
T, get_greedy = operator_factory(cw)
v_star = solve_model(cw, verbose=False)
greedy_star = get_greedy(v_star)

fig, ax = plt.subplots(figsize=(6, 6))
```

```
tg, eg = np.meshgrid(cw.θ, cw.ϵ)
lvls = (0.5, 1.5, 2.5, 3.5)
ax.contourf(tg, eg, greedy_star.T, levels=lvls, cmap=cm.winter, alpha=0.5)
ax.contour(tg, eg, greedy_star.T, colors='k', levels=lvls, linewidths=2)
ax.set(xlabel='θ', ylabel='ϵ')
ax.text(1.8, 2.5, 'new life', fontsize=14)
ax.text(4.5, 1.5, 'new job', fontsize=14, rotation='vertical')
ax.text(4.0, 4.5, 'stay put', fontsize=14)
plt.show()
```



In the new figure, you see that the region for which the worker stays put has grown because the distribution for ϵ has become more concentrated around the mean, making high-paying jobs less realistic.

CHAPTER

FIFTY

JOB SEARCH VII: ON-THE-JOB SEARCH

Contents

50.1 Overview

In this section, we solve a simple on-the-job search model

- based on [Ljungqvist and Sargent, 2018], exercise 6.18, and [Jovanovic, 1979]

Let's start with some imports:

```
import matplotlib.pyplot as plt
import numpy as np
import scipy.stats as stats
from numba import jit, prange
```

50.1.1 Model Features

- job-specific human capital accumulation combined with on-the-job search
- infinite-horizon dynamic programming with one state variable and two controls

50.2 Model

Let x_t denote the time-t job-specific human capital of a worker employed at a given firm and let w_t denote current wages.

Let $w_t = x_t(1 - s_t - \phi_t)$, where

- ϕ_t is investment in job-specific human capital for the current role and
- s_t is search effort, devoted to obtaining new offers from other firms.

For as long as the worker remains in the current job, evolution of $\{x_t\}$ is given by $x_{t+1} = g(x_t, \phi_t)$.

When search effort at t is s_t, the worker receives a new job offer with probability $\pi(s_t) \in [0, 1]$.

The value of the offer, measured in job-specific human capital, is u_{t+1}, where $\{u_t\}$ is IID with common distribution f.

The worker can reject the current offer and continue with existing job.

Hence $x_{t+1} = u_{t+1}$ if he/she accepts and $x_{t+1} = g(x_t, \phi_t)$ otherwise.

Let $b_{t+1} \in \{0, 1\}$ be a binary random variable, where $b_{t+1} = 1$ indicates that the worker receives an offer at the end of time t.

We can write

$$x_{t+1} = (1 - b_{t+1})g(x_t, \phi_t) + b_{t+1} \max\{g(x_t, \phi_t), u_{t+1}\} \tag{50.1}$$

Agent's objective: maximize expected discounted sum of wages via controls $\{s_t\}$ and $\{\phi_t\}$.

Taking the expectation of $v(x_{t+1})$ and using (50.1), the Bellman equation for this problem can be written as

$$v(x) = \max_{s+\phi \leq 1} \left\{ x(1 - s - \phi) + \beta(1 - \pi(s))v[g(x, \phi)] + \beta\pi(s) \int v[g(x, \phi) \vee u] f(du) \right\} \tag{50.2}$$

Here nonnegativity of s and ϕ is understood, while $a \vee b := \max\{a, b\}$.

50.2.1 Parameterization

In the implementation below, we will focus on the parameterization

$$g(x, \phi) = A(x\phi)^{\alpha}, \quad \pi(s) = \sqrt{s} \quad \text{and} \quad f = \text{Beta}(2, 2)$$

with default parameter values

- $A = 1.4$
- $\alpha = 0.6$
- $\beta = 0.96$

The Beta$(2, 2)$ distribution is supported on $(0, 1)$ - it has a unimodal, symmetric density peaked at 0.5.

50.2.2 Back-of-the-Envelope Calculations

Before we solve the model, let's make some quick calculations that provide intuition on what the solution should look like.

To begin, observe that the worker has two instruments to build capital and hence wages:

1. invest in capital specific to the current job via ϕ
2. search for a new job with better job-specific capital match via s

Since wages are $x(1 - s - \phi)$, marginal cost of investment via either ϕ or s is identical.

Our risk-neutral worker should focus on whatever instrument has the highest expected return.

The relative expected return will depend on x.

For example, suppose first that $x = 0.05$

- If $s = 1$ and $\phi = 0$, then since $g(x, \phi) = 0$, taking expectations of (50.1) gives expected next period capital equal to $\pi(s)\mathbb{E}u = \mathbb{E}u = 0.5$.
- If $s = 0$ and $\phi = 1$, then next period capital is $g(x, \phi) = g(0.05, 1) \approx 0.23$.

Both rates of return are good, but the return from search is better.

Next, suppose that $x = 0.4$

- If $s = 1$ and $\phi = 0$, then expected next period capital is again 0.5
- If $s = 0$ and $\phi = 1$, then $g(x, \phi) = g(0.4, 1) \approx 0.8$

Return from investment via ϕ dominates expected return from search.

Combining these observations gives us two informal predictions:

1. At any given state x, the two controls ϕ and s will function primarily as substitutes — worker will focus on whichever instrument has the higher expected return.
2. For sufficiently small x, search will be preferable to investment in job-specific human capital. For larger x, the reverse will be true.

Now let's turn to implementation, and see if we can match our predictions.

50.3 Implementation

We will set up a class `JVWorker` that holds the parameters of the model described above

```python
class JVWorker:
    r"""
    A Jovanovic-type model of employment with on-the-job search.

    """

    def __init__(self,
                 A=1.4,
                 α=0.6,
                 β=0.96,         # Discount factor
                 π=np.sqrt,      # Search effort function
                 a=2,            # Parameter of f
                 b=2,            # Parameter of f
                 grid_size=50,
                 mc_size=100,
                 ɛ=1e-4):

        self.A, self.α, self.β, self.π = A, α, β, π
        self.mc_size, self.ɛ = mc_size, ɛ

        self.g = jit(lambda x, ϕ: A * (x * ϕ)**α)    # Transition function
        self.f_rvs = np.random.beta(a, b, mc_size)

        # Max of grid is the max of a large quantile value for f and the
```

(continues on next page)

(continued from previous page)

```
        # fixed point y = g(y, 1)
        ε = 1e-4
        grid_max = max(A**(1 / (1 - α)), stats.beta(a, b).ppf(1 - ε))

        # Human capital
        self.x_grid = np.linspace(ε, grid_max, grid_size)
```

The function `operator_factory` takes an instance of this class and returns a jitted version of the Bellman operator `T`, i.e.

$$Tv(x) = \max_{s+\phi \leq 1} w(s, \phi)$$

where

$$w(s, \phi) := x(1 - s - \phi) + \beta(1 - \pi(s))v[g(x, \phi)] + \beta\pi(s) \int v[g(x, \phi) \vee u]f(du) \tag{50.3}$$

When we represent v, it will be with a NumPy array `v` giving values on grid `x_grid`.

But to evaluate the right-hand side of (50.3), we need a function, so we replace the arrays `v` and `x_grid` with a function `v_func` that gives linear interpolation of `v` on `x_grid`.

Inside the `for` loop, for each `x` in the grid over the state space, we set up the function $w(z) = w(s, \phi)$ defined in (50.3).

The function is maximized over all feasible (s, ϕ) pairs.

Another function, `get_greedy` returns the optimal choice of s and ϕ at each x, given a value function.

```
def operator_factory(jv, parallel_flag=True):

    """
    Returns a jitted version of the Bellman operator T

    jv is an instance of JVWorker

    """

    π, β = jv.π, jv.β
    x_grid, ε, mc_size = jv.x_grid, jv.ε, jv.mc_size
    f_rvs, g = jv.f_rvs, jv.g

    @jit
    def state_action_values(z, x, v):
        s, ϕ = z
        v_func = lambda x: np.interp(x, x_grid, v)

        integral = 0
        for m in range(mc_size):
            u = f_rvs[m]
            integral += v_func(max(g(x, ϕ), u))
        integral = integral / mc_size

        q = π(s) * integral + (1 - π(s)) * v_func(g(x, ϕ))
        return x * (1 - ϕ - s) + β * q

    @jit(parallel=parallel_flag)
    def T(v):
        """
```

(continues on next page)

(continued from previous page)

```
        The Bellman operator
        """

        v_new = np.empty_like(v)
        for i in prange(len(x_grid)):
            x = x_grid[i]

            # Search on a grid
            search_grid = np.linspace(ε, 1, 15)
            max_val = -1
            for s in search_grid:
                for ϕ in search_grid:
                    current_val = state_action_values((s, ϕ), x, v) if s + ϕ <= 1␣
↪else -1
                    if current_val > max_val:
                        max_val = current_val
            v_new[i] = max_val

        return v_new

    @jit
    def get_greedy(v):
        """
        Computes the v-greedy policy of a given function v
        """
        s_policy, ϕ_policy = np.empty_like(v), np.empty_like(v)

        for i in range(len(x_grid)):
            x = x_grid[i]
            # Search on a grid
            search_grid = np.linspace(ε, 1, 15)
            max_val = -1
            for s in search_grid:
                for ϕ in search_grid:
                    current_val = state_action_values((s, ϕ), x, v) if s + ϕ <= 1␣
↪else -1
                    if current_val > max_val:
                        max_val = current_val
                        max_s, max_ϕ = s, ϕ
                        s_policy[i], ϕ_policy[i] = max_s, max_ϕ
        return s_policy, ϕ_policy

    return T, get_greedy
```

To solve the model, we will write a function that uses the Bellman operator and iterates to find a fixed point.

```
def solve_model(jv,
                use_parallel=True,
                tol=1e-4,
                max_iter=1000,
                verbose=True,
                print_skip=25):

    """
    Solves the model by value function iteration

    * jv is an instance of JVWorker
```

(continues on next page)

(continued from previous page)

```
    """

    T, _ = operator_factory(jv, parallel_flag=use_parallel)

    # Set up loop
    v = jv.x_grid * 0.5  # Initial condition
    i = 0
    error = tol + 1

    while i < max_iter and error > tol:
        v_new = T(v)
        error = np.max(np.abs(v - v_new))
        i += 1
        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")
        v = v_new

    if error > tol:
        print("Failed to converge!")
    elif verbose:
        print(f"\nConverged in {i} iterations.")

    return v_new
```

50.4 Solving for Policies

Let's generate the optimal policies and see what they look like.

```
jv = JVWorker()
T, get_greedy = operator_factory(jv)
v_star = solve_model(jv)
s_star, ϕ_star = get_greedy(v_star)
```

```
Error at iteration 25 is 0.15111140366118292.
Error at iteration 50 is 0.05446005373350182.
Error at iteration 75 is 0.01962722456546473.
Error at iteration 100 is 0.007073587294430084.
Error at iteration 125 is 0.002549297637326475 3.
Error at iteration 150 is 0.0009187584987859765.
Error at iteration 175 is 0.0003311175465494642 4.
Error at iteration 200 is 0.00011933367667715 8.

Converged in 205 iterations.
```

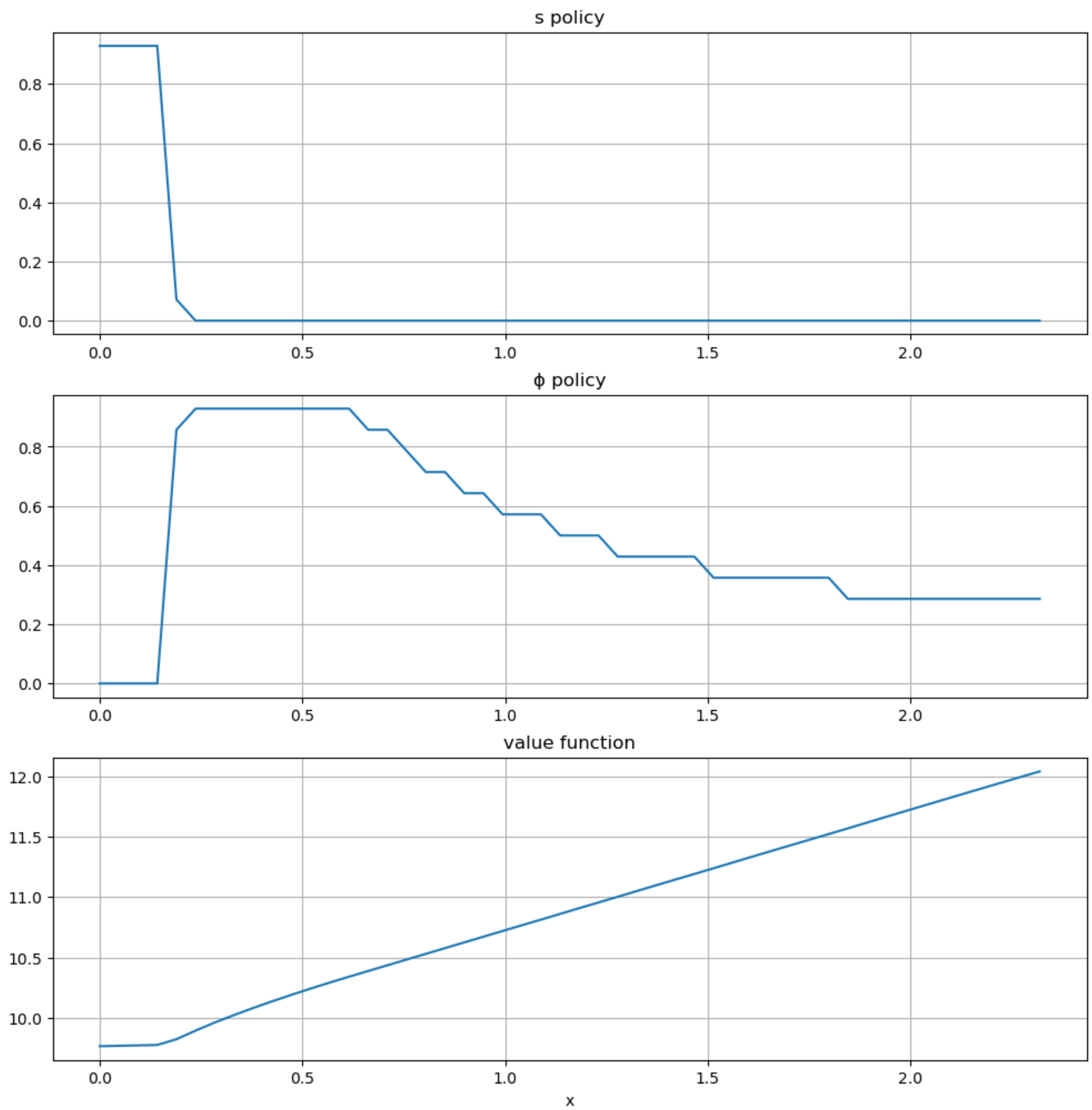
Here are the plots:

```
plots = [s_star, ϕ_star, v_star]
titles = ["s policy", "ϕ policy",  "value function"]

fig, axes = plt.subplots(3, 1, figsize=(12, 12))

for ax, plot, title in zip(axes, plots, titles):
    ax.plot(jv.x_grid, plot)
```

(continues on next page)

(continued from previous page)

```
    ax.set(title=title)
    ax.grid()

axes[-1].set_xlabel("x")
plt.show()
```



The horizontal axis is the state x, while the vertical axis gives $s(x)$ and $\phi(x)$.

Overall, the policies match well with our predictions from *above*

- Worker switches from one investment strategy to the other depending on relative return.
- For low values of x, the best option is to search for a new job.
- Once x is larger, worker does better by investing in human capital specific to the current position.

50.5 Exercises

Exercise 50.5.1

Let's look at the dynamics for the state process $\{x_t\}$ associated with these policies.

The dynamics are given by (50.1) when ϕ_t and s_t are chosen according to the optimal policies, and $\mathbb{P}\{b_{t+1} = 1\} = \pi(s_t)$.

Since the dynamics are random, analysis is a bit subtle.

One way to do it is to plot, for each x in a relatively fine grid called `plot_grid`, a large number K of realizations of x_{t+1} given $x_t = x$.

Plot this with one dot for each realization, in the form of a 45 degree diagram, setting

```
jv = JVWorker(grid_size=25, mc_size=50)
plot_grid_max, plot_grid_size = 1.2, 100
plot_grid = np.linspace(0, plot_grid_max, plot_grid_size)
fig, ax = plt.subplots()
ax.set_xlim(0, plot_grid_max)
ax.set_ylim(0, plot_grid_max)
```

By examining the plot, argue that under the optimal policies, the state x_t will converge to a constant value $\bar{x}$ close to unity.

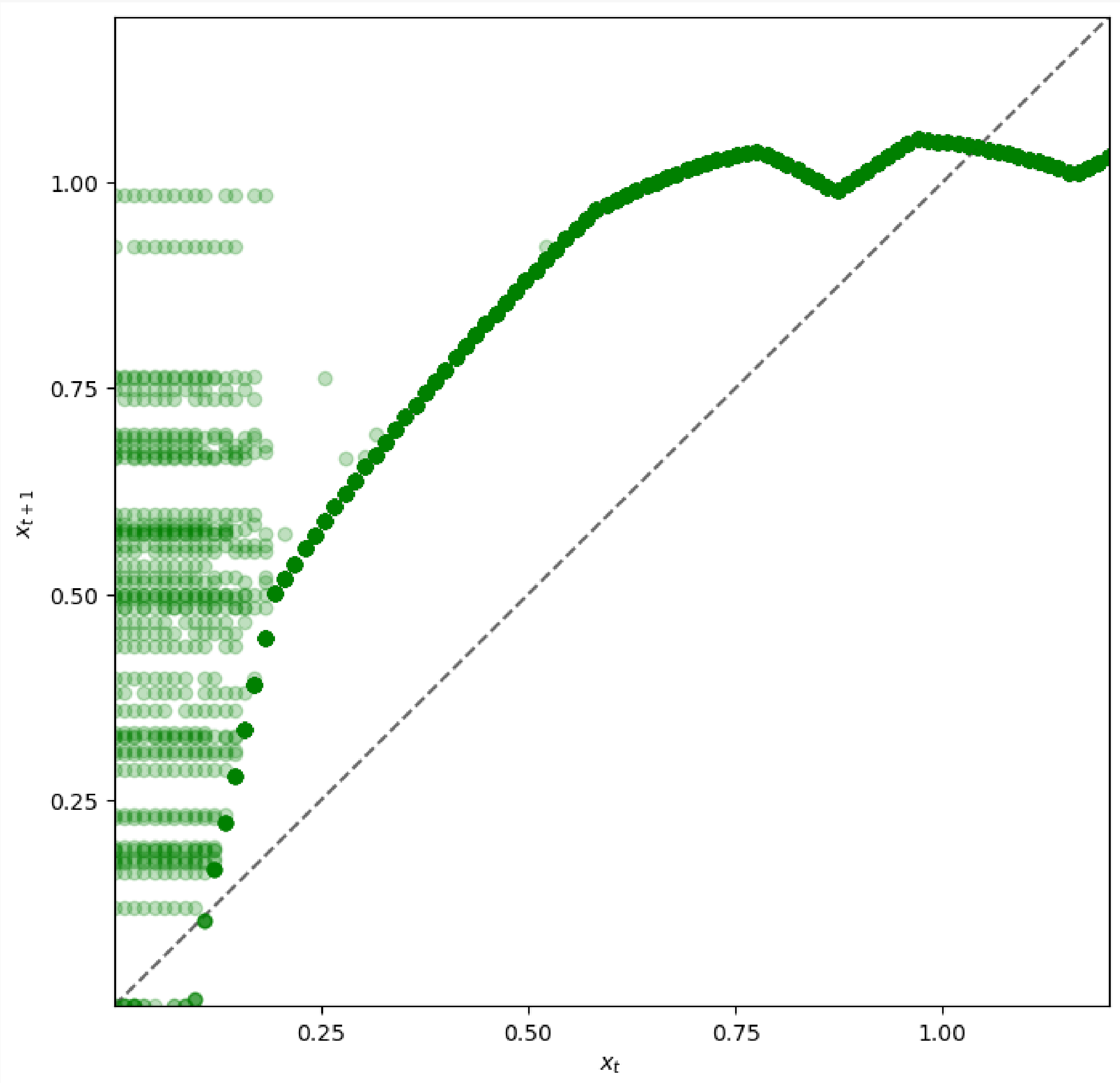
Argue that at the steady state, $s_t \approx 0$ and $\phi_t \approx 0.6$.

Solution

Here's code to produce the 45 degree diagram

```
jv = JVWorker(grid_size=25, mc_size=50)
π, g, f_rvs, x_grid = jv.π, jv.g, jv.f_rvs, jv.x_grid
T, get_greedy = operator_factory(jv)
v_star = solve_model(jv, verbose=False)
s_policy, ϕ_policy = get_greedy(v_star)

# Turn the policy function arrays into actual functions
s = lambda y: np.interp(y, x_grid, s_policy)
ϕ = lambda y: np.interp(y, x_grid, ϕ_policy)

def h(x, b, u):
    return (1 - b) * g(x, ϕ(x)) + b * max(g(x, ϕ(x)), u)


plot_grid_max, plot_grid_size = 1.2, 100
plot_grid = np.linspace(0, plot_grid_max, plot_grid_size)
fig, ax = plt.subplots(figsize=(8, 8))
ticks = (0.25, 0.5, 0.75, 1.0)
ax.set(xticks=ticks, yticks=ticks,
       xlim=(0, plot_grid_max),
       ylim=(0, plot_grid_max),
       xlabel='$x_t$', ylabel='$x_{t+1}$')

ax.plot(plot_grid, plot_grid, 'k--', alpha=0.6)  # 45 degree line
for x in plot_grid:
```

```
    for i in range(jv.mc_size):
        b = 1 if np.random.uniform(0, 1) < π(s(x)) else 0
        u = f_rvs[i]
        y = h(x, b, u)
        ax.plot(x, y, 'go', alpha=0.25)

plt.show()
```



Looking at the dynamics, we can see that

- If x_t is below about 0.2 the dynamics are random, but $x_{t+1} > x_t$ is very likely.
- As x_t increases the dynamics become deterministic, and x_t converges to a steady state value close to 1.

Referring back to the figure *here* we see that $x_t \approx 1$ means that $s_t = s(x_t) \approx 0$ and $\phi_t = \phi(x_t) \approx 0.6$.

Exercise 50.5.2

In Exercise 50.5.1, we found that s_t converges to zero and ϕ_t converges to about 0.6.

Since these results were calculated at a value of β close to one, let's compare them to the best choice for an *infinitely* patient worker.

Intuitively, an infinitely patient worker would like to maximize steady state wages, which are a function of steady state capital.

You can take it as given—it's certainly true—that the infinitely patient worker does not search in the long run (i.e., $s_t = 0$ for large t).

Thus, given ϕ, steady state capital is the positive fixed point $x^*(\phi)$ of the map $x \mapsto g(x, \phi)$.

Steady state wages can be written as $w^*(\phi) = x^*(\phi)(1 - \phi)$.

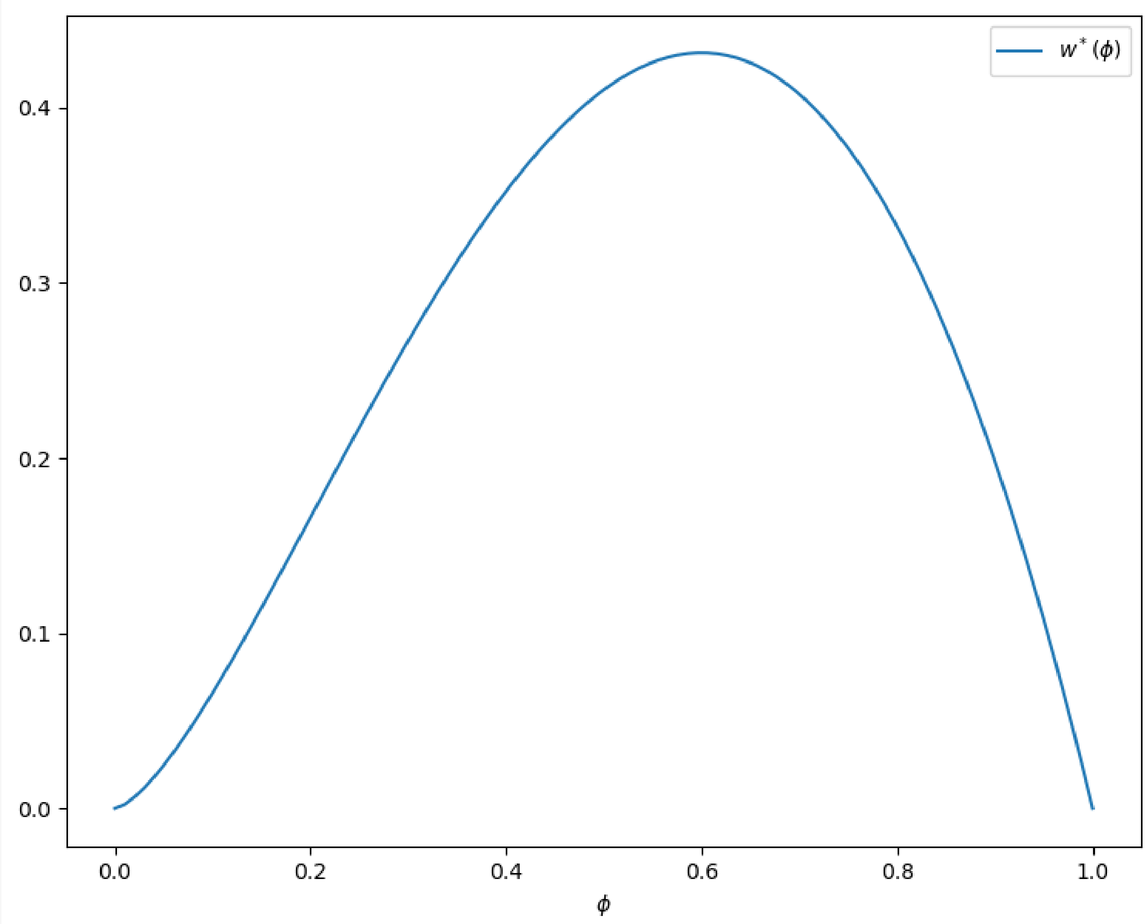
Graph $w^*(\phi)$ with respect to ϕ, and examine the best choice of ϕ.

Can you give a rough interpretation for the value that you see?

Solution

The figure can be produced as follows

```
jv = JVWorker()

def xbar(ϕ):
    A, α = jv.A, jv.α
    return (A * ϕ**α)**(1 / (1 - α))

ϕ_grid = np.linspace(0, 1, 100)
fig, ax = plt.subplots(figsize=(9, 7))
ax.set(xlabel=r'$\phi$')
ax.plot(ϕ_grid, [xbar(ϕ) * (1 - ϕ) for ϕ in ϕ_grid], label=r'$w^*(\phi)$')
ax.legend()

plt.show()
```



Observe that the maximizer is around 0.6.

This is similar to the long-run value for ϕ obtained in Exercise 50.5.1.

Hence the behavior of the infinitely patent worker is similar to that of the worker with $\beta = 0.96$.

This seems reasonable and helps us confirm that our dynamic programming solutions are probably correct.

CHAPTER

FIFTYONE

JOB SEARCH VIII: SEARCH WITH LEARNING

Contents

In addition to what's in Anaconda, this lecture deploys the libraries:

```
!pip install interpolation
```

51.1 Overview

In this lecture, we consider an extension of the previously studied job search model of McCall [McCall, 1970].

We'll build on a model of Bayesian learning discussed in this lecture on the topic of exchangeability and its relationship to the concept of IID (identically and independently distributed) random variables and to Bayesian updating.

In the McCall model, an unemployed worker decides when to accept a permanent job at a specific fixed wage, given

- his or her discount factor
- the level of unemployment compensation

- the distribution from which wage offers are drawn

In the version considered below, the wage distribution is unknown and must be learned.

- The following is based on the presentation in [Ljungqvist and Sargent, 2018], section 6.6.

Let's start with some imports

```
import matplotlib.pyplot as plt
from numba import jit, prange, vectorize
from interpolation import mlinterp
from math import gamma
import numpy as np
from matplotlib import cm
import scipy.optimize as op
from scipy.stats import cumfreq, beta
```

51.1.1 Model Features

- Infinite horizon dynamic programming with two states and one binary control.
- Bayesian updating to learn the unknown distribution.

51.2 Model

Let's first review the basic McCall model [McCall, 1970] and then add the variation we want to consider.

51.2.1 The Basic McCall Model

Recall that, *in the baseline model*, an unemployed worker is presented in each period with a permanent job offer at wage W_t.

At time t, our worker either

1. accepts the offer and works permanently at constant wage W_t
2. rejects the offer, receives unemployment compensation c and reconsiders next period

The wage sequence W_t is IID and generated from known density q.

The worker aims to maximize the expected discounted sum of earnings $\mathbb{E} \sum_{t=0}^{\infty} \beta^t y_t$.

Let $v(w)$ be the optimal value of the problem for a previously unemployed worker who has just received offer w and is yet to decide whether to accept or reject the offer.

The value function v satisfies the recursion

$$v(w) = \max \left\{ \frac{w}{1-\beta}, \, c + \beta \int v(w')q(w')dw' \right\} \tag{51.1}$$

The optimal policy has the form $\mathbf{1}\{w \geq \bar{w}\}$, where $\bar{w}$ is a constant called the **reservation wage**.

51.2.2 Offer Distribution Unknown

Now let's extend the model by considering the variation presented in [Ljungqvist and Sargent, 2018], section 6.6.

The model is as above, apart from the fact that

- the density q is unknown
- the worker learns about q by starting with a prior and updating based on wage offers that he/she observes

The worker knows there are two possible distributions F and G.

These two distributions have densities f and g, repectively.

Just before time starts, "nature" selects q to be either f or g.

This is then the wage distribution from which the entire sequence W_t will be drawn.

The worker does not know which distribution nature has drawn, but the worker does know the two possible distributions f and g.

The worker puts a (subjective) prior probability π_0 on f having been chosen.

The worker's time 0 subjective distribution for the distribution of W_0 is

$$\pi_0 f + (1 - \pi_0) g$$

The worker's time t subjective belief about the the distribution of W_t is

$$\pi_t f + (1 - \pi_t) g,$$

where π_t updates via

$$\pi_{t+1} = \frac{\pi_t f(w_{t+1})}{\pi_t f(w_{t+1}) + (1 - \pi_t) g(w_{t+1})} \tag{51.2}$$

This last expression follows from Bayes' rule, which tells us that

$$\mathbb{P}\{q = f \,|\, W = w\} = \frac{\mathbb{P}\{W = w \,|\, q = f\} \mathbb{P}\{q = f\}}{\mathbb{P}\{W = w\}}$$

and

$$\mathbb{P}\{W = w\} = \sum_{\omega \in \{f, g\}} \mathbb{P}\{W = w \,|\, q = \omega\} \mathbb{P}\{q = \omega\}$$

The fact that (51.2) is recursive allows us to progress to a recursive solution method.

Letting

$$q_\pi(w) := \pi f(w) + (1 - \pi) g(w)$$

and

$$\kappa(w, \pi) := \frac{\pi f(w)}{\pi f(w) + (1 - \pi) g(w)}$$

we can express the value function for the unemployed worker recursively as follows

$$v(w, \pi) = \max \left\{ \frac{w}{1 - \beta}, \, c + \beta \int v(w', \pi') \, q_\pi(w') \, dw' \right\} \quad \text{where} \quad \pi' = \kappa(w', \pi) \tag{51.3}$$

Notice that the current guess π is a state variable, since it affects the worker's perception of probabilities for future rewards.

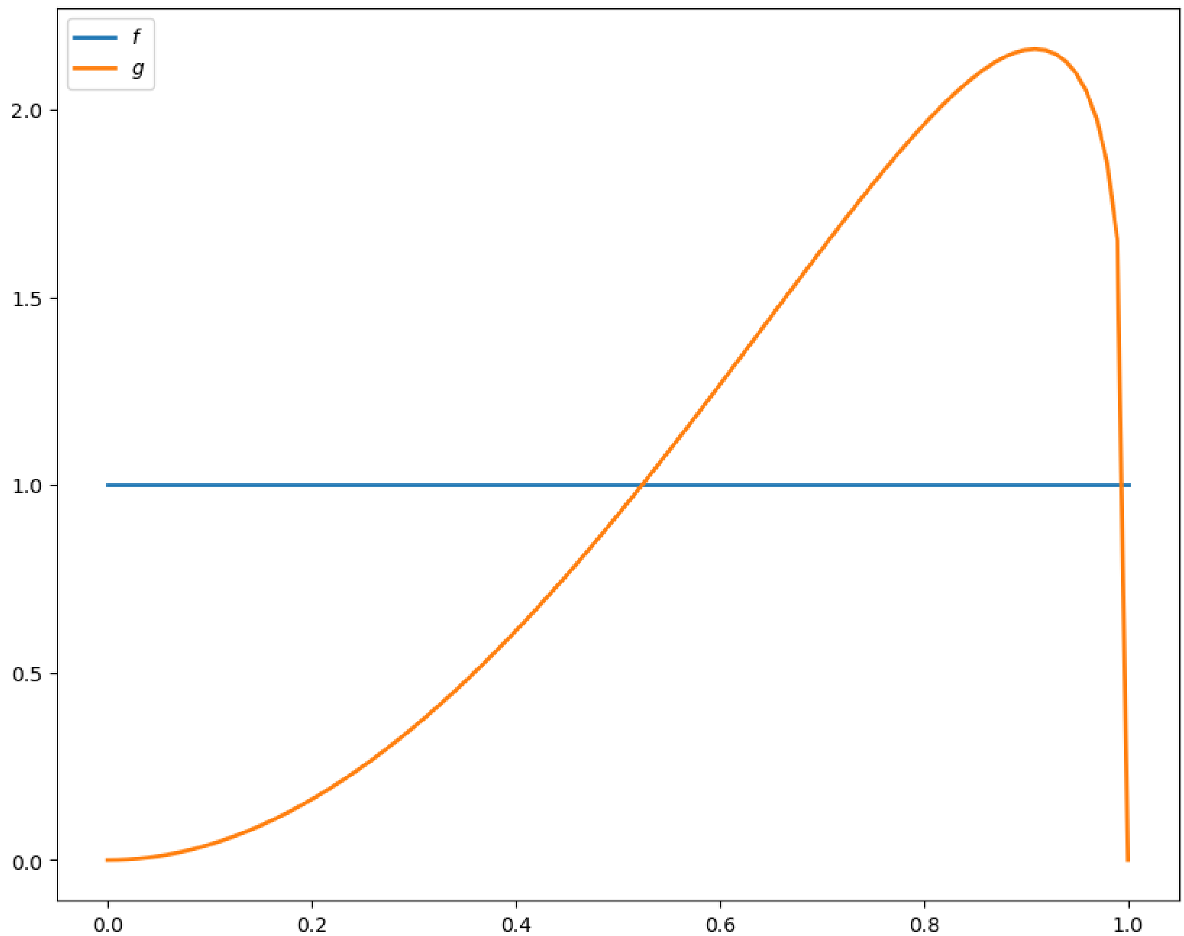
51.2.3 Parameterization

Following section 6.6 of [Ljungqvist and Sargent, 2018], our baseline parameterization will be

- f is $\text{Beta}(1, 1)$
- g is $\text{Beta}(3, 1.2)$
- $\beta = 0.95$ and $c = 0.3$

The densities f and g have the following shape

```
@vectorize
def p(x, a, b):
    r = gamma(a + b) / (gamma(a) * gamma(b))
    return r * x**(a-1) * (1 - x)**(b-1)


x_grid = np.linspace(0, 1, 100)
f = lambda x: p(x, 1, 1)
g = lambda x: p(x, 3, 1.2)

fig, ax = plt.subplots(figsize=(10, 8))
ax.plot(x_grid, f(x_grid), label='$f$', lw=2)
ax.plot(x_grid, g(x_grid), label='$g$', lw=2)

ax.legend()
plt.show()
```



51.2.4 Looking Forward

What kind of optimal policy might result from (51.3) and the parameterization specified above?

Intuitively, if we accept at w_a and $w_a \leq w_b$, then — all other things being given — we should also accept at w_b.

This suggests a policy of accepting whenever w exceeds some threshold value $\bar{w}$.

But $\bar{w}$ should depend on π — in fact, it should be decreasing in π because

- f is a less attractive offer distribution than g
- larger π means more weight on f and less on g

Thus, larger π depresses the worker's assessment of her future prospects, so relatively low current offers become more attractive.

Summary: We conjecture that the optimal policy is of the form $\mathbb{1}w \geq \bar{w}(\pi)$ for some decreasing function $\bar{w}$.

51.3 Take 1: Solution by VFI

Let's set about solving the model and see how our results match with our intuition.

We begin by solving via value function iteration (VFI), which is natural but ultimately turns out to be second best.

The class `SearchProblem` is used to store parameters and methods needed to compute optimal actions.

```
class SearchProblem:
    """
    A class to store a given parameterization of the "offer distribution
    unknown" model.

    """

    def __init__(self,
                 β=0.95,            # Discount factor
                 c=0.3,             # Unemployment compensation
                 F_a=1,
                 F_b=1,
                 G_a=3,
                 G_b=1.2,
                 w_max=1,           # Maximum wage possible
                 w_grid_size=100,
                 π_grid_size=100,
                 mc_size=500):

        self.β, self.c, self.w_max = β, c, w_max

        self.f = jit(lambda x: p(x, F_a, F_b))
        self.g = jit(lambda x: p(x, G_a, G_b))

        self.π_min, self.π_max = 1e-3, 1-1e-3    # Avoids instability
        self.w_grid = np.linspace(0, w_max, w_grid_size)
        self.π_grid = np.linspace(self.π_min, self.π_max, π_grid_size)

        self.mc_size = mc_size

        self.w_f = np.random.beta(F_a, F_b, mc_size)
        self.w_g = np.random.beta(G_a, G_b, mc_size)
```

The following function takes an instance of this class and returns jitted versions of the Bellman operator `T`, and a `get_greedy()` function to compute the approximate optimal policy from a guess `v` of the value function

```
def operator_factory(sp, parallel_flag=True):

    f, g = sp.f, sp.g
    w_f, w_g = sp.w_f, sp.w_g
    β, c = sp.β, sp.c
    mc_size = sp.mc_size
    w_grid, π_grid = sp.w_grid, sp.π_grid

    @jit
    def v_func(x, y, v):
        return mlinterp((w_grid, π_grid), v, (x, y))

    @jit
    def κ(w, π):
```

(continues on next page)

(continued from previous page)

```
        """
        Updates π using Bayes' rule and the current wage observation w.
        """
        pf, pg = π * f(w), (1 - π) * g(w)
        π_new = pf / (pf + pg)

        return π_new

    @jit(parallel=parallel_flag)
    def T(v):
        """
        The Bellman operator.

        """
        v_new = np.empty_like(v)

        for i in prange(len(w_grid)):
            for j in prange(len(π_grid)):
                w = w_grid[i]
                π = π_grid[j]

                v_1 = w / (1 - β)

                integral_f, integral_g = 0, 0
                for m in prange(mc_size):
                    integral_f += v_func(w_f[m], κ(w_f[m], π), v)
                    integral_g += v_func(w_g[m], κ(w_g[m], π), v)
                integral = (π * integral_f + (1 - π) * integral_g) / mc_size

                v_2 = c + β * integral
                v_new[i, j] = max(v_1, v_2)

        return v_new

    @jit(parallel=parallel_flag)
    def get_greedy(v):
        """"
        Compute optimal actions taking v as the value function.

        """
        σ = np.empty_like(v)

        for i in prange(len(w_grid)):
            for j in prange(len(π_grid)):
                w = w_grid[i]
                π = π_grid[j]

                v_1 = w / (1 - β)

                integral_f, integral_g = 0, 0
                for m in prange(mc_size):
                    integral_f += v_func(w_f[m], κ(w_f[m], π), v)
                    integral_g += v_func(w_g[m], κ(w_g[m], π), v)
                integral = (π * integral_f + (1 - π) * integral_g) / mc_size

                v_2 = c + β * integral
```

(continues on next page)

(continued from previous page)

```
                σ[i, j] = v_1 > v_2  # Evaluates to 1 or 0

        return σ

    return T, get_greedy
```

We will omit a detailed discussion of the code because there is a more efficient solution method that we will use later.

To solve the model we will use the following function that iterates using T to find a fixed point

```
def solve_model(sp,
                use_parallel=True,
                tol=1e-4,
                max_iter=1000,
                verbose=True,
                print_skip=5):

    """
    Solves for the value function

    * sp is an instance of SearchProblem
    """

    T, _ = operator_factory(sp, use_parallel)

    # Set up loop
    i = 0
    error = tol + 1
    m, n = len(sp.w_grid), len(sp.π_grid)

    # Initialize v
    v = np.zeros((m, n)) + sp.c / (1 - sp.β)

    while i < max_iter and error > tol:
        v_new = T(v)
        error = np.max(np.abs(v - v_new))
        i += 1
        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")
        v = v_new

    if error > tol:
        print("Failed to converge!")
    elif verbose:
        print(f"\nConverged in {i} iterations.")


    return v_new
```

Let's look at solutions computed from value function iteration

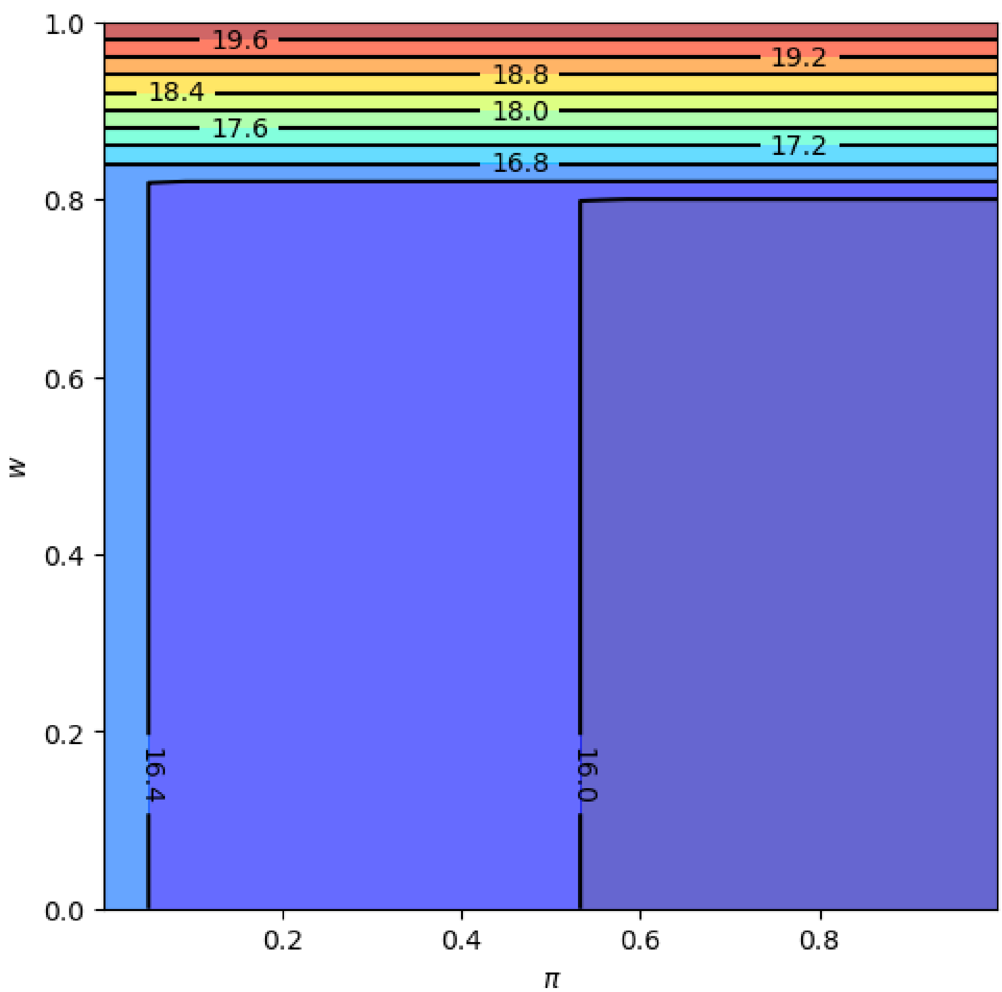
```
sp = SearchProblem()
v_star = solve_model(sp)
fig, ax = plt.subplots(figsize=(6, 6))
ax.contourf(sp.π_grid, sp.w_grid, v_star, 12, alpha=0.6, cmap=cm.jet)
cs = ax.contour(sp.π_grid, sp.w_grid, v_star, 12, colors="black")
ax.clabel(cs, inline=1, fontsize=10)
```

(continues on next page)

(continued from previous page)

```
ax.set(xlabel=r'$\pi$', ylabel='$w$')

plt.show()
```

```
Error at iteration 5 is 0.5969764417354373.
Error at iteration 10 is 0.08771064236289305.
Error at iteration 15 is 0.017140165378641825.
Error at iteration 20 is 0.003567524961193058.
Error at iteration 25 is 0.0007428257738641975.
Error at iteration 30 is 0.00015467915391376152.

Converged in 32 iterations.
```



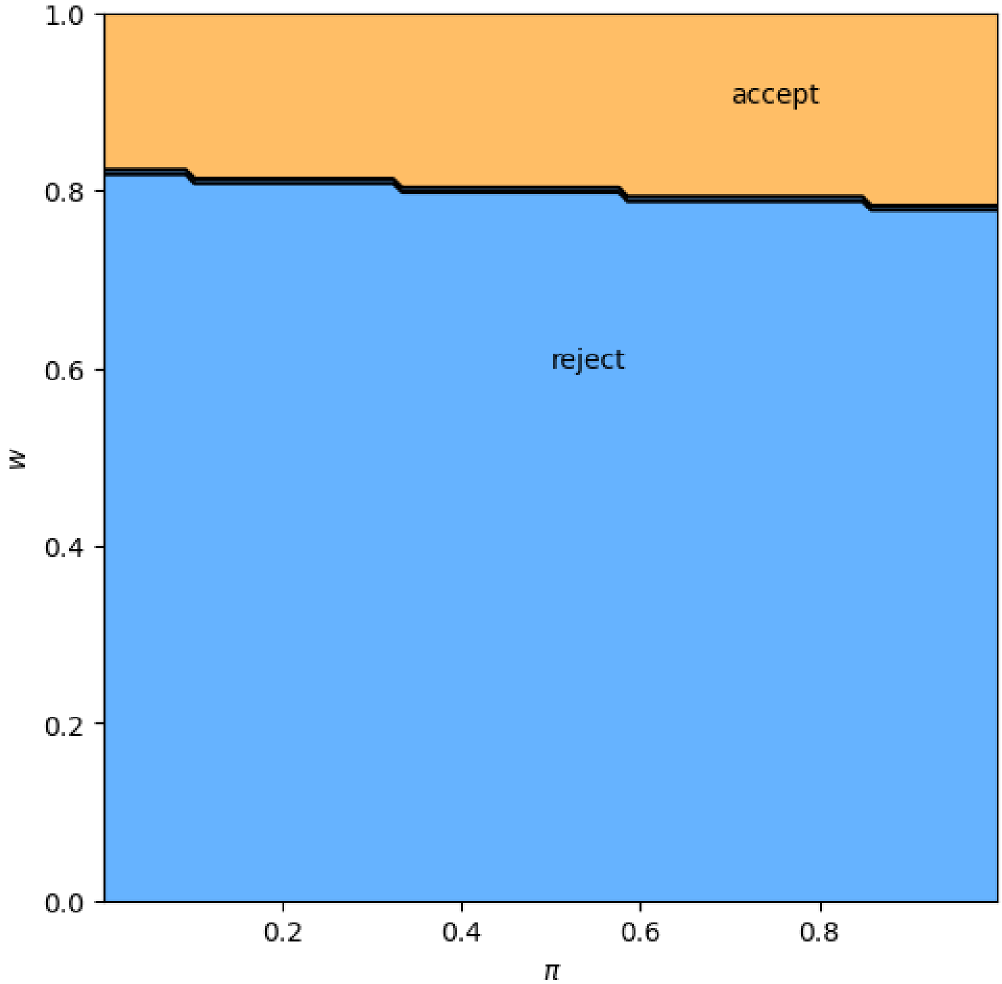
We will also plot the optimal policy

```
T, get_greedy = operator_factory(sp)
σ_star = get_greedy(v_star)

fig, ax = plt.subplots(figsize=(6, 6))
ax.contourf(sp.π_grid, sp.w_grid, σ_star, 1, alpha=0.6, cmap=cm.jet)
ax.contour(sp.π_grid, sp.w_grid, σ_star, 1, colors="black")
```

(continues on next page)

(continued from previous page)

```
ax.set(xlabel=r'$\pi$', ylabel='$w$')

ax.text(0.5, 0.6, 'reject')
ax.text(0.7, 0.9, 'accept')

plt.show()
```



The results fit well with our intuition from section *looking forward*.

- The black line in the figure above corresponds to the function $\bar{w}(\pi)$ introduced there.
- It is decreasing as expected.

51.4 Take 2: A More Efficient Method

Let's consider another method to solve for the optimal policy.

We will use iteration with an operator that has the same contraction rate as the Bellman operator, but

- one dimensional rather than two dimensional
- no maximization step

As a consequence, the algorithm is orders of magnitude faster than VFI.

This section illustrates the point that when it comes to programming, a bit of mathematical analysis goes a long way.

51.5 Another Functional Equation

To begin, note that when $w = \bar{w}(\pi)$, the worker is indifferent between accepting and rejecting.

Hence the two choices on the right-hand side of (51.3) have equal value:

$$\frac{\bar{w}(\pi)}{1-\beta} = c + \beta \int v(w', \pi') \, q_\pi(w') \, dw' \tag{51.4}$$

Together, (51.3) and (51.4) give

$$v(w, \pi) = \max \left\{ \frac{w}{1-\beta}, \frac{\bar{w}(\pi)}{1-\beta} \right\} \tag{51.5}$$

Combining (51.4) and (51.5), we obtain

$$\frac{\bar{w}(\pi)}{1-\beta} = c + \beta \int \max \left\{ \frac{w'}{1-\beta}, \frac{\bar{w}(\pi')}{1-\beta} \right\} \, q_\pi(w') \, dw'$$

Multiplying by $1 - \beta$, substituting in $\pi' = \kappa(w', \pi)$ and using $\circ$ for composition of functions yields

$$\bar{w}(\pi) = (1-\beta)c + \beta \int \max \left\{ w', \bar{w} \circ \kappa(w', \pi) \right\} \, q_\pi(w') \, dw' \tag{51.6}$$

Equation (51.6) can be understood as a functional equation, where $\bar{w}$ is the unknown function.

- Let's call it the **reservation wage functional equation** (RWFE).
- The solution $\bar{w}$ to the RWFE is the object that we wish to compute.

51.6 Solving the RWFE

To solve the RWFE, we will first show that its solution is the fixed point of a contraction mapping.

To this end, let

- $b[0, 1]$ be the bounded real-valued functions on $[0, 1]$
- $\|\omega\| := \sup_{x \in [0,1]} |\omega(x)|$

Consider the operator Q mapping $\omega \in b[0, 1]$ into $Q\omega \in b[0, 1]$ via

$$(Q\omega)(\pi) = (1-\beta)c + \beta \int \max \left\{ w', \omega \circ \kappa(w', \pi) \right\} \, q_\pi(w') \, dw' \tag{51.7}$$

Comparing (51.6) and (51.7), we see that the set of fixed points of Q exactly coincides with the set of solutions to the RWFE.

- If $Q\bar{w} = \bar{w}$ then $\bar{w}$ solves (51.6) and vice versa.

Moreover, for any $\omega, \omega' \in b[0,1]$, basic algebra and the triangle inequality for integrals tells us that

$$|(Q\omega)(\pi) - (Q\omega')(\pi)| \leq \beta \int |\max\{w', \omega \circ \kappa(w', \pi)\} - \max\{w', \omega' \circ \kappa(w', \pi)\}| \; q_\pi(w') \, dw' \tag{51.8}$$

Working case by case, it is easy to check that for real numbers a, b, c we always have

$$|\max\{a, b\} - \max\{a, c\}| \leq |b - c| \tag{51.9}$$

Combining (51.8) and (51.9) yields

$$|(Q\omega)(\pi) - (Q\omega')(\pi)| \leq \beta \int |\omega \circ \kappa(w', \pi) - \omega' \circ \kappa(w', \pi)| \; q_\pi(w') \, dw' \leq \beta \|\omega - \omega'\| \tag{51.10}$$

Taking the supremum over π now gives us

$$\|Q\omega - Q\omega'\| \leq \beta \|\omega - \omega'\| \tag{51.11}$$

In other words, Q is a contraction of modulus β on the complete metric space $(b[0,1], \|\cdot\|)$.

Hence

- A unique solution $\bar{w}$ to the RWFE exists in $b[0,1]$.
- $Q^k \omega \to \bar{w}$ uniformly as $k \to \infty$, for any $\omega \in b[0,1]$.

51.7 Implementation

The following function takes an instance of `SearchProblem` and returns the operator `Q`

```
def Q_factory(sp, parallel_flag=True):

    f, g = sp.f, sp.g
    w_f, w_g = sp.w_f, sp.w_g
    β, c = sp.β, sp.c
    mc_size = sp.mc_size
    w_grid, π_grid = sp.w_grid, sp.π_grid

    @jit
    def ω_func(p, ω):
        return np.interp(p, π_grid, ω)

    @jit
    def κ(w, π):
        """
        Updates π using Bayes' rule and the current wage observation w.
        """
        pf, pg = π * f(w), (1 - π) * g(w)
        π_new = pf / (pf + pg)

        return π_new

    @jit(parallel=parallel_flag)
    def Q(ω):
        """
```

(continues on next page)

(continued from previous page)

```
        Updates the reservation wage function guess ω via the operator
        Q.

        """
        ω_new = np.empty_like(ω)

        for i in prange(len(π_grid)):
            π = π_grid[i]
            integral_f, integral_g = 0, 0

            for m in prange(mc_size):
                integral_f += max(w_f[m], ω_func(κ(w_f[m], π), ω))
                integral_g += max(w_g[m], ω_func(κ(w_g[m], π), ω))
            integral = (π * integral_f + (1 - π) * integral_g) / mc_size

            ω_new[i] = (1 - β) * c + β * integral

        return ω_new

    return Q
```

In the next exercise, you are asked to compute an approximation to $\bar{w}$.

51.8 Exercises

Exercise 51.8.1

Use the default parameters and `Q_factory` to compute an optimal policy.

Your result should coincide closely with the figure for the optimal policy *shown above*.

Try experimenting with different parameters, and confirm that the change in the optimal policy coincides with your intuition.

51.9 Solutions

Solution

This code solves the "Offer Distribution Unknown" model by iterating on a guess of the reservation wage function.

You should find that the run time is shorter than that of the value function approach.

Similar to above, we set up a function to iterate with `Q` to find the fixed point

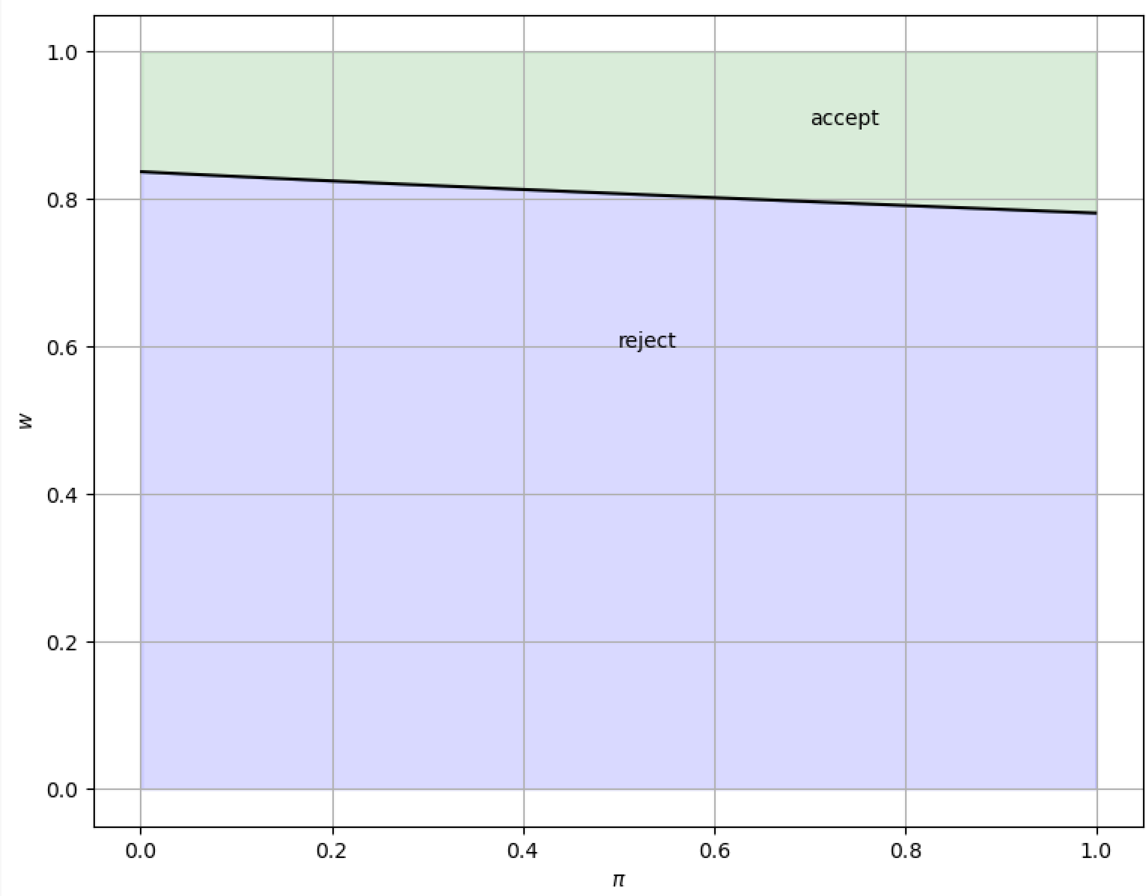
```
def solve_wbar(sp,
               use_parallel=True,
               tol=1e-4,
               max_iter=1000,
               verbose=True,
               print_skip=5):

    Q = Q_factory(sp, use_parallel)
```

```
    # Set up loop
    i = 0
    error = tol + 1
    m, n = len(sp.w_grid), len(sp.π_grid)

    # Initialize w
    w = np.ones_like(sp.π_grid)

    while i < max_iter and error > tol:
        w_new = Q(w)
        error = np.max(np.abs(w - w_new))
        i += 1
        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")
        w = w_new

    if error > tol:
        print("Failed to converge!")
    elif verbose:
        print(f"\nConverged in {i} iterations.")

    return w_new
```

The solution can be plotted as follows

```
sp = SearchProblem()
w_bar = solve_wbar(sp)

fig, ax = plt.subplots(figsize=(9, 7))

ax.plot(sp.π_grid, w_bar, color='k')
ax.fill_between(sp.π_grid, 0, w_bar, color='blue', alpha=0.15)
ax.fill_between(sp.π_grid, w_bar, sp.w_max, color='green', alpha=0.15)
ax.text(0.5, 0.6, 'reject')
ax.text(0.7, 0.9, 'accept')
ax.set(xlabel=r'$\pi$', ylabel='$w$')
ax.grid()
plt.show()
```

```
Error at iteration 5 is 0.020764139953211358.
Error at iteration 10 is 0.00651643788166878.
Error at iteration 15 is 0.0016898998756010863.
Error at iteration 20 is 0.0004108877926140009.
Error at iteration 25 is 9.787252812576419e-05.

Converged in 25 iterations.
```



51.10 Appendix A

The next piece of code generates a fun simulation to see what the effect of a change in the underlying distribution on the unemployment rate is.

At a point in the simulation, the distribution becomes significantly worse.

It takes a while for agents to learn this, and in the meantime, they are too optimistic and turn down too many jobs.

As a result, the unemployment rate spikes

```
F_a, F_b, G_a, G_b = 1, 1, 3, 1.2

sp = SearchProblem(F_a=F_a, F_b=F_b, G_a=G_a, G_b=G_b)
f, g = sp.f, sp.g

# Solve for reservation wage
w_bar = solve_wbar(sp, verbose=False)

# Interpolate reservation wage function
π_grid = sp.π_grid
w_func = jit(lambda x: np.interp(x, π_grid, w_bar))
```

(continues on next page)

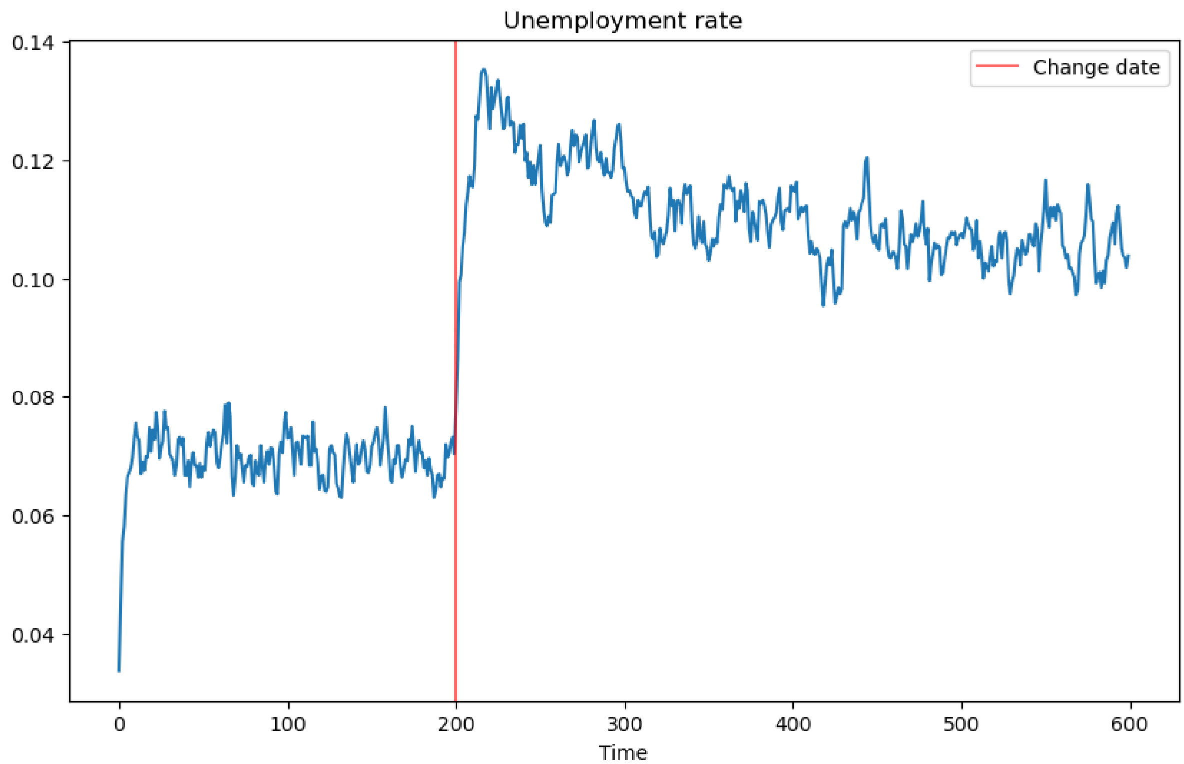
(continued from previous page)

```
@jit
def update(a, b, e, π):
    "Update e and π by drawing wage offer from beta distribution with parameters a␣
↪and b"

    if e == False:
        w = np.random.beta(a, b)       # Draw random wage
        if w >= w_func(π):
            e = True                   # Take new job
        else:
            π = 1 / (1 + ((1 - π) * g(w)) / (π * f(w)))

    return e, π

@jit
def simulate_path(F_a=F_a,
                  F_b=F_b,
                  G_a=G_a,
                  G_b=G_b,
                  N=5000,       # Number of agents
                  T=600,        # Simulation length
                  d=200,        # Change date
                  s=0.025):     # Separation rate

    """Simulates path of employment for N number of works over T periods"""

    e = np.ones((N, T+1))
    π = np.full((N, T+1), 1e-3)

    a, b = G_a, G_b   # Initial distribution parameters

    for t in range(T+1):

        if t == d:
            a, b = F_a, F_b  # Change distribution parameters

        # Update each agent
        for n in range(N):
            if e[n, t] == 1:                    # If agent is currently employment
                p = np.random.uniform(0, 1)
                if p <= s:                      # Randomly separate with probability s
                    e[n, t] = 0

            new_e, new_π = update(a, b, e[n, t], π[n, t])
            e[n, t+1] = new_e
            π[n, t+1] = new_π

    return e[:, 1:]

d = 200  # Change distribution at time d
unemployment_rate = 1 - simulate_path(d=d).mean(axis=0)

fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(unemployment_rate)
ax.axvline(d, color='r', alpha=0.6, label='Change date')
ax.set_xlabel('Time')
ax.set_title('Unemployment rate')
```

(continues on next page)

(continued from previous page)

```
ax.legend()
plt.show()
```



51.11 Appendix B

In this appendix we provide more details about how Bayes' Law contributes to the workings of the model.

We present some graphs that bring out additional insights about how learning works.

We build on graphs proposed in this lecture.

In particular, we'll add actions of our searching worker to a key graph presented in that lecture.

To begin, we first define two functions for computing the empirical distributions of unemployment duration and π at the time of employment.

```
@jit
def empirical_dist(F_a, F_b, G_a, G_b, w_bar, π_grid,
                   N=10000, T=600):
    """
    Simulates population for computing empirical cumulative
    distribution of unemployment duration and π at time when
    the worker accepts the wage offer. For each job searching
    problem, we simulate for two cases that either f or g is
    the true offer distribution.

    Parameters
    ----------
```

(continues on next page)

(continued from previous page)

```
    F_a, F_b, G_a, G_b : parameters of beta distributions F and G.
    w_bar : the reservation wage
    π_grid : grid points of π, for interpolation
    N : number of workers for simulation, optional
    T : maximum of time periods for simulation, optional

    Returns
    -------
    accpet_t : 2 by N ndarray. the empirical distribution of
               unemployment duration when f or g generates offers.
    accept_π : 2 by N ndarray. the empirical distribution of
               π at the time of employment when f or g generates offers.
    """

    accept_t = np.empty((2, N))
    accept_π = np.empty((2, N))

    # f or g generates offers
    for i, (a, b) in enumerate([(F_a, F_b), (G_a, G_b)]):
        # update each agent
        for n in range(N):

            # initial priori
            π = 0.5

            for t in range(T+1):

                # Draw random wage
                w = np.random.beta(a, b)
                lw = p(w, F_a, F_b) / p(w, G_a, G_b)
                π = π * lw / (π * lw + 1 - π)

                # move to next agent if accepts
                if w >= np.interp(π, π_grid, w_bar):
                    break

            # record the unemployment duration
            # and π at the time of acceptance
            accept_t[i, n] = t
            accept_π[i, n] = π

    return accept_t, accept_π

def cumfreq_x(res):
    """
    A helper function for calculating the x grids of
    the cumulative frequency histogram.
    """

    cumcount = res.cumcount
    lowerlimit, binsize = res.lowerlimit, res.binsize

    x = lowerlimit + np.linspace(0, binsize*cumcount.size, cumcount.size)

    return x
```

Now we define a wrapper function for analyzing job search models with learning under different parameterizations.

The wrapper takes parameters of beta distributions and unemployment compensation as inputs and then displays various things we want to know to interpret the solution of our search model.

In addition, it computes empirical cumulative distributions of two key objects.

```
def job_search_example(F_a=1, F_b=1, G_a=3, G_b=1.2, c=0.3):
    """
    Given the parameters that specify F and G distributions,
    calculate and display the rejection and acceptance area,
    the evolution of belief π, and the probability of accepting
    an offer at different π level, and simulate and calculate
    the empirical cumulative distribution of the duration of
    unemployment and π at the time the worker accepts the offer.
    """

    # construct a search problem
    sp = SearchProblem(F_a=F_a, F_b=F_b, G_a=G_a, G_b=G_b, c=c)
    f, g = sp.f, sp.g
    π_grid = sp.π_grid

    # Solve for reservation wage
    w_bar = solve_wbar(sp, verbose=False)

    # l(w) = f(w) / g(w)
    l = lambda w: f(w) / g(w)
    # objective function for solving l(w) = 1
    obj = lambda w: l(w) - 1.

    # the mode of beta distribution
    # use this to divide w into two intervals for root finding
    G_mode = (G_a - 1) / (G_a + G_b - 2)
    roots = np.empty(2)
    roots[0] = op.root_scalar(obj, bracket=[1e-10, G_mode]).root
    roots[1] = op.root_scalar(obj, bracket=[G_mode, 1-1e-10]).root

    fig, axs = plt.subplots(2, 2, figsize=(12, 9))

    # part 1: display the details of the model settings and some results
    w_grid = np.linspace(1e-12, 1-1e-12, 100)

    axs[0, 0].plot(l(w_grid), w_grid, label='$l$', lw=2)
    axs[0, 0].vlines(1., 0., 1., linestyle="--")
    axs[0, 0].hlines(roots, 0., 2., linestyle="--")
    axs[0, 0].set_xlim([0., 2.])
    axs[0, 0].legend(loc=4)
    axs[0, 0].set(xlabel='$l(w)=f(w)/g(w)$', ylabel='$w$')

    axs[0, 1].plot(sp.π_grid, w_bar, color='k')
    axs[0, 1].fill_between(sp.π_grid, 0, w_bar, color='blue', alpha=0.15)
    axs[0, 1].fill_between(sp.π_grid, w_bar, sp.w_max, color='green', alpha=0.15)
    axs[0, 1].text(0.5, 0.6, 'reject')
    axs[0, 1].text(0.7, 0.9, 'accept')

    W = np.arange(0.01, 0.99, 0.08)
    Π = np.arange(0.01, 0.99, 0.08)

    ΔW = np.zeros((len(W), len(Π)))
```

(continues on next page)

(continued from previous page)

```
    ΔΠ = np.empty((len(W), len(Π)))
    for i, w in enumerate(W):
        for j, π in enumerate(Π):
            lw = l(w)
            ΔΠ[i, j] = π * (lw / (π * lw + 1 - π) - 1)

    q = axs[0, 1].quiver(Π, W, ΔΠ, ΔW, scale=2, color='r', alpha=0.8)

    axs[0, 1].hlines(roots, 0., 1., linestyle="--")
    axs[0, 1].set(xlabel=r'$\pi$', ylabel='$w$')
    axs[0, 1].grid()

    axs[1, 0].plot(f(x_grid), x_grid, label='$f$', lw=2)
    axs[1, 0].plot(g(x_grid), x_grid, label='$g$', lw=2)
    axs[1, 0].vlines(1., 0., 1., linestyle="--")
    axs[1, 0].hlines(roots, 0., 2., linestyle="--")
    axs[1, 0].legend(loc=4)
    axs[1, 0].set(xlabel='$f(w), g(w)$', ylabel='$w$')

    axs[1, 1].plot(sp.π_grid, 1 - beta.cdf(w_bar, F_a, F_b), label='$f$')
    axs[1, 1].plot(sp.π_grid, 1 - beta.cdf(w_bar, G_a, G_b), label='$g$')
    axs[1, 1].set_ylim([0., 1.])
    axs[1, 1].grid()
    axs[1, 1].legend(loc=4)
    axs[1, 1].set(xlabel=r'$\pi$', ylabel=r'$\mathbb{P}\{w > \overline{w} (\pi)\}$')

    plt.show()

    # part 2: simulate empirical cumulative distribution
    accept_t, accept_π = empirical_dist(F_a, F_b, G_a, G_b, w_bar, π_grid)
    N = accept_t.shape[1]

    cfq_t_F = cumfreq(accept_t[0, :], numbins=100)
    cfq_π_F = cumfreq(accept_π[0, :], numbins=100)

    cfq_t_G = cumfreq(accept_t[1, :], numbins=100)
    cfq_π_G = cumfreq(accept_π[1, :], numbins=100)

    fig, axs = plt.subplots(2, 1, figsize=(12, 9))

    axs[0].plot(cumfreq_x(cfq_t_F), cfq_t_F.cumcount/N, label="f generates")
    axs[0].plot(cumfreq_x(cfq_t_G), cfq_t_G.cumcount/N, label="g generates")
    axs[0].grid(linestyle='--')
    axs[0].legend(loc=4)
    axs[0].title.set_text('CDF of duration of unemployment')
    axs[0].set(xlabel='time', ylabel='Prob(time)')

    axs[1].plot(cumfreq_x(cfq_π_F), cfq_π_F.cumcount/N, label="f generates")
    axs[1].plot(cumfreq_x(cfq_π_G), cfq_π_G.cumcount/N, label="g generates")
    axs[1].grid(linestyle='--')
    axs[1].legend(loc=4)
    axs[1].title.set_text('CDF of π at time worker accepts wage and leaves␣
↪unemployment')
    axs[1].set(xlabel='π', ylabel='Prob(π)')

    plt.show()
```

We now provide some examples that provide insights about how the model works.

51.12 Examples

51.12.1 Example 1 (Baseline)

F ~ Beta(1, 1), G ~ Beta(3, 1.2), c=0.3.

In the graphs below, the red arrows in the upper right figure show how π_t is updated in response to the new information w_t.

Recall the following formula from this lecture

$$\frac{\pi_{t+1}}{\pi_t} = \frac{l\left(w_{t+1}\right)}{\pi_t l\left(w_{t+1}\right) + \left(1 - \pi_t\right)} \begin{cases} > 1 & \text{if } l\left(w_{t+1}\right) > 1 \\ \leq 1 & \text{if } l\left(w_{t+1}\right) \leq 1 \end{cases}$$

The formula implies that the direction of motion of π_t is determined by the relationship between $l(w_t)$ and 1.

The magnitude is small if

- $l(w)$ is close to 1, which means the new w is not very informative for distinguishing two distributions,
- π_{t-1} is close to either 0 or 1, which means the priori is strong.

Will an unemployed worker accept an offer earlier or not, when the actual ruling distribution is g instead of f?

Two countervailing effects are at work.

- if f generates successive wage offers, then w is more likely to be low, but π is moving up toward to 1, which lowers the reservation wage, i.e., the worker becomes less selective the longer he or she remains unemployed.
- if g generates wage offers, then w is more likely to be high, but π is moving downward toward 0, increasing the reservation wage, i.e., the worker becomes more selective the longer he or she remains unemployed.

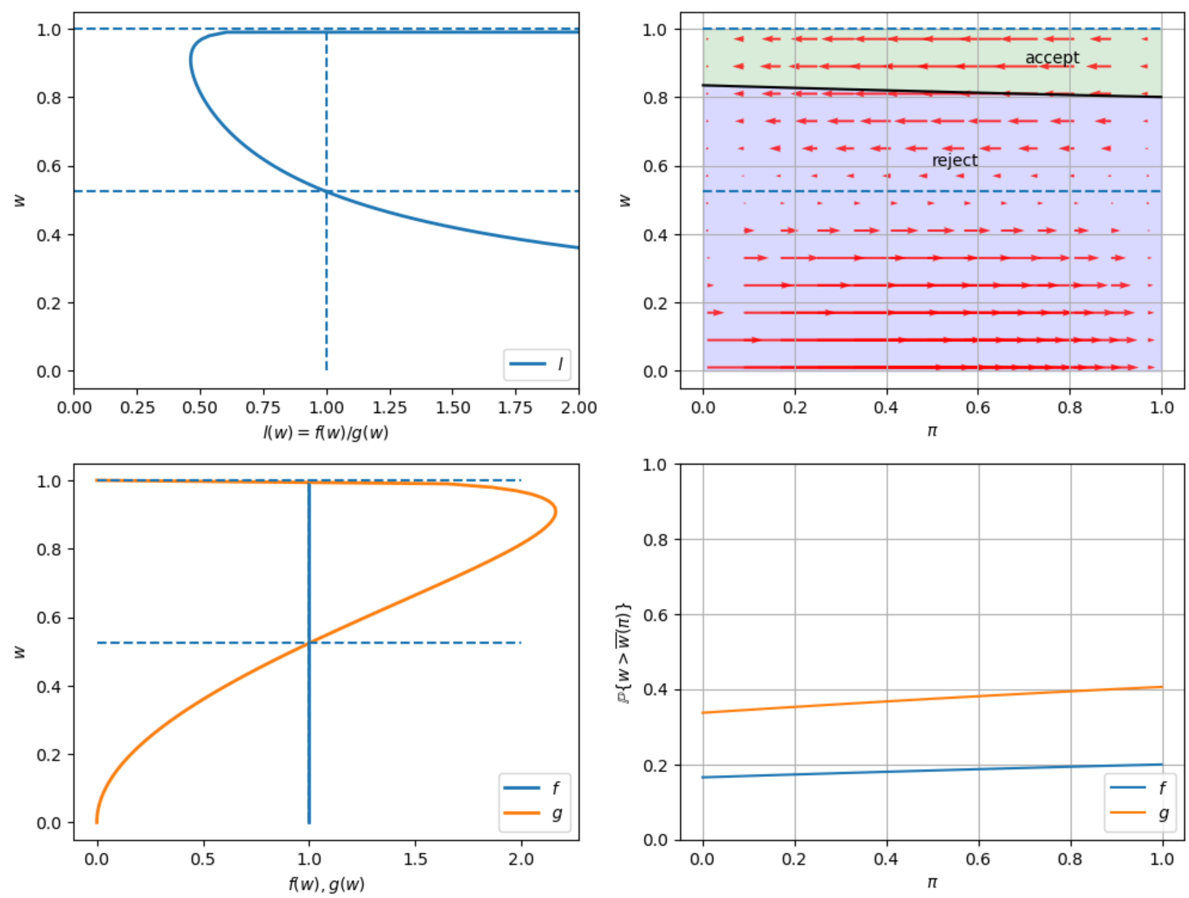
Quantitatively, the lower right figure sheds light on which effect dominates in this example.

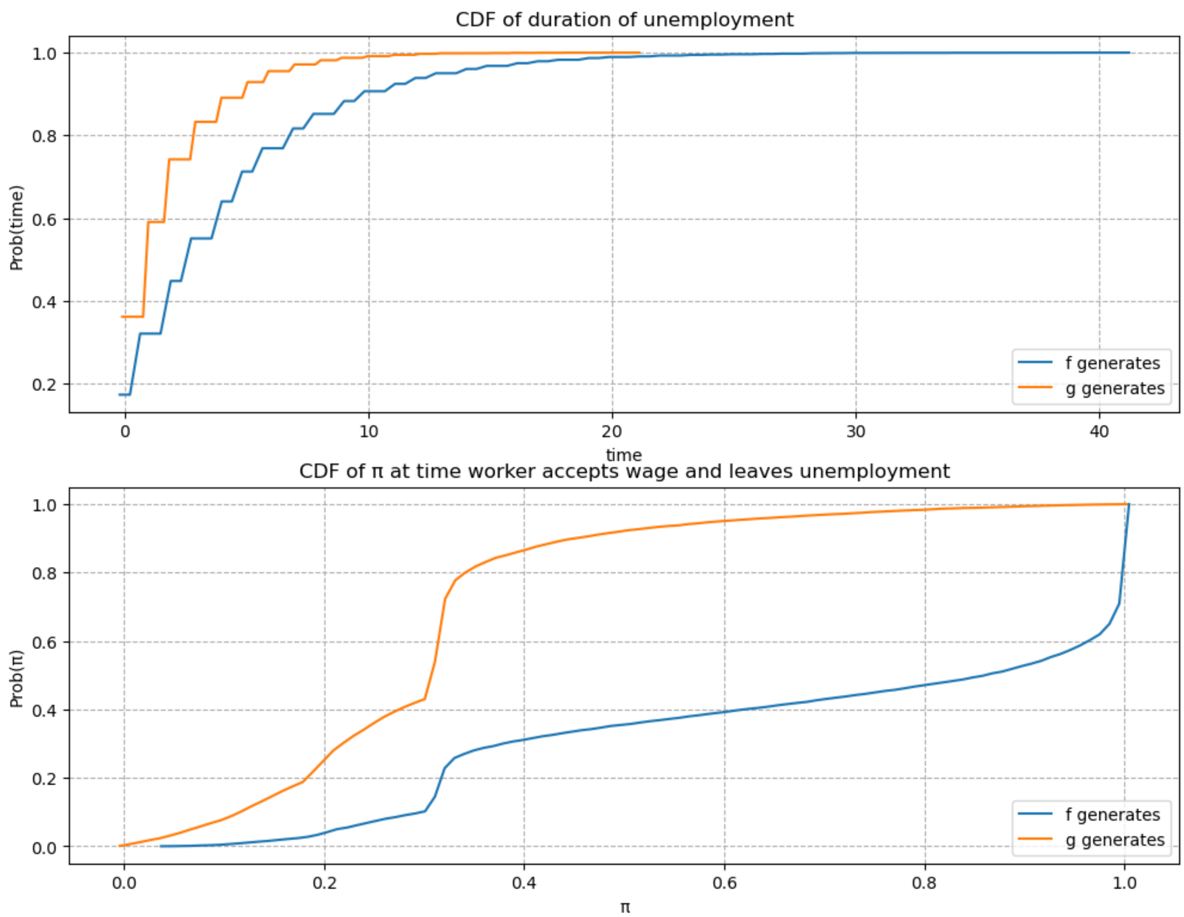
It shows the probability that a previously unemployed worker accepts an offer at different values of π when f or g generates wage offers.

That graph shows that for the particular f and g in this example, the worker is always more likely to accept an offer when f generates the data even when π is close to zero so that the worker believes the true distribution is g and therefore is relatively more selective.

The empirical cumulative distribution of the duration of unemployment verifies our conjecture.

```
job_search_example()
```


w
$l(w) = f(w)/g(w)$
l
accept
reject
π
$f(w), g(w)$
f
g
$\mathbb{P}\{w > \overline{w}(\pi)\}$



51.12.2 Example 2

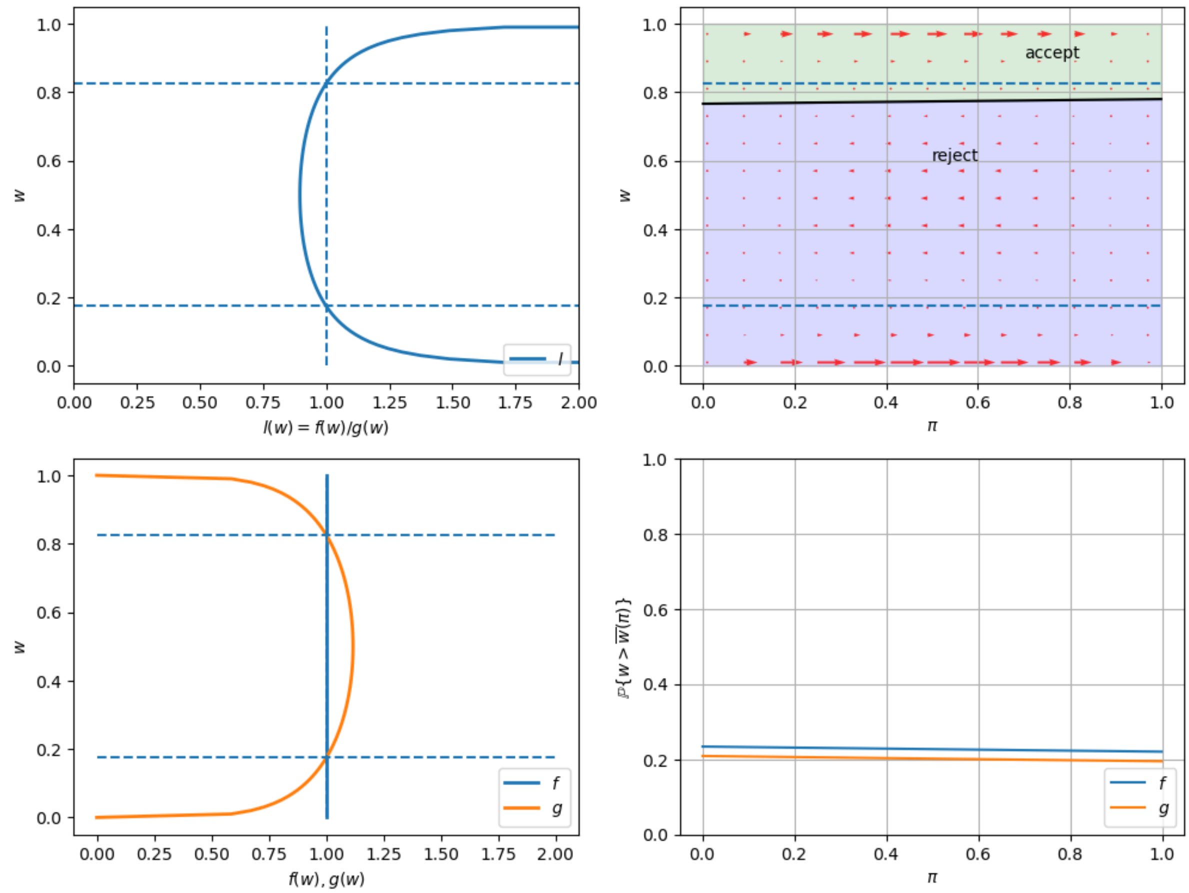
F ~ Beta(1, 1), G ~ Beta(1.2, 1.2), c=0.3.

Now G has the same mean as F with a smaller variance.

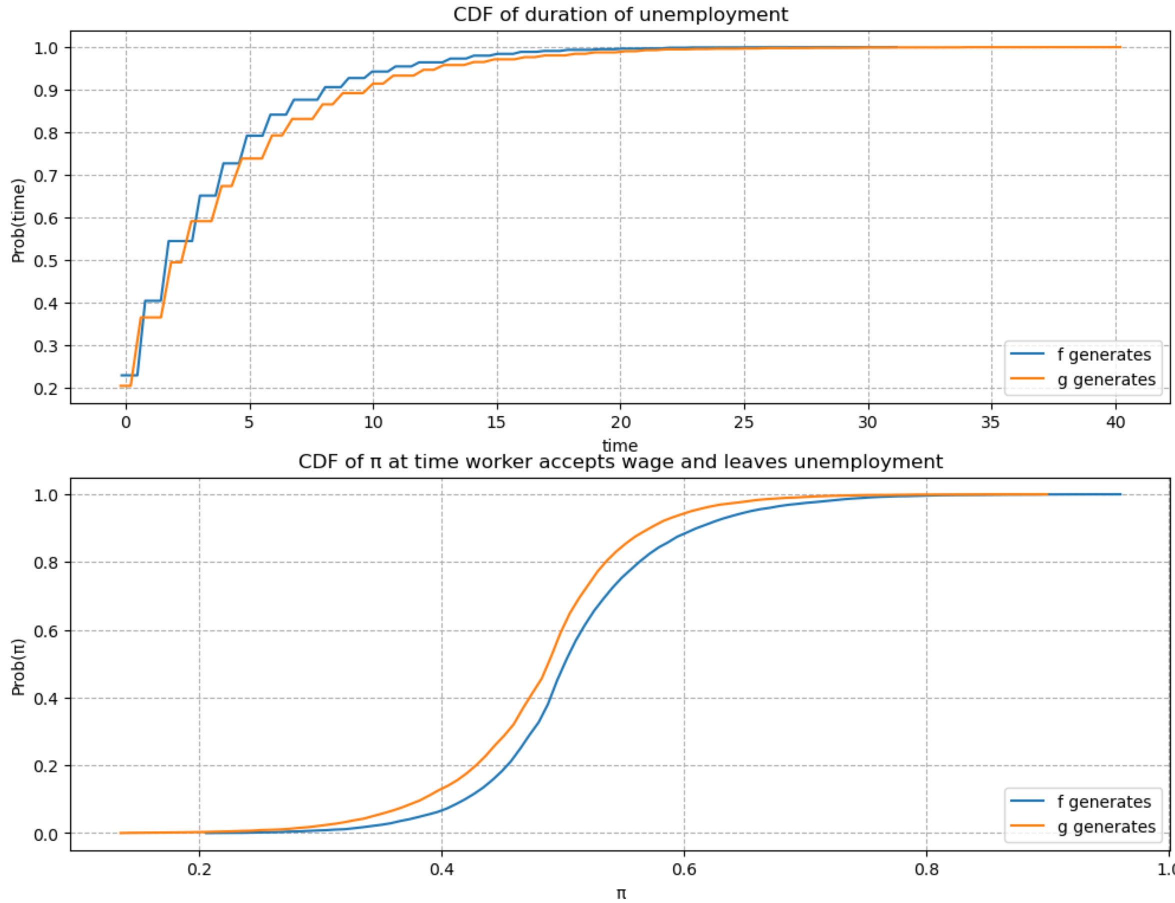
Since the unemployment compensation c serves as a lower bound for bad wage offers, G is now an "inferior" distribution to F.

Consequently, we observe that the optimal policy $\overline{w}(\pi)$ is increasing in π.

```
job_search_example(1, 1, 1.2, 1.2, 0.3)
```



w
l(w) = f(w)/g(w)
l
accept
reject
w
π
w
f(w), g(w)
f
g
ℙ{w > w̄(π)}
π
f
g

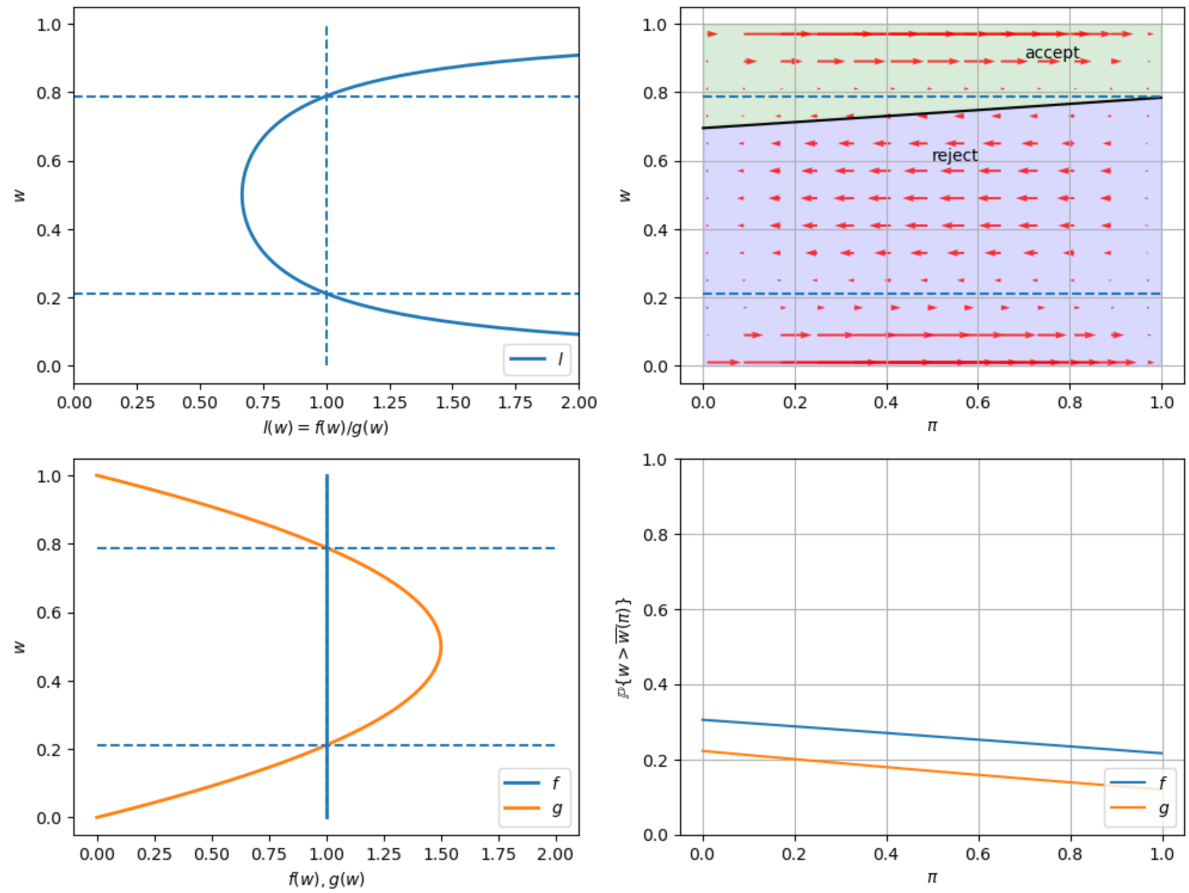


51.12.3 Example 3

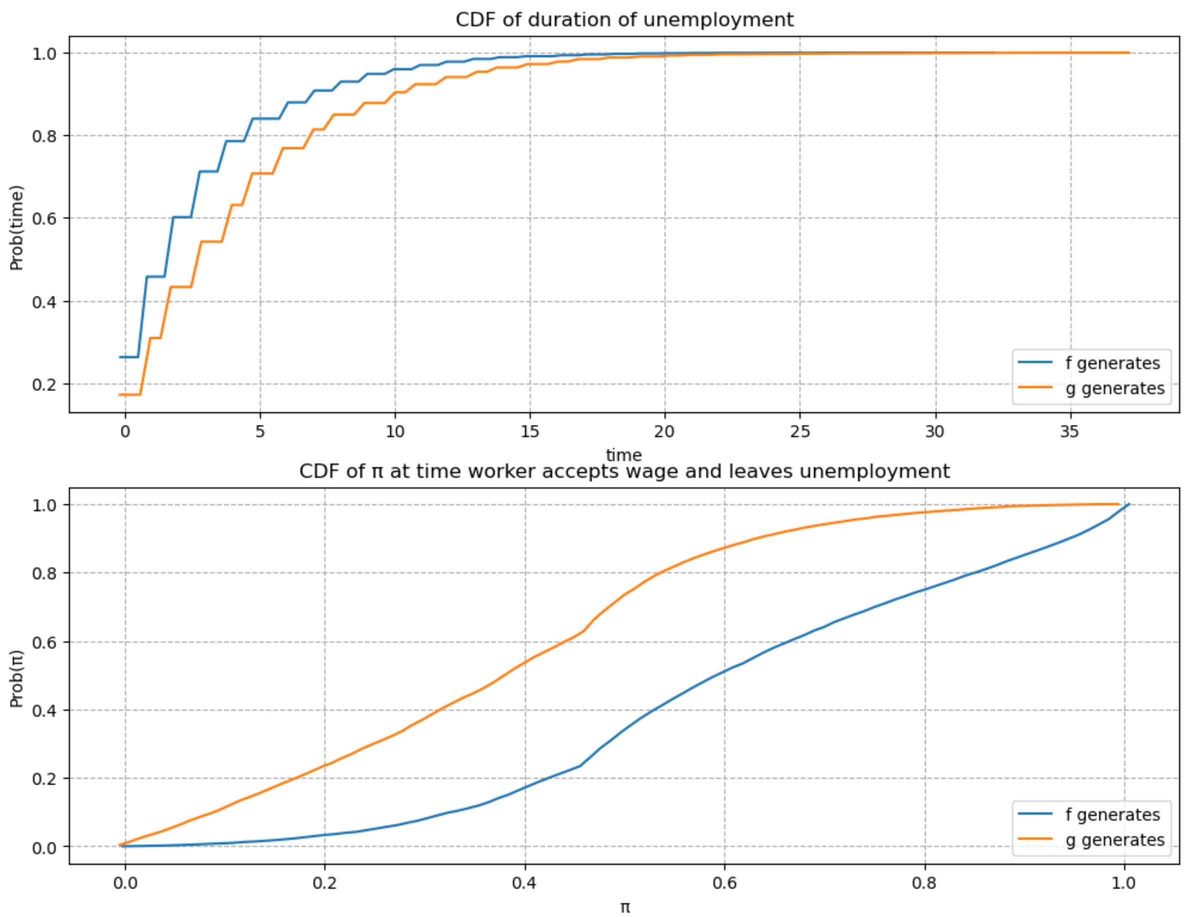
F ~ Beta(1, 1), G ~ Beta(2, 2), c=0.3.

If the variance of G is smaller, we observe in the result that G is even more "inferior" and the slope of $\overline{w}(\pi)$ is larger.

```
job_search_example(1, 1, 2, 2, 0.3)
```



$l(w) = f(w)/g(w)$
w
l
accept
reject
π
$f(w), g(w)$
f
g
$\mathbb{P}\{w > \overline{w}(\pi)\}$



51.12.4 Example 4

F ~ Beta(1, 1), G ~ Beta(3, 1.2), and c=0.8.

In this example, we keep the parameters of beta distributions to be the same with the baseline case but increase the unemployment compensation c.

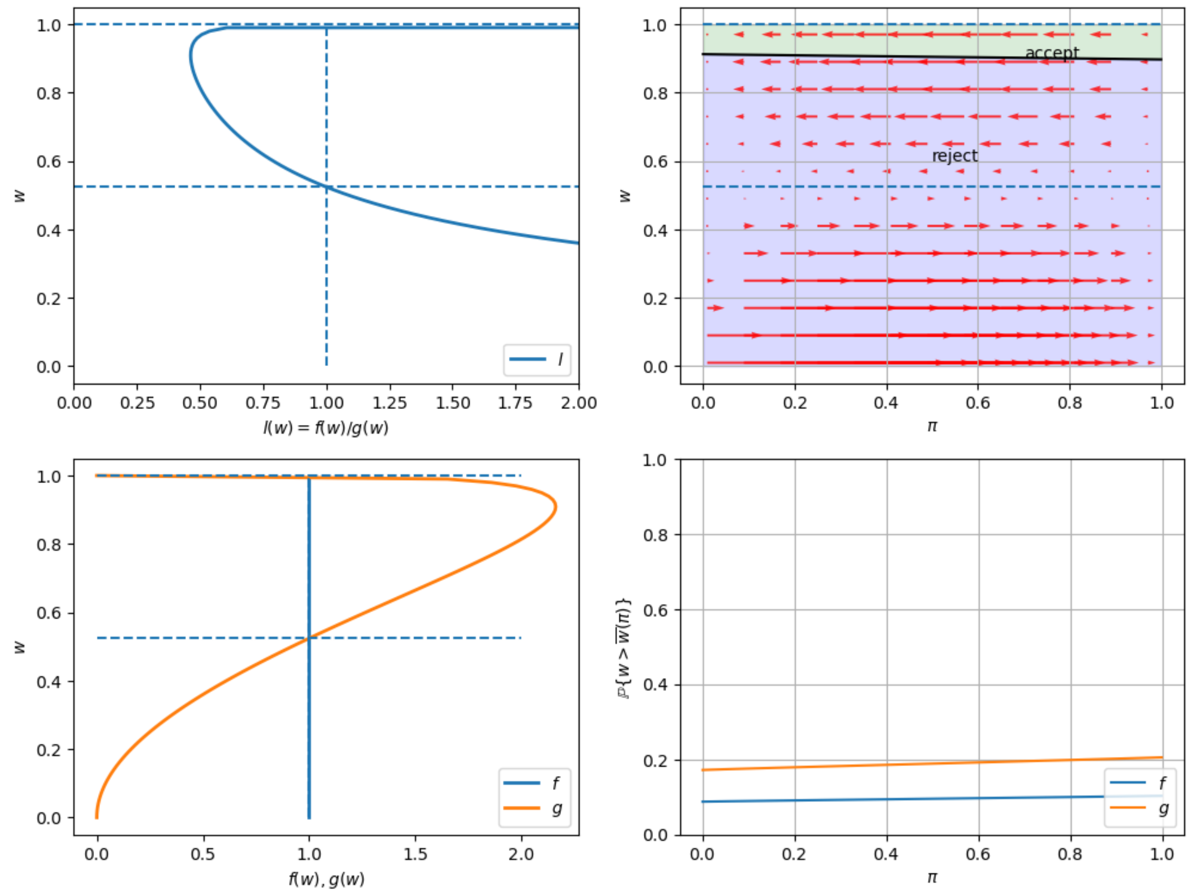
Comparing outcomes to the baseline case (example 1) in which unemployment compensation if low (c=0.3), now the worker can afford a longer learning period.

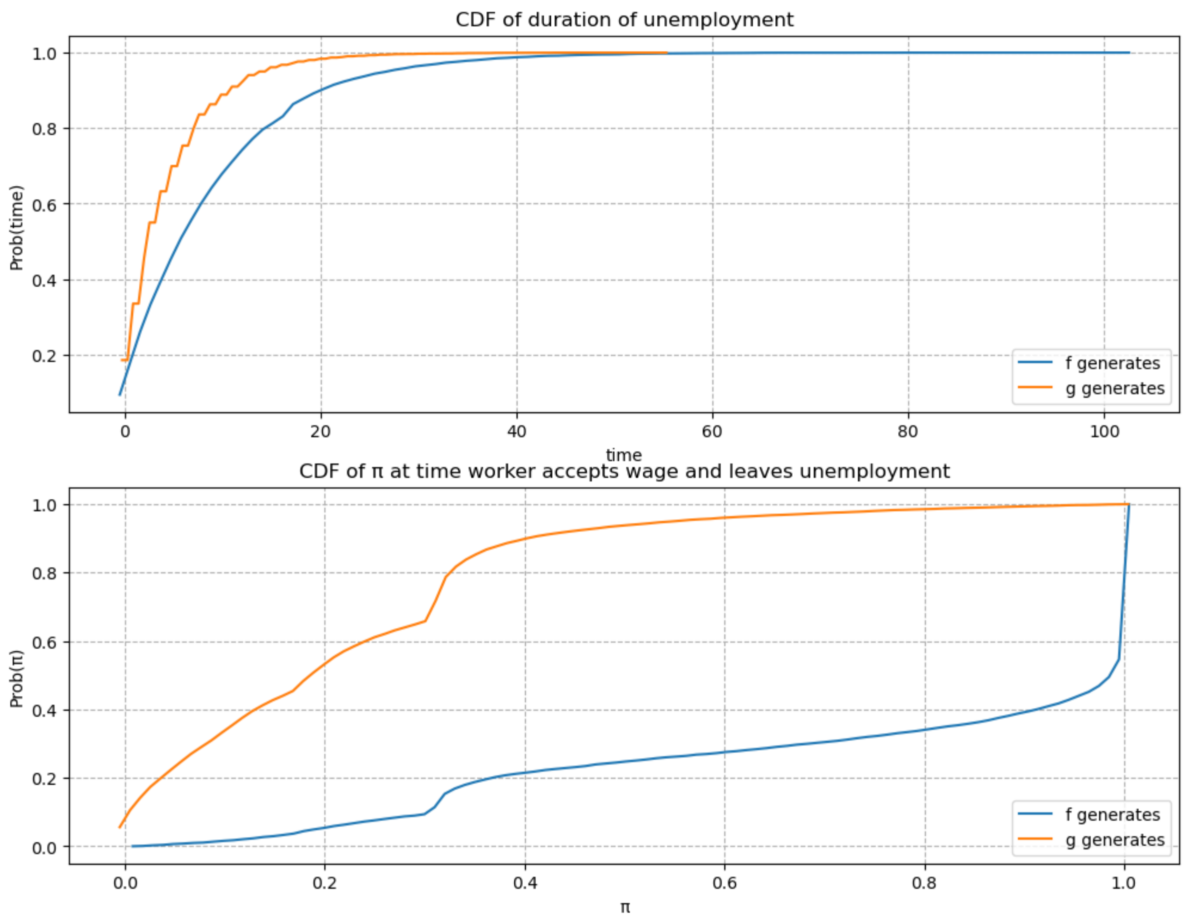
As a result, the worker tends to accept wage offers much later.

Furthermore, at the time of accepting employment, the belief π is closer to either 0 or 1.

That means that the worker has a better idea about what the true distribution is when he eventually chooses to accept a wage offer.

```
job_search_example(1, 1, 3, 1.2, c=0.8)
```



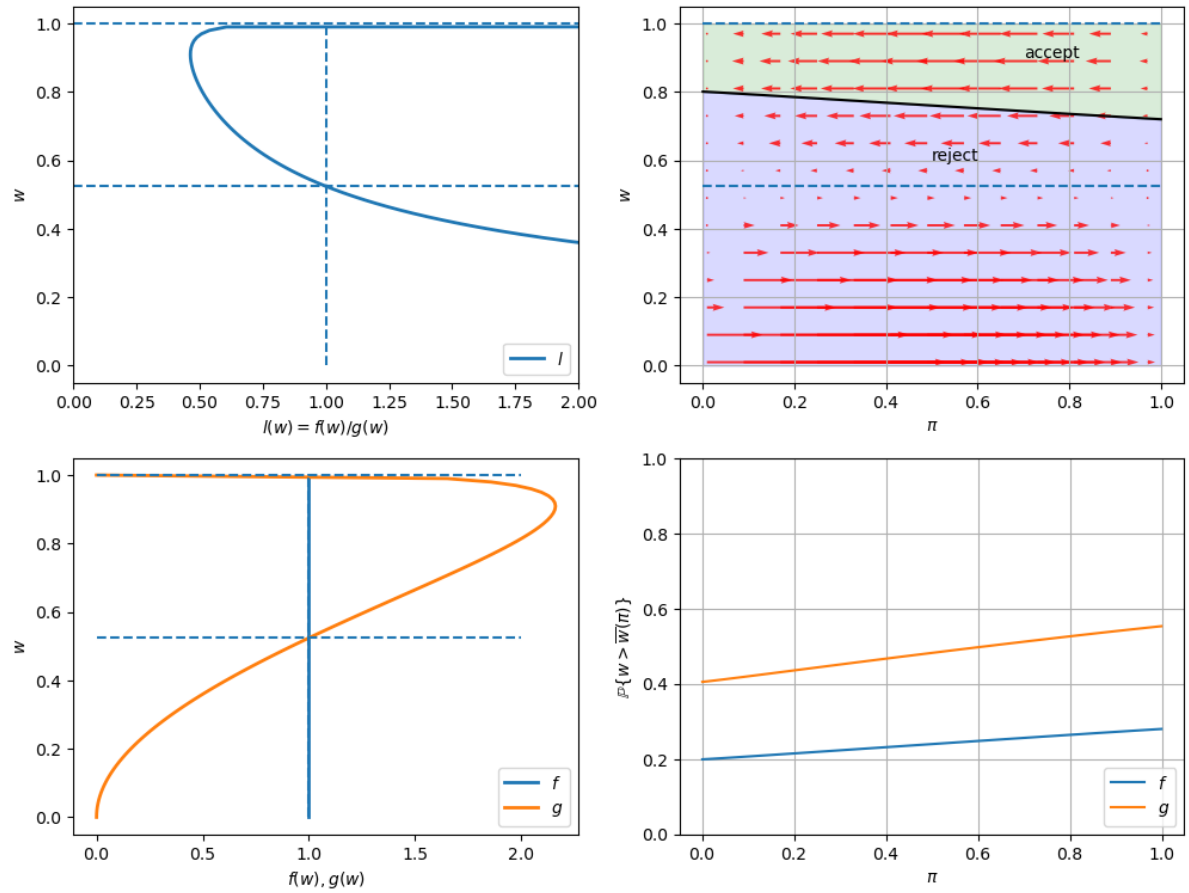


51.12.5 Example 5

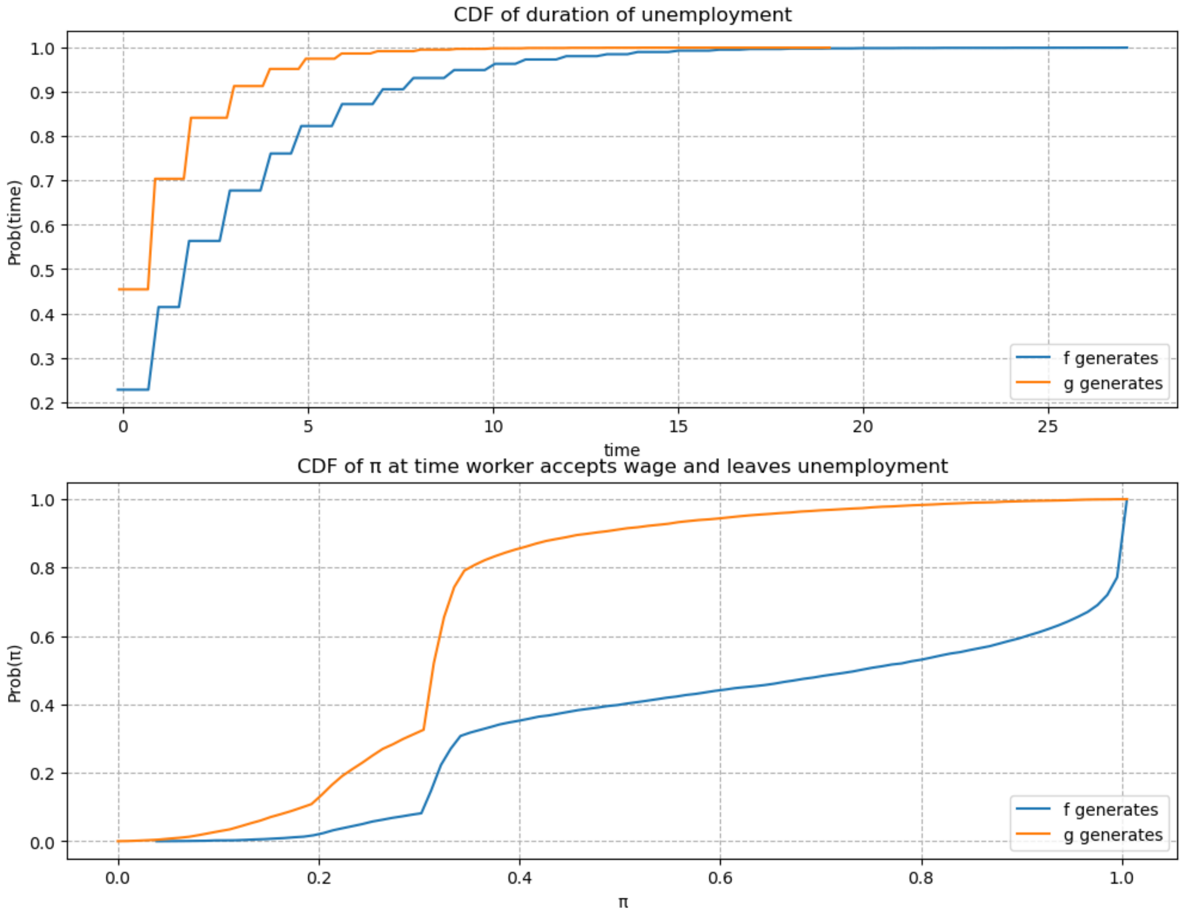
F ~ Beta(1, 1), G ~ Beta(3, 1.2), and c=0.1.

As expected, a smaller c makes an unemployed worker accept wage offers earlier after having acquired less information about the wage distribution.

```
job_search_example(1, 1, 3, 1.2, c=0.1)
```



l
$l(w) = f(w)/g(w)$
w
accept
reject
π
f
g
$f(w), g(w)$
$\mathbb{P}\{w > \overline{w}(\pi)\}$


CDF of duration of unemployment
Prob(time)
time
f generates
g generates
CDF of π at time worker accepts wage and leaves unemployment
Prob(π)
π
f generates
g generates

CHAPTER FIFTYTWO

JOB SEARCH IX: SEARCH WITH Q-LEARNING

52.1 Overview

This lecture illustrates a powerful machine learning technique called Q-learning.

[Sutton and Barto, 2018] presents Q-learning and a variety of other statistical learning procedures.

The Q-learning algorithm combines ideas from

- dynamic programming
- a recursive version of least squares known as temporal difference learning.

This lecture applies a Q-learning algorithm to the situation faced by a McCall worker.

This lecture also considers the case where a McCall worker is given an option to quit the current job.

Relative to the dynamic programming formulation of the McCall worker model that we studied in *quantecon lecture*, a Q-learning algorithm gives the worker less knowledge about

- the random process that generates a sequence of wages
- the reward function that tells consequences of accepting or rejecting a job

The Q-learning algorithm invokes a statistical learning model to learn about these things.

Statistical learning often comes down to some version of least squares, and it will be here too.

Any time we say **statistical learning**, we have to say what object is being learned.

For Q-learning, the object that is learned is not the **value function** that is a focus of dynamic programming.

But it is something that is closely affiliated with it.

In the finite-action, finite state context studied in this lecture, the object to be learned statistically is a **Q-table**, an instance of a **Q-function** for finite sets.

Sometimes a Q-function or Q-table is called a quality-function or quality-table.

The rows and columns of a Q-table correspond to possible states that an agent might encounter, and possible actions that he can take in each state.

An equation that resembles a Bellman equation plays an important role in the algorithm.

It differs from the Bellman equation for the McCall model that we have seen in *this quantecon lecture*

In this lecture, we'll learn a little about

- the **Q-function** or **quality function** that is affiliated with any Markov decision problem whose optimal value function satisfies a Bellman equation
- **temporal difference learning**, a key component of a Q-learning algorithm

As usual, let's import some Python modules.

```
!pip install quantecon
```

```
import numpy as np

from numba import jit, float64, int64
from numba.experimental import jitclass
from quantecon.distributions import BetaBinomial

import matplotlib.pyplot as plt

np.random.seed(123)
```

52.2 Review of McCall Model

We begin by reviewing the McCall model described in *this quantecon lecture*.

We'll compute an optimal value function and a policy that attains it.

We'll eventually compare that optimal policy to what the Q-learning McCall worker learns.

The McCall model is characterized by parameters β, c and a known distribution of wage offers F.

A McCall worker wants to maximize an expected discounted sum of lifetime incomes

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t y_t$$

The worker's income y_t equals his wage w if he is employed, and unemployment compensation c if he is unemployed.

An optimal value $V(w)$ for a McCall worker who has just received a wage offer w and is deciding whether to accept or reject it satisfies the Bellman equation

$$V(w) = \max_{\text{accept, reject}} \left\{ \frac{w}{1-\beta}, c + \beta \int V(w') \, dF(w') \right\} \tag{52.1}$$

To form a benchmark to compare with results from Q-learning, we first approximate the optimal value function.

With possible states residing in a finite discrete state space indexed by $\{1, 2, ..., n\}$, we make an initial guess for the value function of $v \in \mathbb{R}^n$ and then iterate on the Bellman equation:

$$v'(i) = \max \left\{ \frac{w(i)}{1-\beta}, c + \beta \sum_{1 \leq j \leq n} v(j) q(j) \right\} \quad \text{for } i = 1, \ldots, n$$

Let's use Python code from *this quantecon lecture*.

We use a Python method called `VFI` to compute the optimal value function using value function iterations.

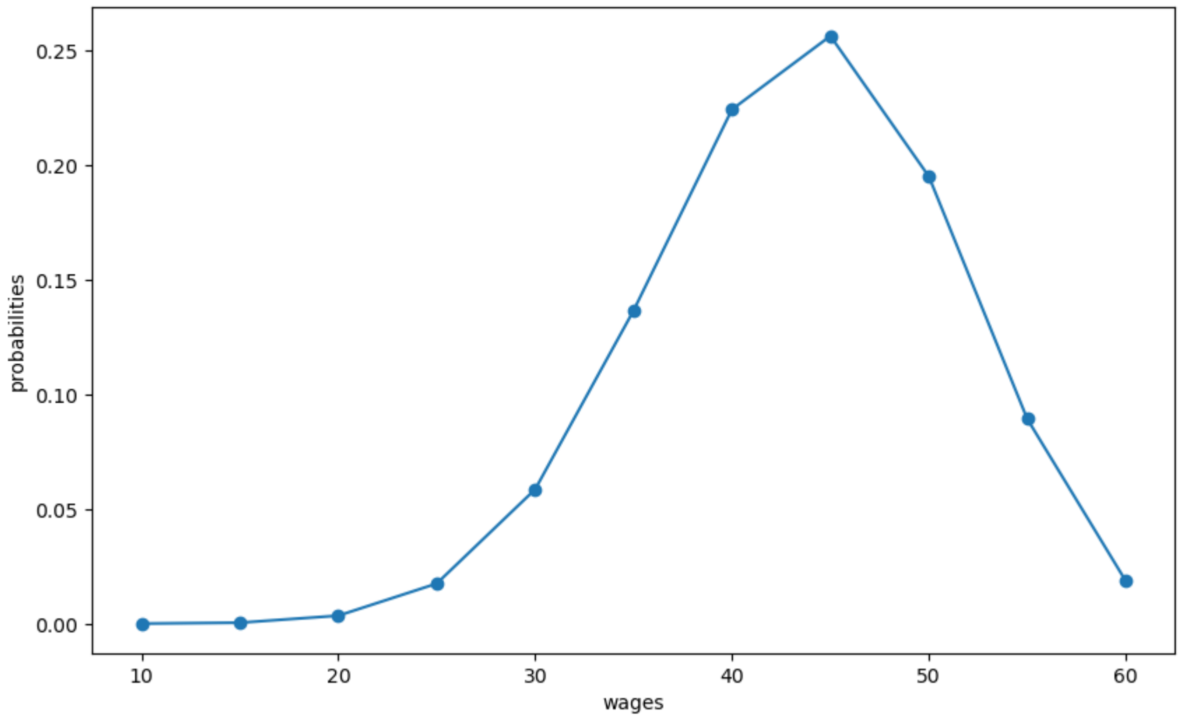
We construct an assumed distribution of wages and plot it with the following Python code

```
n, a, b = 10, 200, 100                        # default parameters
q_default = BetaBinomial(n, a, b).pdf()       # default choice of q

w_min, w_max = 10, 60
w_default = np.linspace(w_min, w_max, n+1)
```

(continues on next page)

(continued from previous page)

```
# plot distribution of wage offer
fig, ax = plt.subplots(figsize=(10,6))
ax.plot(w_default, q_default, '-o', label='$q(w(i))$')
ax.set_xlabel('wages')
ax.set_ylabel('probabilities')

plt.show()
```



Next we'll compute the worker's optimal value function by iterating to convergence on the Bellman equation.

Then we'll plot various iterates on the Bellman operator.

```
mccall_data = [
    ('c', float64),      # unemployment compensation
    ('β', float64),      # discount factor
    ('w', float64[::1]),  # array of wage values, w[i] = wage at state i
    ('q', float64[::1])   # array of probabilities
]


@jitclass(mccall_data)
class McCallModel:

    def __init__(self, c=25, β=0.99, w=w_default, q=q_default):

        self.c, self.β = c, β
        self.w, self.q = w, q

    def state_action_values(self, i, v):
        """
        The values of state-action pairs.
```

(continues on next page)

(continued from previous page)

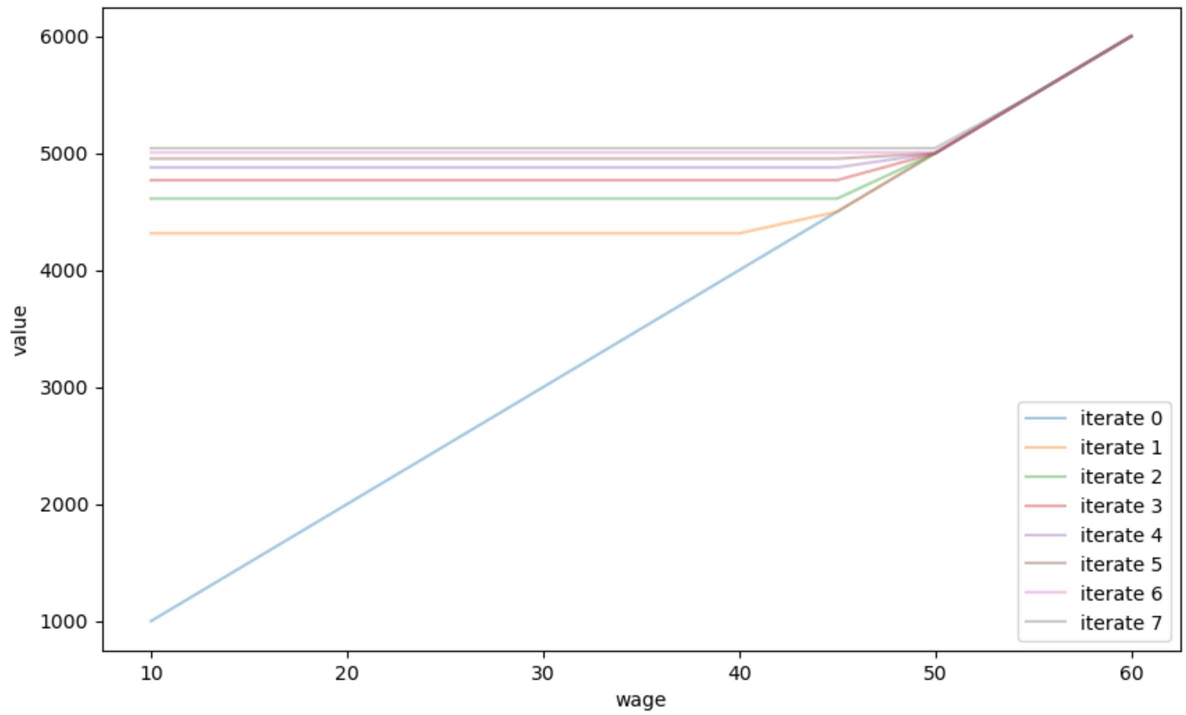
```
        """
        # Simplify names
        c, β, w, q = self.c, self.β, self.w, self.q
        # Evaluate value for each state-action pair
        # Consider action = accept or reject the current offer
        accept = w[i] / (1 - β)
        reject = c + β * (v @ q)

        return np.array([accept, reject])

    def VFI(self, eps=1e-5, max_iter=500):
        """
        Find the optimal value function.
        """

        n = len(self.w)
        v = self.w / (1 - self.β)
        v_next = np.empty_like(v)
        flag=0

        for i in range(max_iter):
            for j in range(n):
                v_next[j] = np.max(self.state_action_values(j, v))

            if np.max(np.abs(v_next - v))<=eps:
                flag=1
                break
            v[:] = v_next

        return v, flag

def plot_value_function_seq(mcm, ax, num_plots=8):
    """
    Plot a sequence of value functions.

        * mcm is an instance of McCallModel
        * ax is an axes object that implements a plot method.

    """

    n = len(mcm.w)
    v = mcm.w / (1 - mcm.β)
    v_next = np.empty_like(v)
    for i in range(num_plots):
        ax.plot(mcm.w, v, '-', alpha=0.4, label=f"iterate {i}")
        # Update guess
        for i in range(n):
            v_next[i] = np.max(mcm.state_action_values(i, v))
        v[:] = v_next  # copy contents into v

    ax.legend(loc='lower right')
```

```
mcm = McCallModel()
valfunc_VFI, flag = mcm.VFI()

fig, ax = plt.subplots(figsize=(10,6))
```

(continues on next page)

(continued from previous page)

```
ax.set_xlabel('wage')
ax.set_ylabel('value')
plot_value_function_seq(mcm, ax)
plt.show()
```



Next we'll print out the limit of the sequence of iterates.

This is the approximation to the McCall worker's value function that is produced by value function iteration.

We'll use this value function as a benchmark later after we have done some Q-learning.

```
print(valfunc_VFI)
```

```
[5322.27935875 5322.27935875 5322.27935875 5322.27935875 5322.27935875
 5322.27935875 5322.27935875 5322.27935875 5322.27935875 5500.
 6000.        ]
```

52.3 Implied Quality Function Q

A **quality function** Q map state-action pairs into optimal values.

They are tightly linked to optimal value functions.

But value functions are functions just of states, and not actions.

For each given state, the quality function gives a list of optimal values that can be attained starting from that state, with each component of the list indicating one of the possible actions that is taken.

For our McCall worker with a finite set of possible wages

- the state space $\mathcal{W} = \{w_1, w_2, ..., w_n\}$ is indexed by integers $1, 2, ..., n$

- the action space is $\mathcal{A} = \{\text{accept}, \text{reject}\}$

Let $a \in \mathcal{A}$ be one of the two possible actions, i.e., accept or reject.

For our McCall worker, an optimal Q-function $Q(w, a)$ equals the maximum value of that a previously unemployed worker who has offer w in hand can attain if he takes action a.

This definition of $Q(w, a)$ presumes that in subsequent periods the worker takes optimal actions.

An optimal Q-function for our McCall worker satisfies

$$
\begin{aligned}
Q(w, \text{accept}) &= \frac{w}{1-\beta} \\
Q(w, \text{reject}) &= c + \beta \int \max_{\text{accept, reject}} \left\{ \frac{w'}{1-\beta}, Q(w', \text{reject}) \right\} dF(w')
\end{aligned} \tag{52.2}
$$

Note that the first equation of system (52.2) presumes that after the agent has accepted an offer, he will not have the objection to reject that same offer in the future.

These equations are aligned with the Bellman equation for the worker's optimal value function that we studied in *this quantecon lecture*.

Evidently, the optimal value function $V(w)$ described in that lecture is related to our Q-function by

$$
V(w) = \max_{\text{accept,reject}} \left\{ Q(w, \text{accept}), Q(w, \text{reject}) \right\}
$$

If we stare at the second equation of system (52.2), we notice that since the wage process is identically and independently distributed over time, $Q(w, \text{reject})$, the right side of the equation is independent of the current state w.

So we can denote it as a scalar

$$
Q_r := Q(w, \text{reject}) \quad \forall \, w \in \mathcal{W}.
$$

This fact provides us with an an alternative, and as it turns out in this case, a faster way to compute an optimal value function and associated optimal policy for the McCall worker.

Instead of using the value function iterations that we deployed above, we can instead iterate to convergence on a version of the second equation in system (52.2) that maps an estimate of Q_r into an improved estimate Q_r':

$$
Q_r' = c + \beta \int \max \left\{ \frac{w'}{1-\beta}, Q_r \right\} dF(w')
$$

After a Q_r sequence has converged, we can recover the optimal value function $V(w)$ for the McCall worker from

$$
V(w) = \max \left\{ \frac{w}{1-\beta}, Q_r \right\}
$$

52.4 From Probabilities to Samples

We noted above that the optimal Q function for our McCall worker satisfies the Bellman equations

$$
\begin{aligned}
w + \beta \max_{\text{accept, reject}} \left\{ Q(w, \text{accept}), Q(w, \text{reject}) \right\} - Q(w, \text{accept}) = 0 \\
c + \beta \int \max_{\text{accept, reject}} \left\{ Q(w', \text{accept}), Q(w', \text{reject}) \right\} dF(w') - Q(w, \text{reject}) = 0
\end{aligned} \tag{52.3}
$$

Notice the integral over $F(w')$ on the second line.

Erasing the integral sign sets the stage for an illegitimate argument that can get us started thinking about Q-learning.

Thus, construct a difference equation system that keeps the first equation of (52.3) but replaces the second by removing integration over $F(w')$:

$$\begin{aligned} w+\beta \max_{\text{accept, reject}} \{Q(w,\text{accept}), Q(w,\text{reject})\} - Q(w,\text{accept}) &= 0 \\ c+\beta \max_{\text{accept, reject}} \{Q(w',\text{accept}), Q(w',\text{reject})\} - Q(w,\text{reject}) &\approx 0 \end{aligned} \tag{52.4}$$

The second equation can't hold for all w, w' pairs in the appropriate Cartesian product of our state space.

But maybe an appeal to a Law of Large numbers could let us hope that it would hold **on average** for a long time series sequence of draws of w_t, w_{t+1} pairs, where we are thinking of w_t as w and w_{t+1} as w'.

The basic idea of Q-learning is to draw a long sample of wage offers from F (we know F though we assume that the worker doesn't) and iterate on a recursion that maps an estimate $\hat{Q}_t$ of a Q-function at date t into an improved estimate $\hat{Q}_{t+1}$ at date $t+1$.

To set up such an algorithm, we first define some errors or "differences"

$$\begin{aligned} w+\beta \max_{\text{accept, reject}} \left\{\hat{Q}_t(w_t,\text{accept}), \hat{Q}_t(w_t,\text{reject})\right\} - \hat{Q}_t(w_t,\text{accept}) &= \text{diff}_{\text{accept},t} \\ c+\beta \max_{\text{accept, reject}} \left\{\hat{Q}_t(w_{t+1},\text{accept}), \hat{Q}_t(w_{t+1},\text{reject})\right\} - \hat{Q}_t(w_t,\text{reject}) &= \text{diff}_{\text{reject},t} \end{aligned} \tag{52.5}$$

The adaptive learning scheme would then be some version of

$$\hat{Q}_{t+1} = \hat{Q}_t + \alpha\ \text{diff}_t \tag{52.6}$$

where $\alpha \in (0,1)$ is a small **gain** parameter that governs the rate of learning and $\hat{Q}_t$ and diff_t are 2×1 vectors corresponding to objects in equation system (52.5).

This informal argument takes us to the threshold of Q-learning.

52.5 Q-Learning

Let's first describe a Q-learning algorithm precisely.

Then we'll implement it.

The algorithm works by using a Monte Carlo method to update estimates of a Q-function.

We begin with an initial guess for a Q-function.

In the example studied in this lecture, we have a finite action space and also a finite state space.

That means that we can represent a Q-function as a matrix or Q-table, $\widetilde{Q}(w,a)$.

Q-learning proceeds by updating the Q-function as the decision maker acquires experience along a path of wage draws generated by simulation.

During the learning process, our McCall worker takes actions and experiences rewards that are consequences of those actions.

He learns simultaneously about the environment, in this case the distribution of wages, and the reward function, in this case the unemployment compensation c and the present value of wages.

The updating algorithm is based on a slight modification (to be described soon) of a recursion like

$$\widetilde{Q}^{new}(w,a) = \widetilde{Q}^{old}(w,a) + \alpha \widetilde{TD}(w,a) \tag{52.7}$$

where

$$
\begin{aligned}
\widetilde{TD}(w, \text{accept}) &= \left[w + \beta \max_{a' \in \mathcal{A}} \widetilde{Q}^{old}(w, a') \right] - \widetilde{Q}^{old}(w, \text{accept}) \\
\widetilde{TD}(w, \text{reject}) &= \left[c + \beta \max_{a' \in \mathcal{A}} \widetilde{Q}^{old}(w', a') \right] - \widetilde{Q}^{old}(w, \text{reject}), \; w' \sim F
\end{aligned}
\tag{52.8}
$$

The terms $\widetilde{TD}(w, a)$ for $a = \{\text{accept,reject}\}$ are the **temporal difference errors** that drive the updates.

This system is thus a version of the adaptive system that we sketched informally in equation (52.6).

An aspect of the algorithm not yet captured by equation system (52.8) is random **experimentation** that we add by occasionally randomly replacing

$$
\text{argmax}_{a' \in \mathcal{A}} \widetilde{Q}^{old}(w, a')
$$

with

$$
\text{argmin}_{a' \in \mathcal{A}} \widetilde{Q}^{old}(w, a')
$$

and occasionally replacing

$$
\text{argmax}_{a' \in \mathcal{A}} \widetilde{Q}^{old}(w', a')
$$

with

$$
\text{argmin}_{a' \in \mathcal{A}} \widetilde{Q}^{old}(w', a')
$$

We activate such experimentation with probability ϵ in step 3 of the following pseudo-code for our McCall worker to do Q-learning:

1. Set an arbitrary initial Q-table.
2. Draw an initial wage offer w from F.
3. From the appropriate row in the Q-table, choose an action using the following ϵ-greedy algorithm:
   - with probability $1 - \epsilon$, choose the action that maximizes the value, and
   - with probability ϵ, choose the alternative action.
4. Update the state associated with the chosen action and compute $\widetilde{TD}$ according to (52.8) and update $\widetilde{Q}$ according to (52.7).
5. Either draw a new state w' if required or else take existing wage if and update the Q-table again according to (52.7).
6. Stop when the old and new Q-tables are close enough, i.e., $\|\tilde{Q}^{new} - \tilde{Q}^{old}\|_\infty \leq \delta$ for given δ or if the worker keeps accepting for T periods for a prescribed T.
7. Return to step 2 with the updated Q-table.

Repeat this procedure for N episodes or until the updated Q-table has converged.

We call one pass through steps 2 to 7 an "episode" or "epoch" of temporal difference learning.

In our context, each episode starts with an agent drawing an initial wage offer, i.e., a new state.

The agent then takes actions based on the preset Q-table, receives rewards, and then enters a new state implied by this period's actions.

The Q-table is updated via temporal difference learning.

We iterate this until convergence of the Q-table or the maximum length of an episode is reached.

Multiple episodes allow the agent to start afresh and visit states that she was less likely to visit from the terminal state of a previos episode.

For example, an agent who has accepted a wage offer based on her Q-table will be less likely to draw a new offer from other parts of the wage distribution.

By using the ϵ-greedy method and also by increasing the number of episodes, the Q-learning algorithm balances gains from exploration and from exploitation.

Remark: Notice that $\widetilde{TD}$ associated with an optimal Q-table defined in (52.7) automatically above satisfies $\widetilde{TD} = 0$ for all state action pairs. Whether a limit of our Q-learning algorithm converges to an optimal Q-table depends on whether the algorithm visits all state-action pairs often enough.

We implement this pseudo code in a Python class.

For simplicity and convenience, we let `s` represent the state index between 0 and $n = 50$ and $w_s = w[s]$.

The first column of the Q-table represents the value associated with rejecting the wage and the second represents accepting the wage.

We use `numba` compilation to accelerate computations.

```
params=[
    ('c', float64),            # unemployment compensation
    ('β', float64),            # discount factor
    ('w', float64[:]),         # array of wage values, w[i] = wage at state i
    ('q', float64[:]),         # array of probabilities
    ('eps', float64),          # for epsilon greedy algorithm
    ('δ', float64),            # Q-table threshold
    ('lr', float64),           # the learning rate α
    ('T', int64),              # maximum periods of accepting
    ('quit_allowed', int64)    # whether quit is allowed after accepting the wage␣
↪offer
]

@jitclass(params)
class Qlearning_McCall:
    def __init__(self, c=25, β=0.99, w=w_default, q=q_default, eps=0.1,
                 δ=1e-5, lr=0.5, T=10000, quit_allowed=0):

        self.c, self.β = c, β
        self.w, self.q = w, q
        self.eps, self.δ, self.lr, self.T = eps, δ, lr, T
        self.quit_allowed = quit_allowed


    def draw_offer_index(self):
        """
        Draw a state index from the wage distribution.
        """

        q = self.q
        return np.searchsorted(np.cumsum(q), np.random.random(), side="right")

    def temp_diff(self, qtable, state, accept):
        """
        Compute the TD associated with state and action.
        """

        c, β, w = self.c, self.β, self.w
```

(continues on next page)

(continued from previous page)

```
        if accept==0:
            state_next = self.draw_offer_index()
            TD = c + β*np.max(qtable[state_next, :]) - qtable[state, accept]
        else:
            state_next = state
            if self.quit_allowed == 0:
                TD = w[state_next] + β*np.max(qtable[state_next, :]) - qtable[state,␣
↪accept]
            else:
                TD = w[state_next] + β*qtable[state_next, 1] - qtable[state, accept]

        return TD, state_next

    def run_one_epoch(self, qtable, max_times=20000):
        """
        Run an "epoch".
        """

        c, β, w = self.c, self.β, self.w
        eps, δ, lr, T = self.eps, self.δ, self.lr, self.T

        s0 = self.draw_offer_index()
        s = s0
        accept_count = 0

        for t in range(max_times):

            # choose action
            accept = np.argmax(qtable[s, :])
            if np.random.random()<=eps:
                accept = 1 - accept

            if accept == 1:
                accept_count += 1
            else:
                accept_count = 0

            TD, s_next = self.temp_diff(qtable, s, accept)

            # update qtable
            qtable_new = qtable.copy()
            qtable_new[s, accept] = qtable[s, accept] + lr*TD

            if np.max(np.abs(qtable_new-qtable))<=δ:
                break

            if accept_count == T:
                break

            s, qtable = s_next, qtable_new

        return qtable_new

@jit
def run_epochs(N, qlmc, qtable):
    """
```

(continues on next page)

(continued from previous page)

```
    Run epochs N times with qtable from the last iteration each time.
    """

    for n in range(N):
        if n%(N/10)==0:
            print(f"Progress: EPOCHs = {n}")
        new_qtable = qlmc.run_one_epoch(qtable)
        qtable = new_qtable

    return qtable

def valfunc_from_qtable(qtable):
    return np.max(qtable, axis=1)

def compute_error(valfunc, valfunc_VFI):
    return np.mean(np.abs(valfunc-valfunc_VFI))
```

```
# create an instance of Qlearning_McCall
qlmc = Qlearning_McCall()

# run
qtable0 = np.zeros((len(w_default), 2))
qtable = run_epochs(20000, qlmc, qtable0)
```

```
Progress: EPOCHs = 0
Progress: EPOCHs = 2000
Progress: EPOCHs = 4000
Progress: EPOCHs = 6000
Progress: EPOCHs = 8000
Progress: EPOCHs = 10000
Progress: EPOCHs = 12000
Progress: EPOCHs = 14000
Progress: EPOCHs = 16000
Progress: EPOCHs = 18000
```

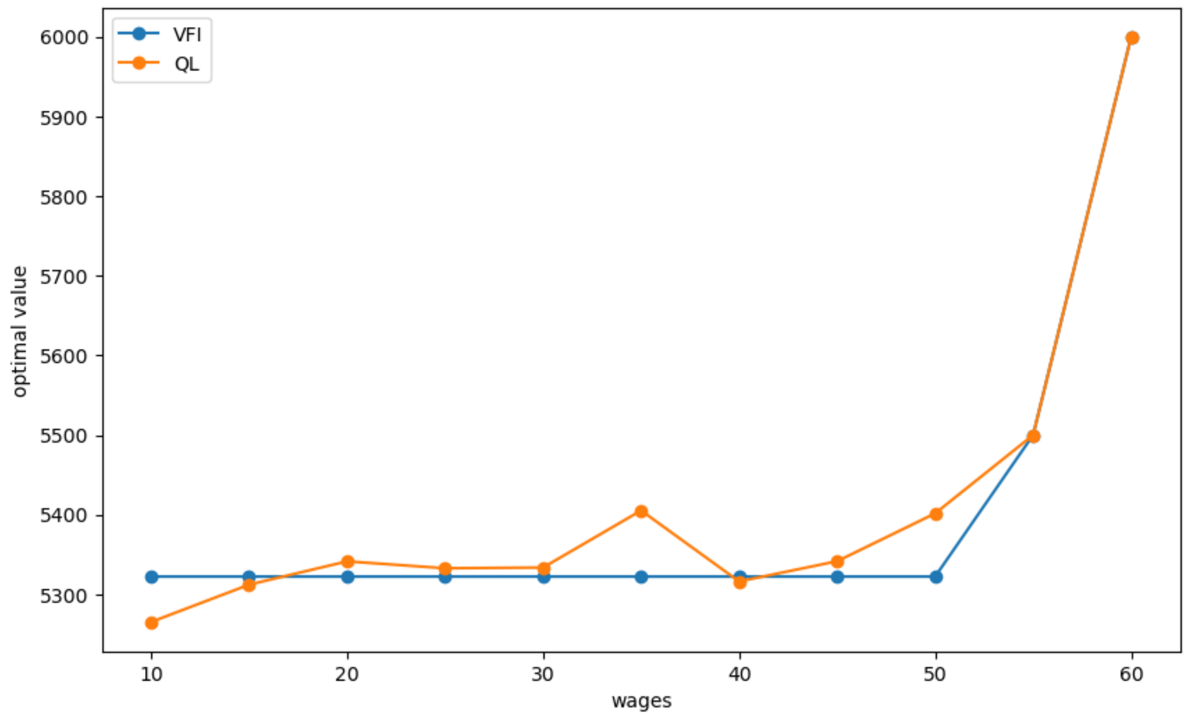
```
print(qtable)
```

```
[[5265.22778424    0.        ]
 [5312.11433307 5243.18248138]
 [5322.18770558 5341.38560333]
 [5332.72923538 5283.84592747]
 [5333.60861011 5288.46288689]
 [5405.23522909 5234.34927276]
 [5312.00057187 5316.14739432]
 [5310.79741249 5341.53153792]
 [5401.59059063 5212.79792124]
 [5380.34889181 5500.00000797]
 [5262.71213971 6000.        ]]
```

```
# inspect value function
valfunc_qlr = valfunc_from_qtable(qtable)

print(valfunc_qlr)
```

```
[5265.22778424 5312.11433307 5341.38560333 5332.72923538 5333.60861011
 5405.23522909 5316.14739432 5341.53153792 5401.59059063 5500.00000797
 6000.        ]
```

```
# plot
fig, ax = plt.subplots(figsize=(10,6))
ax.plot(w_default, valfunc_VFI, '-o', label='VFI')
ax.plot(w_default, valfunc_qlr, '-o', label='QL')
ax.set_xlabel('wages')
ax.set_ylabel('optimal value')
ax.legend()

plt.show()
```



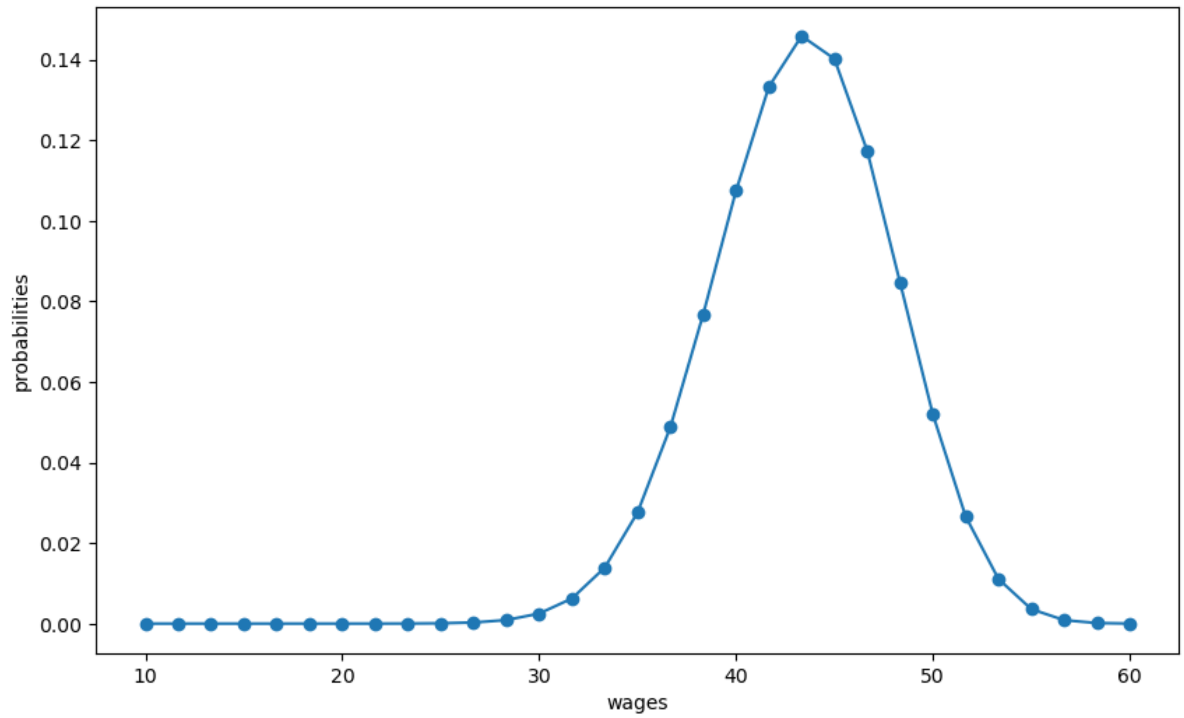
Now, let us compute the case with a larger state space: $n = 30$ instead of $n = 10$.

```
n, a, b = 30, 200, 100                          # default parameters
q_new = BetaBinomial(n, a, b).pdf()             # default choice of q

w_min, w_max = 10, 60
w_new = np.linspace(w_min, w_max, n+1)


# plot distribution of wage offer
fig, ax = plt.subplots(figsize=(10,6))
ax.plot(w_new, q_new, '-o', label='$q(w(i))$')
ax.set_xlabel('wages')
ax.set_ylabel('probabilities')

plt.show()
```

(continues on next page)

(continued from previous page)

```
# VFI
mcm = McCallModel(w=w_new, q=q_new)
valfunc_VFI, flag = mcm.VFI()
```




```
mcm = McCallModel(w=w_new, q=q_new)
valfunc_VFI, flag = mcm.VFI()
valfunc_VFI
```

```
array([4859.77015703, 4859.77015703, 4859.77015703, 4859.77015703,
       4859.77015703, 4859.77015703, 4859.77015703, 4859.77015703,
       4859.77015703, 4859.77015703, 4859.77015703, 4859.77015703,
       4859.77015703, 4859.77015703, 4859.77015703, 4859.77015703,
       4859.77015703, 4859.77015703, 4859.77015703, 4859.77015703,
       4859.77015703, 4859.77015703, 4859.77015703, 4859.77015703,
       5000.        , 5166.66666667, 5333.33333333, 5500.        ,
       5666.66666667, 5833.33333333, 6000.        ])
```

```
def plot_epochs(epochs_to_plot, quit_allowed=1):
    "Plot value function implied by outcomes of an increasing number of epochs."
    qlmc_new = Qlearning_McCall(w=w_new, q=q_new, quit_allowed=quit_allowed)
    qtable = np.zeros((len(w_new),2))
    epochs_to_plot = np.asarray(epochs_to_plot)
    # plot
    fig, ax = plt.subplots(figsize=(10,6))
    ax.plot(w_new, valfunc_VFI, '-o', label='VFI')

    max_epochs = np.max(epochs_to_plot)
    # iterate on epoch numbers
    for n in range(max_epochs + 1):
        if n%(max_epochs/10)==0:
```

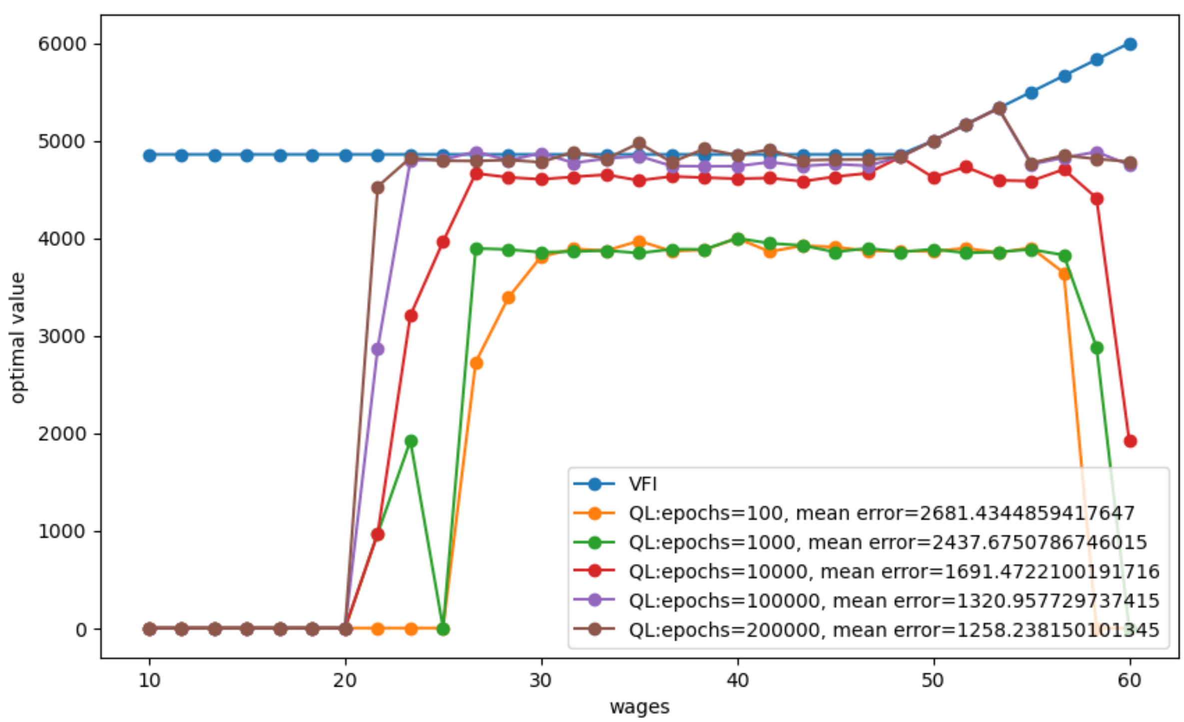
(continues on next page)

(continued from previous page)

```
            print(f"Progress: EPOCHs = {n}")
        if n in epochs_to_plot:
            valfunc_qlr = valfunc_from_qtable(qtable)
            error = compute_error(valfunc_qlr, valfunc_VFI)

            ax.plot(w_new, valfunc_qlr, '-o', label=f'QL:epochs={n}, mean error=
↪{error}')


        new_qtable = qlmc_new.run_one_epoch(qtable)
        qtable = new_qtable

    ax.set_xlabel('wages')
    ax.set_ylabel('optimal value')
    ax.legend(loc='lower right')
    plt.show()
```

```
plot_epochs(epochs_to_plot=[100, 1000, 10000, 100000, 200000])
```

```
Progress: EPOCHs = 0
Progress: EPOCHs = 20000
Progress: EPOCHs = 40000
Progress: EPOCHs = 60000
Progress: EPOCHs = 80000
Progress: EPOCHs = 100000
Progress: EPOCHs = 120000
Progress: EPOCHs = 140000
Progress: EPOCHs = 160000
Progress: EPOCHs = 180000
Progress: EPOCHs = 200000
```



The above graphs indicates that

- the Q-learning algorithm has trouble learning the Q-table well for wages that are rarely drawn
- the quality of approximation to the "true" value function computed by value function iteration improves for longer epochs

52.6 Employed Worker Can't Quit

The preceding version of temporal difference Q-learning described in equation system (52.8) lets an employed worker quit, i.e., reject her wage as an incumbent and instead receive unemployment compensation this period and draw a new offer next period.

This is an option that the McCall worker described in *this quantecon lecture* would not take.

See [Ljungqvist and Sargent, 2018], chapter 6 on search, for a proof.

But in the context of Q-learning, giving the worker the option to quit and get unemployment compensation while unemployed turns out to accelerate the learning process by promoting experimentation vis a vis premature exploitation only.

To illustrate this, we'll amend our formulas for temporal differences to forbid an employed worker from quitting a job she had accepted earlier.

With this understanding about available choices, we obtain the following temporal difference values:

$$
\begin{aligned}
\widetilde{TD}(w, \text{accept}) &= \left[w + \beta \widetilde{Q}^{old}(w, \text{accept}) \right] - \widetilde{Q}^{old}(w, \text{accept}) \\
\widetilde{TD}(w, \text{reject}) &= \left[c + \beta \max_{a' \in \mathcal{A}} \widetilde{Q}^{old}(w', a') \right] - \widetilde{Q}^{old}(w, \text{reject}), \ w' \sim F
\end{aligned} \tag{52.9}
$$

It turns out that formulas (52.9) combined with our Q-learning recursion (52.7) can lead our agent to eventually learn the optimal value function as well as in the case where an option to redraw can be exercised.

But learning is slower because an agent who ends up accepting a wage offer prematurally loses the option to explore new states in the same episode and to adjust the value associated with that state.

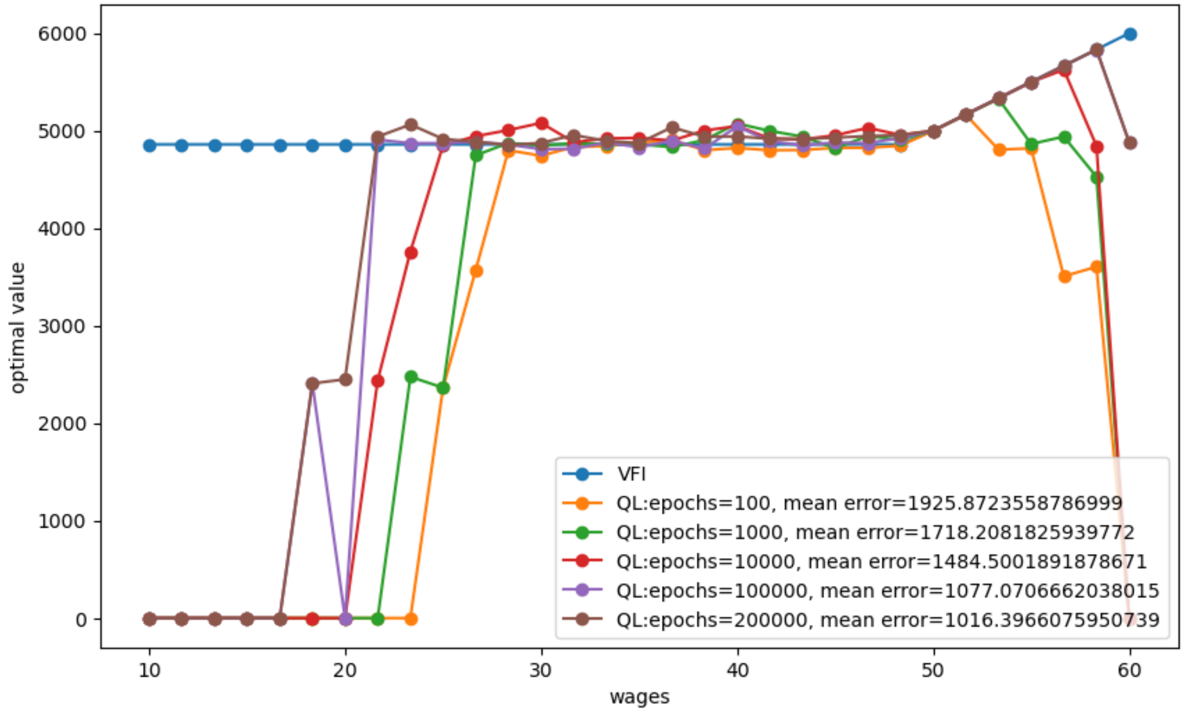
This can lead to inferior outcomes when the number of epochs/episodes is low.

But if we increase the number of epochs/episodes, we can observe that the error decreases and the outcomes get better.

We illustrate these possibilities with the following code and graph.

```
plot_epochs(epochs_to_plot=[100, 1000, 10000, 100000, 200000], quit_allowed=0)
```

```
Progress: EPOCHs = 0
Progress: EPOCHs = 20000
Progress: EPOCHs = 40000
Progress: EPOCHs = 60000
Progress: EPOCHs = 80000
Progress: EPOCHs = 100000
Progress: EPOCHs = 120000
Progress: EPOCHs = 140000
Progress: EPOCHs = 160000
Progress: EPOCHs = 180000
Progress: EPOCHs = 200000
```



52.7 Possible Extensions

To extend the algorthm to handle problems with continuous state spaces, a typical approach is to restrict Q-functions and policy functions to take particular functional forms.

This is the approach in **deep Q-learning** where the idea is to use a multilayer neural network as a good function approximator.

We will take up this topic in a subsequent quantecon lecture.

Part VIII

Introduction to Optimal Savings

CHAPTER

FIFTYTHREE

OPTIMAL SAVINGS I: CAKE EATING

Contents

53.1 Overview

In this lecture we introduce a simple "cake eating" problem.

The intertemporal problem is: how much to enjoy today and how much to leave for the future?

This trade-off between current and future rewards is at the heart of many savings and consumption problems.

Once we master the ideas in a simple environment, we will apply them to progressively more challenging problems.

The main tool we will use to solve the cake eating problem is dynamic programming.

The following lectures contain background on dynamic programming and might be worth reviewing:

- The shortest paths lecture
- The *basic McCall model*
- The *McCall model with separation*
- The *McCall model with separation and a continuous wage distribution*

We require the following imports:

```
import matplotlib.pyplot as plt
import numpy as np
```

53.2 The model

We consider a discrete time model with an infinite time horizon.

At $t = 0$ the agent is given a complete cake with size $\bar{x}$.

Let x_t denote the size of the cake at the beginning of each period, so that, in particular, $x_0 = \bar{x}$.

We choose how much of the cake to eat in any given period t.

After choosing to consume c_t of the cake in period t, the amount left in period $t + 1$ is

$$x_{t+1} = x_t - c_t$$

Consuming quantity c of the cake gives current utility $u(c)$.

We adopt the CRRA utility function

$$u(c) = \frac{c^{1-\gamma}}{1-\gamma} \qquad (\gamma > 0, \, \gamma \neq 1) \tag{53.1}$$

In Python this is

```
def u(c, γ):

    return c**(1 - γ) / (1 - γ)
```

Future cake consumption utility is discounted according to $\beta \in (0, 1)$.

In particular, consumption of c units t periods hence has present value $\beta^t u(c)$

The agent's problem can be written as

$$\max_{\{c_t\}} \sum_{t=0}^{\infty} \beta^t u(c_t) \tag{53.2}$$

subject to

$$x_{t+1} = x_t - c_t \quad \text{and} \quad 0 \leq c_t \leq x_t \tag{53.3}$$

for all t.

A consumption path $\{c_t\}$ satisfying (53.3) and $x_0 = \bar{x}$ is called **feasible**.

In this problem, the following terminology is standard:

- x_t is called the **state variable**
- c_t is called the **control variable** or the **action**
- β and γ are **parameters**

53.2.1 Trade-off

The key trade-off in the cake-eating problem is this:

- Delaying consumption is costly because of the discount factor.
- But delaying some consumption is also attractive because u is concave.

The concavity of u implies that the consumer gains value from **consumption smoothing**, which means spreading consumption out over time.

This is because concavity implies diminishing marginal utility—a progressively smaller gain in utility for each additional spoonful of cake consumed within one period.

53.2.2 Intuition

The reasoning given above suggests that the discount factor β and the curvature parameter γ will play a key role in determining the rate of consumption.

Here's an educated guess as to what impact these parameters will have.

1. Higher β implies less discounting, and hence the agent is more patient, which should reduce the rate of consumption.
2. Higher γ implies more curvature in u, more desire for consumption smoothing, and hence a lower rate of consumption.

Note

More formally, higher γ implies a lower intertemporal elasticity of substitution, since IES = $1/\gamma$.

This means the consumer is less willing to substitute consumption between periods.

This stronger preference for consumption smoothing results in a lower consumption rate.

In summary, we expect the rate of consumption to be decreasing in both parameters.

Let's see if this is true.

53.3 The value function

The first step of our dynamic programming treatment is to obtain the Bellman equation.

The next step is to use it to calculate the solution.

53.3.1 The Bellman equation

To this end, we let $v(x)$ be maximum lifetime utility attainable from the current time when x units of cake are left.

That is,

$$v(x) = \max \sum_{t=0}^{\infty} \beta^t u(c_t) \tag{53.4}$$

where the maximization is over all paths $\{c_t\}$ that are feasible from $x_0 = x$.

At this point, we do not have an expression for v, but we can still make inferences about it.

For example, as was the case with the *McCall model*, the value function will satisfy a version of the *Bellman equation*.

In the present case, this equation states that v satisfies

$$v(x) = \max_{0 \leq c \leq x} \{u(c) + \beta v(x - c)\} \quad \text{for any given } x \geq 0. \tag{53.5}$$

The intuition here is essentially the same as it was for the McCall model.

Choosing c optimally means trading off current vs future rewards.

Current rewards from choice c are just $u(c)$.

Future rewards given current cake size x, measured from next period and assuming optimal behavior, are $v(x - c)$.

These are the two terms on the right hand side of (53.5), after suitable discounting.

If c is chosen optimally using this trade off strategy, then we obtain maximal lifetime rewards from our current state x.

Hence, $v(x)$ equals the right hand side of (53.5), as claimed.

53.3.2 An analytical solution

It has been shown that, with u as the CRRA utility function in (53.1), the function v^* given by

$$v^*(x) = \left(1 - \beta^{1/\gamma}\right)^{-\gamma} u(x) \tag{53.6}$$

solves the Bellman equation and hence is equal to the value function.

You are asked to confirm that this is true in the exercises below.

Note

The solution (53.6) depends heavily on the CRRA utility function.

In fact, if we move away from CRRA utility, usually there is no analytical solution at all.

In other words, beyond CRRA utility, we know that the value function still satisfies the Bellman equation, but we do not have a way of writing it explicitly, as a function of the state variable and the parameters.

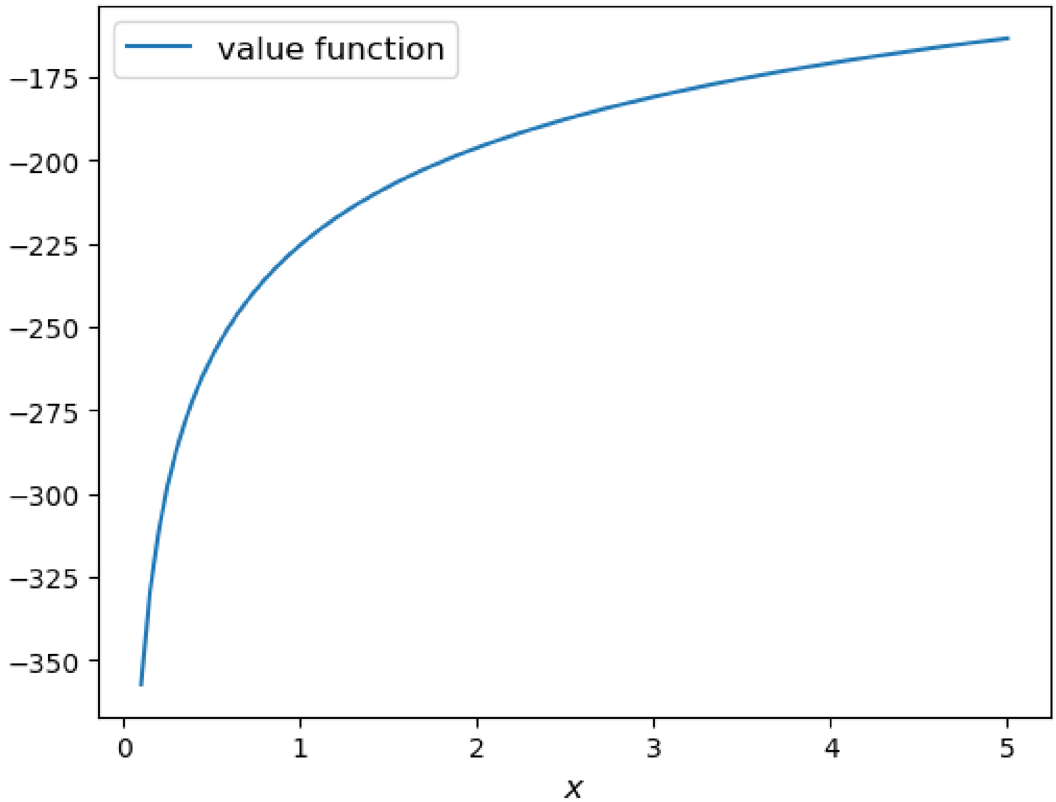
We will deal with that situation numerically in the following lectures.

Here is a Python representation of the value function:

```
def v_star(x, β, γ):

    return (1 - β**(1 / γ))**(-γ) * u(x, γ)
```

And here's a figure showing the function for fixed parameters:

```
β, γ = 0.95, 1.2
x_grid = np.linspace(0.1, 5, 100)

fig, ax = plt.subplots()

ax.plot(x_grid, v_star(x_grid, β, γ), label='value function')

ax.set_xlabel('$x$', fontsize=12)
ax.legend(fontsize=12)

plt.show()
```



53.4 The optimal policy

Now that we have the value function v^*, it is straightforward to calculate the optimal action at each state.

We should choose consumption to maximize the right hand side of the Bellman equation (53.5).

$$c^* = \arg \max_{0 \leq c \leq x} \{u(c) + \beta v^*(x - c)\}$$

We can think of this optimal choice as a *function* of the state x, in which case we call it the **optimal policy**.

We denote the optimal policy by σ^*, so that

$$\sigma^*(x) := \arg \max_{c} \{u(c) + \beta v^*(x - c)\} \quad \text{for all} \quad x \geq 0$$

If we plug the analytical expression (53.6) for the value function into the right hand side and compute the optimum, we find that

$$\sigma^*(x) = \left(1 - \beta^{1/\gamma}\right) x \tag{53.7}$$

Now let's recall our intuition on the impact of parameters.

We guessed that consumption would be decreasing in both parameters.

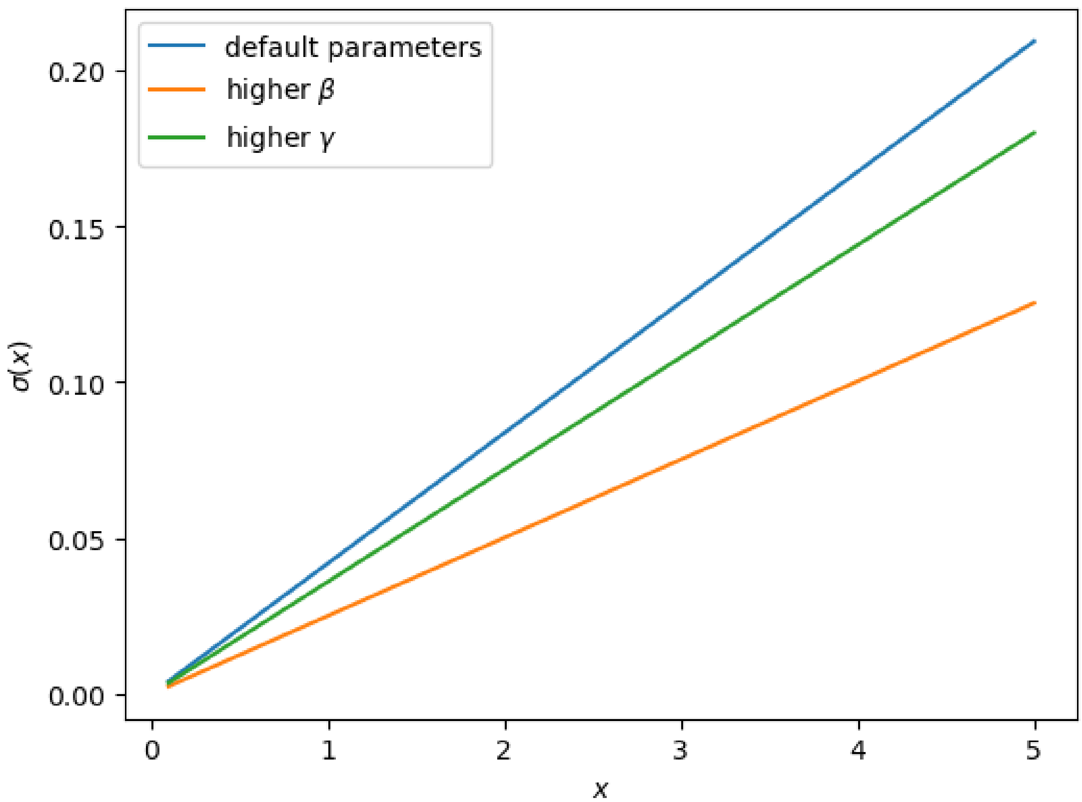
This is in fact the case, as can be seen from (53.7).

Here are some plots that illustrate.

```
def c_star(x, β, γ):

    return (1 - β ** (1/γ)) * x
```

Continuing with the values for β and γ used above, the plot is

```
fig, ax = plt.subplots()
ax.plot(x_grid, c_star(x_grid, β, γ), label='default parameters')
ax.plot(x_grid, c_star(x_grid, β + 0.02, γ), label=r'higher $\beta$')
ax.plot(x_grid, c_star(x_grid, β, γ + 0.2), label=r'higher $\gamma$')
ax.set_ylabel(r'$\sigma(x)$')
ax.set_xlabel('$x$')
ax.legend()

plt.show()
```



53.5 The Euler equation

In the discussion above we have provided a complete solution to the cake eating problem in the case of CRRA utility.

There is in fact another way to solve for the optimal policy, based on the so-called **Euler equation**.

Although we already have a complete solution, now is a good time to study the Euler equation.

This is because, for more difficult problems, this equation provides key insights that are hard to obtain by other methods.

53.5.1 Statement and implications

The Euler equation for the present problem can be stated as

$$u'(c_t^*) = \beta u'(c_{t+1}^*) \tag{53.8}$$

This is a necessary condition for an optimal consumption path $\{c_t^*\}_{t \geq 0}$.

It says that, along the optimal path, marginal rewards are equalized across time, after appropriate discounting.

This makes sense: optimality is obtained by smoothing consumption up to the point where no marginal gains remain.

We can also state the Euler equation in terms of the policy function.

A **feasible consumption policy** is a map $x \mapsto \sigma(x)$ satisfying $0 \leq \sigma(x) \leq x$.

(The last restriction says that we cannot consume more than the remaining quantity of cake.)

A feasible consumption policy σ is said to **satisfy the Euler equation** if, for all $x > 0$,

$$u'(\sigma(x)) = \beta u'(\sigma(x - \sigma(x))) \tag{53.9}$$

Evidently (53.9) is just the policy equivalent of (53.8).

It turns out that a feasible policy is optimal if and only if it satisfies the Euler equation.

In the exercises, you are asked to verify that the optimal policy (53.7) does indeed satisfy this functional equation.

Note

A **functional equation** is an equation where the unknown object is a function.

For a proof of sufficiency of the Euler equation in a very general setting, see proposition 2.2 of [Ma *et al.*, 2020].

The following arguments focus on necessity, explaining why an optimal path or policy should satisfy the Euler equation.

53.5.2 Derivation I: a perturbation approach

Let's write c as a shorthand for consumption path $\{c_t\}_{t=0}^\infty$.

The overall cake-eating maximization problem can be written as

$$\max_{c \in F} U(c) \quad \text{where } U(c) := \sum_{t=0}^\infty \beta^t u(c_t)$$

and F is the set of feasible consumption paths.

We know that differentiable functions have a zero gradient at a maximizer.

So the optimal path $c^* := \{c_t^*\}_{t=0}^\infty$ must satisfy $U'(c^*) = 0$.

Note

If you want to know exactly how the derivative $U'(c^*)$ is defined, given that the argument c^* is a vector of infinite length, you can start by learning about Gateaux derivatives. However, such knowledge is not assumed in what follows.

In other words, the rate of change in U must be zero for any infinitesimally small (and feasible) perturbation away from the optimal path.

So consider a feasible perturbation that reduces consumption at time t to $c_t^* - h$ and increases it in the next period to $c_{t+1}^* + h$.

Consumption does not change in any other period.

We call this perturbed path c^h.

By the preceding argument about zero gradients, we have

$$\lim_{h \to 0} \frac{U(c^h) - U(c^*)}{h} = U'(c^*) = 0$$

Recalling that consumption only changes at t and $t+1$, this becomes

$$\lim_{h \to 0} \frac{\beta^t u(c_t^* - h) + \beta^{t+1} u(c_{t+1}^* + h) - \beta^t u(c_t^*) - \beta^{t+1} u(c_{t+1}^*)}{h} = 0$$

After rearranging, the same expression can be written as

$$\lim_{h \to 0} \frac{u(c_t^* - h) - u(c_t^*)}{h} + \beta \lim_{h \to 0} \frac{u(c_{t+1}^* + h) - u(c_{t+1}^*)}{h} = 0$$

or, taking the limit,

$$-u'(c_t^*) + \beta u'(c_{t+1}^*) = 0$$

This is just the Euler equation.

53.5.3 Derivation II: using the Bellman equation

Another way to derive the Euler equation is to use the Bellman equation (53.5).

> **Note**
>
> The argument that follows assumes that the value function is differentiable. A proof of differentiability of the value function can be found in EDTC, theorem 10.1.13.

Taking the derivative on the right hand side of the Bellman equation with respect to c and setting it to zero, we get

$$u'(c) = \beta v'(x - c) \tag{53.10}$$

To obtain $v'(x-c)$, we set $g(c, x) = u(c) + \beta v(x - c)$, so that, at the optimal choice of consumption,

$$v(x) = g(c, x) \tag{53.11}$$

Differentiating both sides while acknowledging that the maximizing consumption will depend on x, we get

$$v'(x) = \frac{\partial}{\partial c} g(c, x) \frac{\partial c}{\partial x} + \frac{\partial}{\partial x} g(c, x)$$

When $g(c, x)$ is maximized at c, we have $\frac{\partial}{\partial c} g(c, x) = 0$.

Hence the derivative simplifies to

$$v'(x) = \frac{\partial g(c, x)}{\partial x} = \frac{\partial}{\partial x} \beta v(x - c) = \beta v'(x - c) \tag{53.12}$$

(This argument is an example of the Envelope Theorem.)

But now an application of (53.10) gives

$$u'(c) = v'(x) \tag{53.13}$$

Thus, the derivative of the value function is equal to marginal utility.

Combining this fact with (53.12) recovers the Euler equation.

53.6 Exercises

Exercise 53.6.1

How does one obtain the expressions for the value function and optimal policy given in (53.6) and (53.7) respectively?

The first step is to make a guess of the functional form for the consumption policy.

So suppose that we do not know the solutions and start with a guess that the optimal policy is linear.

In other words, we conjecture that there exists a positive θ such that setting $c_t^* = \theta x_t$ for all t produces an optimal path.

Starting from this conjecture, try to obtain the solutions (53.6) and (53.7).

In doing so, you will need to use the definition of the value function and the Bellman equation.

Solution

We start with the conjecture $c_t^* = \theta x_t$, which leads to a path for the state variable (cake size) given by

$$x_{t+1} = x_t(1-\theta)$$

Then $x_t = x_0(1-\theta)^t$ and hence

$$\begin{aligned}
v(x_0) &= \sum_{t=0}^{\infty} \beta^t u(\theta x_t) \\
&= \sum_{t=0}^{\infty} \beta^t u(\theta x_0 (1-\theta)^t) \\
&= \sum_{t=0}^{\infty} \theta^{1-\gamma} \beta^t (1-\theta)^{t(1-\gamma)} u(x_0) \\
&= \frac{\theta^{1-\gamma}}{1-\beta(1-\theta)^{1-\gamma}} u(x_0)
\end{aligned}$$

From the Bellman equation, then,

$$\begin{aligned}
v(x) &= \max_{0 \le c \le x} \left\{ u(c) + \beta \frac{\theta^{1-\gamma}}{1-\beta(1-\theta)^{1-\gamma}} \cdot u(x-c) \right\} \\
&= \max_{0 \le c \le x} \left\{ \frac{c^{1-\gamma}}{1-\gamma} + \beta \frac{\theta^{1-\gamma}}{1-\beta(1-\theta)^{1-\gamma}} \cdot \frac{(x-c)^{1-\gamma}}{1-\gamma} \right\}
\end{aligned}$$

From the first order condition, we obtain

$$c^{-\gamma} + \beta \frac{\theta^{1-\gamma}}{1-\beta(1-\theta)^{1-\gamma}} \cdot (x-c)^{-\gamma}(-1) = 0$$

or

$$c^{-\gamma} = \beta \frac{\theta^{1-\gamma}}{1-\beta(1-\theta)^{1-\gamma}} \cdot (x-c)^{-\gamma}$$

With $c = \theta x$ we get

$$(\theta x)^{-\gamma} = \beta \frac{\theta^{1-\gamma}}{1-\beta(1-\theta)^{1-\gamma}} \cdot (x(1-\theta))^{-\gamma}$$

Some rearrangement produces

$$\theta = 1 - \beta^{\frac{1}{\gamma}}$$

This confirms our earlier expression for the optimal policy:

$$c_t^* = \left(1 - \beta^{\frac{1}{\gamma}}\right) x_t$$

Substituting θ into the value function above gives

$$v^*(x_t) = \frac{\left(1-\beta^{\frac{1}{\gamma}}\right)^{1-\gamma}}{1-\beta\left(\beta^{\frac{1-\gamma}{\gamma}}\right)} u(x_t)$$

Rearranging gives

$$v^*(x_t) = \left(1-\beta^{\frac{1}{\gamma}}\right)^{-\gamma} u(x_t)$$

Our claims are now verified.

Exercise 53.6.2

Verify that the optimal policy (53.7) satisfies the Euler equation (53.9).

Solution

Recall that the optimal policy is

$$\sigma^*(x) = (1 - \beta^{1/\gamma})x$$

and the Euler equation in policy form is

$$u'(\sigma(x)) = \beta u'(\sigma(x - \sigma(x)))$$

With CRRA utility $u(c) = \frac{c^{1-\gamma}}{1-\gamma}$, the marginal utility is

$$u'(c) = c^{-\gamma}$$

Now let's verify the Euler equation. The left-hand side is

$$u'(\sigma^*(x)) = [\sigma^*(x)]^{-\gamma} = [(1-\beta^{1/\gamma})x]^{-\gamma} = (1-\beta^{1/\gamma})^{-\gamma} x^{-\gamma}$$

For the right-hand side, we first compute the state in the next period:

$$x - \sigma^*(x) = x - (1 - \beta^{1/\gamma})x = x\beta^{1/\gamma}$$

Next period's consumption under the optimal policy is

$$\sigma^*(x - \sigma^*(x)) = \sigma^*(x\beta^{1/\gamma}) = (1 - \beta^{1/\gamma}) \cdot x\beta^{1/\gamma}$$

Therefore, the right-hand side of the Euler equation is

$$\begin{aligned}
\beta u'(\sigma^*(x - \sigma^*(x))) &= \beta[(1 - \beta^{1/\gamma}) \cdot x\beta^{1/\gamma}]^{-\gamma} \\
&= \beta(1 - \beta^{1/\gamma})^{-\gamma} x^{-\gamma} (\beta^{1/\gamma})^{-\gamma} \\
&= \beta(1 - \beta^{1/\gamma})^{-\gamma} x^{-\gamma} \beta^{-1} \\
&= (1 - \beta^{1/\gamma})^{-\gamma} x^{-\gamma}
\end{aligned}$$

Since the left-hand side equals the right-hand side, the optimal policy σ^* satisfies the Euler equation.

CHAPTER

FIFTYFOUR

OPTIMAL SAVINGS II: NUMERICAL CAKE EATING

Contents

54.1 Overview

In this lecture we continue the study of the problem described in *Optimal Savings I: Cake Eating*.

The aim of this lecture is to solve the problem using numerical methods.

At first this might appear unnecessary, since we already obtained the optimal policy analytically.

However, the cake eating problem is too simple to be useful without modifications, and once we start modifying the problem, numerical methods become essential.

Hence it makes sense to introduce numerical methods now, and test them on this simple problem.

Since we know the analytical solution, this will allow us to assess the accuracy of alternative numerical methods.

Note

The code below aims for clarity rather than maximum efficiency.

In the lectures below we will explore best practice for speed and efficiency.

Let's put these algorithm and code optimizations to one side for now.

We will use the following imports:

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.optimize import minimize_scalar, bisect
from typing import NamedTuple
```

54.2 Reviewing the Model

You might like to review the details in *Optimal Savings I: Cake Eating* before we start.

Recall in particular that the Bellman equation is

$$v(x) = \max_{0 \leq c \leq x} \{u(c) + \beta v(x - c)\} \quad \text{for all } x \geq 0. \tag{54.1}$$

where u is the CRRA utility function.

The analytical solutions for the value function and optimal policy were found to be as follows.

```
def c_star(x, β, γ):

    return (1 - β ** (1/γ)) * x


def v_star(x, β, γ):

    return (1 - β**(1 / γ))**(-γ) * (x**(1-γ) / (1-γ))
```

Our first aim is to obtain these analytical solutions numerically.

54.3 Value Function Iteration

The first approach we will take is **value function iteration**.

This is a form of **successive approximation**, and was discussed in our *lecture on job search*.

The basic idea is:

1. Take an arbitrary initial guess of v.
2. Obtain an update $\hat{v}$ defined by

$$\hat{v}(x) = \max_{0 \leq c \leq x} \{u(c) + \beta v(x - c)\}$$

3. Stop if $\hat{v}$ is approximately equal to v, otherwise set $v = \hat{v}$ and go back to step 2.

Let's write this a bit more mathematically.

54.3.1 The Bellman Operator

We introduce the **Bellman operator** T that takes a function v as an argument and returns a new function Tv defined by

$$Tv(x) = \max_{0 \leq c \leq x} \{u(c) + \beta v(x - c)\}$$

From v we get Tv, and applying T to this yields $T^2 v := T(Tv)$ and so on.

This is called **iterating with the Bellman operator** from initial guess v.

As we discuss in more detail in later lectures, one can use Banach's contraction mapping theorem to prove that the sequence of functions $T^n v$ converges to the solution to the Bellman equation.

54.3.2 Fitted Value Function Iteration

Both consumption c and the state variable x are continuous.

This causes complications when it comes to numerical work.

For example, we need to store each function $T^n v$ in order to compute the next iterate $T^{n+1} v$.

But this means we have to store $T^n v(x)$ at infinitely many x, which is, in general, impossible.

To circumvent this issue we will use fitted value function iteration, as discussed previously in *one of the lectures* on job search.

The process looks like this:

1. Begin with an array of values $\{v_0, \ldots, v_I\}$ representing the values of some initial function v on the grid points $\{x_0, \ldots, x_I\}$.
2. Build a function $\hat{v}$ on the state space $\mathbb{R}_+$ by interpolation, based on the interpolation points $\{(x_i, v_i)\}$.
3. Insert $\hat{v}$ into the right hand side of the Bellman equation and obtain and record the value $T\hat{v}(x_i)$ on each grid point x_i.
4. Unless some stopping condition is satisfied, set $\{v_0, \ldots, v_I\} = \{T\hat{v}(x_0), \ldots, T\hat{v}(x_I)\}$ and go to step 2.

In step 2 we'll use piecewise linear interpolation.

54.3.3 Implementation

The `maximize` function below is a small helper function that converts a SciPy minimization routine into a maximization routine.

```python
def maximize(g, upper_bound):
    """
    Maximize the function g over the interval [0, upper_bound].

    We use the fact that the maximizer of g on any interval is
    also the minimizer of -g.

    """

    objective = lambda x: -g(x)
    bounds = (0, upper_bound)
    result = minimize_scalar(objective, bounds=bounds, method='bounded')
    maximizer, maximum = result.x, -result.fun
    return maximizer, maximum
```

We'll store the parameters β and γ and the grid in a `NamedTuple` called `Model`.

We'll also create a helper function called `create_cake_eating_model` to store default parameters and build an instance of `Model`.

```python
class Model(NamedTuple):
    β: float
    γ: float
    x_grid: np.ndarray

def create_cake_eating_model(
        β: float = 0.96,           # discount factor
        γ: float = 1.5,            # degree of relative risk aversion
```

(continues on next page)

(continued from previous page)

```
        x_grid_min: float = 1e-3,  # exclude zero for numerical stability
        x_grid_max: float = 2.5,   # size of cake
        x_grid_size: int = 120
    ):
    """
    Creates an instance of the cake eating model.

    """
    x_grid = np.linspace(x_grid_min, x_grid_max, x_grid_size)
    return Model(β, γ, x_grid)
```

Here's the CRRA utility function.

```
def u(c, γ):
    return (c ** (1 - γ)) / (1 - γ)
```

To work with the Bellman equation, let's write it as

$$v(x) = \max_{0 \leq c \leq x} B(x, c, v)$$

where

$$B(x, c, v) := u(c) + \beta v(x - c)$$

Now we implement the function B.

```
def B(
        x: float,               # the current state (remaining cake)
        c: float,               # current consumption
        v: np.ndarray,          # current guess of the value function
        model: Model            # instance of cake eating model
    ):
    """
    Right hand side of the Bellman equation given x and c.

    """
    # Unpack (simplify names)
    β, γ, x_grid = model

    # Convert array v into a function by linear interpolation
    vf = lambda x: np.interp(x, x_grid, v)

    # Return B(x, c, v)
    return u(c, γ) + β * vf(x - c)
```

We now define the Bellman operator acting on grid points:

$$Tv(x_i) = \max_{0 \leq c \leq x_i} B(x_i, c, v) \qquad \text{for all } i$$

```
def T(
        v: np.ndarray,          # Current guess of the value function
        model: Model            # Instance of the cake eating model
    ):
    " The Bellman operator.  Updates the guess of the value function. "

    # Allocate memory for the new array v_new = Tv
```

(continues on next page)

(continued from previous page)

```
    v_new = np.empty_like(v)

    # Calculate Tv(x) for all x
    for i, x in enumerate(model.x_grid):
        # Maximize RHS of Bellman equation with respect to c over [0, x]
        _, v_new[i] = maximize(lambda c: B(x, c, v, model), x)

    return v_new
```

After defining the Bellman operator, we are ready to solve the model.

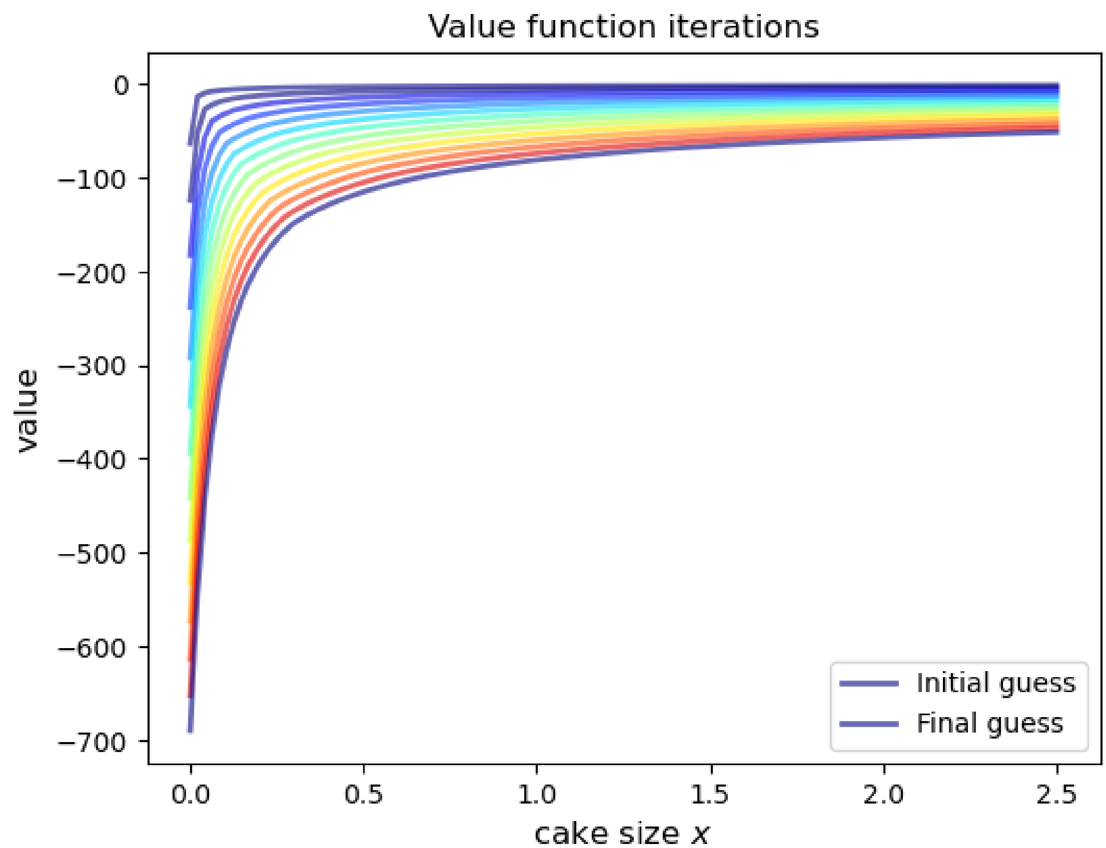
Let's start by creating a model using the default parameterization.

```
model = create_cake_eating_model()
β, γ, x_grid = model
```

Now let's see iteration of the value function in action.

We start from guess v given by $v(x) = u(x)$ for every x grid point.

```
v = u(x_grid, γ)  # Initial guess
n = 12              # Number of iterations
fig, ax = plt.subplots()

# Initial plot
ax.plot(x_grid, v, color=plt.cm.jet(0),
        lw=2, alpha=0.6, label='Initial guess')

# Iterate
for i in range(n):
    v = T(v, model)  # Apply the Bellman operator
    ax.plot(x_grid, v, color=plt.cm.jet(i / n), lw=2, alpha=0.6)

# One last update and plot
v = T(v, model)
ax.plot(x_grid, v, color=plt.cm.jet(0),
        lw=2, alpha=0.6, label='Final guess')

ax.legend()
ax.set_ylabel('value', fontsize=12)
ax.set_xlabel('cake size $x$', fontsize=12)
ax.set_title('Value function iterations')
plt.show()
```



To iterate more systematically, we introduce a wrapper function called `compute_value_function`.

It's task is to iterate using T until some convergence conditions are satisfied.

```
def compute_value_function(
        model: Model,
        tol: float = 1e-4,
        max_iter: int = 1_000,
        verbose: bool = True,
        print_skip: int = 25
    ):

    # Set up loop
    v = np.zeros(len(model.x_grid)) # Initial guess
    i = 0
    error = tol + 1

    while i < max_iter and error > tol:
        v_new = T(v, model)

        error = np.max(np.abs(v - v_new))
        i += 1

        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")

        v = v_new
```

(continues on next page)

(continued from previous page)

```
    if error > tol:
        print("Failed to converge!")
    elif verbose:
        print(f"\nConverged in {i} iterations.")

    return v_new
```

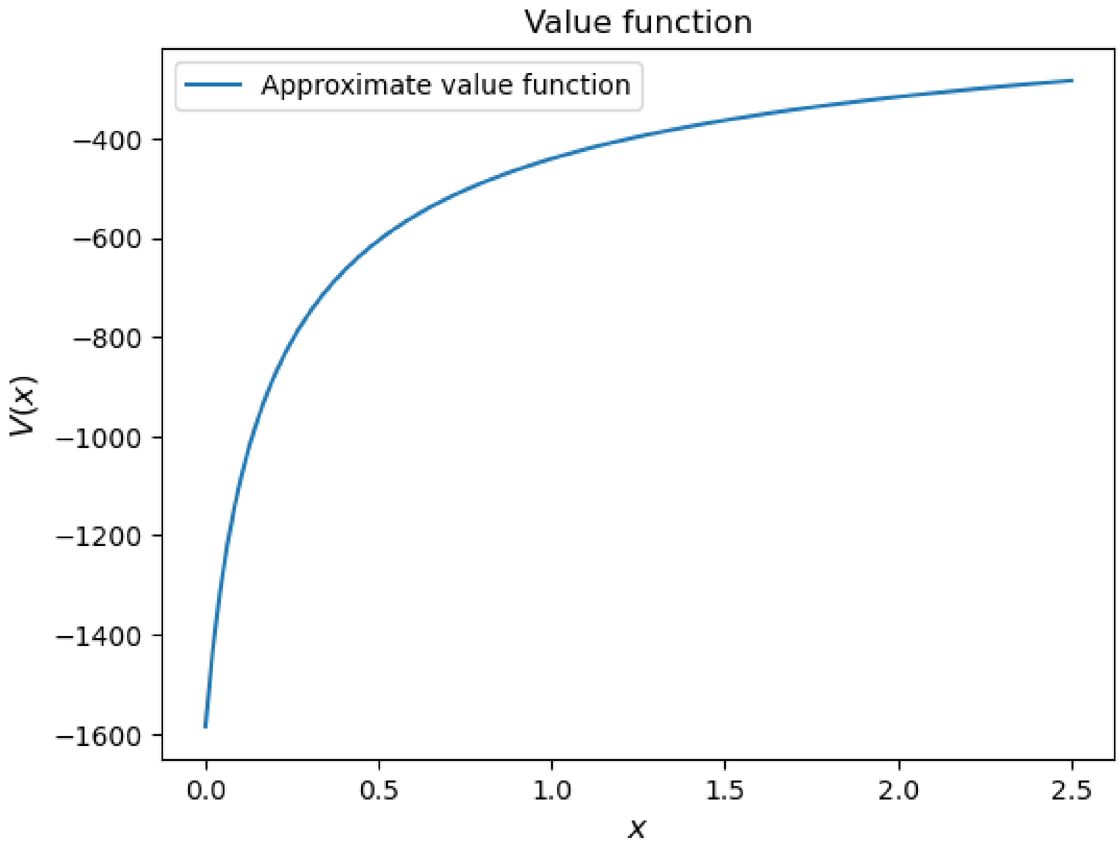
Now let's call it, noting that it takes a little while to run.

```
v = compute_value_function(model)
```

```
Error at iteration 25 is 23.80037552320357.
Error at iteration 50 is 8.577577198550443.
Error at iteration 75 is 3.091330660953872.
Error at iteration 100 is 1.1141054209301728.
Error at iteration 125 is 0.4015199359369035.
Error at iteration 150 is 0.1447064666647293.
Error at iteration 175 is 0.05215173549413521.
Error at iteration 200 is 0.018795314250610318.
Error at iteration 225 is 0.006773769548317432.
Error at iteration 250 is 0.002441244306055233.
Error at iteration 275 is 0.0008798164330414693.
Error at iteration 300 is 0.0003170829540977138.
Error at iteration 325 is 0.00011427565573285392.

Converged in 329 iterations.
```

Now we can plot and see what the converged value function looks like.

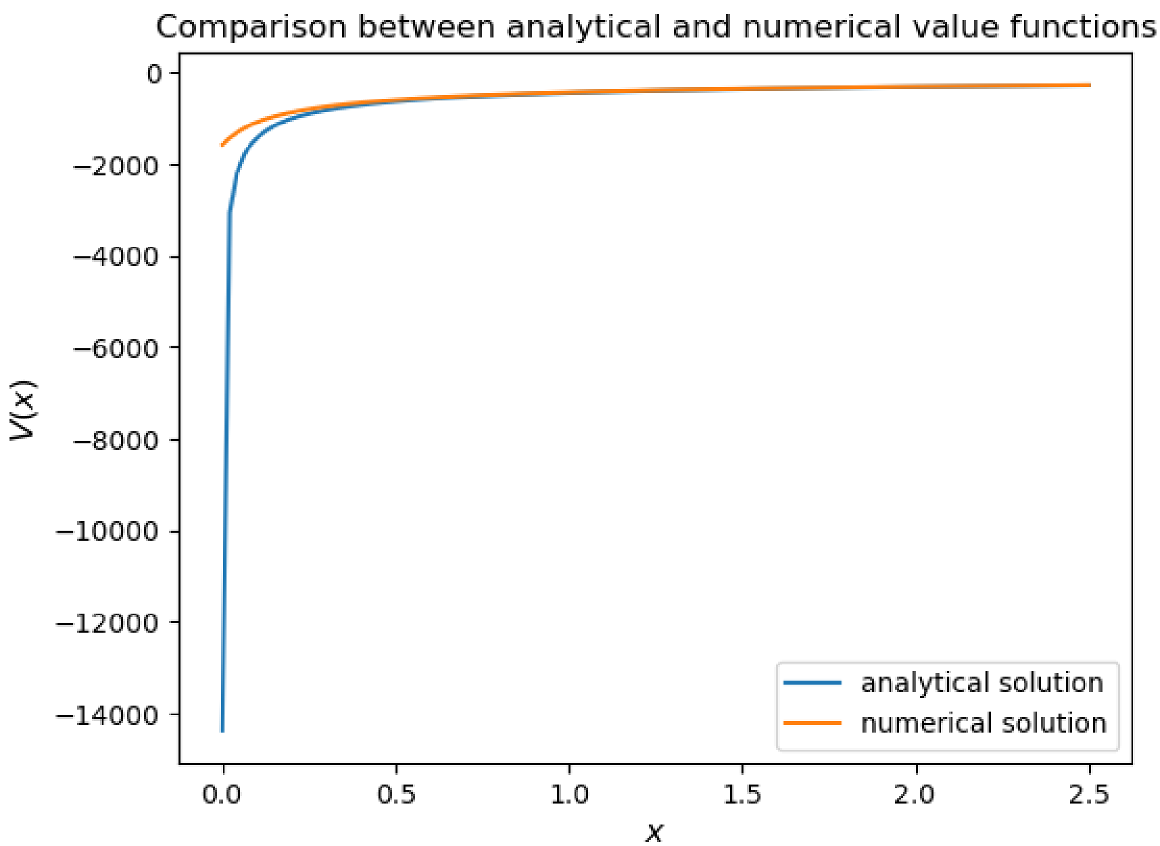
```
fig, ax = plt.subplots()

ax.plot(x_grid, v, label='Approximate value function')
ax.set_ylabel('$V(x)$', fontsize=12)
ax.set_xlabel('$x$', fontsize=12)
ax.set_title('Value function')
ax.legend()
plt.show()
```



Next let's compare it to the analytical solution.

```
v_analytical = v_star(x_grid, β, γ)
```

```
fig, ax = plt.subplots()

ax.plot(x_grid, v_analytical, label='analytical solution')
ax.plot(x_grid, v, label='numerical solution')
ax.set_ylabel('$V(x)$', fontsize=12)
ax.set_xlabel('$x$', fontsize=12)
ax.legend()
ax.set_title('Comparison between analytical and numerical value functions')
plt.show()
```



The quality of approximation is reasonably good for large x, but less so near the lower boundary.

The reason is that the utility function and hence value function is very steep near the lower boundary, and hence hard to approximate.

Note

One way to fix this issue is to use a nonlinear grid, with more points in the neighborhood of zero.

Instead of pursuing this idea, however, we will turn our attention to working with policy functions.

We will see that value function iteration can be avoided by iterating on a guess of the policy function instead.

The policy function has less curvature and hence is easier to interpolate than the value function.

These ideas will be explored over the next few lectures.

54.3.4 Policy Function

Let's try computing the optimal policy.

In *Optimal Savings I: Cake Eating*, the optimal consumption policy was shown to be

$$\sigma^*(x) = \left(1 - \beta^{1/\gamma}\right) x$$

Let's see if our numerical results lead to something similar.

Our numerical strategy will be to compute, for any given v, the policy

$$\sigma(x) = \arg\max_{0 \leq c \leq x} \{u(c) + \beta v(x - c)\}$$

This policy is called the v-**greedy policy**.

In practice we will compute σ on a grid of x points and then interpolate.

For v we will use the approximation of the value function we obtained above.

Here's the function:

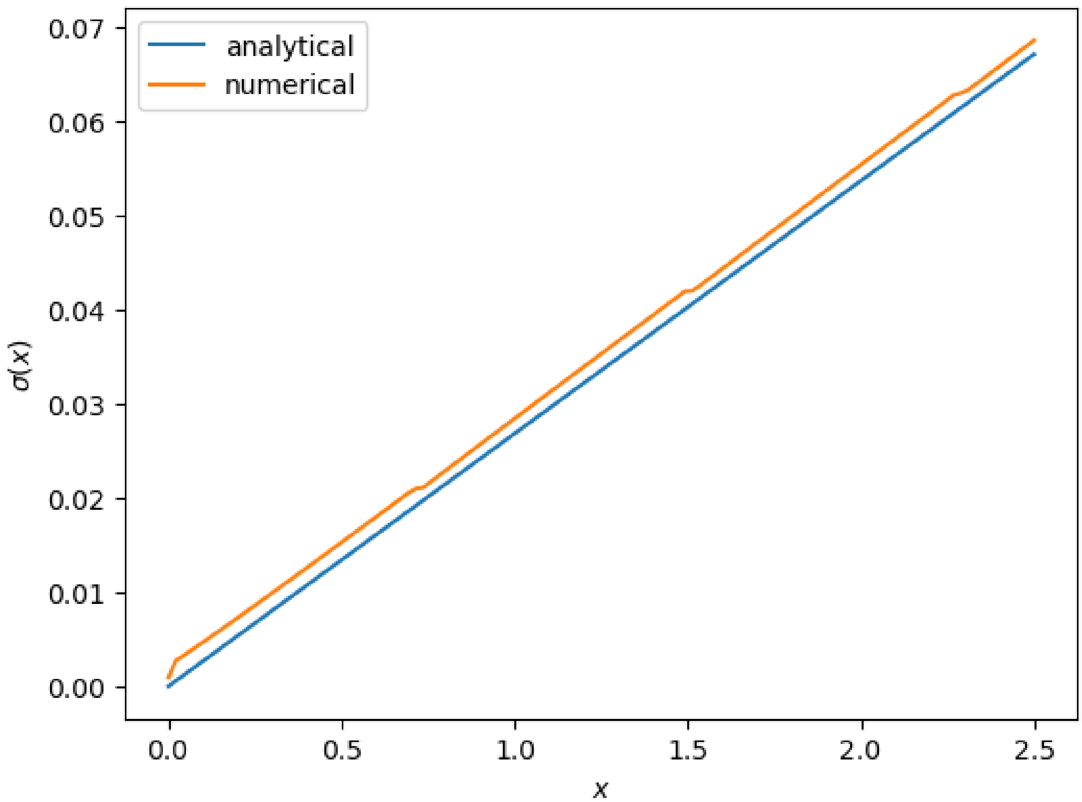
```
def get_greedy(
        v: np.ndarray,          # current guess of the value function
        model: Model            # instance of cake eating model
    ):
    " Compute the v-greedy policy on x_grid."

    σ = np.empty_like(v)

    for i, x in enumerate(model.x_grid):
        # Maximize RHS of Bellman equation at state x
        σ[i], _ = maximize(lambda c: B(x, c, v, model), x)

    return σ
```

Now let's pass the approximate value function and compute optimal consumption:

```
σ = get_greedy(v, model)
```

Let's plot this next to the true analytical solution

```
c_analytical = c_star(model.x_grid, model.β, model.γ)

fig, ax = plt.subplots()

ax.plot(model.x_grid, c_analytical, label='analytical')
ax.plot(model.x_grid, σ, label='numerical')
ax.set_ylabel(r'$\sigma(x)$')
ax.set_xlabel('$x$')
ax.legend()

plt.show()
```



The fit is reasonable but not perfect.

We can improve it by increasing the grid size or reducing the error tolerance in the value function iteration routine.

However, both changes will lead to a longer compute time.

Another possibility is to use an alternative algorithm, which offers the possibility of faster compute time and, at the same time, more accuracy.

We explore this in *Optimal Savings IV: Time Iteration*.

54.4 Exercises

Exercise 54.4.1

Try the following modification of the problem.

Instead of the cake size changing according to $x_{t+1} = x_t - c_t$, let it change according to

$$x_{t+1} = (x_t - c_t)^\alpha$$

where α is a parameter satisfying $0 < \alpha < 1$.

(We will see this kind of update rule when we study optimal growth models.)

Make the required changes to value function iteration code and plot the value and policy functions.

Try to reuse as much code as possible.

Solution

We need to create an extended version of our model and state-action value function.

We'll create a new `NamedTuple` for the extended cake model and a helper function.

```
# Create extended cake model data structure
class ExtendedModel(NamedTuple):
    β: float
    γ: float
    α: float
    x_grid: np.ndarray

def create_extended_model(β=0.96,           # discount factor
                          γ=1.5,            # degree of relative risk aversion
                          α=0.4,            # productivity parameter
                          x_grid_min=1e-3,  # exclude zero for numerical stability
                          x_grid_max=2.5,   # size of cake
                          x_grid_size=120):
    """
    Creates an instance of the extended cake eating model.
    """
    x_grid = np.linspace(x_grid_min, x_grid_max, x_grid_size)
    return ExtendedModel(β, γ, α, x_grid)

def extended_B(c, x, v, model):
    """
    Right hand side of the Bellman equation for the extended cake model given x
↪and c.

    """
    β, γ, α, x_grid = model
    vf = lambda x: np.interp(x, x_grid, v)
    return u(c, γ) + β * vf((x - c)**α)
```

We also need a modified Bellman operator:

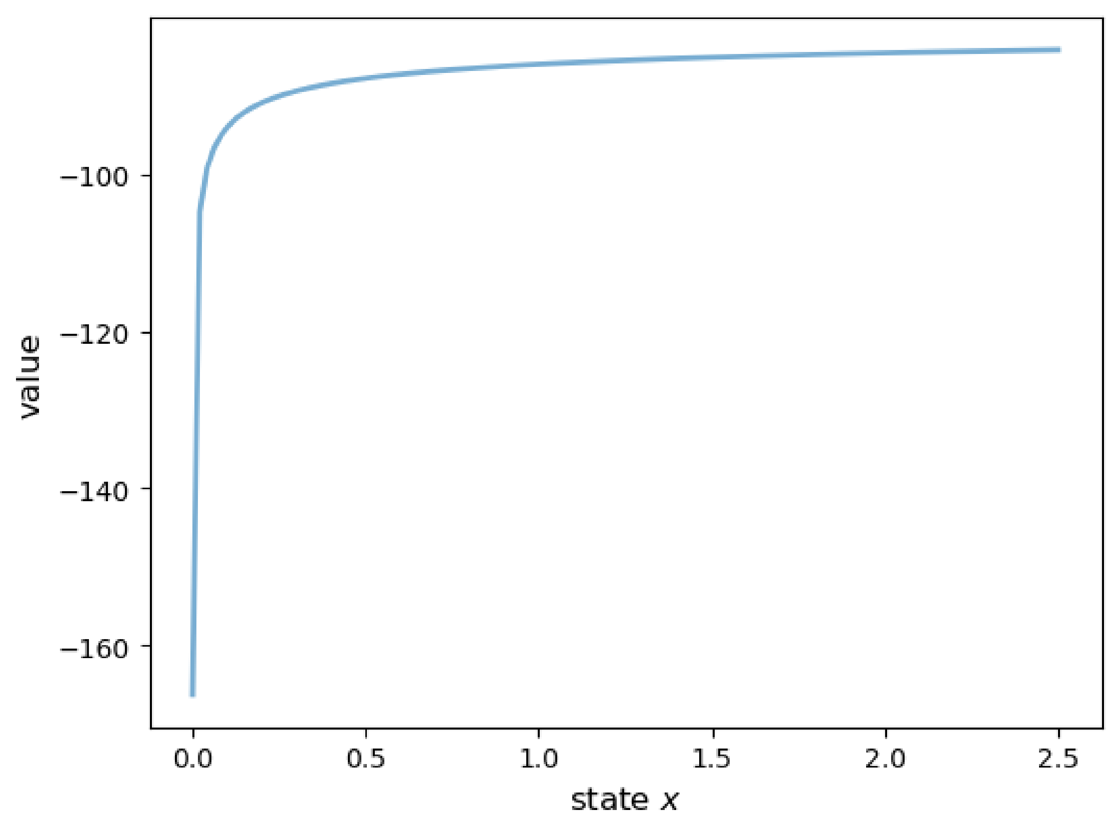
```
def extended_T(v, model):
    " The Bellman operator for the extended cake model. "

    v_new = np.empty_like(v)
    for i, x in enumerate(model.x_grid):
        _, v_new[i] = maximize(lambda c: extended_B(c, x, v, model), x)
    return v_new
```

Now create the model:

```
model = create_extended_model()
```

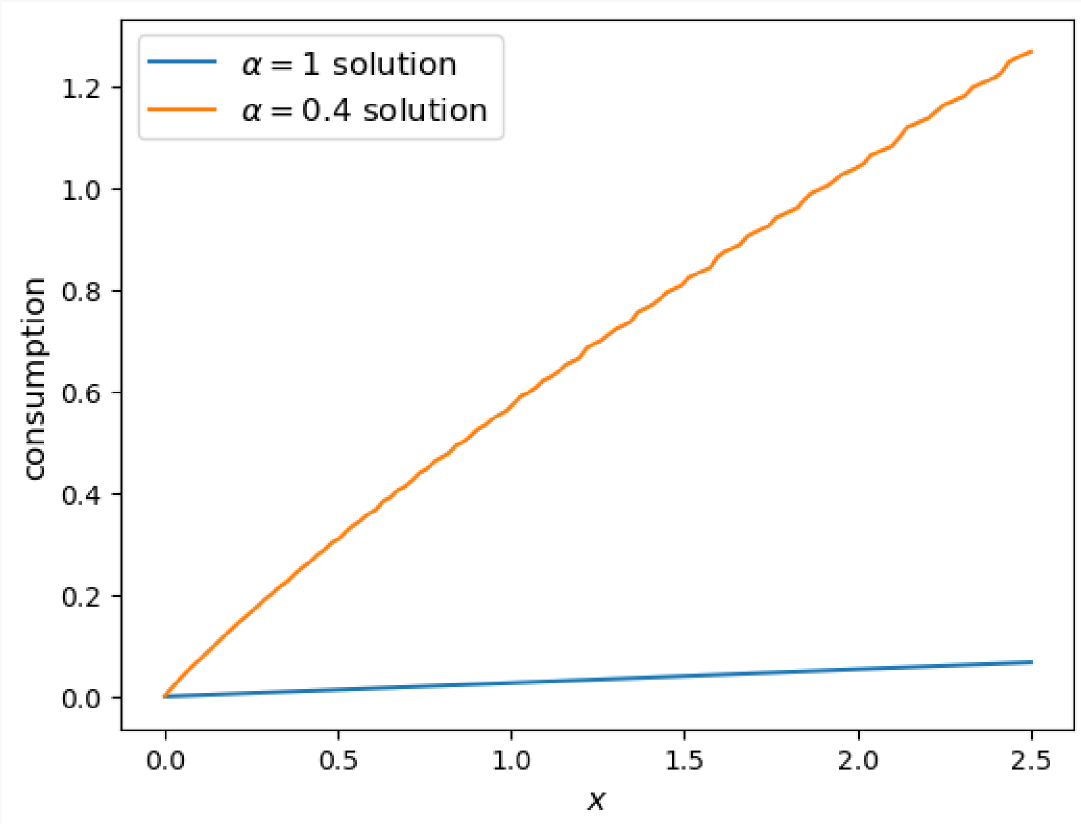
Here's a function to compute the value function.

```
def compute_value_function_extended(model,
                                    tol=1e-4,
                                    max_iter=1000,
                                    verbose=True,
                                    print_skip=25):
    """
    Compute value function for extended cake model.
    """
    v = np.zeros(len(model.x_grid))
    i = 0
    error = tol + 1

    while i < max_iter and error > tol:
        v_new = extended_T(v, model)
        error = np.max(np.abs(v - v_new))
        i += 1
        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")
        v = v_new

    if error > tol:
        print("Failed to converge!")
    elif verbose:
        print(f"\nConverged in {i} iterations.")

    return v_new

v = compute_value_function_extended(model, verbose=False)

fig, ax = plt.subplots()

ax.plot(model.x_grid, v, lw=2, alpha=0.6)
ax.set_ylabel('value', fontsize=12)
ax.set_xlabel('state $x$', fontsize=12)

plt.show()
```



Here's the computed policy, combined with the solution we derived above for the standard cake eating case $\alpha = 1$.

```
def extended_get_greedy(model, v):
    """
    The optimal policy function for the extended cake model.
    """
    σ = np.empty_like(v)

    for i, x in enumerate(model.x_grid):
        # Maximize extended_B with respect to c over [0, x]
        σ[i], _ = maximize(lambda c: extended_B(c, x, v, model), x)

    return σ

σ = extended_get_greedy(model, v)

# Get the baseline model for comparison
baseline_model = create_cake_eating_model()
c_analytical = c_star(baseline_model.x_grid, baseline_model.β, baseline_model.γ)

fig, ax = plt.subplots()

ax.plot(baseline_model.x_grid, c_analytical, label=r'$\alpha=1$ solution')
ax.plot(model.x_grid, σ, label=fr'$\alpha={model.α}$ solution')

ax.set_ylabel('consumption', fontsize=12)
ax.set_xlabel('$x$', fontsize=12)

ax.legend(fontsize=12)

plt.show()
```



Consumption is higher when $\alpha < 1$ because, at least for large x, the return to savings is lower.

CHAPTER

FIFTYFIVE

OPTIMAL SAVINGS III: STOCHASTIC RETURNS

Contents

55.1 Overview

In this lecture, we continue our study of optimal savings problems, building on *Optimal Savings I: Cake Eating* and *Optimal Savings II: Numerical Cake Eating*.

The key difference from the previous lectures is that wealth now evolves stochastically.

We can think of wealth as a harvest that regrows if we save some seeds.

Specifically, if we save and invest part of today's harvest x_t, it grows into next period's harvest x_{t+1} according to a stochastic production process.

The extensions in this lecture introduce several new elements:

- nonlinear returns to saving, through a production function, and
- stochastic returns, due to shocks to production.

Despite these additions, the model remains relatively tractable.

As a first pass, we will solve the model using dynamic programming and value function iteration (VFI).

Note

In later lectures we'll explore more efficient methods for this class of problems.

At the same time, VFI is foundational and globally convergent.

Hence we want to be sure we can use this method too.

More information on this savings problem can be found in

- [Ljungqvist and Sargent, 2018], Section 3.1
- EDTC, Chapter 1
- [Sundaram, 1996], Chapter 12

Let's start with some imports:

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.interpolate import interp1d
from scipy.optimize import minimize_scalar
from typing import NamedTuple, Callable
```

55.2 The Model

Here we described the new model and the optimization problem.

55.2.1 Setup

Consider an agent who owns an amount $x_t \in \mathbb{R}_+ := [0, \infty)$ of a consumption good at time t.

This output can either be consumed or saved and used for production.

Production is stochastic, in that it also depends on a shock ξ_{t+1} realized at the end of the current period.

Next period output is

$$x_{t+1} := f(s_t)\xi_{t+1}$$

where $f: \mathbb{R}_+ \to \mathbb{R}_+$ is the **production function** and

$$s_t = x_t - c_t \tag{55.1}$$

is **current savings**.

and all variables are required to be nonnegative.

In what follows,

- The sequence $\{\xi_t\}$ is assumed to be IID.
- The common distribution of each ξ_t will be denoted by ϕ.
- The production function f is assumed to be increasing and continuous.

55.2.2 Optimization

Taking x_0 as given, the agent wishes to maximize

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t u(c_t) \tag{55.2}$$

subject to

$$x_{t+1} = f(x_t - c_t)\xi_{t+1} \quad \text{and} \quad 0 \leq c_t \leq x_t \quad \text{for all } t \tag{55.3}$$

where

- u is a bounded, continuous and strictly increasing utility function and
- $\beta \in (0, 1)$ is a discount factor.

In summary, the agent's aim is to select a path $c_0, c_1, c_2, \ldots$ for consumption that is

1. nonnegative,
2. feasible,
3. optimal, in the sense that it maximizes (55.2) relative to all other feasible consumption sequences, and
4. **adapted**, in the sense that the current action c_t depends only on current and historical outcomes, not on future outcomes such as ξ_{t+1}.

In the present context

- x_t is called the **state** variable — it summarizes the "state of the world" at the start of each period.
- c_t is called the **control** variable — a value chosen by the agent each period after observing the state.

55.2.3 Optimal Policies

Let us look at **policy functions**, each one of which is a map σ from the current state x_t into a current action c_t.

> **Note**
>
> These kinds of policies are called Markov policies (or stationary Markov policies).
>
> For this dynamic program, the optimal policy is always a Markov policy (see, e.g., DP1).
>
> In essence, the current state x_t provides a sufficient statistic for the history in terms of making an optimal decision today.

In what follows, we will call σ a **feasible consumption policy** if it satisfies

$$0 \leq \sigma(x) \leq x \quad \text{for all} \quad x \in \mathbb{R}_+ \tag{55.4}$$

In other words, a feasible policy is a policy function that respects the resource constraint.

The set of all feasible consumption policies will be denoted by Σ.

Each $\sigma \in \Sigma$ determines a Markov dynamics for output $\{x_t\}$ via

$$x_{t+1} = f(x_t - \sigma(x_t))\xi_{t+1}, \quad x_0 \text{ given} \tag{55.5}$$

This is the time path for output when we choose and stick with the policy σ.

We insert this process into the objective function to get

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t u(c_t) = \mathbb{E} \sum_{t=0}^{\infty} \beta^t u(\sigma(x_t)) \tag{55.6}$$

This is the total expected present value of following policy σ forever, given initial income x_0.

The aim is to select a policy that makes this number as large as possible.

The next section covers these ideas more formally.

55.2.4 Optimality

The lifetime value v_σ associated with a given policy σ is the mapping defined by

$$v_\sigma(x) = \mathbb{E} \sum_{t=0}^{\infty} \beta^t u(\sigma(x_t)) \tag{55.7}$$

when $\{x_t\}$ is given by (55.5) with $x_0 = x$.

In other words, it is the lifetime value of following policy σ forever, starting at initial condition x.

The **value function** is then defined as

$$v^*(x) := \sup_{\sigma \in \Sigma} \; v_\sigma(x) \tag{55.8}$$

The value function gives the maximal value that can be obtained from state x, after considering all feasible policies.

A policy $\sigma \in \Sigma$ is called **optimal** if $v_\sigma(x) = v^*(x)$ for all $x \in \mathbb{R}_+$.

55.2.5 The Bellman Equation

The following equation is called the **Bellman equation** associated with this dynamic programming problem.

$$v(x) = \max_{0 \leq c \leq x} \left\{ u(c) + \beta \int v(f(x-c)z)\phi(dz) \right\} \qquad (x \in \mathbb{R}_+) \tag{55.9}$$

This is a *functional equation in* v, in the sense that a given v can either satisfy it or not satisfy it.

The term $\int v(f(x-c)z)\phi(dz)$ can be understood as the expected next period value when

- v is used to measure value
- the state is x
- consumption is set to c

As shown in DP1 and a range of other texts, the value function v^* satisfies the Bellman equation.

In other words, (55.9) holds when $v = v^*$.

The intuition is that maximal value from a given state can be obtained by optimally trading off

- current reward from a given action, vs
- expected discounted future value of the state resulting from that action

The Bellman equation is important because it

1. gives us more information about the value function and
2. suggests a way of computing the value function, which we discuss below.

55.2.6 Greedy Policies

The value function can be used to compute optimal policies.

Given a continuous function v on $\mathbb{R}_+$, we say that $\sigma \in \Sigma$ is v-**greedy** if

$$\sigma(x) \in \arg\max_{0 \leq c \leq x} \left\{ u(c) + \beta \int v(f(x-c)z)\phi(dz) \right\} \tag{55.10}$$

for every $x \in \mathbb{R}_+$.

In other words, $\sigma \in \Sigma$ is v-greedy if it optimally trades off current and future rewards when v is taken to be the value function.

In our setting, we have the following key result

Theorem 55.2.1

A feasible consumption policy is optimal if and only if it is v^*-greedy.

See, for example, Theorem 10.1.11 of EDTC.

Hence, once we have a good approximation to v^*, we can compute the (approximately) optimal policy by computing the corresponding greedy policy.

The advantage is that we are now solving a much lower dimensional optimization problem.

55.2.7 The Bellman Operator

How, then, should we compute the value function?

One way is to use the so-called **Bellman operator**.

(The term **operator** is usually reserved for functions that send functions into functions!)

The Bellman operator is denoted by T and defined by

$$Tv(x) := \max_{0 \leq c \leq x} \left\{ u(c) + \beta \int v(f(x-c)z)\phi(dz) \right\} \qquad (x \in \mathbb{R}_+) \tag{55.11}$$

In other words, T sends the function v into the new function Tv defined by (55.11).

By construction, the set of solutions to the Bellman equation (55.9) *exactly coincides with* the set of fixed points of T.

For example, if $Tv = v$, then, for any $x \geq 0$,

$$v(x) = Tv(x) = \max_{0 \leq c \leq x} \left\{ u(c) + \beta \int v(f(x-c)z)\phi(dz) \right\}$$

which says precisely that v is a solution to the Bellman equation.

It follows that v^* is a fixed point of T.

55.2.8 Review of Theoretical Results

One can also show that T is a contraction mapping on the set of continuous bounded functions on $\mathbb{R}_+$ under the supremum distance

$$\rho(g, h) = \sup_{x \geq 0} |g(x) - h(x)|$$

See EDTC, Lemma 10.1.18.

Hence, it has exactly one fixed point in this set, which we know is equal to the value function.

It follows that

- The value function v^* is bounded and continuous.
- Starting from any bounded and continuous v, the sequence $v, Tv, T^2v, \ldots$ generated by iteratively applying T converges uniformly to v^*.

This iterative method is called **value function iteration**.

We also know that a feasible policy is optimal if and only if it is v^*-greedy.

It's not too hard to show that a v^*-greedy policy exists.

Hence, at least one optimal policy exists.

Our problem now is how to compute it.

55.2.9 Unbounded Utility

The results stated above assume that u is bounded.

In practice economists often work with unbounded utility functions — and so will we.

In the unbounded setting, various optimality theories exist.

Nevertheless, their main conclusions are usually in line with those stated for the bounded case just above (as long as we drop the word "bounded").

Note

Consult the following references for more on the unbounded case:

- The lecture *The Income Fluctuation Problem V: Stochastic Returns on Assets*.
- Section 12.2 of EDTC.

55.3 Computation

Let's now look at computing the value function and the optimal policy.

Our implementation in this lecture will focus on clarity and flexibility.

(In subsequent lectures we will focus on efficiency and speed.)

We will use fitted value function iteration, which was already described in *Optimal Savings II: Numerical Cake Eating*.

55.3.1 Scalar Maximization

To maximize the right hand side of the Bellman equation (55.9), we are going to use the `minimize_scalar` routine from SciPy.

To keep the interface tidy, we will wrap `minimize_scalar` in an outer function as follows:

```
def maximize(g, upper_bound):
    """
    Maximize the function g over the interval [0, upper_bound].

    We use the fact that the maximizer of g on any interval is
    also the minimizer of -g.

    """

    objective = lambda x: -g(x)
```

(continues on next page)

(continued from previous page)

```
    bounds = (0, upper_bound)
    result = minimize_scalar(objective, bounds=bounds, method='bounded')
    maximizer, maximum = result.x, -result.fun
    return maximizer, maximum
```

55.3.2 Model

We will assume for now that ϕ is the distribution of $\xi := \exp(\mu + \nu\zeta)$ where

- ζ is standard normal,
- μ is a shock location parameter and
- ν is a shock scale parameter.

We will store the primitives of the model in a `NamedTuple`.

```
class Model(NamedTuple):
    u: Callable        # utility function
    f: Callable        # production function
    β: float           # discount factor
    μ: float           # shock location parameter
    ν: float           # shock scale parameter
    x_grid: np.ndarray # state grid
    shocks: np.ndarray # shock draws


def create_model(
        u: Callable,
        f: Callable,
        β: float = 0.96,
        μ: float = 0.0,
        ν: float = 0.1,
        grid_max: float = 4.0,
        grid_size: int = 120,
        shock_size: int = 250,
        seed: int = 1234
    ) -> Model:
    """
    Creates an instance of the optimal savings model.
    """
    # Set up grid
    x_grid = np.linspace(1e-4, grid_max, grid_size)

    # Store shocks (with a seed, so results are reproducible)
    np.random.seed(seed)
    shocks = np.exp(μ + ν * np.random.randn(shock_size))

    return Model(u, f, β, μ, ν, x_grid, shocks)
```

We set up the right-hand side of the Bellman equation

$$B(x, c, v) := u(c) + \beta \int v(f(x - c)z)\phi(dz)$$

```
def B(
        x: float,               # State
        c: float,               # Action
        v_array: np.ndarray,    # Array representing a guess of the value fn
        model: Model            # An instance of Model containing parameters
    ):

    u, f, β, μ, ν, x_grid, shocks = model
    v = interp1d(x_grid, v_array)

    return u(c) + β * np.mean(v(f(x - c) * shocks))
```

In the second last line we are using linear interpolation.

In the last line, the expectation in (55.11) is computed via Monte Carlo, using the approximation

$$\int v(f(x-c)z)\phi(dz) \approx \frac{1}{n}\sum_{i=1}^{n} v(f(x-c)\xi_i)$$

where $\{\xi_i\}_{i=1}^n$ are IID draws from ϕ.

Monte Carlo is not always the most efficient way to compute integrals numerically but it does have some theoretical advantages in the present setting.

(For example, it preserves the contraction mapping property of the Bellman operator — see, e.g., [Pál and Stachurski, 2013].)

55.3.3 The Bellman Operator

The next function implements the Bellman operator.

```
def T(v: np.ndarray, model: Model) -> tuple[np.ndarray, np.ndarray]:
    """
    The Bellman operator.  Updates the guess of the value function.

      * model is an instance of Model
      * v is an array representing a guess of the value function

    """
    x_grid = model.x_grid
    v_new = np.empty_like(v)

    for i in range(len(x_grid)):
        x = x_grid[i]
        _, v_max = maximize(lambda c: B(x, c, v, model), x)
        v_new[i] = v_max

    return v_new
```

Here's the function:

```
def get_greedy(
        v: np.ndarray,          # current guess of the value function
        model: Model            # instance of optimal savings model
    ):
    " Compute the v-greedy policy on x_grid."
```

(continues on next page)

(continued from previous page)

```
    σ = np.empty_like(v)

    for i, x in enumerate(model.x_grid):
        # Maximize RHS of Bellman equation at state x
        σ[i], _ = maximize(lambda c: B(x, c, v, model), x)

    return σ
```

55.3.4 An Example

Let's suppose now that

$$f(x-c) = (x-c)^{\alpha} \quad \text{and} \quad u(c) = \ln c$$

For this particular problem, an exact analytical solution is available (see [Ljungqvist and Sargent, 2018], section 3.1.2), with

$$v^*(x) = \frac{\ln(1-\alpha\beta)}{1-\beta} + \frac{(\mu + \alpha\ln(\alpha\beta))}{1-\alpha}\left[\frac{1}{1-\beta} - \frac{1}{1-\alpha\beta}\right] + \frac{1}{1-\alpha\beta}\ln x \tag{55.12}$$

and optimal consumption policy

$$\sigma^*(x) = (1-\alpha\beta)x$$

It is valuable to have these closed-form solutions because it lets us check whether our code works for this particular case.

In Python, the functions above can be expressed as:

```
def v_star(x, α, β, μ):
    """
    True value function
    """
    c1 = np.log(1 - α * β) / (1 - β)
    c2 = (μ + α * np.log(α * β)) / (1 - α)
    c3 = 1 / (1 - β)
    c4 = 1 / (1 - α * β)
    return c1 + c2 * (c3 - c4) + c4 * np.log(x)

def σ_star(x, α, β):
    """
    True optimal policy
    """
    return (1 - α * β) * x
```

Next let's create an instance of the model with the above primitives and assign it to the variable `model`.

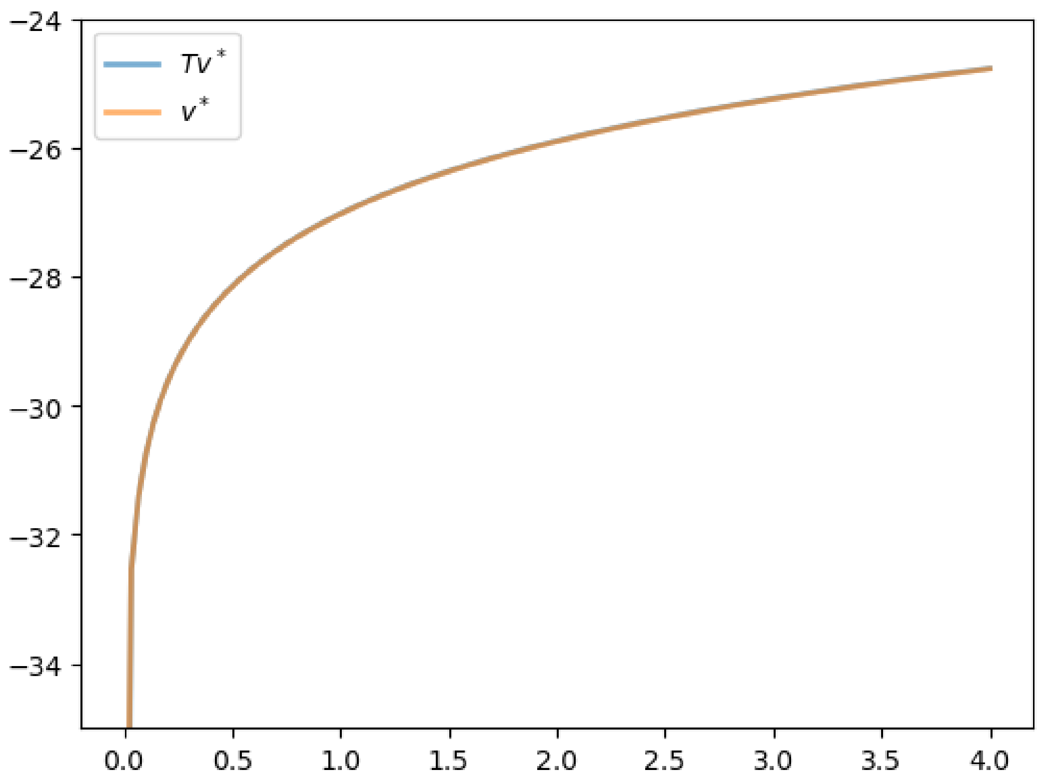
```
α = 0.4
def fcd(s):
    return s**α

model = create_model(u=np.log, f=fcd)
```

Now let's see what happens when we apply our Bellman operator to the exact solution v^* in this case.

In theory, since v^* is a fixed point, the resulting function should again be v^*.

In practice, we expect some small numerical error.

```
x_grid = model.x_grid

v_init = v_star(x_grid, α, model.β, model.μ)     # Start at the solution
v = T(v_init, model)                 # Apply T once

fig, ax = plt.subplots()
ax.set_ylim(-35, -24)
ax.plot(x_grid, v, lw=2, alpha=0.6, label='$Tv^*$')
ax.plot(x_grid, v_init, lw=2, alpha=0.6, label='$v^*$')
ax.legend()
plt.show()
```



The two functions are essentially indistinguishable, so we are off to a good start.

Now let's have a look at iterating with the Bellman operator, starting from an arbitrary initial condition.

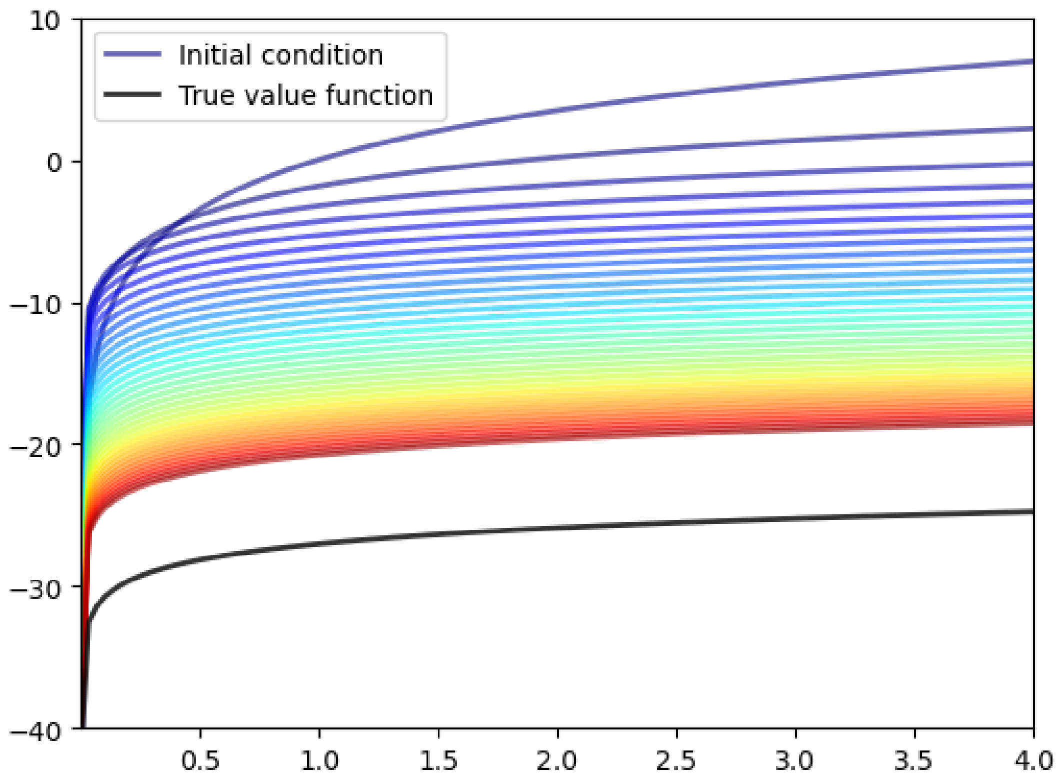
The initial condition we'll start with is, somewhat arbitrarily, $v(x) = 5\ln(x)$.

```
v = 5 * np.log(x_grid)   # An initial condition
n = 35

fig, ax = plt.subplots()

ax.plot(x_grid, v, color=plt.cm.jet(0),
        lw=2, alpha=0.6, label='Initial condition')

for i in range(n):
    v = T(v, model)   # Apply the Bellman operator
    ax.plot(x_grid, v, color=plt.cm.jet(i / n), lw=2, alpha=0.6)
```

(continues on next page)

(continued from previous page)

```
ax.plot(x_grid, v_star(x_grid, α, model.β, model.μ), 'k-', lw=2,
        alpha=0.8, label='True value function')

ax.legend()
ax.set(ylim=(-40, 10), xlim=(np.min(x_grid), np.max(x_grid)))
plt.show()
```



The figure shows

1. the first 36 functions generated by the fitted value function iteration algorithm, with hotter colors given to higher iterates
2. the true value function v^* drawn in black

The sequence of iterates converges towards v^*.

We are clearly getting closer.

55.3.5 Iterating to Convergence

We can write a function that iterates until the difference is below a particular tolerance level.

```
def solve_model(
        model: Model,           # instance of optimal savings model
        tol: float = 1e-4,      # convergence tolerance
        max_iter: int = 1000,   # maximum iterations
        verbose: bool = True,   # print iteration info
        print_skip: int = 25    # iterations between prints
```

(continues on next page)

(continued from previous page)

```
    ):
    " Solve by value function iteration. "

    v = model.u(model.x_grid)   # Initial condition
    i = 0
    error = tol + 1

    while i < max_iter and error > tol:
        v_new = T(v, model)
        error = np.max(np.abs(v - v_new))
        i += 1
        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")
        v = v_new

    if error > tol:
        print("Failed to converge!")
    elif verbose:
        print(f"\nConverged in {i} iterations.")

    v_greedy = get_greedy(v_new, model)
    return v_greedy, v_new
```

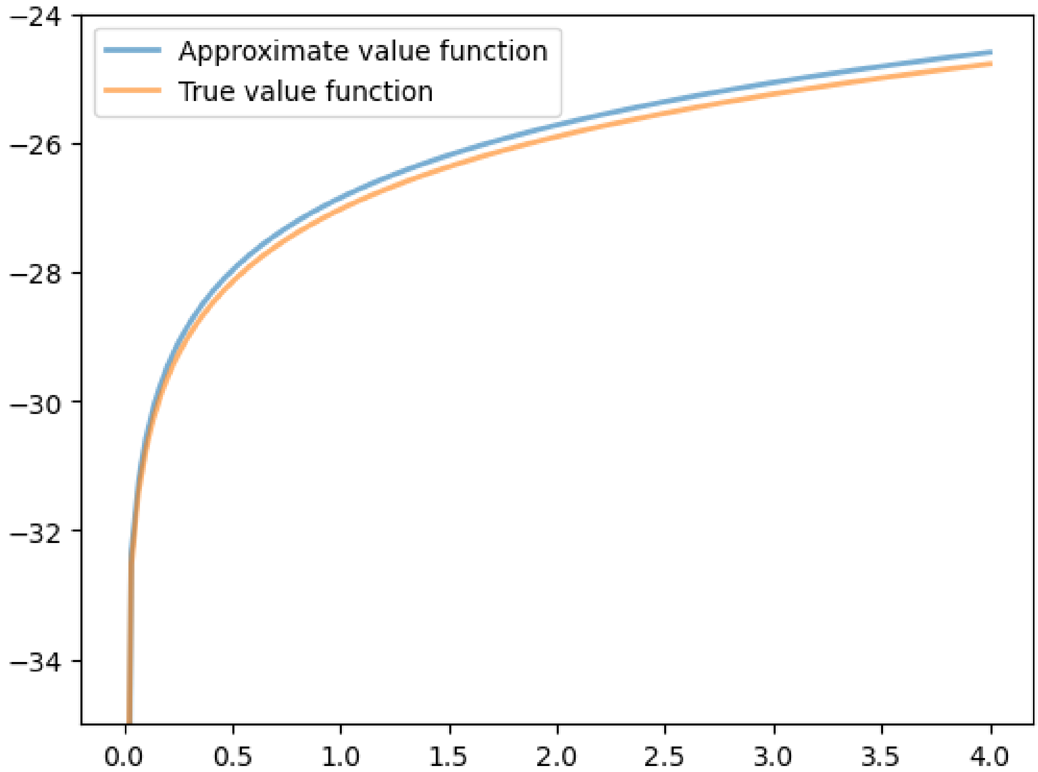
Let's use this function to compute an approximate solution at the defaults.

```
v_greedy, v_solution = solve_model(model)
```

```
Error at iteration 25 is 0.40975776844490497.
Error at iteration 50 is 0.1476753540823772.
Error at iteration 75 is 0.05322171277213883.
Error at iteration 100 is 0.019180930548646558.
Error at iteration 125 is 0.006912744396021964.
Error at iteration 150 is 0.0024913303848137502.
Error at iteration 175 is 0.0008978672913073638.
Error at iteration 200 is 0.00032358842396718046.
Error at iteration 225 is 0.00011662020561686859.

Converged in 229 iterations.
```

Now we check our result by plotting it against the true value:

```
fig, ax = plt.subplots()

ax.plot(x_grid, v_solution, lw=2, alpha=0.6,
        label='Approximate value function')

ax.plot(x_grid, v_star(x_grid, α, model.β, model.μ), lw=2,
        alpha=0.6, label='True value function')

ax.legend()
ax.set_ylim(-35, -24)
plt.show()
```



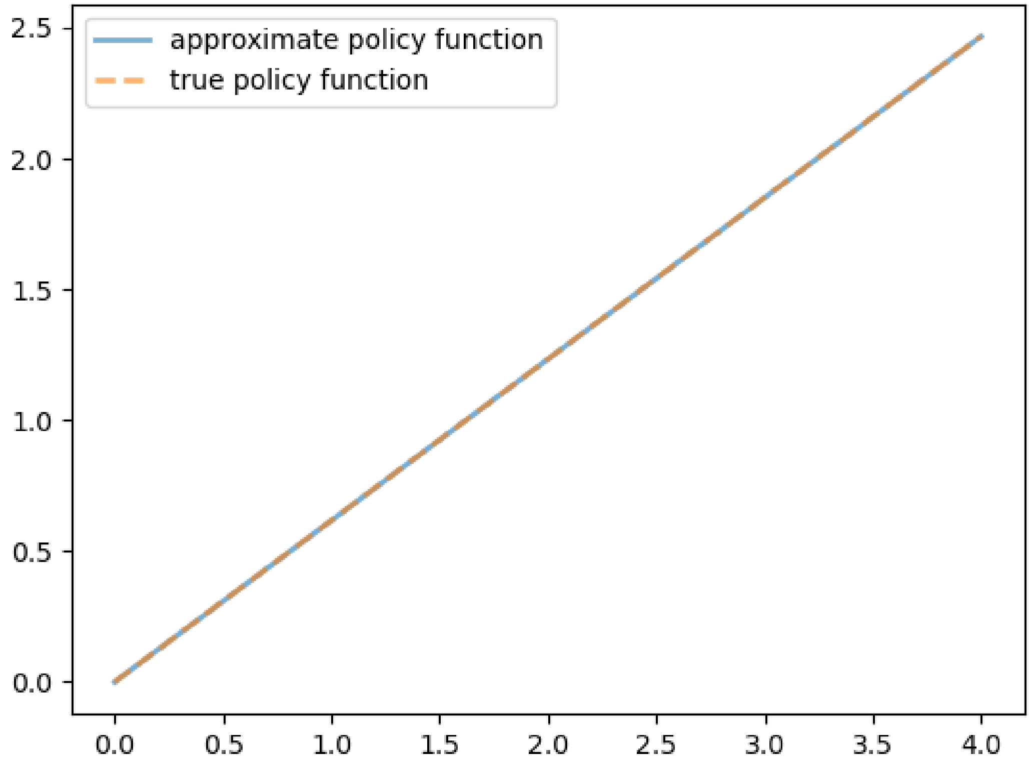
The figure shows that we are pretty much on the money.

55.3.6 The Policy Function

The policy `v_greedy` computed above corresponds to an approximate optimal policy.

The next figure compares it to the exact solution, which, as mentioned above, is $\sigma(x) = (1 - \alpha\beta)x$

```
fig, ax = plt.subplots()

ax.plot(x_grid, v_greedy, lw=2,
        alpha=0.6, label='approximate policy function')

ax.plot(x_grid, σ_star(x_grid, α, model.β), '--',
        lw=2, alpha=0.6, label='true policy function')

ax.legend()
plt.show()
```



The figure shows that we've done a good job in this instance of approximating the true policy.

55.4 Exercises

Exercise 55.4.1

A common choice for utility function in this kind of work is the CRRA specification

$$u(c) = \frac{c^{1-\gamma}}{1-\gamma}$$

Maintaining the other defaults, including the Cobb-Douglas production function, solve the optimal savings model with this utility specification.

Setting $\gamma = 1.5$, compute and plot an estimate of the optimal policy.

Solution

Here we set up the model.

```
γ = 1.5    # Preference parameter

def u_crra(c):
    return (c**(1 - γ) - 1) / (1 - γ)

model = create_model(u=u_crra, f=fcd)
```

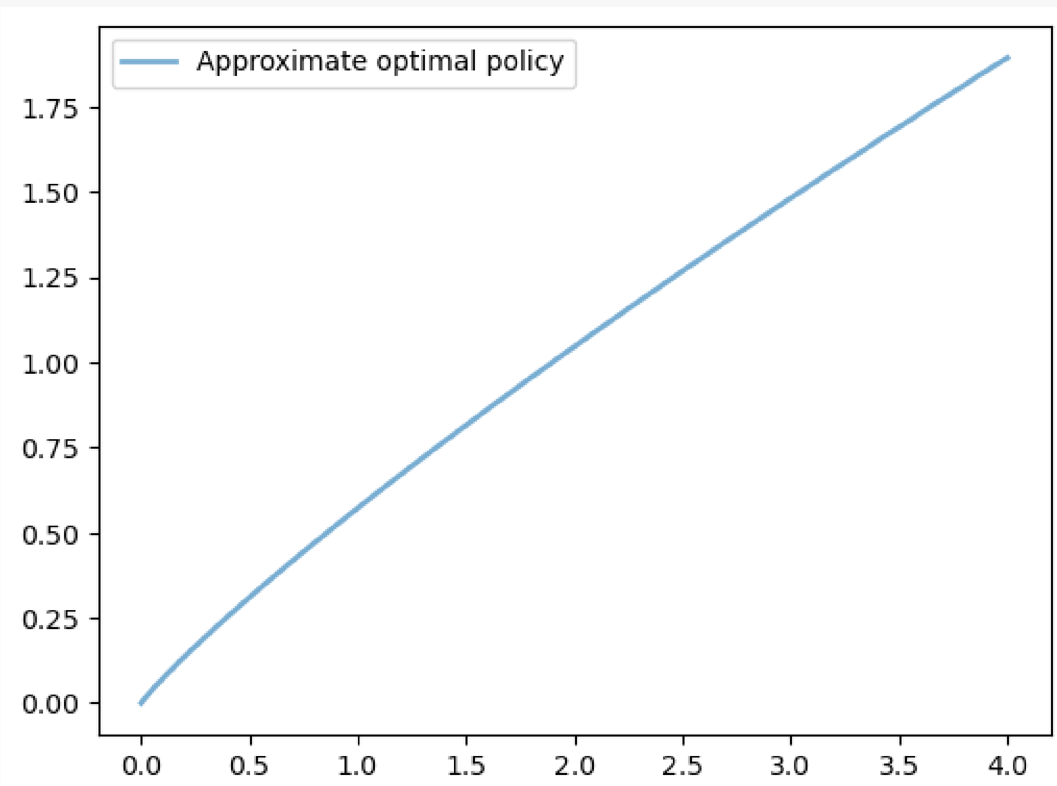
Now let's run it, with a timer.

```
%%time
v_greedy, v_solution = solve_model(model)
```

```
Error at iteration 25 is 0.5528151810315194.
Error at iteration 50 is 0.19923228425591333.
Error at iteration 75 is 0.07180266113797629.
Error at iteration 100 is 0.025877443335843964.
Error at iteration 125 is 0.009326145618956616.
Error at iteration 150 is 0.003361112262041388.
Error at iteration 175 is 0.0012113338242443206.
Error at iteration 200 is 0.0004365607333056687.
Error at iteration 225 is 0.00015733505506432266.

Converged in 237 iterations.
CPU times: user 29.6 s, sys: 6 ms, total: 29.6 s
Wall time: 29.6 s
```

Let's plot the policy function just to see what it looks like:

```
fig, ax = plt.subplots()
ax.plot(x_grid, v_greedy, lw=2,
        alpha=0.6, label='Approximate optimal policy')
ax.legend()
plt.show()
```



CHAPTER

FIFTYSIX

OPTIMAL SAVINGS IV: TIME ITERATION

Contents

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
```

56.1 Overview

In this lecture, we introduce the core idea of **time iteration**: iterating on a guess of the optimal policy using the Euler equation.

This approach differs from the value function iteration we used in *Optimal Savings III: Stochastic Returns*, where we iterated on the value function itself.

Time iteration exploits the structure of the Euler equation to find the optimal policy directly, rather than computing the value function as an intermediate step.

The key advantage is computational efficiency: by working directly with the policy function, we can often solve problems faster than with value function iteration.

However, time iteration is not the most efficient Euler equation-based method available.

In *Optimal Savings V: The Endogenous Grid Method*, we'll introduce the **endogenous grid method** (EGM), which provides an even more efficient way to solve the problem.

For now, our goal is to understand the basic mechanics of time iteration and how it leverages the Euler equation.

Let's start with some imports:

```
import matplotlib.pyplot as plt
import numpy as np
```

(continues on next page)

(continued from previous page)

```
from scipy.optimize import brentq
from typing import NamedTuple, Callable
```

56.2 The Euler Equation

Our first step is to derive the Euler equation, which is a generalization of the Euler equation we obtained in *Optimal Savings I: Cake Eating*.

We take the model set out in *Optimal Savings III: Stochastic Returns* and add the following assumptions:

1. u and f are continuously differentiable and strictly concave
2. $f(0) = 0$
3. $\lim_{c \to 0} u'(c) = \infty$ and $\lim_{c \to \infty} u'(c) = 0$
4. $\lim_{k \to 0} f'(k) = \infty$ and $\lim_{k \to \infty} f'(k) = 0$

The last two conditions are usually called **Inada conditions**.

Recall the Bellman equation

$$v(x) = \max_{0 \leq c \leq x} \left\{ u(c) + \beta \int v(f(x-c)z)\phi(dz) \right\} \quad \text{for all} \quad x \in \mathbb{R}_+ \tag{56.1}$$

Let v^* be the value function and let σ^* be the optimal consumption policy.

We know that σ^* is a v^*-greedy policy.

The conditions above imply that

- σ^* is the unique optimal policy for the optimal savings problem
- the optimal policy is continuous, strictly increasing and also **interior**, in the sense that $0 < \sigma^*(x) < x$ for all strictly positive x, and
- the value function is strictly concave and continuously differentiable, with

$$(v^*)'(x) = u'(\sigma^*(x)) := (u' \circ \sigma^*)(x) \tag{56.2}$$

The last result is called the **envelope condition** due to its relationship with the envelope theorem.

To see why (56.2) holds, write the Bellman equation in the equivalent form

$$v^*(x) = \max_{0 \leq k \leq x} \left\{ u(x-k) + \beta \int v^*(f(k)z)\phi(dz) \right\},$$

Differentiating with respect to x, and then evaluating at the optimum yields (56.2).

(Section 12.1 of EDTC contains full proofs of these results, and closely related discussions can be found in many other texts.)

Differentiability of the value function and interiority of the optimal policy imply that optimal consumption satisfies the first-order condition associated with (56.1), which is

$$u'(\sigma^*(x)) = \beta \int (v^*)'(f(x - \sigma^*(x))z) f'(x - \sigma^*(x)) z \phi(dz) \tag{56.3}$$

Combining (56.2) and the first-order condition (56.3) gives the **Euler equation**

$$(u' \circ \sigma^*)(x) = \beta \int (u' \circ \sigma^*)(f(x - \sigma^*(x))z) f'(x - \sigma^*(x)) z \phi(dz) \tag{56.4}$$

We can think of the Euler equation as a functional equation

$$(u' \circ \sigma)(x) = \beta \int (u' \circ \sigma)(f(x - \sigma(x))z) f'(x - \sigma(x)) z \phi(dz) \tag{56.5}$$

over interior consumption policies σ, one solution of which is the optimal policy σ^*.

Our aim is to solve the functional equation (56.5) and hence obtain σ^*.

56.2.1 The Coleman-Reffett Operator

Recall the Bellman operator

$$Tv(x) := \max_{0 \leq c \leq x} \left\{ u(c) + \beta \int v(f(x - c)z) \phi(dz) \right\} \tag{56.6}$$

Just as we introduced the Bellman operator to solve the Bellman equation, we will now introduce an operator over policies to help us solve the Euler equation.

This operator K will act on the set of all $\sigma \in \Sigma$ that are continuous, strictly increasing and interior.

Henceforth we denote this set of policies by $\mathscr{P}$

1. The operator K takes as its argument a $\sigma \in \mathscr{P}$ and
2. returns a new function $K\sigma$, where $K\sigma(x)$ is the $c \in (0, x)$ that solves

$$u'(c) = \beta \int (u' \circ \sigma)(f(x - c)z) f'(x - c) z \phi(dz) \tag{56.7}$$

We call this operator the **Coleman-Reffett operator** to acknowledge the work of [Coleman, 1990] and [Reffett, 1996].

In essence, $K\sigma$ is the consumption policy that the Euler equation tells you to choose today when your future consumption policy is σ.

The important thing to note about K is that, by construction, its fixed points coincide with solutions to the functional equation (56.5).

In particular, the optimal policy σ^* is a fixed point.

Indeed, for fixed x, the value $K\sigma^*(x)$ is the c that solves

$$u'(c) = \beta \int (u' \circ \sigma^*)(f(x - c)z) f'(x - c) z \phi(dz)$$

In view of the Euler equation, this is exactly $\sigma^*(x)$.

56.2.2 Is the Coleman-Reffett Operator Well Defined?

In particular, is there always a unique $c \in (0, x)$ that solves (56.7)?

The answer is yes, under our assumptions.

For any $\sigma \in \mathscr{P}$, the right side of (56.7)

- is continuous and strictly increasing in c on $(0, x)$
- diverges to $+\infty$ as $c \uparrow x$

The left side of (56.7)

- is continuous and strictly decreasing in c on $(0, x)$

- diverges to $+\infty$ as $c \downarrow 0$

Sketching these curves and using the information above will convince you that they cross exactly once as c ranges over $(0, x)$.

With a bit more analysis, one can show in addition that $K\sigma \in \mathscr{P}$ whenever $\sigma \in \mathscr{P}$.

56.2.3 Comparison with VFI (Theory)

It is possible to prove that there is a tight relationship between iterates of K and iterates of the Bellman operator.

Mathematically, T and K are **topologically conjugate** under a translation that involves differentiation in one direction and integration in the other.

This conjugacy implies that if iterates of one operator converge then so do iterates of the other, and vice versa.

Moreover, there is a sense in which they converge *at the same rate*, at least in theory.

However, it turns out that the operator K is more stable *numerically* and hence more efficient in the applications we consider.

This is because

- K exploits additional structure because it uses first-order conditions, and
- policies near the optimal policy have less curvature and hence are easier to approximate than value functions near the optimal value function.

Examples are given below.

56.3 Implementation

Let's turn to implementation.

Note

In this lecture we mainly focus on the algorithm, favoring clarity over efficiency in the code.

In later lectures we will optimize both the algorithm and the code.

As in *Optimal Savings III: Stochastic Returns*, we assume that

- $u(c) = \ln c$
- $f(x - c) = (x - c)^\alpha$
- ϕ is the distribution of $\xi := \exp(\mu + \nu\zeta)$ when ζ is standard normal

This allows us to compare our results to the analytical solutions we obtained in that lecture:

```
def v_star(x, α, β, μ):
    """
    True value function
    """
    c1 = np.log(1 - α * β) / (1 - β)
    c2 = (μ + α * np.log(α * β)) / (1 - α)
    c3 = 1 / (1 - β)
    c4 = 1 / (1 - α * β)
```

(continues on next page)

(continued from previous page)

```
    return c1 + c2 * (c3 - c4) + c4 * np.log(x)

def σ_star(x, α, β):
    """
    True optimal policy
    """
    return (1 - α * β) * x
```

As discussed above, our plan is to solve the model using time iteration, which means iterating with the operator K.

For this we need access to the functions u' and f, f'.

We use the same `Model` structure from *Optimal Savings III: Stochastic Returns*.

```
class Model(NamedTuple):
    u: Callable        # utility function
    f: Callable        # production function
    β: float           # discount factor
    μ: float           # shock location parameter
    ν: float           # shock scale parameter
    grid: np.ndarray   # state grid
    shocks: np.ndarray # shock draws
    α: float = 0.4     # production function parameter
    u_prime: Callable = None        # derivative of utility
    f_prime: Callable = None        # derivative of production


def create_model(
        u: Callable,
        f: Callable,
        β: float = 0.96,
        μ: float = 0.0,
        ν: float = 0.1,
        grid_max: float = 4.0,
        grid_size: int = 120,
        shock_size: int = 250,
        seed: int = 1234,
        α: float = 0.4,
        u_prime: Callable = None,
        f_prime: Callable = None
    ) -> Model:
    """
    Creates an instance of the optimal savings model.
    """
    # Set up grid
    grid = np.linspace(1e-4, grid_max, grid_size)

    # Store shocks (with a seed, so results are reproducible)
    np.random.seed(seed)
    shocks = np.exp(μ + ν * np.random.randn(shock_size))

    return Model(u, f, β, μ, ν, grid, shocks, α, u_prime, f_prime)
```

Now we implement a method called `euler_diff`, which returns

$$u'(c) - \beta \int (u' \circ \sigma)(f(x-c)z) f'(x-c) z \phi(dz) \tag{56.8}$$

```python
def euler_diff(c: float, σ: np.ndarray, x: float, model: Model) -> float:
    """
    Set up a function such that the root with respect to c,
    given x and σ, is equal to Kσ(x).

    """

    # Unpack
    u, f, β, μ, ν, grid, shocks, α, u_prime, f_prime = model

    # Turn σ into a function via interpolation
    σ_func = lambda x: np.interp(x, grid, σ)

    # Now set up the function we need to find the root of.
    vals = u_prime(σ_func(f(x - c, α) * shocks)) * f_prime(x - c, α) * shocks
    return u_prime(c) - β * np.mean(vals)
```

The function `euler_diff` evaluates integrals by Monte Carlo and approximates functions using linear interpolation.

We will use a root-finding algorithm to solve (56.8) for c given state x and σ, the current guess of the policy.

Here's the operator K, that implements the root-finding step.

```python
def K(σ: np.ndarray, model: Model) -> np.ndarray:
    """
    The Coleman-Reffett operator

    """

    # Unpack
    u, f, β, μ, ν, grid, shocks, α, u_prime, f_prime = model

    σ_new = np.empty_like(σ)
    for i, x in enumerate(grid):
        # Solve for optimal c at x
        c_star = brentq(euler_diff, 1e-10, x-1e-10, args=(σ, x, model))
        σ_new[i] = c_star

    return σ_new
```

56.3.1 Testing

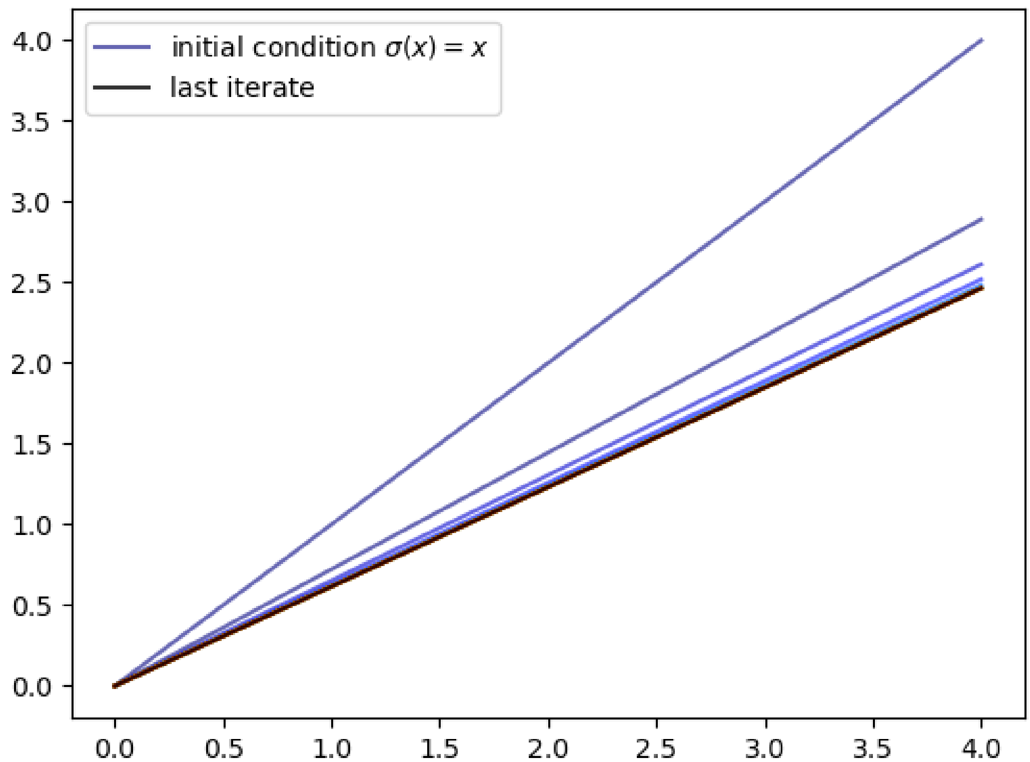
Let's generate an instance and plot some iterates of K, starting from $\sigma(x) = x$.

```python
# Define utility and production functions with derivatives
α = 0.4
u = lambda c: np.log(c)
u_prime = lambda c: 1 / c
f = lambda k, α: k**α
f_prime = lambda k, α: α * k**(α - 1)

model = create_model(u=u, f=f, α=α, u_prime=u_prime, f_prime=f_prime)
grid = model.grid

n = 15
σ = grid.copy()  # Set initial condition
```

(continues on next page)

(continued from previous page)

```
fig, ax = plt.subplots()
lb = r'initial condition $\sigma(x) = x$'
ax.plot(grid, σ, color=plt.cm.jet(0), alpha=0.6, label=lb)

for i in range(n):
    σ = K(σ, model)
    ax.plot(grid, σ, color=plt.cm.jet(i / n), alpha=0.6)

# Update one more time and plot the last iterate in black
σ = K(σ, model)
ax.plot(grid, σ, color='k', alpha=0.8, label='last iterate')

ax.legend()

plt.show()
```



We see that the iteration process converges quickly to a limit that resembles the solution we obtained in *Optimal Savings III: Stochastic Returns*.

Here is a function called `solve_model_time_iter` that takes an instance of `Model` and returns an approximation to the optimal policy, using time iteration.

```
def solve_model_time_iter(
        model: Model,
        σ_init: np.ndarray,
        tol: float = 1e-5,
        max_iter: int = 1000,
        verbose: bool = True
```

(continues on next page)

(continued from previous page)

```
    ) -> np.ndarray:
    """
    Solve the model using time iteration.

    """
    σ = σ_init
    error = tol + 1
    i = 0

    while error > tol and i < max_iter:
        σ_new = K(σ, model)
        error = np.max(np.abs(σ_new - σ))
        σ = σ_new
        i += 1
        if verbose:
            print(f"Iteration {i}, error = {error}")

    if i == max_iter:
        print("Warning: maximum iterations reached")

    return σ
```

Let's call it:

```
# Unpack
grid = model.grid

σ_init = np.copy(grid)
σ = solve_model_time_iter(model, σ_init)
```

```
Iteration 1, error = 1.1098265895953756
Iteration 2, error = 0.27827989207957415
Iteration 3, error = 0.09312729948559406
Iteration 4, error = 0.034020038271351805
Iteration 5, error = 0.012820752818722525
Iteration 6, error = 0.004888081560539437
Iteration 7, error = 0.0018718902256105174
Iteration 8, error = 0.0007180512309568066
Iteration 9, error = 0.0002756205293255043
Iteration 10, error = 0.00010582190181418483
Iteration 11, error = 4.063319516811603e-05
Iteration 12, error = 1.560279084289462e-05
Iteration 13, error = 5.991419175455093e-06
```

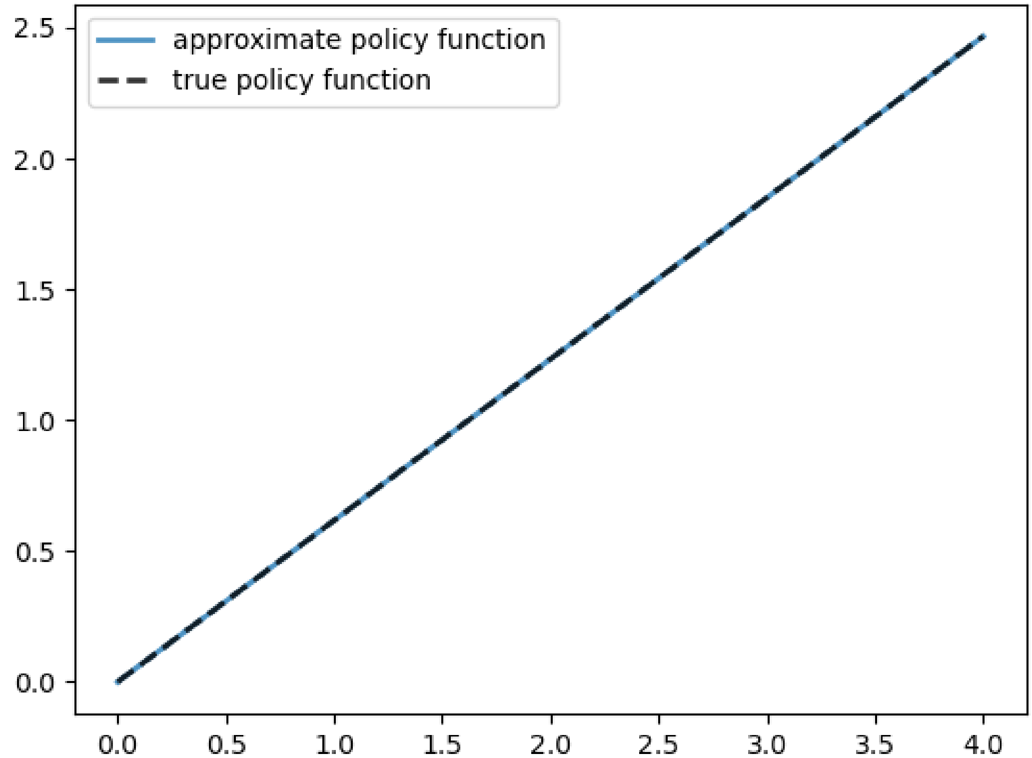
Here is a plot of the resulting policy, compared with the true policy:

```
# Unpack
grid, α, β = model.grid, model.α, model.β

fig, ax = plt.subplots()

ax.plot(grid, σ, lw=2,
        alpha=0.8, label='approximate policy function')

ax.plot(grid, σ_star(grid, α, β), 'k--',
        lw=2, alpha=0.8, label='true policy function')
```

(continues on next page)

(continued from previous page)

```
ax.legend()
plt.show()
```



Again, the fit is excellent.

The maximal absolute deviation between the two policies is

```
# Unpack
grid, α, β = model.grid, model.α, model.β

np.max(np.abs(σ - σ_star(grid, α, β)))
```

```
np.float64(3.7348959489591493e-06)
```

Time iteration runs faster than value function iteration, as discussed in *Optimal Savings III: Stochastic Returns*.

This is because time iteration exploits differentiability and the first-order conditions, while value function iteration does not use this available structure.

At the same time, there is a variation of time iteration that runs even faster.

This is the endogenous grid method, which we will introduce in *Optimal Savings V: The Endogenous Grid Method*.

56.4 Exercises

Exercise 56.4.1

Solve the optimal savings problem with CRRA utility

$$u(c) = \frac{c^{1-\gamma}}{1-\gamma}$$

Set `γ = 1.5`.

Compute and plot the optimal policy.

Solution

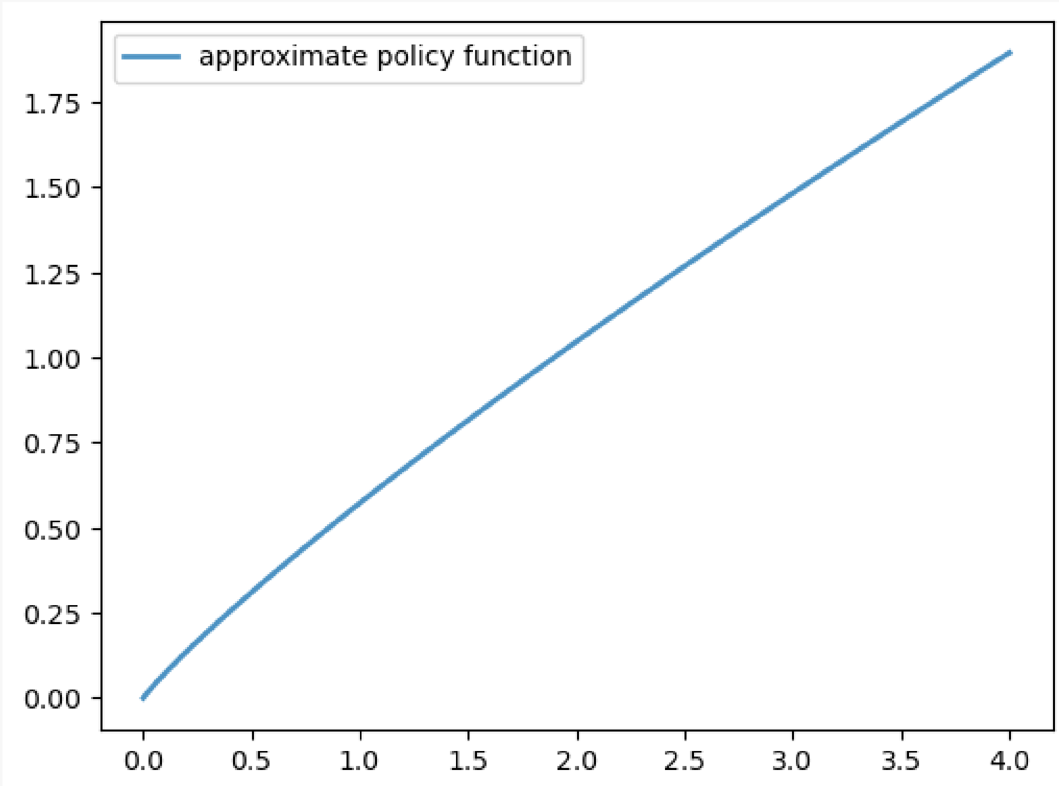
We define the CRRA utility function and its derivative.

```
γ = 1.5

def u_crra(c):
    return c**(1 - γ) / (1 - γ)

def u_prime_crra(c):
    return c**(-γ)

# Use the same production function as before
model_crra = create_model(u=u_crra, f=f, α=α,
                          u_prime=u_prime_crra, f_prime=f_prime)
```

Now we solve and plot the policy:

```
%%time
# Unpack
grid = model_crra.grid

σ_init = np.copy(grid)
σ = solve_model_time_iter(model_crra, σ_init)


fig, ax = plt.subplots()

ax.plot(grid, σ, lw=2,
        alpha=0.8, label='approximate policy function')

ax.legend()
plt.show()
```

```
Iteration 1, error = 1.449952719114732
Iteration 2, error = 0.3967698022828947
Iteration 3, error = 0.14845269076775747
Iteration 4, error = 0.06192954031818365
Iteration 5, error = 0.027017665601367424
Iteration 6, error = 0.012019070058330028
Iteration 7, error = 0.005393694573905705
Iteration 8, error = 0.0024299846499917788
Iteration 9, error = 0.0010967197524933692
Iteration 10, error = 0.0004953902833375601
Iteration 11, error = 0.00022384722341417 53
Iteration 12, error = 0.0001011641350074921
Iteration 13, error = 4.572272482672446e-05
Iteration 14, error = 2.066580711579391e-05
Iteration 15, error = 9.340704450133686e-06
CPU times: user 644 ms, sys: 3.02 ms, total: 647 ms
```


approximate policy function
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0

CHAPTER

FIFTYSEVEN

OPTIMAL SAVINGS V: THE ENDOGENOUS GRID METHOD

Contents

57.1 Overview

Previously, we solved the optimal savings problem using

1. *value function iteration*
2. *Euler equation based time iteration*

We found time iteration to be significantly more accurate and efficient.

In this lecture, we'll look at a clever twist on time iteration called the **endogenous grid method** (EGM).

EGM is a numerical method for implementing policy iteration invented by Chris Carroll.

The original reference is [Carroll, 2006].

For now we will focus on a clean and simple implementation of EGM that stays close to the underlying mathematics.

Then, in *Optimal Savings VI: EGM with JAX*, we will construct a fully vectorized and parallelized version of EGM based on JAX.

Let's start with some standard imports:

```
import matplotlib.pyplot as plt
import numpy as np
import quantecon as qe
```

57.2 Key Idea

First we remind ourselves of the theory and then we turn to numerical methods.

57.2.1 Theory

We work with the model set out in *Optimal Savings IV: Time Iteration*, following the same terminology and notation.

As we saw, the Coleman-Reffett operator is a nonlinear operator K engineered so that the optimal policy σ^* is a fixed point of K.

It takes as its argument a continuous strictly increasing consumption policy $\sigma \in \Sigma$.

It returns a new function $K\sigma$, where $(K\sigma)(x)$ is the $c \in (0, \infty)$ that solves

$$u'(c) = \beta \int (u' \circ \sigma)(f(x-c)z)f'(x-c)z\phi(dz) \tag{57.1}$$

57.2.2 Exogenous Grid

As discussed in *Optimal Savings IV: Time Iteration*, to implement the method on a computer, we represent a policy function by a set of values on a finite grid.

The function itself is reconstructed from this representation when necessary, using interpolation or some other method.

Our previous strategy in *Optimal Savings IV: Time Iteration* for obtaining a finite representation of an updated consumption policy was to

- fix a grid of income points $\{x_i\}$
- calculate the consumption value c_i corresponding to each x_i using (57.1) and a root-finding routine

Each c_i is then interpreted as the value of the function $K\sigma$ at x_i.

Thus, with the pairs $\{(x_i, c_i)\}$ in hand, we can reconstruct $K\sigma$ via approximation.

Iteration then continues…

57.2.3 Endogenous Grid

The method discussed above requires a root-finding routine to find the c_i corresponding to a given income value x_i.

Root-finding is costly because it typically involves a significant number of function evaluations.

As pointed out by Carroll [Carroll, 2006], we can avoid this step if x_i is chosen endogenously.

The only assumption required is that u' is invertible on $(0, \infty)$.

Let $(u')^{-1}$ be the inverse function of u'.

The idea is this:

- First, we fix an *exogenous* grid $\{s_i\}$ for savings ($s = x - c$).
- Then we obtain c_i via

$$c_i = (u')^{-1} \left\{ \beta \int (u' \circ \sigma)(f(s_i)z)\, f'(s_i)\, z\, \phi(dz) \right\} \tag{57.2}$$

- Finally, for each c_i we set $x_i = c_i + s_i$.

Importantly, each (x_i, c_i) pair constructed in this manner satisfies (57.1).

With the points $\{x_i, c_i\}$ in hand, we can reconstruct $K\sigma$ via approximation as before.

The name EGM comes from the fact that the grid $\{x_i\}$ is determined **endogenously**.

57.3 Implementation

As in *Optimal Savings IV: Time Iteration*, we will start with a simple setting where

- $u(c) = \ln c$,
- the function f has a Cobb-Douglas specification, and
- the shocks are lognormal.

This will allow us to make comparisons with the analytical solutions.

```python
def v_star(x, α, β, μ):
    """
    True value function
    """
    c1 = np.log(1 - α * β) / (1 - β)
    c2 = (μ + α * np.log(α * β)) / (1 - α)
    c3 = 1 / (1 - β)
    c4 = 1 / (1 - α * β)
    return c1 + c2 * (c3 - c4) + c4 * np.log(x)

def σ_star(x, α, β):
    """
    True optimal policy
    """
    return (1 - α * β) * x
```

We reuse the `Model` structure from *Optimal Savings IV: Time Iteration*.

```python
from typing import NamedTuple, Callable

class Model(NamedTuple):
    u: Callable           # utility function
    f: Callable           # production function
    β: float              # discount factor
    μ: float              # shock location parameter
    ν: float              # shock scale parameter
    s_grid: np.ndarray    # exogenous savings grid
    shocks: np.ndarray    # shock draws
    α: float              # production function parameter
    u_prime: Callable     # derivative of utility
    f_prime: Callable     # derivative of production
    u_prime_inv: Callable # inverse of u_prime


def create_model(
        u: Callable,
        f: Callable,
        β: float = 0.96,
        μ: float = 0.0,
        ν: float = 0.1,
```

(continues on next page)

(continued from previous page)

```
        grid_max: float = 4.0,
        grid_size: int = 120,
        shock_size: int = 250,
        seed: int = 1234,
        α: float = 0.4,
        u_prime: Callable = None,
        f_prime: Callable = None,
        u_prime_inv: Callable = None
    ) -> Model:
    """
    Creates an instance of the optimal savings model.
    """
    # Set up exogenous savings grid
    s_grid = np.linspace(1e-4, grid_max, grid_size)

    # Store shocks (with a seed, so results are reproducible)
    np.random.seed(seed)
    shocks = np.exp(μ + ν * np.random.randn(shock_size))

    return Model(
        u, f, β, μ, ν, s_grid, shocks, α, u_prime, f_prime, u_prime_inv
    )
```

57.3.1 The Operator

Here's an implementation of K using EGM as described above.

```
def K(
        c_in: np.ndarray,    # Consumption values on the endogenous grid
        x_in: np.ndarray,    # Current endogenous grid
        model: Model         # Model specification
    ):
    """
    An implementation of the Coleman-Reffett operator using EGM.

    """

    # Simplify names
    u, f, β, μ, ν, s_grid, shocks, α, u_prime, f_prime, u_prime_inv = model

    # Linear interpolation of policy on the endogenous grid
    σ = lambda x: np.interp(x, x_in, c_in)

    # Allocate memory for new consumption array
    c_out = np.empty_like(s_grid)

    for i, s in enumerate(s_grid):
        # Approximate marginal utility ∫ u'(σ(f(s, α)z)) f'(s, α) z ϕ(z)dz
        vals = u_prime(σ(f(s, α) * shocks)) * f_prime(s, α) * shocks
        mu = np.mean(vals)
        # Compute consumption
        c_out[i] = u_prime_inv(β * mu)

    # Determine corresponding endogenous grid
    x_out = s_grid + c_out        # x_i = s_i + c_i
```

(continues on next page)

(continued from previous page)

```
    return c_out, x_out
```

Note the lack of any root-finding algorithm.

Note

The routine is still not particularly fast because we are using pure Python loops.

But in the next lecture (*Optimal Savings VI: EGM with JAX*) we will use a fully vectorized and efficient solution.

57.3.2 Testing

First we create an instance.

```
# Define utility and production functions with derivatives
u = lambda c: np.log(c)
u_prime = lambda c: 1 / c
u_prime_inv = lambda x: 1 / x
f = lambda k, α: k**α
f_prime = lambda k, α: α * k**(α - 1)

model = create_model(u=u, f=f, u_prime=u_prime,
                     f_prime=f_prime, u_prime_inv=u_prime_inv)
s_grid = model.s_grid
```

Here's our solver routine:

```
def solve_model_time_iter(
        model: Model,              # Model details
        c_init: np.ndarray,        # initial guess of consumption on EG
        x_init: np.ndarray,        # initial guess of endogenous grid
        tol: float = 1e-5,         # Error tolerance
        max_iter: int = 1000,      # Max number of iterations of K
        verbose: bool = True       # If true print output
    ):
    """
    Solve the model using time iteration with EGM.
    """
    c, x = c_init, x_init
    error = tol + 1
    i = 0

    while error > tol and i < max_iter:
        c_new, x_new = K(c, x, model)
        error = np.max(np.abs(c_new - c))
        c, x = c_new, x_new
        i += 1
        if verbose:
            print(f"Iteration {i}, error = {error}")

    if i == max_iter:
        print("Warning: maximum iterations reached")
```

(continues on next page)

(continued from previous page)

```
    return c, x
```

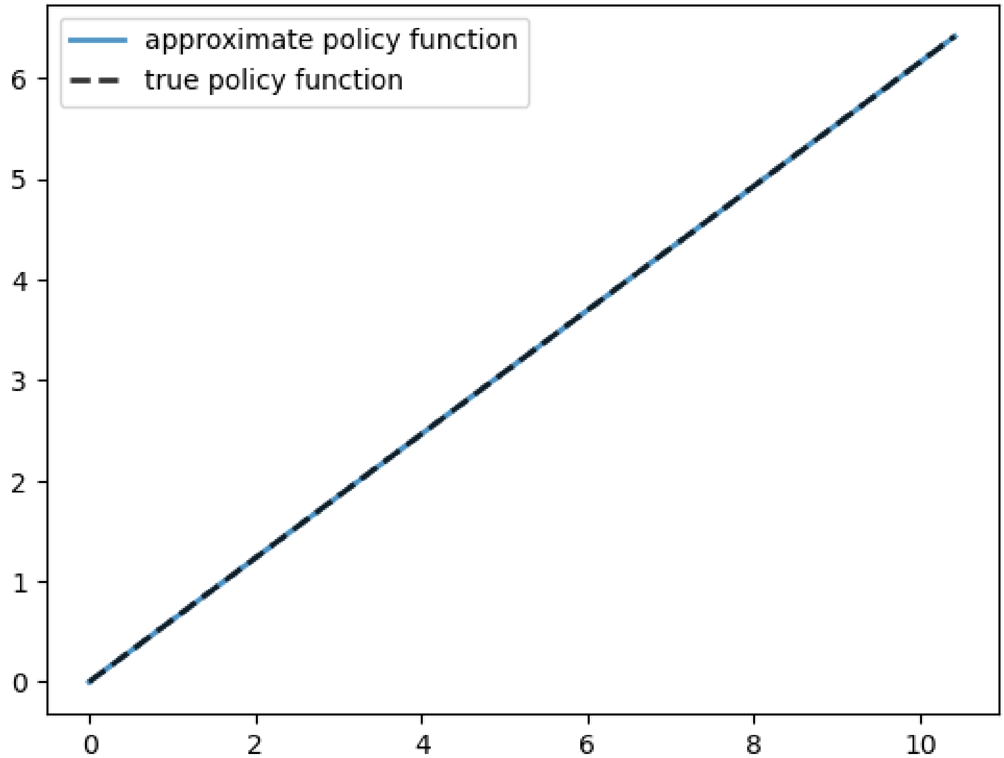
Let's call it:

```
c_init = np.copy(s_grid)
x_init = s_grid + c_init
c, x = solve_model_time_iter(model, c_init, x_init)
```

```
Iteration 1, error = 1.2083333333333333
Iteration 2, error = 0.6834464555052788
Iteration 3, error = 0.3126351338414741
Iteration 4, error = 0.12905177785629984
Iteration 5, error = 0.05099394329538409
Iteration 6, error = 0.01980055511172729
Iteration 7, error = 0.007636088144566955
Iteration 8, error = 0.002937098891578671
Iteration 9, error = 0.0011285611196765188
Iteration 10, error = 0.00043347299641638415
Iteration 11, error = 0.00016646919532892213
Iteration 12, error = 6.392646634978405e-05
Iteration 13, error = 2.4548101555943447e-05
Iteration 14, error = 9.426520908739633e-06
```

Here is a plot of the resulting policy, compared with the true policy:

```
fig, ax = plt.subplots()

ax.plot(x, c, lw=2,
        alpha=0.8, label='approximate policy function')

ax.plot(x, σ_star(x, model.α, model.β), 'k--',
        lw=2, alpha=0.8, label='true policy function')

ax.legend()
plt.show()
```



The maximal absolute deviation between the two policies is

```
np.max(np.abs(c - σ_star(x, model.α, model.β)))
```

```
np.float64(2.2564941266622895e-06)
```

Here's the execution time:

```
with qe.Timer():
    c, x = solve_model_time_iter(model, c_init, x_init, verbose=False)
```

```
0.03 seconds elapsed
```

EGM is faster than time iteration because it avoids numerical root-finding.

Instead, we invert the marginal utility function directly, which is much more efficient.

In *Optimal Savings VI: EGM with JAX*, we will use a fully vectorized and efficient version of EGM that is also parallelized using JAX.

This provides an extremely fast way to solve the optimal consumption problem we have been studying for the last few lectures.

CHAPTER

FIFTYEIGHT

OPTIMAL SAVINGS VI: EGM WITH JAX

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

Contents

58.1 Overview

In this lecture, we'll implement the endogenous grid method (EGM) using JAX.

This lecture builds on *Optimal Savings V: The Endogenous Grid Method*, which introduced EGM using NumPy.

By converting to JAX, we can leverage fast linear algebra, hardware accelerators, and JIT compilation for improved performance.

We'll also use JAX's `vmap` function to fully vectorize the Coleman-Reffett operator.

Let's start with some standard imports:

```
import matplotlib.pyplot as plt
import jax
import jax.numpy as jnp
import quantecon as qe
from typing import NamedTuple
```

58.2 Implementation

For details on the savings problem and the endogenous grid method (EGM), please see *Optimal Savings V: The Endogenous Grid Method*.

Here we focus on the JAX implementation of EGM.

We use the same setting as in *Optimal Savings V: The Endogenous Grid Method*:

- $u(c) = \ln c$,
- production is Cobb-Douglas, and
- the shocks are lognormal.

Here are the analytical solutions for comparison.

```
def v_star(x, α, β, μ):
    """
    True value function
    """
    c1 = jnp.log(1 - α * β) / (1 - β)
    c2 = (μ + α * jnp.log(α * β)) / (1 - α)
    c3 = 1 / (1 - β)
    c4 = 1 / (1 - α * β)
    return c1 + c2 * (c3 - c4) + c4 * jnp.log(x)

def σ_star(x, α, β):
    """
    True optimal policy
    """
    return (1 - α * β) * x
```

The `Model` class stores only the data (grids, shocks, and parameters).

Utility and production functions will be defined globally to work with JAX's JIT compiler.

```
class Model(NamedTuple):
    β: float              # discount factor
    μ: float              # shock location parameter
    s: float              # shock scale parameter
    s_grid: jnp.ndarray   # exogenous savings grid
    shocks: jnp.ndarray   # shock draws
    α: float              # production function parameter


def create_model(
        β: float = 0.96,
        μ: float = 0.0,
        s: float = 0.1,
        grid_max: float = 4.0,
        grid_size: int = 120,
        shock_size: int = 250,
        seed: int = 1234,
        α: float = 0.4
    ) -> Model:
    """
    Creates an instance of the optimal savings model.
    """
```

(continues on next page)

(continued from previous page)

```
    # Set up exogenous savings grid
    s_grid = jnp.linspace(1e-4, grid_max, grid_size)

    # Store shocks (with a seed, so results are reproducible)
    key = jax.random.PRNGKey(seed)
    shocks = jnp.exp(μ + s * jax.random.normal(key, shape=(shock_size,)))

    return Model(β, μ, s, s_grid, shocks, α)
```

We define utility and production functions globally.

```
# Define utility and production functions with derivatives
u = lambda c: jnp.log(c)
u_prime = lambda c: 1 / c
u_prime_inv = lambda x: 1 / x
f = lambda k, α: k**α
f_prime = lambda k, α: α * k**(α - 1)
```

Here's the Coleman-Reffett operator using EGM.

The key JAX feature here is `vmap`, which vectorizes the computation over the grid points.

```
def K(
        c_in: jnp.ndarray,  # Consumption values on the endogenous grid
        x_in: jnp.ndarray,  # Current endogenous grid
        model: Model        # Model specification
    ):
    """
    The Coleman-Reffett operator using EGM

    """
    β, μ, s, s_grid, shocks, α = model
    σ = lambda x_val: jnp.interp(x_val, x_in, c_in)

    # Define function to compute consumption at a single grid point
    def compute_c(s):
        # Approximate marginal utility ∫ u'(σ(f(s, α)z)) f'(s, α) z ϕ(z)dz
        vals = u_prime(σ(f(s, α) * shocks)) * f_prime(s, α) * shocks
        mu = jnp.mean(vals)
        # Calculate consumption
        return u_prime_inv(β * mu)

    # Vectorize and calculate on all exogenous grid points
    compute_c_vectorized = jax.vmap(compute_c)
    c_out = compute_c_vectorized(s_grid)

    # Determine corresponding endogenous grid
    x_out = s_grid + c_out  # x_i = s_i + c_i

    return c_out, x_out
```

Now we create a model instance.

```
model = create_model()
s_grid = model.s_grid
```

The solver uses JAX's `jax.lax.while_loop` for the iteration and is JIT-compiled for speed.

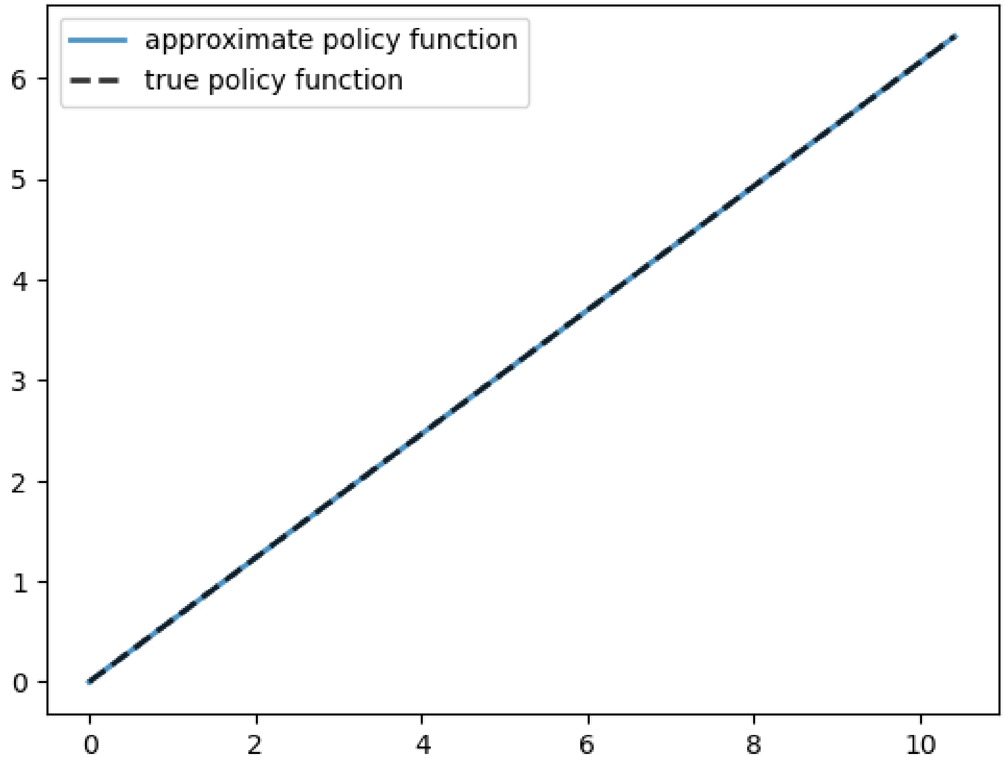
```python
@jax.jit
def solve_model_time_iter(
        model: Model,
        c_init: jnp.ndarray,
        x_init: jnp.ndarray,
        tol: float = 1e-5,
        max_iter: int = 1000
    ):
    """
    Solve the model using time iteration with EGM.
    """

    def condition(loop_state):
        i, c, x, error = loop_state
        return (error > tol) & (i < max_iter)

    def body(loop_state):
        i, c, x, error = loop_state
        c_new, x_new = K(c, x, model)
        error = jnp.max(jnp.abs(c_new - c))
        return i + 1, c_new, x_new, error

    # Initialize loop state
    initial_state = (0, c_init, x_init, tol + 1)

    # Run the loop
    i, c, x, error = jax.lax.while_loop(condition, body, initial_state)

    return c, x
```

We solve the model starting from an initial guess.

```python
c_init = jnp.copy(s_grid)
x_init = s_grid + c_init
c, x = solve_model_time_iter(model, c_init, x_init)
```

Let's plot the resulting policy against the analytical solution.

```python
fig, ax = plt.subplots()

ax.plot(x, c, lw=2,
        alpha=0.8, label='approximate policy function')

ax.plot(x, σ_star(x, model.α, model.β), 'k--',
        lw=2, alpha=0.8, label='true policy function')

ax.legend()
plt.show()
```



The fit is very good.

```
max_dev = jnp.max(jnp.abs(c - σ_star(x, model.α, model.β)))
print(f"Maximum absolute deviation: {max_dev:.7}")
```

```
Maximum absolute deviation: 1.430511e-06
```

The JAX implementation is very fast thanks to JIT compilation and vectorization.

```
with qe.Timer(precision=8):
    c, x = solve_model_time_iter(model, c_init, x_init)
    jax.block_until_ready(c)
```

```
0.00263739 seconds elapsed
```

This speed comes from:

- JIT compilation of the entire solver
- Vectorization via `vmap` in the Coleman-Reffett operator
- Use of `jax.lax.while_loop` instead of a Python loop
- Efficient JAX array operations throughout

58.3 Exercises

Exercise 58.3.1

Solve the optimal savings problem with CRRA utility

$$u(c) = \frac{c^{1-\gamma} - 1}{1 - \gamma}$$

Compare the optimal policies for values of γ approaching 1 from above (e.g., 1.05, 1.1, 1.2).

Show that as $\gamma \to 1$, the optimal policy converges to the policy obtained with log utility ($\gamma = 1$).

Hint: Use values of γ close to 1 to ensure the endogenous grids have similar coverage and make visual comparison easier.

Solution

We need to create a version of the Coleman-Reffett operator and solver that work with CRRA utility.

The key is to parameterize the utility functions by γ.

```
def u_crra(c, γ):
    return (c**(1 - γ) - 1) / (1 - γ)

def u_prime_crra(c, γ):
    return c**(-γ)

def u_prime_inv_crra(x, γ):
    return x**(-1/γ)
```

Now we create a version of the Coleman-Reffett operator that takes γ as a parameter.

```
def K_crra(
        c_in: jnp.ndarray,  # Consumption values on the endogenous grid
        x_in: jnp.ndarray,  # Current endogenous grid
        model: Model,       # Model specification
        γ: float            # CRRA parameter
    ):
    """
    The Coleman-Reffett operator using EGM with CRRA utility
    """
    # Simplify names
    β, α = model.β, model.α
    s_grid, shocks = model.s_grid, model.shocks

    # Linear interpolation of policy using endogenous grid
    σ = lambda x_val: jnp.interp(x_val, x_in, c_in)

    # Define function to compute consumption at a single grid point
    def compute_c(s):
        vals = u_prime_crra(σ(f(s, α) * shocks), γ) * f_prime(s, α) * shocks
        return u_prime_inv_crra(β * jnp.mean(vals), γ)

    # Vectorize over grid using vmap
    compute_c_vectorized = jax.vmap(compute_c)
    c_out = compute_c_vectorized(s_grid)

    # Determine corresponding endogenous grid
    x_out = s_grid + c_out  # x_i = s_i + c_i

    return c_out, x_out
```

We also need a solver that uses this operator.

```
@jax.jit
def solve_model_crra(model: Model,
                     c_init: jnp.ndarray,
                     x_init: jnp.ndarray,
                     γ: float,
                     tol: float = 1e-5,
                     max_iter: int = 1000):
    """
    Solve the model using time iteration with EGM and CRRA utility.
    """

    def condition(loop_state):
        i, c, x, error = loop_state
        return (error > tol) & (i < max_iter)

    def body(loop_state):
        i, c, x, error = loop_state
        c_new, x_new = K_crra(c, x, model, γ)
        error = jnp.max(jnp.abs(c_new - c))
        return i + 1, c_new, x_new, error

    # Initialize loop state
    initial_state = (0, c_init, x_init, tol + 1)

    # Run the loop
    i, c, x, error = jax.lax.while_loop(condition, body, initial_state)

    return c, x
```

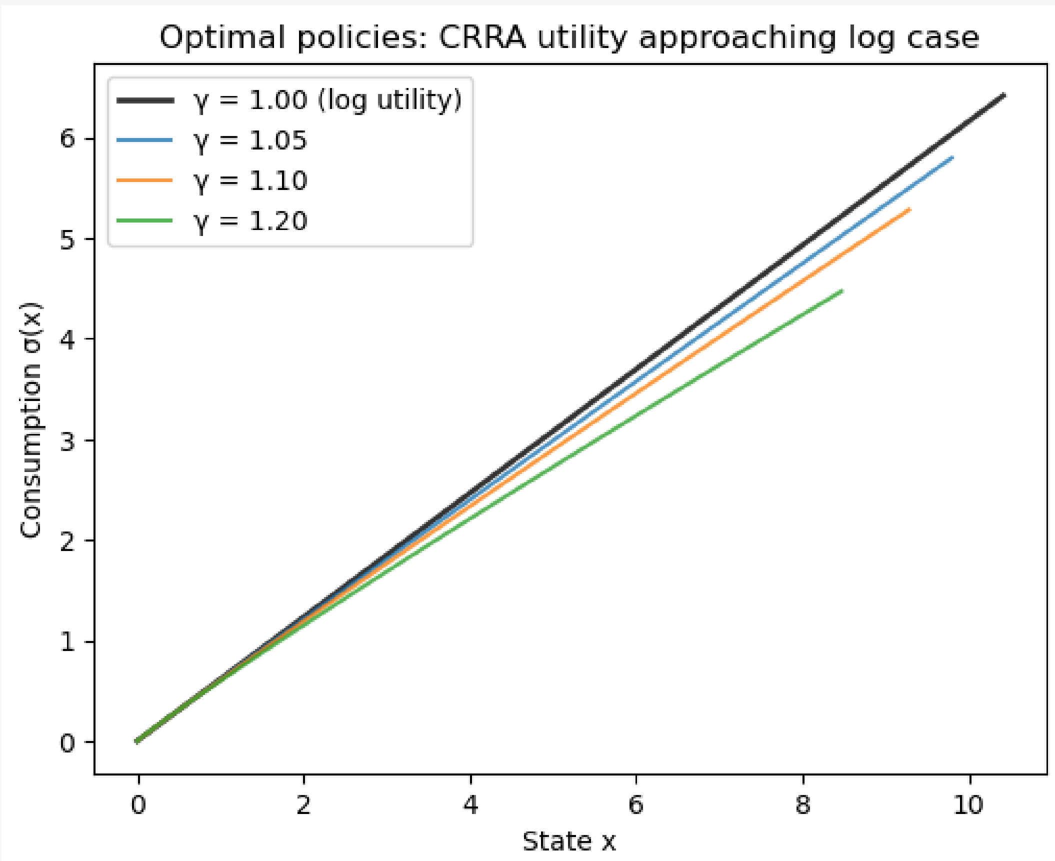
Now we solve for $\gamma = 1$ (log utility) and values approaching 1 from above.

```
γ_values = [1.0, 1.05, 1.1, 1.2]
policies = {}
endogenous_grids = {}

model_crra = create_model()

for γ in γ_values:
    c_init = jnp.copy(model_crra.s_grid)
    x_init = model_crra.s_grid + c_init
    c_gamma, x_gamma = solve_model_crra(model_crra, c_init, x_init, γ)
    jax.block_until_ready(c_gamma)
    policies[γ] = c_gamma
    endogenous_grids[γ] = x_gamma
    print(f"Solved for γ = {γ}")
```

```
Solved for γ = 1.0
Solved for γ = 1.05
Solved for γ = 1.1
Solved for γ = 1.2
```

Plot the policies on their endogenous grids.

```
fig, ax = plt.subplots()

for γ in γ_values:
    x = endogenous_grids[γ]
    if γ == 1.0:
        ax.plot(x, policies[γ], 'k-', linewidth=2,
                label=f'γ = {γ:.2f} (log utility)', alpha=0.8)
    else:
        ax.plot(x, policies[γ], label=f'γ = {γ:.2f}', alpha=0.8)

ax.set_xlabel('State x')
ax.set_ylabel('Consumption σ(x)')
ax.legend()
ax.set_title('Optimal policies: CRRA utility approaching log case')
plt.show()
```



Note that the plots for $\gamma > 1$ do not cover the entire x-axis range shown.

This is because the endogenous grid $x = s + \sigma(s)$ depends on the consumption policy, which varies with γ.

Let's check the maximum deviation between the log utility case ($\gamma = 1.0$) and values approaching from above.

```
for γ in [1.05, 1.1, 1.2]:
    max_diff = jnp.max(jnp.abs(policies[1.0] - policies[γ]))
    print(f"Max difference between γ=1.0 and γ={γ}: {max_diff:.6}")
```

```
Max difference between γ=1.0 and γ=1.05: 0.619199
Max difference between γ=1.0 and γ=1.1: 1.1362
Max difference between γ=1.0 and γ=1.2: 1.94592
```

As expected, the differences decrease as γ approaches 1 from above, confirming convergence.

Part IX

Household Problems

CHAPTER

FIFTYNINE

THE INCOME FLUCTUATION PROBLEM I: DISCRETIZATION AND VFI

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

59.1 Overview

In this lecture, we study an optimal savings problem for an infinitely lived consumer—the "common ancestor" described in [Ljungqvist and Sargent, 2018], section 1.3.

This savings problem is often called an **income fluctuation problem** or a **household problem**.

It is an essential sub-problem for many representative macroeconomic models

- [Aiyagari, 1994]
- [Huggett, 1993]
- etc.

It is related to the decision problem in *Optimal Savings III: Stochastic Returns* but differs in significant ways.

For example,

1. The choice problem for the agent includes an additive income term that leads to an occasionally binding constraint.
2. Shocks affecting the budget constraint are correlated, forcing us to track an extra state variable.

We will begin by working with a relatively basic version of the model and solving it via old-fashioned discretization + value function iteration.

Although this approach is not the fastest or the most efficient, it is very robust and flexible.

For example, if we suddenly decided to add Epstein–Zin preferences, or modify ordinary conditional expectations to quantiles, the technique would continue to work well.

Note

The same is not true of some other methods we will deploy, such as the endogenous grid method.

This is a general rule of computation and analysis — while we can often come up with faster algorithms by exploiting structure, these new algorithms are typically less robust.

They are less robust precisely because they exploit more structure — which implies that they are, inevitably, more vulnerable to change.

In addition to Anaconda, this lecture will need the following libraries:

```
!pip install quantecon jax
```

We will use the following imports:

```
import quantecon as qe
import jax
import jax.numpy as jnp
import matplotlib.pyplot as plt
from typing import NamedTuple
from time import time
```

We'll use 64 bit floats to gain extra precision.

```
jax.config.update("jax_enable_x64", True)
```

59.2 Set Up

We study a household that chooses a state-contingent consumption plan $\{c_t\}_{t \geq 0}$ to maximize

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t u(c_t)$$

subject to

$$a_{t+1} + c_t \leq R a_t + y_t$$

Here

- c_t is consumption and $c_t \geq 0$,
- a_t is assets and $a_t \geq 0$,
- $R = 1 + r$ is a gross rate of return, and
- $(y_t)_{t \geq 0}$ is labor income, taking values in some finite set Y.

We assume below that labor income dynamics follow a discretized AR(1) process.

We set $\mathsf{S} := \mathbb{R}_+ \times \mathsf{Y}$, which represents the state space.

The **value function** $V : \mathsf{S} \to \mathbb{R}$ is defined by

$$V(a, y) := \max \mathbb{E} \left\{ \sum_{t=0}^{\infty} \beta^t u(c_t) \right\} \tag{59.1}$$

where the maximization is over all feasible consumption sequences given $(a_0, y_0) = (a, y)$.

The Bellman equation is

$$v(a, y) = \max_{0 \le a' \le Ra+y} \left\{ u(Ra + y - a') + \beta \sum_{y'} v(a', y') Q(y, y') \right\}$$

where

$$u(c) = \frac{c^{1-\gamma}}{1-\gamma}$$

In the code we use the function

$$B((a, y), a', v) = u(Ra + y - a') + \beta \sum_{y'} v(a', y') Q(y, y').$$

the encapsulate the right hand side of the Bellman equation.

59.3 Code

The following code defines a `NamedTuple` to store the model parameters and grids.

```
class Model(NamedTuple):
    β: float              # Discount factor
    R: float              # Gross interest rate
    γ: float              # CRRA parameter
    a_grid: jnp.ndarray   # Asset grid
    y_grid: jnp.ndarray   # Income grid
    Q: jnp.ndarray        # Markov matrix for income


def create_consumption_model(
        R=1.01,                      # Gross interest rate
        β=0.98,                      # Discount factor
        γ=2,                         # CRRA parameter
        a_min=0.01,                  # Min assets
        a_max=10.0,                  # Max assets
        a_size=150,                  # Grid size
        ρ=0.9, ν=0.1, y_size=100    # Income parameters
    ):
    """
    Creates an instance of the consumption-savings model.

    """
    a_grid = jnp.linspace(a_min, a_max, a_size)
    mc = qe.tauchen(n=y_size, rho=ρ, sigma=ν)
    y_grid, Q = jnp.exp(mc.state_values), jax.device_put(mc.P)
    return Model(β, R, γ, a_grid, y_grid, Q)
```

Now we define the right hand side of the Bellman equation.

We'll use a vectorized coding style reminiscent of Matlab and NumPy (avoiding all loops).

Your are invited to explore an alternative style based around `jax.vmap` in the Exercises.

```
@jax.jit
def B(v, model):
    """
    A vectorized version of the right-hand side of the Bellman equation
    (before maximization), which is a 3D array representing

        B(a, y, a′) = u(Ra + y - a′) + β Σ_y′ v(a′, y′) Q(y, y′)

    for all (a, y, a′).
    """

    # Unpack
    β, R, γ, a_grid, y_grid, Q = model
    a_size, y_size = len(a_grid), len(y_grid)

    # Compute current rewards r(a, y, ap) as array r[i, j, ip]
    a  = jnp.reshape(a_grid, (a_size, 1, 1))    # a[i]   ->  a[i, j, ip]
    y  = jnp.reshape(y_grid, (1, y_size, 1))    # z[j]   ->  z[i, j, ip]
    ap = jnp.reshape(a_grid, (1, 1, a_size))    # ap[ip] -> ap[i, j, ip]
    c = R * a + y - ap

    # Calculate continuation rewards at all combinations of (a, y, ap)
    v = jnp.reshape(v, (1, 1, a_size, y_size))  # v[ip, jp] -> v[i, j, ip, jp]
    Q = jnp.reshape(Q, (1, y_size, 1, y_size))  # Q[j, jp]  -> Q[i, j, ip, jp]
    EV = jnp.sum(v * Q, axis=3)                 # sum over last index jp

    # Compute the right-hand side of the Bellman equation
    return jnp.where(c > 0, c**(1-γ)/(1-γ) + β * EV, -jnp.inf)
```

Some readers might be concerned that we are creating high dimensional arrays, leading to inefficiency.

Could they be avoided by more careful vectorization?

In fact this is not necessary: this function will be JIT-compiled by JAX, and the JIT compiler will optimize compiled code to minimize memory use.

The Bellman operator T can be implemented by

```
@jax.jit
def T(v, model):
    "The Bellman operator."
    return jnp.max(B(v, model), axis=2)
```

The next function computes a v-greedy policy given v (i.e., the policy that maximizes the right-hand side of the Bellman equation.)

```
@jax.jit
def get_greedy(v, model):
    "Computes a v-greedy policy, returned as a set of indices."
    return jnp.argmax(B(v, model), axis=2)
```

59.3.1 Value function iteration

Now we define a solver that implements VFI.

First we write a simple version using a standard Python loop.

```
def value_function_iteration_python(model, tol=1e-5, max_iter=10_000):
    """
    Implements VFI using successive approximation with a Python loop.
    """
    v = jnp.zeros((len(model.a_grid), len(model.y_grid)))
    error = tol + 1
    k = 0

    while error > tol and k < max_iter:
        v_new = T(v, model)
        error = jnp.max(jnp.abs(v_new - v))
        v = v_new
        k += 1

    return v, get_greedy(v, model)
```

Next we write a version that uses `jax.lax.while_loop`.

```
@jax.jit
def value_function_iteration(model, tol=1e-5, max_iter=10_000):
    """
    Implements VFI using successive approximation.
    """
    def body_fun(k_v_err):
        k, v, error = k_v_err
        v_new = T(v, model)
        error = jnp.max(jnp.abs(v_new - v))
        return k + 1, v_new, error

    def cond_fun(k_v_err):
        k, v, error = k_v_err
        return jnp.logical_and(error > tol, k < max_iter)

    v_init = jnp.zeros((len(model.a_grid), len(model.y_grid)))
    k, v_star, error = jax.lax.while_loop(cond_fun, body_fun,
                                          (1, v_init, tol + 1))
    return v_star, get_greedy(v_star, model)
```

59.3.2 Timing

Let's create an instance and compare the two implementations.

```
model = create_consumption_model()
```

First let's time the Python version.

```
print("Starting VFI using Python loop.")
start = time()
v_star_python, σ_star_python = value_function_iteration_python(model)
python_time = time() - start
print(f"VFI completed in {python_time} seconds.")
```

```
Starting VFI using Python loop.
VFI completed in 2.7177579402923584 seconds.
```

Now let's time the `jax.lax.while_loop` version.

```
print("Starting VFI using jax.lax.while_loop.")
start = time()
v_star_jax, σ_star_jax = value_function_iteration(model)
v_star_jax.block_until_ready()
jax_with_compile = time() - start
print(f"VFI completed in {jax_with_compile} seconds.")
```

```
Starting VFI using jax.lax.while_loop.
VFI completed in 1.7616283893585205 seconds.
```

Let's run it again to eliminate compile time.

```
start = time()
v_star_jax, σ_star_jax = value_function_iteration(model)
v_star_jax.block_until_ready()
jax_without_compile = time() - start
print(f"VFI completed in {jax_without_compile} seconds.")
```

```
VFI completed in 1.3665611743927002 seconds.
```

Let's check that the two implementations produce the same result.

```
print(f"Values match: {jnp.allclose(v_star_python, v_star_jax)}")
print(f"Policies match: {jnp.allclose(σ_star_python, σ_star_jax)}")
```

```
Values match: True
Policies match: True
```

Here's the speedup from using `jax.lax.while_loop`.

```
print(f"Relative speed = {python_time / jax_without_compile:.2f}")
```

```
Relative speed = 1.99
```

59.3.3 Asset Dynamics

To understand long-run behavior, let's examine the asset accumulation dynamics under the optimal policy.

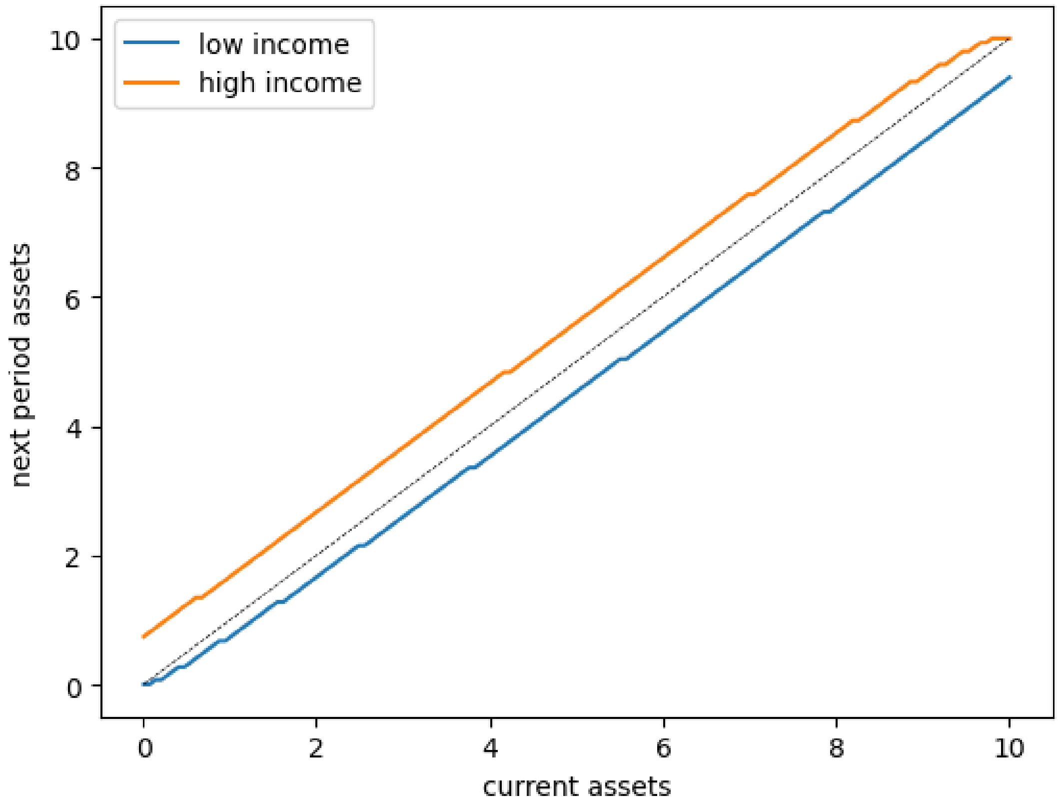
The following 45-degree diagram shows how assets evolve over time:

```
fig, ax = plt.subplots()

# Plot asset accumulation for first and last income states
for j, label in zip([0, -1], ['low income', 'high income']):
    # Get next-period assets for each current asset level
    a_next = model.a_grid[σ_star_jax[:, j]]
    ax.plot(model.a_grid, a_next, label=label)

# Add 45-degree line
```

(continues on next page)

(continued from previous page)

```
ax.plot(model.a_grid, model.a_grid, 'k--', linewidth=0.5)
ax.set(xlabel='current assets', ylabel='next period assets')
ax.legend()
plt.show()
```



The plot shows the asset accumulation rule for each income state.

The dotted line is the 45-degree line, representing points where $a_{t+1} = a_t$.

We see that:

- For low income levels, assets tend to decrease (points below the 45-degree line)
- For high income levels, assets tend to increase at low asset levels
- The dynamics suggest convergence to a stationary distribution

59.4 Exercises

Exercise 59.4.1

In this exercise, we explore an alternative approach to implementing value function iteration using `jax.vmap`.

For this simple optimal savings problem, direct vectorization is relatively easy.

In particular, it's straightforward to express the right hand side of the Bellman equation as an array that stores evaluations of the function at every state and control.

However, for more complex models, direct vectorization can be much harder.

For this reason, it helps to have another approach to fast JAX implementations up our sleeves.

Your task is to implement a version that:

1. writes the right hand side of the Bellman operator as a function of individual states and controls, and
2. applies `jax.vmap` on the outside to achieve a parallelized solution.

Specifically:

1. Rewrite `B` to take indices `(i, j, ip)` corresponding to `(a, y, a')` and compute the Bellman equation for those specific indices.
2. Use `jax.vmap` successively to vectorize over all indices (use staged vmap as shown in earlier examples).
3. Implement `T_vmap` and `get_greedy_vmap` functions using the vectorized `B`.
4. Implement `value_iteration_vmap` using `jax.lax.while_loop`.
5. Test that your implementation produces the same results as the direct vectorization approach.
6. Compare the execution times of both approaches.

Solution

Here's one solution.

First let's rewrite `B` to work with individual indices:

```
def B(v, model, i, j, ip):
    """
    The right-hand side of the Bellman equation before maximization, which takes
    the form

        B(a, y, a') = u(Ra + y - a') + β Σ_y' v(a', y') Q(y, y')

    The indices are (i, j, ip) -> (a, y, a').
    """
    β, R, γ, a_grid, y_grid, Q = model
    a, y, ap  = a_grid[i], y_grid[j], a_grid[ip]
    c = R * a + y - ap
    EV = jnp.sum(v[ip, :] * Q[j, :])
    return jnp.where(c > 0, c**(1-γ)/(1-γ) + β * EV, -jnp.inf)
```

Now we successively apply `vmap` to simulate nested loops.

```
B_1    = jax.vmap(B,   in_axes=(None, None, None, None, 0))
B_2    = jax.vmap(B_1, in_axes=(None, None, None, 0,    None))
B_vmap = jax.vmap(B_2, in_axes=(None, None, 0,    None, None))
```

Here's the Bellman operator and the `get_greedy` functions for the `vmap` case.

```
@jax.jit
def T_vmap(v, model):
    "The Bellman operator."
    a_indices = jnp.arange(len(model.a_grid))
    y_indices = jnp.arange(len(model.y_grid))
    B_values = B_vmap(v, model, a_indices, y_indices, a_indices)
    return jnp.max(B_values, axis=-1)

@jax.jit
def get_greedy_vmap(v, model):
    "Computes a v-greedy policy, returned as a set of indices."
    a_indices = jnp.arange(len(model.a_grid))
    y_indices = jnp.arange(len(model.y_grid))
    B_values = B_vmap(v, model, a_indices, y_indices, a_indices)
    return jnp.argmax(B_values, axis=-1)
```

Here's the iteration routine.

```
def value_iteration_vmap(model, tol=1e-5, max_iter=10_000):
    """
    Implements VFI using vmap and successive approximation.
    """
    def body_fun(k_v_err):
        k, v, error = k_v_err
        v_new = T_vmap(v, model)
        error = jnp.max(jnp.abs(v_new - v))
        return k + 1, v_new, error

    def cond_fun(k_v_err):
        k, v, error = k_v_err
        return jnp.logical_and(error > tol, k < max_iter)

    v_init = jnp.zeros((len(model.a_grid), len(model.y_grid)))
    k, v_star, error = jax.lax.while_loop(cond_fun, body_fun,
                                          (1, v_init, tol + 1))
    return v_star, get_greedy_vmap(v_star, model)
```

Let's see how long it takes to solve the model using the `vmap` method.

```
print("Starting VFI using vmap.")
start = time()
v_star_vmap, σ_star_vmap = value_iteration_vmap(model)
v_star_vmap.block_until_ready()
jax_vmap_with_compile = time() - start
print(f"VFI completed in {jax_vmap_with_compile} seconds.")
```

```
Starting VFI using vmap.
VFI completed in 0.9995379447937012 seconds.
```

Let's run it again to get rid of compile time.

```
start = time()
v_star_vmap, σ_star_vmap = value_iteration_vmap(model)
v_star_vmap.block_until_ready()
jax_vmap_without_compile = time() - start
print(f"VFI completed in {jax_vmap_without_compile} seconds.")
```

```
VFI completed in 0.4652867317199707 seconds.
```

We need to make sure that we got the same result.

```
print(jnp.allclose(v_star_vmap, v_star_jax))
print(jnp.allclose(σ_star_vmap, σ_star_jax))
```

```
True
True
```

Here's the comparison with the first JAX implementation (which used direct vectorization).

```
print(f"Relative speed = {jax_without_compile / jax_vmap_without_compile}")
```

```
Relative speed = 2.9370301821010334
```

The execution times for the two JAX versions are relatively similar.

However, as emphasized above, having a second method up our sleeves (i.e, the `vmap` approach) will be helpful when confronting dynamic programs with more sophisticated Bellman equations.

CHAPTER

SIXTY

THE INCOME FLUCTUATION PROBLEM II: OPTIMISTIC POLICY ITERATION

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

60.1 Overview

In *The Income Fluctuation Problem I: Discretization and VFI* we studied the income fluctuation problem and solved it using value function iteration (VFI).

In this lecture we'll solve the same problem using **optimistic policy iteration** (OPI), which is very general, typically faster than VFI and only slightly more complex.

OPI combines elements of both value function iteration and policy iteration.

A detailed discussion of the algorithm can be found in DP1.

Here our aim is to implement OPI and test whether or not it yields significant speed improvements over standard VFI for the income fluctuation problem.

In addition to Anaconda, this lecture will need the following libraries:

```
!pip install quantecon jax
```

We will use the following imports:

```
import quantecon as qe
import jax
import jax.numpy as jnp
import matplotlib.pyplot as plt
from typing import NamedTuple
from time import time
```

60.2 Model and Primitives

The model and parameters are the same as in *The Income Fluctuation Problem I: Discretization and VFI*.

We repeat the key elements here for convenience.

The household's problem is to maximize

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t u(c_t)$$

subject to

$$a_{t+1} + c_t \leq R a_t + y_t$$

where $u(c) = c^{1-\gamma}/(1-\gamma)$.

Here's the model structure:

```
class Model(NamedTuple):
    β: float              # Discount factor
    R: float              # Gross interest rate
    γ: float              # CRRA parameter
    a_grid: jnp.ndarray   # Asset grid
    y_grid: jnp.ndarray   # Income grid
    Q: jnp.ndarray        # Markov matrix for income


def create_consumption_model(
        R=1.01,                      # Gross interest rate
        β=0.98,                      # Discount factor
        γ=2,                         # CRRA parameter
        a_min=0.01,                  # Min assets
        a_max=10.0,                  # Max assets
        a_size=150,                  # Grid size
        ρ=0.9, ν=0.1, y_size=100     # Income parameters
    ):
    """
    Creates an instance of the consumption-savings model.

    """
    a_grid = jnp.linspace(a_min, a_max, a_size)
    mc = qe.tauchen(n=y_size, rho=ρ, sigma=ν)
    y_grid, Q = jnp.exp(mc.state_values), jax.device_put(mc.P)
    return Model(β, R, γ, a_grid, y_grid, Q)
```

60.3 Operators and Policies

We repeat some functions from *The Income Fluctuation Problem I: Discretization and VFI*.

Here is the right hand side of the Bellman equation:

```
def B(v, model, i, j, ip):
    """
    The right-hand side of the Bellman equation before maximization, which takes
    the form
```

(continues on next page)

(continued from previous page)

```
        B(a, y, a') = u(Ra + y - a') + β Σ_y' v(a', y') Q(y, y')

    The indices are (i, j, ip) -> (a, y, a').
    """
    β, R, γ, a_grid, y_grid, Q = model
    a, y, ap  = a_grid[i], y_grid[j], a_grid[ip]
    c = R * a + y - ap
    EV = jnp.sum(v[ip, :] * Q[j, :])
    return jnp.where(c > 0, c**(1-γ)/(1-γ) + β * EV, -jnp.inf)
```

Now we successively apply `vmap` to vectorize over all indices:

```
B_1    = jax.vmap(B,   in_axes=(None, None, None, None, 0))
B_2    = jax.vmap(B_1, in_axes=(None, None, None, 0,    None))
B_vmap = jax.vmap(B_2, in_axes=(None, None, 0,    None, None))
```

Here's the Bellman operator:

```
def T(v, model):
    "The Bellman operator."
    a_indices = jnp.arange(len(model.a_grid))
    y_indices = jnp.arange(len(model.y_grid))
    B_values = B_vmap(v, model, a_indices, y_indices, a_indices)
    return jnp.max(B_values, axis=-1)
```

Here's the function that computes a v-greedy policy:

```
def get_greedy(v, model):
    "Computes a v-greedy policy, returned as a set of indices."
    a_indices = jnp.arange(len(model.a_grid))
    y_indices = jnp.arange(len(model.y_grid))
    B_values = B_vmap(v, model, a_indices, y_indices, a_indices)
    return jnp.argmax(B_values, axis=-1)
```

Now we define the policy operator T_σ, which is the Bellman operator with policy σ fixed.

For a given policy σ, the policy operator is defined by

$$(T_\sigma v)(a, y) = u(Ra + y - \sigma(a, y)) + \beta \sum_{y'} v(\sigma(a, y), y') Q(y, y')$$

```
def T_σ(v, σ, model, i, j):
    """
    The σ-policy operator for indices (i, j) -> (a, y).
    """
    β, R, γ, a_grid, y_grid, Q = model

    # Get values at current state
    a, y = a_grid[i], y_grid[j]
    # Get policy choice
    ap = a_grid[σ[i, j]]

    # Compute current reward
    c = R * a + y - ap
    r = jnp.where(c > 0, c**(1-γ)/(1-γ), -jnp.inf)
```

(continues on next page)

(continued from previous page)

```
    # Compute expected value
    EV = jnp.sum(v[σ[i, j], :] * Q[j, :])

    return r + β * EV
```

Apply vmap to vectorize:

```
T_σ_1    = jax.vmap(T_σ,   in_axes=(None, None, None, None, 0))
T_σ_vmap = jax.vmap(T_σ_1, in_axes=(None, None, None, 0,    None))

def T_σ_vec(v, σ, model):
    """Vectorized version of T_σ."""
    a_size, y_size = len(model.a_grid), len(model.y_grid)
    a_indices = jnp.arange(a_size)
    y_indices = jnp.arange(y_size)
    return T_σ_vmap(v, σ, model, a_indices, y_indices)
```

Now we need a function to apply the policy operator m times:

```
def iterate_policy_operator(σ, v, m, model):
    """
    Apply the policy operator T_σ exactly m times to v.
    """
    def update(i, v):
        return T_σ_vec(v, σ, model)

    v = jax.lax.fori_loop(0, m, update, v)
    return v
```

60.4 Value Function Iteration

For comparison, here's VFI from *The Income Fluctuation Problem I: Discretization and VFI*:

```
@jax.jit
def value_function_iteration(model, tol=1e-5, max_iter=10_000):
    """
    Implements VFI using successive approximation.
    """
    def body_fun(k_v_err):
        k, v, error = k_v_err
        v_new = T(v, model)
        error = jnp.max(jnp.abs(v_new - v))
        return k + 1, v_new, error

    def cond_fun(k_v_err):
        k, v, error = k_v_err
        return jnp.logical_and(error > tol, k < max_iter)

    v_init = jnp.zeros((len(model.a_grid), len(model.y_grid)))
    k, v_star, error = jax.lax.while_loop(cond_fun, body_fun,
                                          (1, v_init, tol + 1))
    return v_star, get_greedy(v_star, model)
```

60.5 Optimistic Policy Iteration

Now we implement OPI.

The algorithm alternates between

1. Performing m policy operator iterations to update the value function
2. Computing a new greedy policy based on the updated value function

```
@jax.jit
def optimistic_policy_iteration(model, m=10, tol=1e-5, max_iter=10_000):
    """
    Implements optimistic policy iteration with step size m.

    Parameters:
    -----------
    model : Model
        The consumption-savings model
    m : int
        Number of policy operator iterations per step
    tol : float
        Tolerance for convergence
    max_iter : int
        Maximum number of iterations
    """
    v_init = jnp.zeros((len(model.a_grid), len(model.y_grid)))

    def condition_function(inputs):
        i, v, error = inputs
        return jnp.logical_and(error > tol, i < max_iter)

    def update(inputs):
        i, v, error = inputs
        last_v = v
        σ = get_greedy(v, model)
        v = iterate_policy_operator(σ, v, m, model)
        error = jnp.max(jnp.abs(v - last_v))
        i += 1
        return i, v, error

    num_iter, v, error = jax.lax.while_loop(condition_function,
                                            update,
                                            (0, v_init, tol + 1))

    return v, get_greedy(v, model)
```

60.6 Timing Comparison

Let's create a model and compare the performance of VFI and OPI.

```
model = create_consumption_model()
```

First, let's time VFI:

```
print("Starting VFI.")
start = time()
v_star_vfi, σ_star_vfi = value_function_iteration(model)
v_star_vfi.block_until_ready()
vfi_time_with_compile = time() - start
print(f"VFI completed in {vfi_time_with_compile:.2f} seconds.")
```

```
Starting VFI.
VFI completed in 0.82 seconds.
```

Run it again to eliminate compile time:

```
start = time()
v_star_vfi, σ_star_vfi = value_function_iteration(model)
v_star_vfi.block_until_ready()
vfi_time = time() - start
print(f"VFI completed in {vfi_time:.2f} seconds.")
```

```
VFI completed in 0.08 seconds.
```

Now let's time OPI with different values of m:

```
print("Starting OPI with m=50.")
start = time()
v_star_opi, σ_star_opi = optimistic_policy_iteration(model, m=50)
v_star_opi.block_until_ready()
opi_time_with_compile = time() - start
print(f"OPI completed in {opi_time_with_compile:.2f} seconds.")
```

```
Starting OPI with m=50.
OPI completed in 0.53 seconds.
```

Run it again:

```
start = time()
v_star_opi, σ_star_opi = optimistic_policy_iteration(model, m=50)
v_star_opi.block_until_ready()
opi_time = time() - start
print(f"OPI completed in {opi_time:.2f} seconds.")
```

```
OPI completed in 0.03 seconds.
```

Check that we get the same result:

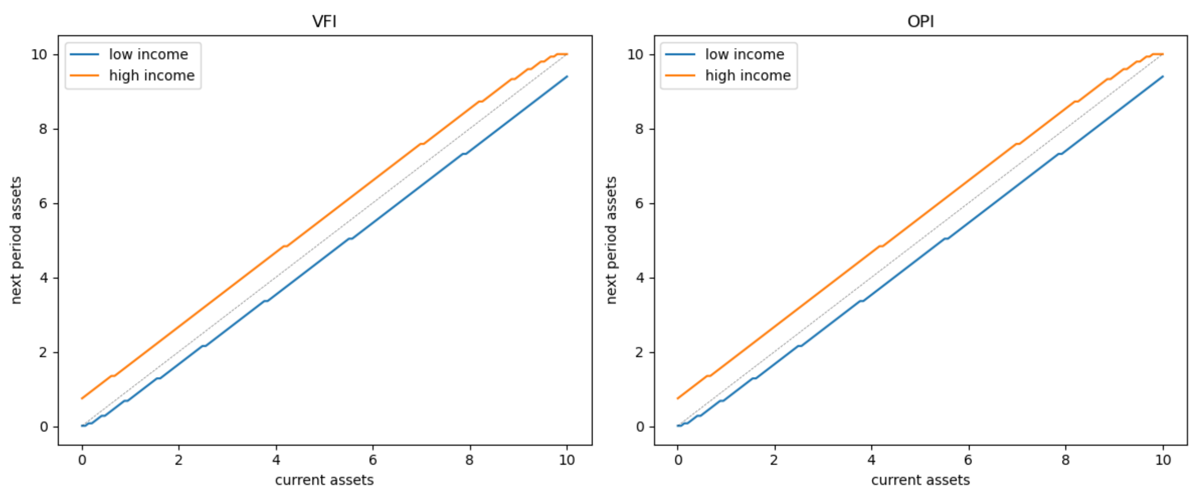
```
print(f"Values match: {jnp.allclose(v_star_vfi, v_star_opi)}")
```

```
Values match: True
```

The value functions match, confirming both algorithms converge to the same solution.

Let's visually compare the asset dynamics under both policies:

```
fig, axes = plt.subplots(1, 2, figsize=(12, 5))

# VFI policy
for j, label in zip([0, -1], ['low income', 'high income']):
    a_next_vfi = model.a_grid[σ_star_vfi[:, j]]
    axes[0].plot(model.a_grid, a_next_vfi, label=label)
axes[0].plot(model.a_grid, model.a_grid, 'k--', linewidth=0.5, alpha=0.5)
axes[0].set(xlabel='current assets', ylabel='next period assets', title='VFI')
axes[0].legend()

# OPI policy
for j, label in zip([0, -1], ['low income', 'high income']):
    a_next_opi = model.a_grid[σ_star_opi[:, j]]
    axes[1].plot(model.a_grid, a_next_opi, label=label)
axes[1].plot(model.a_grid, model.a_grid, 'k--', linewidth=0.5, alpha=0.5)
axes[1].set(xlabel='current assets', ylabel='next period assets', title='OPI')
axes[1].legend()

plt.tight_layout()
plt.show()
```



The policies are visually indistinguishable, confirming both methods produce the same solution.

Here's the speedup:

```
print(f"Speedup factor: {vfi_time / opi_time:.2f}")
```

```
Speedup factor: 2.46
```

Let's try different values of m to see how it affects performance:

```
m_vals = [1, 5, 10, 25, 50, 100, 200, 400]
opi_times = []
```

(continues on next page)

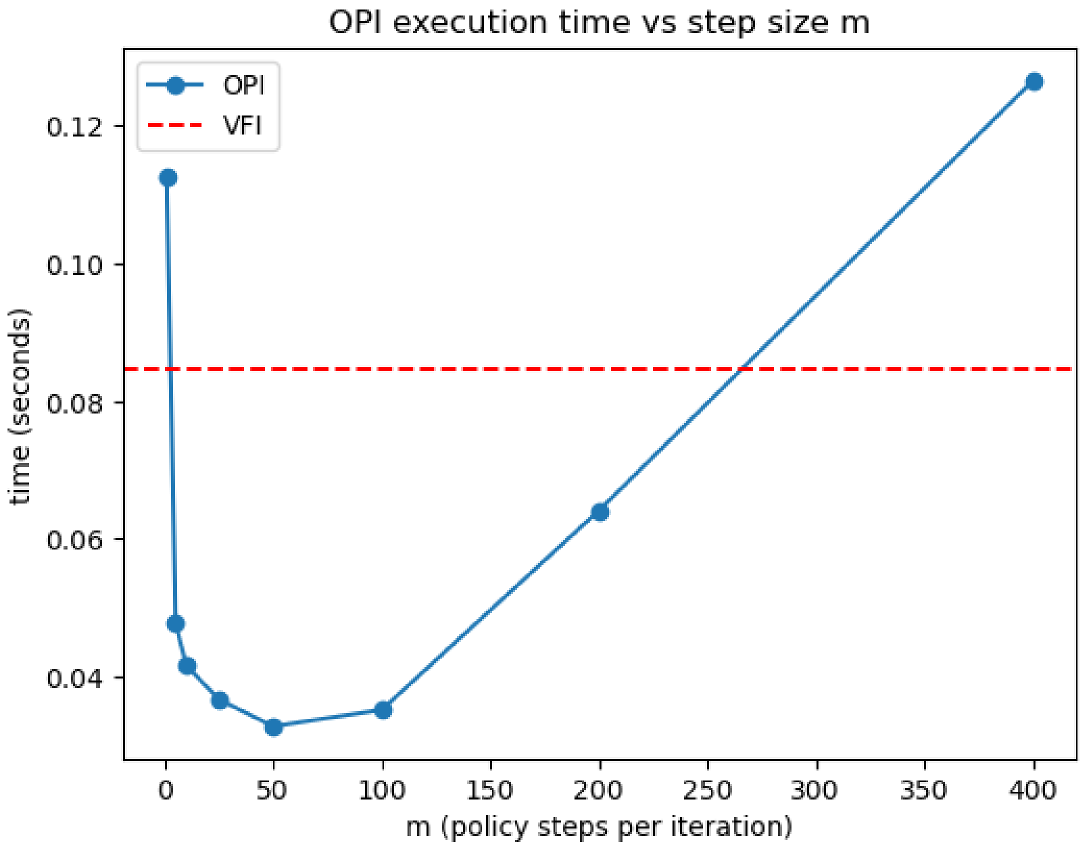
(continued from previous page)

```
for m in m_vals:
    start = time()
    v_star, σ_star = optimistic_policy_iteration(model, m=m)
    v_star.block_until_ready()
    elapsed = time() - start
    opi_times.append(elapsed)
    print(f"OPI with m={m:3d} completed in {elapsed:.2f} seconds.")
```

```
OPI with m=  1 completed in 0.11 seconds.
OPI with m=  5 completed in 0.05 seconds.
OPI with m= 10 completed in 0.04 seconds.
OPI with m= 25 completed in 0.04 seconds.
OPI with m= 50 completed in 0.03 seconds.
OPI with m=100 completed in 0.04 seconds.
OPI with m=200 completed in 0.06 seconds.
OPI with m=400 completed in 0.13 seconds.
```

Plot the results:

```
fig, ax = plt.subplots()
ax.plot(m_vals, opi_times, 'o-', label='OPI')
ax.axhline(vfi_time, linestyle='--', color='red', label='VFI')
ax.set_xlabel('m (policy steps per iteration)')
ax.set_ylabel('time (seconds)')
ax.legend()
ax.set_title('OPI execution time vs step size m')
plt.show()
```



Here's a summary of the results

- OPI outperforms VFI for a large range of m values.
- For very large m, OPI performance begins to degrade as we spend too much time iterating the policy operator.

60.7 Exercises

Exercise 60.7.1

The speed gains achieved by OPI are quite robust to parameter changes.

Confirm this by experimenting with different parameter values for the income process (ρ and ν).

Measure how they affect the relative performance of VFI vs OPI.

Try:

- $\rho \in \{0.8, 0.9, 0.95\}$
- $\nu \in \{0.05, 0.1, 0.2\}$

For each combination, compute the speedup factor (VFI time / OPI time) and report your findings.

Solution

Here's one solution:

```
ρ_vals = [0.8, 0.9, 0.95]
ν_vals = [0.05, 0.1, 0.2]

results = []

for ρ in ρ_vals:
    for ν in ν_vals:
        print(f"\nTesting ρ={ρ}, ν={ν}")

        # Create model
        model = create_consumption_model(ρ=ρ, ν=ν)

        # Time VFI
        start = time()
        v_vfi, σ_vfi = value_function_iteration(model)
        v_vfi.block_until_ready()
        vfi_t = time() - start

        # Time OPI
        start = time()
        v_opi, σ_opi = optimistic_policy_iteration(model, m=10)
        v_opi.block_until_ready()
        opi_t = time() - start

        speedup = vfi_t / opi_t
        results.append((ρ, ν, speedup))
        print(f"  VFI: {vfi_t:.2f}s, OPI: {opi_t:.2f}s, Speedup: {speedup:.2f}x")

# Print summary
print("\nSummary of speedup factors:")
for ρ, ν, speedup in results:
    print(f"ρ={ρ}, ν={ν}: {speedup:.2f}x")
```

```
Testing ρ=0.8, ν=0.05
  VFI: 0.06s, OPI: 0.03s, Speedup: 1.67x

Testing ρ=0.8, ν=0.1
  VFI: 0.06s, OPI: 0.03s, Speedup: 1.78x

Testing ρ=0.8, ν=0.2
  VFI: 0.06s, OPI: 0.03s, Speedup: 1.72x

Testing ρ=0.9, ν=0.05
  VFI: 0.06s, OPI: 0.03s, Speedup: 1.76x

Testing ρ=0.9, ν=0.1
  VFI: 0.06s, OPI: 0.04s, Speedup: 1.71x

Testing ρ=0.9, ν=0.2
  VFI: 0.06s, OPI: 0.03s, Speedup: 1.77x

Testing ρ=0.95, ν=0.05
  VFI: 0.06s, OPI: 0.04s, Speedup: 1.68x

Testing ρ=0.95, ν=0.1
  VFI: 0.06s, OPI: 0.04s, Speedup: 1.73x

Testing ρ=0.95, ν=0.2
  VFI: 0.07s, OPI: 0.04s, Speedup: 1.85x

Summary of speedup factors:
ρ=0.8, ν=0.05: 1.67x
```

CHAPTER

SIXTYONE

THE INCOME FLUCTUATION PROBLEM III: THE ENDOGENOUS GRID METHOD

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

Contents

61.1 Overview

In this lecture we continue examining a version of the IFP from

- *The Income Fluctuation Problem I: Discretization and VFI* and
- *The Income Fluctuation Problem II: Optimistic Policy Iteration*.

We will make two changes.

1. Change the timing to one that is more efficient for our set up.
2. Use the endogenous grid method (EGM) to solve the model.

We use EGM because we know it to be fast and accurate from *Optimal Savings VI: EGM with JAX*.

The primary source for the technical details discussed below is [Ma *et al.*, 2020].

Other references include [Deaton, 1991], [Den Haan, 2010], [Kuhn, 2013], [Rabault, 2002], [Reiter, 2009] and [Schechtman and Escudero, 1977].

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon jax
```

We'll also need the following imports:

```
import matplotlib.pyplot as plt
import numpy as np
import numba
from quantecon import MarkovChain
import jax
import jax.numpy as jnp
from typing import NamedTuple
```

61.2 The Household Problem

Let's write down the model and then discuss how to solve it.

61.2.1 Set-Up

A household chooses a state-contingent consumption plan $\{c_t\}_{t\geq 0}$ to maximize

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t u(c_t)$$

subject to

$$a_{t+1} = R(a_t - c_t) + Y_{t+1} \quad c_t \geq 0, \quad a_t \geq 0 \quad t = 0, 1, \ldots \tag{61.1}$$

Here

- $\beta \in (0, 1)$ is the discount factor
- a_t is asset holdings at time t, with borrowing constraint $a_t \geq 0$
- c_t is consumption
- Y_t is non-capital income (wages, unemployment compensation, etc.)
- $R := 1 + r$, where $r > 0$ is the interest rate on savings

The timing here is as follows:

1. At the start of period t, the household observes current asset holdings a_t.
2. The household chooses current consumption c_t.
3. Savings $s_t := a_t - c_t$ earns interest at rate r.
4. Labor income Y_{t+1} is realized and time shifts to $t + 1$.

Non-capital income Y_t is given by $Y_t = y(Z_t)$, where

- $\{Z_t\}$ is an exogenous state process, and
- y is a function taking values in $\mathbb{R}_+$.

We take $\{Z_t\}$ to be a finite state Markov chain taking values in Z with Markov matrix Π.

Note

In previous lectures we used the more standard household budget constraint $a_{t+1} + c_t \leq Ra_t + Y_t$.

This setup, which is pervasive in quantitative economics, was developed for discretization.

It means that the control variable is also the next period state a_{t+1}, which makes it straightforward to restrict assets to a finite grid.

But fixing the control to be the next period state forces us to include more information in the current state, which expands the size of the state space.

Moreover, aiming for discretization is not always a good idea, since it suffers heavily from the curse of dimensionality.

These ideas will become clearer in the *next lecture*.

We further assume that

1. $\beta R < 1$
2. u is smooth, strictly increasing and strictly concave with $\lim_{c \to 0} u'(c) = \infty$ and $\lim_{c \to \infty} u'(c) = 0$
3. $y(z) = \exp(z)$

The asset space is $\mathbb{R}_+$ and the state is the pair $(a, z) \in \mathsf{S} := \mathbb{R}_+ \times \mathsf{Z}$.

A **feasible consumption path** from $(a, z) \in \mathsf{S}$ is a consumption sequence $\{c_t\}$ such that $\{c_t\}$ and its induced asset path $\{a_t\}$ satisfy

1. $(a_0, z_0) = (a, z)$
2. the feasibility constraints in (61.1), and
3. adaptedness, which means that c_t is a function of random outcomes up to date t but not after.

The meaning of the third point is just that consumption at time t cannot be a function of outcomes are yet to be observed.

In fact, for this problem, consumption can be chosen optimally by taking it to be contingent only on the current state.

Optimality is defined below.

61.2.2 Value Function and Euler Equation

The **value function** $V : \mathsf{S} \to \mathbb{R}$ is defined by

$$V(a, z) := \max \mathbb{E} \left\{ \sum_{t=0}^{\infty} \beta^t u(c_t) \right\} \tag{61.2}$$

where the maximization is overall feasible consumption paths from (a, z).

An **optimal consumption path** from (a, z) is a feasible consumption path from (a, z) that maximizes (59.1).

To pin down such paths we can use a version of the Euler equation, which in the present setting is

$$u'(c_t) \geq \beta R \, \mathbb{E}_t u'(c_{t+1}) \tag{61.3}$$

with

$$c_t < a_t \implies u'(c_t) = \beta R \, \mathbb{E}_t u'(c_{t+1}) \tag{61.4}$$

When c_t hits the upper bound a_t, the strict inequality $u'(c_t) > \beta R \, \mathbb{E}_t u'(c_{t+1})$ can occur because c_t cannot increase sufficiently to attain equality.

The case $c_t = 0$ never arises along the optimal path because $u'(0) = \infty$.

61.2.3 Optimality Results

As shown in [Ma *et al.*, 2020],

1. For each $(a, z) \in \mathsf{S}$, a unique optimal consumption path from (a, z) exists
2. This path is the unique feasible path from (a, z) satisfying the Euler equations (61.3)-(61.4) and the transversality condition

$$\lim_{t \to \infty} \beta^t \, \mathbb{E} \left[u'(c_t) a_{t+1} \right] = 0 \tag{61.5}$$

Moreover, there exists an **optimal consumption policy** $\sigma^* \colon \mathsf{S} \to \mathbb{R}_+$ such that the path from (a, z) generated by

$$(a_0, z_0) = (a, z), \quad c_t = \sigma^*(a_t, Z_t) \quad \text{and} \quad a_{t+1} = R(a_t - c_t) + Y_{t+1}$$

satisfies both the Euler equations (61.3)-(61.4) and (61.5), and hence is the unique optimal path from (a, z).

Thus, to solve the optimization problem, we need to compute the policy σ^*.

61.3 Computation

We solve for the optimal consumption policy using time iteration and the endogenous grid method, which were previously discussed in

- *Optimal Savings IV: Time Iteration*
- *Optimal Savings V: The Endogenous Grid Method*

61.3.1 Solution Method

We rewrite (61.4) to make it a statement about functions rather than random variables:

$$(u' \circ \sigma)(a, z) = \beta R \sum_{z'} (u' \circ \sigma)[R(a - \sigma(a, z)) + y(z'), \, z'] \, \Pi(z, z') \tag{61.6}$$

Here

- $(u' \circ \sigma)(s) := u'(\sigma(s))$,
- primes indicate next period states (as well as derivatives), and
- σ is the unknown function.

The equality (61.6) holds at all interior choices, meaning $\sigma(a, z) < a$.

We aim to find a fixed point σ of (61.6).

To do so we use the EGM.

Below we use the relationships $a_t = c_t + s_t$ and $a_{t+1} = R s_t + Y_{t+1}$.

We begin with an exogenous savings grid $s_0 < s_1 < \cdots < s_m$ with $s_0 = 0$.

We fix a current guess of the policy function σ.

For each exogenous savings level s_i with $i \geq 1$ and current state z_j, we set

$$c_{ij} := (u')^{-1} \left[\beta R \sum_{z'} u'[\sigma(R s_i + y(z'), z')] \, \Pi(z_j, z') \right] \tag{61.7}$$

The Euler equation holds here because $i \geq 1$ implies $s_i > 0$ and hence consumption is interior.

For the boundary case $s_0 = 0$ we set

$$c_{0j} := 0 \quad \text{for all j}$$

We then obtain a corresponding endogenous grid of current assets via

$$a_{ij} := c_{ij} + s_i.$$

Notice that, for each j, we have $a_{0j} = c_{0j} = 0$.

This anchors the interpolation at the correct value at the origin, since, without borrowing, consumption is zero when assets are zero.

Our next guess of the policy function, which we write as $K\sigma$, is the linear interpolation of the interpolation points

$$\{(a_{0j}, c_{0j}), \ldots, (a_{mj}, c_{mj})\}$$

for each j.

(The number of one-dimensional linear interpolations is equal to the size of Z.)

61.4 NumPy Implementation

In this section we'll code up a NumPy version of the code that aims only for clarity, rather than efficiency.

Once we have it working, we'll produce a JAX version that's far more efficient and check that we obtain the same results.

We use the CRRA utility specification

$$u(c) = \frac{c^{1-\gamma}}{1-\gamma}$$

61.4.1 Set Up

Here we build a class called `IFPNumPy` that stores the model primitives.

The exogenous state process $\{Z_t\}$ defaults to a two-state Markov chain with transition matrix Π.

```
class IFPNumPy(NamedTuple):
    R: float                    # Gross interest rate R = 1 + r
    β: float                    # Discount factor
    γ: float                    # Preference parameter
    Π: np.ndarray               # Markov matrix for exogenous shock
    z_grid: np.ndarray          # Markov state values for Z_t
    s: np.ndarray               # Exogenous savings grid
```

(continues on next page)

(continued from previous page)

```
def create_ifp(r=0.01,
               β=0.96,
               γ=1.5,
               Π=((0.6, 0.4),
                  (0.05, 0.95)),
               z_grid=(-10.0, np.log(2.0)),
               savings_grid_max=16,
               savings_grid_size=200):

    s = np.linspace(0, savings_grid_max, savings_grid_size)
    Π, z_grid = np.array(Π), np.array(z_grid)
    R = 1 + r
    assert R * β < 1, "Stability condition violated."
    return IFPNumPy(R, β, γ, Π, z_grid, s)
```

61.4.2 Solver

Here is the operator K that transforms current guess σ into next period guess $K\sigma$.

In practice, it takes in

- a guess of optimal consumption values c_{ij}, stored as `c_vec`
- and a corresponding set of endogenous grid points a^e_{ij}, stored as `a_vec`

These are converted into a consumption policy $a \mapsto \sigma(a, z_j)$ by linear interpolation of (a^e_{ij}, c_{ij}) over i for each j.

Since there are no shocks to integrate out in this version of the model, we can compute (61.7) directly by summing over the finite state space Z.

```
@numba.jit
def K_numpy(
        c_in: np.ndarray,    # Initial guess of σ on grid endogenous grid
        a_in: np.ndarray,    # Initial endogenous grid
        ifp_numpy: IFPNumPy
    ) -> np.ndarray:
    """
    The Euler equation operator for the IFP model using the
    Endogenous Grid Method.

    This operator implements one iteration of the EGM algorithm to
    update the consumption policy function.

    """
    R, β, γ, Π, z_grid, s = ifp_numpy
    n_a, n_z = len(s), len(z_grid)
    c_out = np.zeros_like(c_in)
    u_prime = lambda c: c**(-γ)
    u_prime_inv = lambda c: c**(-1/γ)
    y = lambda z: np.exp(z)

    for i in range(1, n_a):  # Start from 1 for positive savings levels
        for j in range(n_z):

            # Compute Σ_z' u'(σ(R s_i + y(z'), z')) Π[z_j, z']
```

(continues on next page)

(continued from previous page)

```
            expectation = 0.0
            for k in range(n_z):
                z_prime = z_grid[k]
                # Calculate next period assets
                next_a = R * s[i] + y(z_prime)
                # Interpolate to get σ(R s_i + y(z'), z')
                next_c = np.interp(next_a, a_in[:, k], c_in[:, k])
                # Weight by transition probability and add to the expectation
                expectation += u_prime(next_c) * Π[j, k]

            # Calculate updated c_{ij} values
            c_out[i, j] = u_prime_inv(β * R * expectation)

    a_out = c_out + s[:, None]
    return c_out, a_out
```

To solve the model we use a simple while loop.

```
def solve_model_numpy(
        ifp_numpy: IFPNumPy,
        c_init: np.ndarray,
        a_init: np.ndarray,
        tol: float = 1e-5,
        max_iter: int = 1_000
    ) -> np.ndarray:
    """
    Solve the model using time iteration with EGM.

    """
    c_in, a_in = c_init, a_init
    i = 0
    error = tol + 1

    while error > tol and i < max_iter:
        c_out, a_out = K_numpy(c_in, a_in, ifp_numpy)
        error = np.max(np.abs(c_out - c_in))
        i = i + 1
        c_in, a_in = c_out, a_out

    return c_out, a_out
```

Let's road test the EGM code.

```
ifp_numpy = create_ifp()
R, β, γ, Π, z_grid, s = ifp_numpy
# Initial conditions -- agent consumes everything
a_init = s[:, None] * np.ones(len(z_grid))
c_init = a_init
# Solve from these initial conditions
c_vec, a_vec = solve_model_numpy(
    ifp_numpy, c_init, a_init
)
```

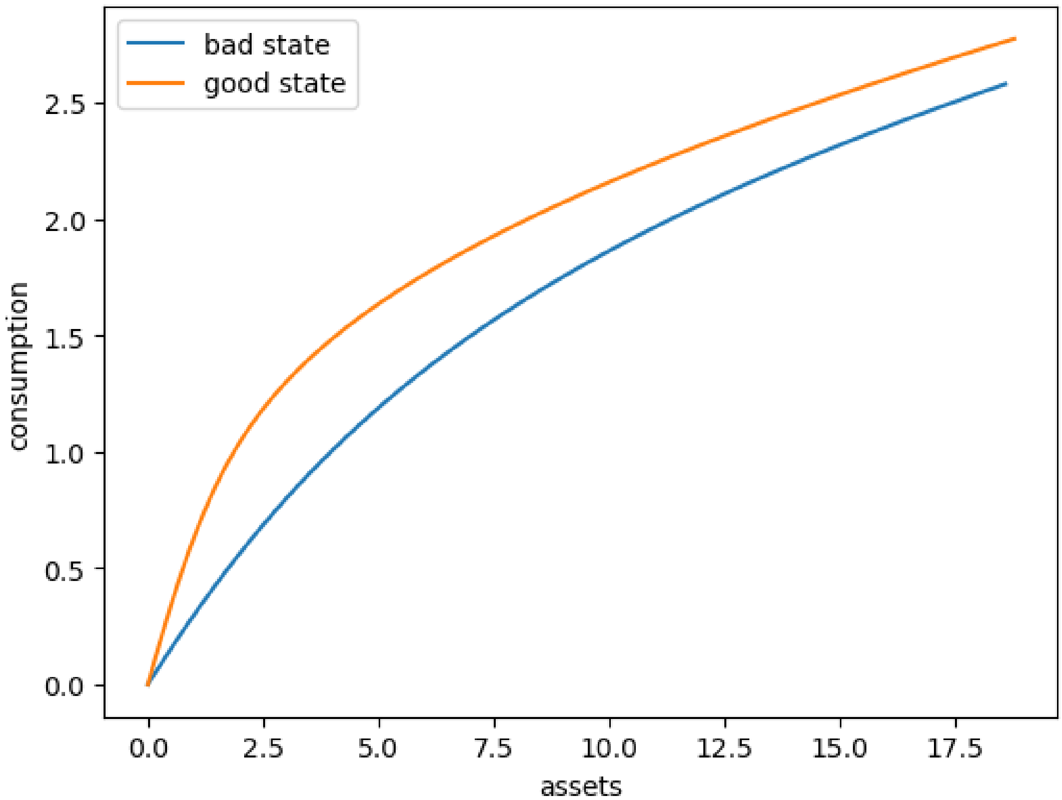
Here's a plot of the optimal consumption policy for each z state

```
fig, ax = plt.subplots()
```

(continues on next page)

(continued from previous page)

```
ax.plot(a_vec[:, 0], c_vec[:, 0], label='bad state')
ax.plot(a_vec[:, 1], c_vec[:, 1], label='good state')
ax.set(xlabel='assets', ylabel='consumption')
ax.legend()
plt.show()
```



61.5 JAX Implementation

Now we write a more efficient JAX version, which can run on a GPU.

61.5.1 Set Up

We start with a class called `IFP` that stores the model primitives.

```
class IFP(NamedTuple):
    R: float                        # Gross interest rate R = 1 + r
    β: float                        # Discount factor
    γ: float                        # Preference parameter
    Π: jnp.ndarray                  # Markov matrix for exogenous shock
    z_grid: jnp.ndarray             # Markov state values for Z_t
    s: jnp.ndarray                  # Exogenous savings grid


def create_ifp(r=0.01,
```

(continues on next page)

(continued from previous page)

```
              β=0.94,
              γ=1.5,
              Π=((0.6, 0.4),
                 (0.05, 0.95)),
              z_grid=(-10.0, jnp.log(2.0)),
              savings_grid_max=16,
              savings_grid_size=200):

    s = jnp.linspace(0, savings_grid_max, savings_grid_size)
    Π, z_grid = jnp.array(Π), jnp.array(z_grid)
    R = 1 + r
    assert R * β < 1, "Stability condition violated."
    return IFP(R, β, γ, Π, z_grid, s)
```

61.5.2 Solver

Here is the operator K that transforms current guess σ into next period guess $K\sigma$.

```
def K(
        c_in: jnp.ndarray,
        a_in: jnp.ndarray,
        ifp: IFP
    ) -> jnp.ndarray:
    """
    The Euler equation operator for the IFP model using the
    Endogenous Grid Method.

    This operator implements one iteration of the EGM algorithm to
    update the consumption policy function.

    """
    R, β, γ, Π, z_grid, s = ifp
    n_z = len(z_grid)
    z_indices = jnp.arange(n_z)
    u_prime = lambda c: c**(-γ)
    u_prime_inv = lambda c: c**(-1/γ)
    y = lambda z: jnp.exp(z)

    def compute_c(i, j):
        " Computes consumption for one (i, j) pair where i >= 1. "

        def compute_mu_k(k):
            " Given i, compute marginal utility u'(σ(R s_i + y(z_k), z_k)) "
            next_a = R * s[i] + y(z_grid[k])
            # Interpolate to get σ(R * s_i + y(z_k), z_k)
            next_c = jnp.interp(next_a, a_in[:, k], c_in[:, k])
            # Return u'(σ(R * s_i + y(z_k), z_k))
            return u_prime(next_c)

        # Compute marginal utility u'(σ(R * s_i + y(z_k), z_k)) for all k
        mu_values = jax.vmap(compute_mu_k)(z_indices)
        # Compute expectation Σ_k u'(σ(...)) * Π[j, k]
        expectation = jnp.sum(mu_values * Π[j, :])
        # Invert to get consumption c_{ij} at (s_i, z_j)
        return u_prime_inv(β * R * expectation)
```

(continues on next page)

(continued from previous page)

```
    # vmap over j for each i
    compute_c_i = jax.vmap(compute_c, in_axes=(None, 0))
    # vmap over i
    compute_c = jax.vmap(lambda i: compute_c_i(i, z_indices))
    # Compute consumption for i >= 1
    c_out_interior = compute_c(jnp.arange(1, len(s)))
    # For i = 0, set consumption to 0
    c_out_boundary = jnp.zeros((1, n_z))
    # Concatenate boundary and interior
    c_out = jnp.concatenate([c_out_boundary, c_out_interior], axis=0)
    # Compute endogenous asset grid a_{ij} = c_{ij} + s_i
    a_out = c_out + s[:, None]
    return c_out, a_out
```

Here's a jit-accelerated iterative routine to solve the model using this operator.

```
@jax.jit
def solve_model(
        ifp: IFP,
        c_init: jnp.ndarray,  # Initial guess of σ on grid endogenous grid
        a_init: jnp.ndarray,  # Initial endogenous grid
        tol: float = 1e-5,
        max_iter: int = 1000
    ) -> jnp.ndarray:
    """
    Solve the model using time iteration with EGM.

    """

    def condition(loop_state):
        c_in, a_in, i, error = loop_state
        return (error > tol) & (i < max_iter)

    def body(loop_state):
        c_in, a_in, i, error = loop_state
        c_out, a_out = K(c_in, a_in, ifp)
        error = jnp.max(jnp.abs(c_out - c_in))
        i += 1
        return c_out, a_out, i, error

    i, error = 0, tol + 1
    initial_state = (c_init, a_init, i, error)
    final_loop_state = jax.lax.while_loop(condition, body, initial_state)
    c_out, a_out, i, error = final_loop_state

    return c_out, a_out
```

61.5.3 Test run

Let's road test the EGM code.

```
ifp = create_ifp()
R, β, γ, Π, z_grid, s = ifp
# Set initial conditions where the agent consumes everything
a_init = s[:, None] * jnp.ones(len(z_grid))
c_init = a_init
# Solve starting from these initial conditions
c_vec_jax, a_vec_jax = solve_model(ifp, c_init, a_init)
```

To verify the correctness of our JAX implementation, let's compare it with the NumPy version we developed earlier.

```
# Compare the results
max_c_diff = np.max(np.abs(np.array(c_vec) - c_vec_jax))
max_ae_diff = np.max(np.abs(np.array(a_vec) - a_vec_jax))

print(f"Maximum difference in consumption policy: {max_c_diff:.2e}")
print(f"Maximum difference in asset grid:        {max_ae_diff:.2e}")
```

```
Maximum difference in consumption policy: 4.45e-01
Maximum difference in asset grid:        4.45e-01
```

These numbers confirm that we are computing essentially the same policy using the two approaches.

61.5.4 Timing

Now let's compare the execution time between NumPy and JAX implementations.

```
import time

# Set up initial conditions for NumPy version
s_np = np.array(s)
z_grid_np = np.array(z_grid)
a_init_np = s_np[:, None] * np.ones(len(z_grid_np))
c_init_np = a_init_np.copy()

# Set up initial conditions for JAX version
a_init_jx = s[:, None] * jnp.ones(len(z_grid))
c_init_jx = a_init_jx

# Time NumPy version
start = time.time()
c_vec_np, a_vec_np = solve_model_numpy(ifp_numpy, c_init_np, a_init_np)
numpy_time = time.time() - start

# Time JAX version (with compilation)
start = time.time()
c_vec_jx, a_vec_jx = solve_model(ifp, c_init_jx, a_init_jx)
c_vec_jx.block_until_ready()
jax_time_with_compile = time.time() - start

# Time JAX version (without compilation - second run)
start = time.time()
c_vec_jx, a_vec_jx = solve_model(ifp, c_init_jx, a_init_jx)
```

(continues on next page)

(continued from previous page)

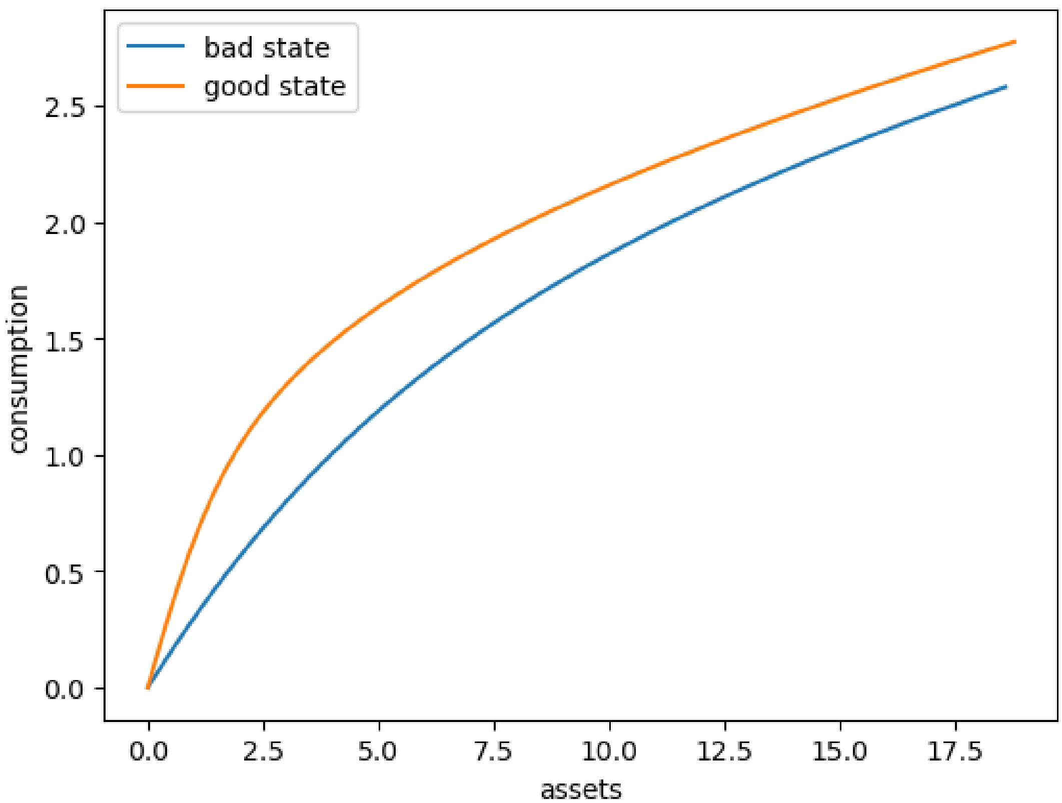
```
c_vec_jx.block_until_ready()
jax_time = time.time() - start

print(f"NumPy time:                {numpy_time:.4f} seconds")
print(f"JAX time (with compile):    {jax_time_with_compile:.4f} seconds")
print(f"JAX time (without compile): {jax_time:.4f} seconds")
print(f"Speedup (NumPy/JAX):        {numpy_time/jax_time:.2f}x")
```

```
NumPy time:                0.0212 seconds
JAX time (with compile):    0.0097 seconds
JAX time (without compile): 0.0096 seconds
Speedup (NumPy/JAX):        2.21x
```

The JAX implementation is faster due to JIT compilation and GPU/TPU acceleration (if available).

Here's a plot of the optimal policy for each z state

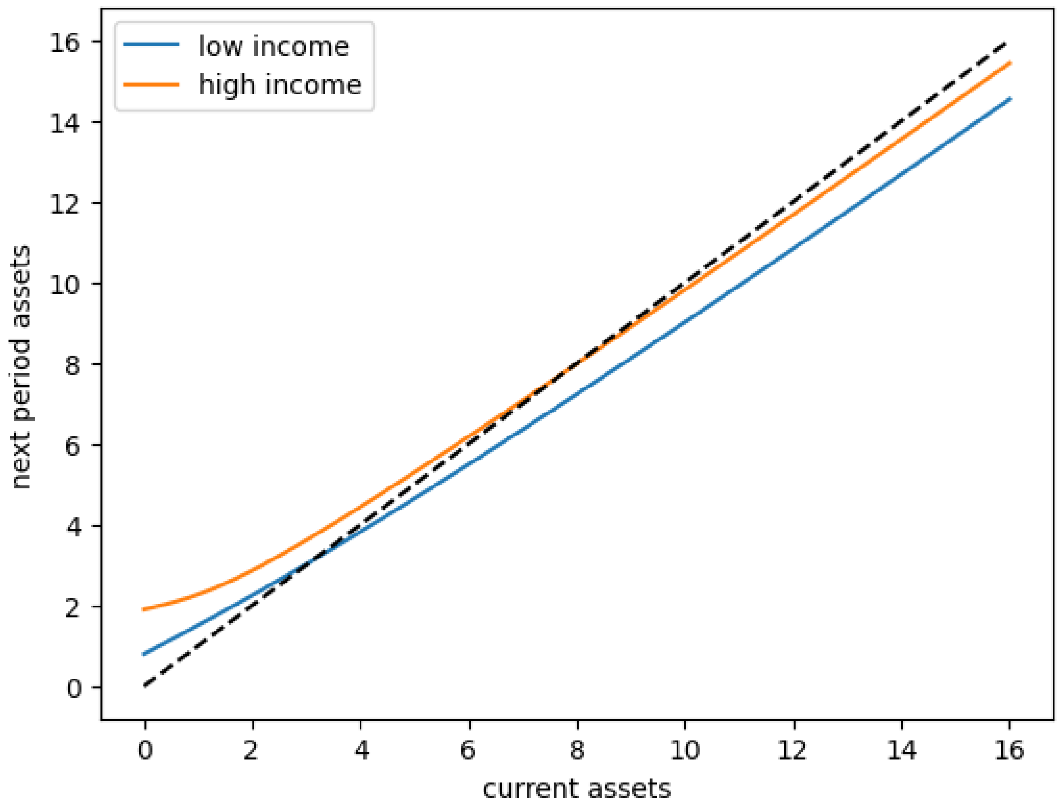
```
fig, ax = plt.subplots()
ax.plot(a_vec[:, 0], c_vec[:, 0], label='bad state')
ax.plot(a_vec[:, 1], c_vec[:, 1], label='good state')
ax.set(xlabel='assets', ylabel='consumption')
ax.legend()
plt.show()
```



61.5.5 Dynamics

To begin to understand the long run asset levels held by households under the default parameters, let's look at the 45 degree diagram showing the law of motion for assets under the optimal consumption policy.

```
fig, ax = plt.subplots()

y = lambda z: jnp.exp(z)

def y_bar(k):
    """
    Taking z = z_grid[k], compute

            E_z Y' = Σ_{z'} y(z') Π[z, z']

    This is the expectation of Y_{t+1} given Z_t = z.
    """
    # Compute y(z') for all z'
    y_values = jax.vmap(y)(z_grid)
    # Weight by transition probabilities and sum
    return jnp.sum(y_values * Π[k, :])

for k, label in zip((0, 1), ('low income', 'high income')):
    # Interpolate consumption policy on the savings grid
    c_on_grid = jnp.interp(s, a_vec[:, k], c_vec[:, k])
    ax.plot(s, R * (s - c_on_grid) + y_bar(k) , label=label)

ax.plot(s, s, 'k--')
ax.set(xlabel='current assets', ylabel='next period assets')

ax.legend()
plt.show()
```



The unbroken lines show the update function for assets at each z, which is

$$a \mapsto R(a - \sigma^*(a, z)) + \bar{y}(z)$$

where

$$\bar{y}(z) := \sum_{z'} y(z')\Pi(z, z')$$

is the expected labor income conditional on current state z.

The dashed line is the 45 degree line.

The figure suggests that, on average, the dynamics will be stable — assets do not diverge even in the highest state.

This turns out to be true: there is a unique stationary distribution of assets.

- For details see [Ma *et al.*, 2020]

This stationary distribution represents the long run dispersion of assets across households when households have idiosyncratic shocks.

61.5.6 A Sanity Check

One way to check our results is to

- set labor income to zero in each state and
- set the gross interest rate R to unity.

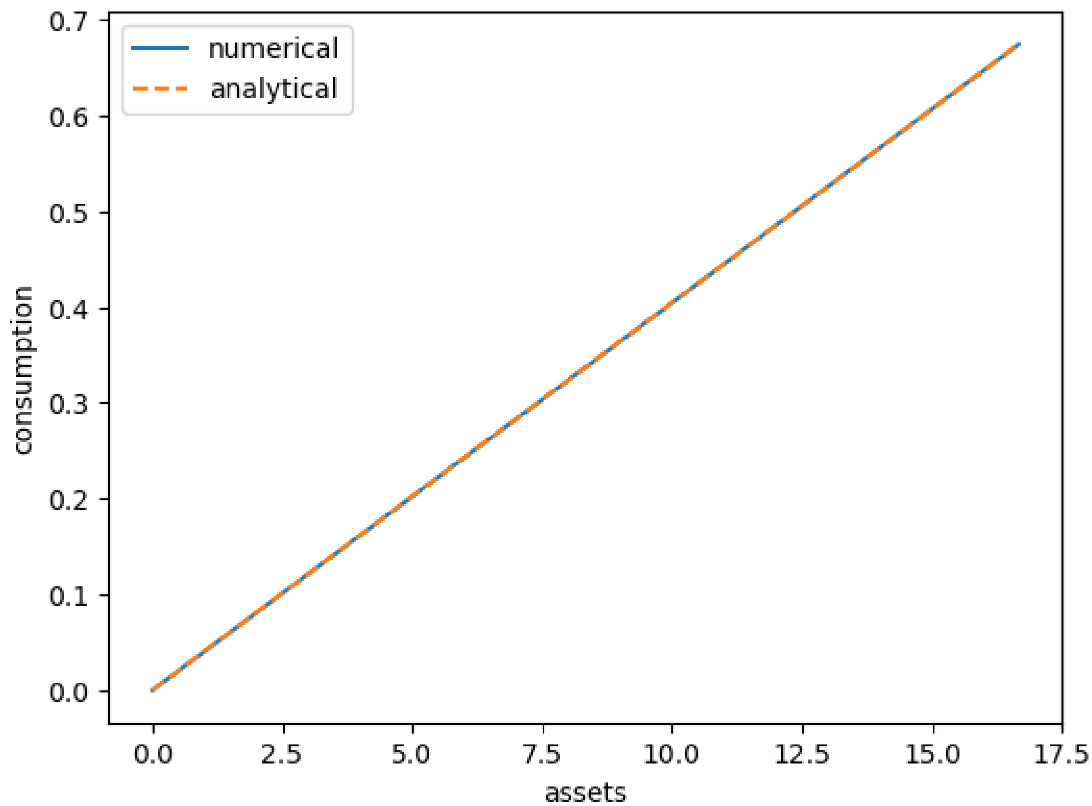
In this case, our income fluctuation problem is just a CRRA cake eating problem.

Then the value function and optimal consumption policy are given by

```
def c_star(x, β, γ):
    return (1 - β ** (1/γ)) * x


def v_star(x, β, γ):
    return (1 - β**(1 / γ))**(-γ) * (x**(1-γ) / (1-γ))
```

Let's see if we match up:

```
ifp_cake_eating = create_ifp(r=0.0, z_grid=(-jnp.inf, -jnp.inf))
R, β, γ, Π, z_grid, s = ifp_cake_eating
a_init = s[:, None] * jnp.ones(len(z_grid))
c_init = a_init
c_vec, a_vec = solve_model(ifp_cake_eating, c_init, a_init)

fig, ax = plt.subplots()
ax.plot(a_vec[:, 0], c_vec[:, 0], label='numerical')
ax.plot(a_vec[:, 0],
        c_star(a_vec[:, 0], ifp_cake_eating.β, ifp_cake_eating.γ),
        '--', label='analytical')
ax.set(xlabel='assets', ylabel='consumption')
ax.legend()
plt.show()
```



This looks pretty good.

61.6 Simulation

Let's return to the default model and study the stationary distribution of assets.

Our plan is to run a large number of households forward for T periods and then histogram the cross-sectional distribution of assets.

Set `num_households=50_000, T=500`.

First we write a function to run a single household forward in time and record the final value of assets.

The function takes a solution pair `c_vec` and `a_vec`, understanding them as representing an optimal policy associated with a given model `ifp`

```
@jax.jit
def simulate_household(
        key, a_0, z_idx_0, c_vec, a_vec, ifp, T
    ):
    """
    Simulates a single household for T periods to approximate the stationary
    distribution of assets.

    - key is the state of the random number generator
    - ifp is an instance of IFP
    - c_vec, a_vec are the optimal consumption policy, endogenous grid for ifp
```

(continues on next page)

(continued from previous page)

```
    """
    R, β, γ, Π, z_grid, s = ifp
    n_z = len(z_grid)

    y = lambda z: jnp.exp(z)
    σ = lambda a, z_idx: jnp.interp(a, a_vec[:, z_idx], c_vec[:, z_idx])

    # Simulate forward T periods
    def update(t, state):
        a, z_idx = state
        # Draw next shock z' from Π[z, z']
        current_key = jax.random.fold_in(key, t)
        z_next_idx = jax.random.choice(current_key, n_z, p=Π[z_idx]).astype(jnp.int32)
        z_next = z_grid[z_next_idx]
        # Update assets: a' = R * (a - c) + Y'
        a_next = R * (a - σ(a, z_idx)) + y(z_next)
        # Return updated state
        return a_next, z_next_idx

    initial_state = a_0, z_idx_0
    final_state = jax.lax.fori_loop(0, T, update, initial_state)
    a_final, _ = final_state
    return a_final
```

Now we write a function to simulate many households in parallel.

```
def compute_asset_stationary(
        c_vec, a_vec, ifp, num_households=50_000, T=500, seed=1234
    ):
    """
    Simulates num_households households for T periods to approximate
    the stationary distribution of assets.

    Returns the final cross-section of asset holdings.

    - ifp is an instance of IFP
    - c_vec, a_vec are the optimal consumption policy and endogenous grid.

    """
    R, β, γ, Π, z_grid, s = ifp
    n_z = len(z_grid)

    # Create interpolation function for consumption policy
    # Interpolate on the endogenous grid
    σ = lambda a, z_idx: jnp.interp(a, a_vec[:, z_idx], c_vec[:, z_idx])

    # Start with assets = savings_grid_max / 2
    a_0_vector = jnp.full(num_households, s[-1] / 2)
    # Initialize the exogenous state of each household
    z_idx_0_vector = jnp.zeros(num_households).astype(jnp.int32)

    # Vectorize over many households
    key = jax.random.PRNGKey(seed)
    keys = jax.random.split(key, num_households)
    # Vectorize simulate_household in (key, a_0, z_idx_0)
    sim_all_households = jax.vmap(
        simulate_household, in_axes=(0, 0, 0, None, None, None, None)
```

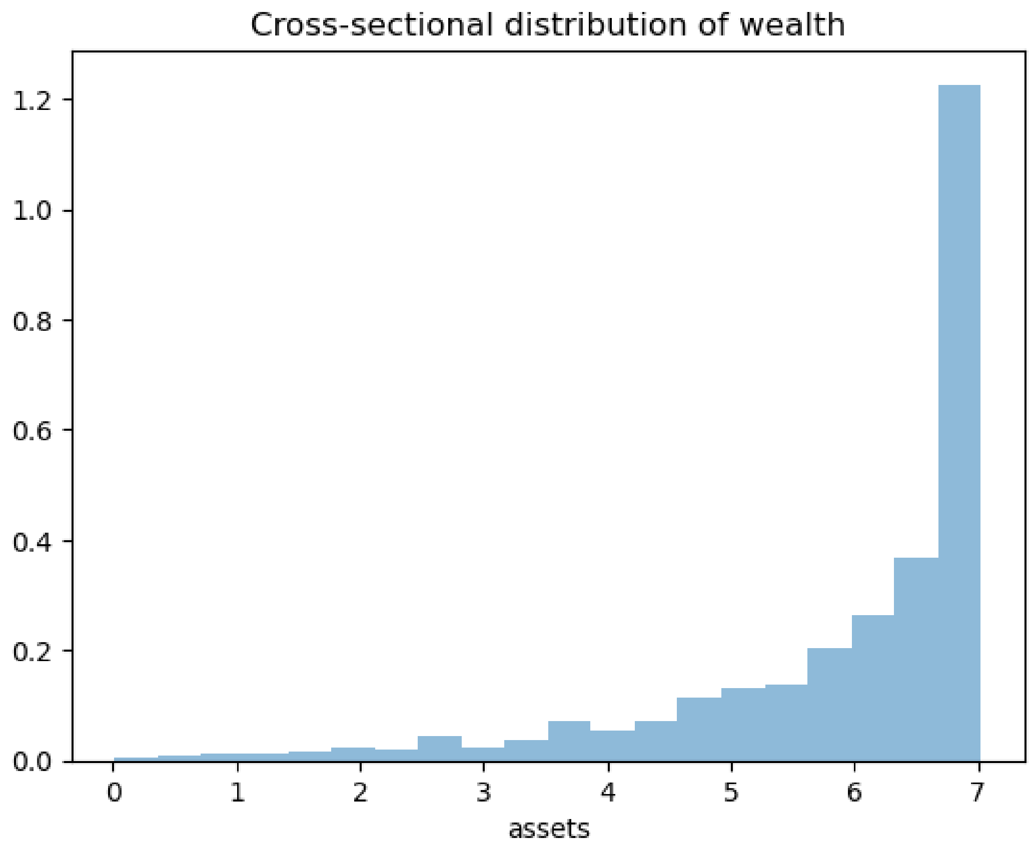
(continues on next page)

(continued from previous page)

```
    )
    assets = sim_all_households(keys, a_0_vector, z_idx_0_vector, c_vec, a_vec, ifp,␣
↪T)

    return np.array(assets)
```

Now we call the function, generate the asset distribution and histogram it:

```
ifp = create_ifp()
R, β, γ, Π, z_grid, s = ifp
a_init = s[:, None] * jnp.ones(len(z_grid))
c_init = a_init
c_vec, a_vec = solve_model(ifp, c_init, a_init)
assets = compute_asset_stationary(c_vec, a_vec, ifp)

fig, ax = plt.subplots()
ax.hist(assets, bins=20, alpha=0.5, density=True)
ax.set(xlabel='assets', title="Cross-sectional distribution of wealth")
plt.show()
```



The wealth distribution looks very different to a typical wealth distribution in the data.

For one thing it is left-skewed rather than right-skewed.

In fact there is essentially no right-hand tail, even though real-world wealth distributions have long right-hand tails.

We'll do our best to fix these issues in the next few lectures.

CHAPTER

SIXTYTWO

THE INCOME FLUCTUATION PROBLEM IV: TRANSIENT INCOME SHOCKS

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

Contents

62.1 Overview

In this lecture we continue extend the IFP from *The Income Fluctuation Problem III: The Endogenous Grid Method* by adding transient shocks to the income process.

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon jax
```

We'll also need the following imports:

```
import matplotlib.pyplot as plt
import numpy as np
import numba
from quantecon import MarkovChain
import jax
import jax.numpy as jnp
from typing import NamedTuple
```

62.2 The Household Problem

We briefly outline the model and then discuss how to solve it.

Readers seeking a more extensive discussion of the model and the EGM solution method can review *The Income Fluctuation Problem III: The Endogenous Grid Method*.

62.2.1 Set-Up

A household chooses a state-contingent consumption plan $\{c_t\}_{t \geq 0}$ to maximize

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t u(c_t)$$

subject to

$$a_{t+1} = R(a_t - c_t) + Y_{t+1} \quad c_t \geq 0, \quad a_t \geq 0 \quad t = 0, 1, \ldots \tag{62.1}$$

The definitions of symbols and the timing are the same as in *The Income Fluctuation Problem III: The Endogenous Grid Method*.

Now, non-capital income Y_t is given by $Y_t = y(Z_t, \eta_t)$, where

- $\{Z_t\}$ is an exogenous state process (persistent component),
- $\{\eta_t\}$ is an IID shock process, and
- y is a function taking values in $\mathbb{R}_+$.

Throughout this lecture, we assume that $\eta_t \sim N(0, 1)$.

We again take $\{Z_t\}$ to be a finite state Markov chain taking values in Z with Markov matrix Π.

The shock process $\{\eta_t\}$ is independent of $\{Z_t\}$ and represents transient income fluctuations.

In addition to previous assumptions, we suppose that $y(z, \eta) = \exp(a_y \eta + z b_y)$ where a_y, b_y are positive constants

The asset space and state space are unchanged, as is the definition of an optimal path.

The functional Euler equation has the form

$$(u' \circ \sigma)(a, z) = \beta R \sum_{z'} \int (u' \circ \sigma)[R(a - \sigma(a, z)) + y(z', \eta'),\ z'] \phi(\eta') d\eta' \, \Pi(z, z') \tag{62.2}$$

Here

- $(u' \circ \sigma)(s) := u'(\sigma(s))$,
- primes indicate next period states (as well as derivatives),
- ϕ is the density of the shock η_t (standard normal), and

- σ is the unknown function.

The equality (62.2) holds at all interior choices, meaning $\sigma(a, z) < a$.

We aim to find a fixed point σ of (62.2).

To do so we use the EGM.

Below we use the relationships $a_t = c_t + s_t$ and $a_{t+1} = Rs_t + Y_{t+1}$.

We begin with an exogenous savings grid $s_0 < s_1 < \cdots < s_m$ with $s_0 = 0$.

We fix a current guess of the policy function σ.

For each exogenous savings level s_i with $i \geq 1$ and current state z_j, we set

$$c_{ij} := (u')^{-1} \left[\beta R \sum_{z'} \int u'[\sigma(Rs_i + y(z', \eta'), z')] \phi(\eta') d\eta' \, \Pi(z_j, z') \right] \tag{62.3}$$

The Euler equation holds here because $i \geq 1$ implies $s_i > 0$ and hence consumption is interior.

For the boundary case $s_0 = 0$ we set

$$c_{0j} := 0 \quad \text{for all j}$$

We then obtain a corresponding endogenous grid of current assets via

$$a_{ij} := c_{ij} + s_i.$$

Our next guess of the policy function, which we write as $K\sigma$, is the linear interpolation of the interpolation points

$$\{(a_{0j}, c_{0j}), \dots, (a_{mj}, c_{mj})\}$$

for each j.

62.3 NumPy Implementation

In this section we'll code up a NumPy version of the code that aims only for clarity, rather than efficiency.

Once we have it working, we'll produce a JAX version that's far more efficient and check that we obtain the same results.

We use the CRRA utility specification

$$u(c) = \frac{c^{1-\gamma}}{1-\gamma}$$

62.3.1 Set Up

Here we build a class called `IFPNumPy` that stores the model primitives.

The exogenous state process $\{Z_t\}$ defaults to a two-state Markov chain with transition matrix Π.

```
class IFPNumPy(NamedTuple):
    R: float                   # Gross interest rate R = 1 + r
    β: float                   # Discount factor
    γ: float                   # Preference parameter
    Π: np.ndarray              # Markov matrix for exogenous shock
```

(continues on next page)

(continued from previous page)

```
    z_grid: np.ndarray          # Markov state values for Z_t
    s: np.ndarray               # Exogenous savings grid
    a_y: float                  # Scale parameter for Y_t
    b_y: float                  # Additive parameter for Y_t
    η_draws: np.ndarray         # Draws of innovation η for MC


def create_ifp(r=0.01,
               β=0.96,
               γ=1.5,
               Π=((0.6, 0.4),
                  (0.05, 0.95)),
               z_grid=(-10.0, np.log(2.0)),
               savings_grid_max=16,
               savings_grid_size=50,
               a_y=0.2,
               b_y=0.5,
               shock_draw_size=100,
               seed=1234):

    np.random.seed(seed)
    s = np.linspace(0, savings_grid_max, savings_grid_size)
    Π, z_grid = np.array(Π), np.array(z_grid)
    R = 1 + r
    η_draws = np.random.randn(shock_draw_size)
    assert R * β < 1, "Stability condition violated."
    return IFPNumPy(R, β, γ, Π, z_grid, s, a_y, b_y, η_draws)
```

62.3.2 Solver

Here is the operator K that transforms current guess σ into next period guess $K\sigma$.

In practice, it takes in

- a guess of optimal consumption values c_{ij}, stored as `c_vec`
- and a corresponding set of endogenous grid points a^e_{ij}, stored as `a_vec`

These are converted into a consumption policy $a \mapsto \sigma(a, z_j)$ by linear interpolation of (a^e_{ij}, c_{ij}) over i for each j.

When we compute consumption in (62.3), we will use Monte Carlo over η', so that the expression becomes

$$c_{ij} := (u')^{-1} \left[\beta R \sum_{z'} \frac{1}{m} \sum_{\ell=1}^{m} u'[\sigma(R s_i + y(z', \eta_\ell), z')] \, \Pi(z_j, z') \right] \tag{62.4}$$

with each η_ℓ being a standard normal draw.

```
@numba.jit
def K_numpy(
        c_in: np.ndarray,    # Initial guess of σ on grid endogenous grid
        a_in: np.ndarray,    # Initial endogenous grid
        ifp_numpy: IFPNumPy
    ) -> np.ndarray:
    """
    The Euler equation operator for the IFP model using the
    Endogenous Grid Method.
```

(continues on next page)

(continued from previous page)

```
    This operator implements one iteration of the EGM algorithm to
    update the consumption policy function.

    """
    R, β, γ, Π, z_grid, s, a_y, b_y, η_draws = ifp_numpy
    n_a = len(s)
    n_z = len(z_grid)

    # Utility functions
    def u_prime(c):
        return c**(-γ)

    def u_prime_inv(c):
        return c**(-1/γ)

    def y(z, η):
        return np.exp(a_y * η + z * b_y)

    c_out = np.zeros_like(c_in)

    for i in range(1, n_a):  # Start from 1 for positive savings levels
        for j in range(n_z):

            # Compute Σ_z' ∫ u'(σ(R s_i + y(z', η'), z')) φ(η') dη' Π[z_j, z']
            expectation = 0.0
            for k in range(n_z):
                z_prime = z_grid[k]
                # Integrate over η draws (Monte Carlo)
                inner_sum = 0.0
                for η in η_draws:
                    # Calculate next period assets
                    next_a = R * s[i] + y(z_prime, η)
                    # Interpolate to get σ(R s_i + y(z', η), z')
                    next_c = np.interp(next_a, a_in[:, k], c_in[:, k])
                    # Add to the inner sum
                    inner_sum += u_prime(next_c)
                # Average over η draws to approximate the integral
                # ∫ u'(σ(R s_i + y(z', η'), z')) φ(η') dη' when z' = z_grid[k]
                inner_mean_k = (inner_sum / len(η_draws))
                # Weight by transition probability and add to the expectation
                expectation += inner_mean_k * Π[j, k]

            # Calculate updated c_{ij} values
            c_out[i, j] = u_prime_inv(β * R * expectation)

    a_out = c_out + s[:, None]

    return c_out, a_out
```

To solve the model we use a simple while loop.

```
def solve_model_numpy(
        ifp_numpy: IFPNumPy,
        c_init: np.ndarray,
        a_init: np.ndarray,
        tol: float = 1e-5,
```

(continues on next page)

(continued from previous page)

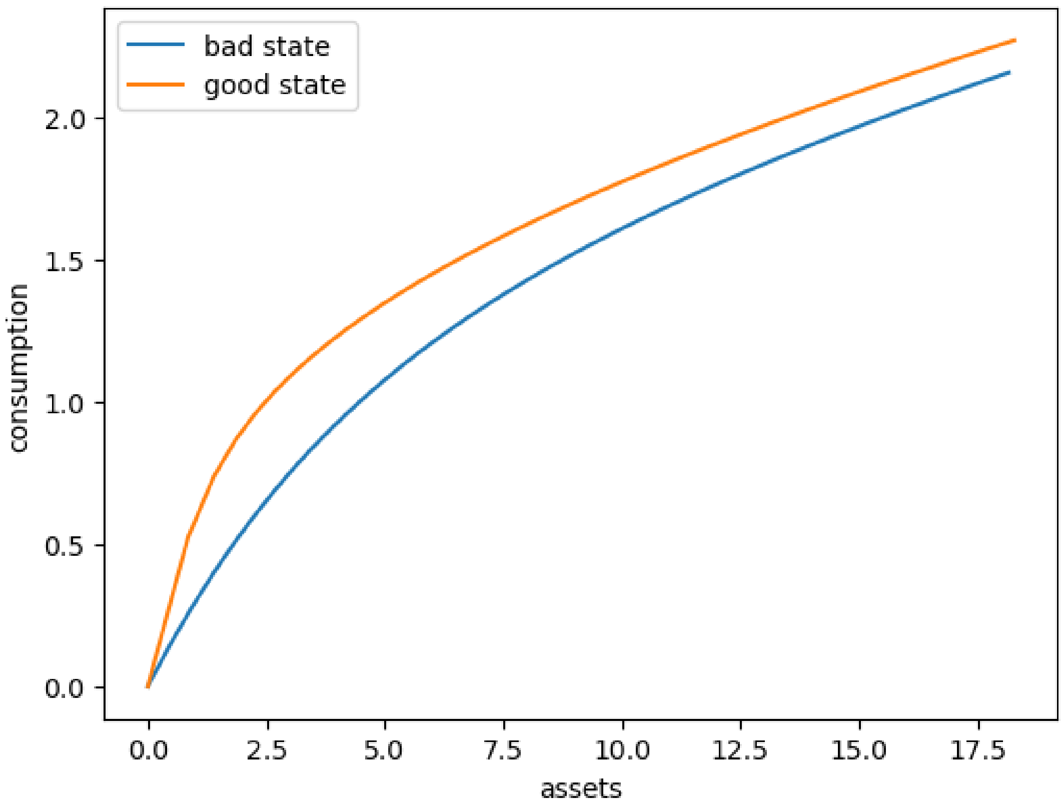
```
        max_iter: int = 1_000
    ) -> np.ndarray:
    """
    Solve the model using time iteration with EGM.

    """
    c_in, a_in = c_init, a_init
    i = 0
    error = tol + 1

    while error > tol and i < max_iter:
        c_out, a_out = K_numpy(c_in, a_in, ifp_numpy)
        error = np.max(np.abs(c_out - c_in))
        i = i + 1
        c_in, a_in = c_out, a_out

    return c_out, a_out
```

Let's road test the EGM code.

```
ifp_numpy = create_ifp()
R, β, γ, Π, z_grid, s, a_y, b_y, η_draws = ifp_numpy
# Initial conditions -- agent consumes everything
a_init = s[:, None] * np.ones(len(z_grid))
c_init = a_init
# Solve from these initial conditions
c_vec, a_vec = solve_model_numpy(
    ifp_numpy, c_init, a_init
)
```

Here's a plot of the optimal consumption policy for each z state

```
fig, ax = plt.subplots()

ax.plot(a_vec[:, 0], c_vec[:, 0], label='bad state')
ax.plot(a_vec[:, 1], c_vec[:, 1], label='good state')
ax.set(xlabel='assets', ylabel='consumption')
ax.legend()
plt.show()
```



62.4 JAX Implementation

Now we write a more efficient JAX version, which can run on a GPU.

62.4.1 Set Up

We start with a class called `IFP` that stores the model primitives.

```
class IFP(NamedTuple):
    R: float                    # Gross interest rate R = 1 + r
    β: float                    # Discount factor
    γ: float                    # Preference parameter
    Π: jnp.ndarray              # Markov matrix for exogenous shock
    z_grid: jnp.ndarray         # Markov state values for Z_t
    s: jnp.ndarray              # Exogenous savings grid
    a_y: float                  # Scale parameter for Y_t
    b_y: float                  # Additive parameter for Y_t
    η_draws: jnp.ndarray        # Draws of innovation η for MC


def create_ifp(r=0.01,
               β=0.94,
               γ=1.5,
               Π=((0.6, 0.4),
                  (0.05, 0.95)),
```

(continues on next page)

(continued from previous page)

```
            z_grid=(-10.0, jnp.log(2.0)),
            savings_grid_max=16,
            savings_grid_size=50,
            a_y=0.2,
            b_y=0.5,
            shock_draw_size=100,
            seed=1234):

    key = jax.random.PRNGKey(seed)
    s = jnp.linspace(0, savings_grid_max, savings_grid_size)
    Π, z_grid = jnp.array(Π), jnp.array(z_grid)
    R = 1 + r
    η_draws = jax.random.normal(key, (shock_draw_size,))
    assert R * β < 1, "Stability condition violated."
    return IFP(R, β, γ, Π, z_grid, s, a_y, b_y, η_draws)
```

62.4.2 Solver

Here is the operator K that transforms current guess σ into next period guess $K\sigma$.

```
def K(
        c_in: jnp.ndarray,
        a_in: jnp.ndarray,
        ifp: IFP
    ) -> jnp.ndarray:
    """
    The Euler equation operator for the IFP model using the
    Endogenous Grid Method.

    This operator implements one iteration of the EGM algorithm to
    update the consumption policy function.

    """
    R, β, γ, Π, z_grid, s, a_y, b_y, η_draws = ifp
    n_a = len(s)
    n_z = len(z_grid)

    # Utility functions
    def u_prime(c):
        return c**(-γ)

    def u_prime_inv(c):
        return c**(-1/γ)

    def y(z, η):
        return jnp.exp(a_y * η + z * b_y)

    def compute_c(i, j):
        " Compute c_ij when i >= 1 (interior choice). "

        def expected_mu(k):
            " Approximate ∫ u'(σ(R s_i + y(z_k, η'), z_k)) φ(η') dη' "

            def compute_mu_at_eta(η):
                " Compute u'(σ(R * s_i + y(z_k, η), z_k)) "
```

(continues on next page)

(continued from previous page)

```
                next_a = R * s[i] + y(z_grid[k], η)
                # Interpolate to get σ(R * s_i + y(z_k, η), z_k)
                next_c = jnp.interp(next_a, a_in[:, k], c_in[:, k])
                # Return u'(σ(R * s_i + y(z_k, η), z_k))
                return u_prime(next_c)

            # Average over η draws to approximate the inner integral
            # ∫ u'(σ(R s_i + y(z_k, η'), z_k)) φ(η') dη'
            all_draws = jax.vmap(compute_mu_at_eta)(η_draws)
            return jnp.mean(all_draws)

        # Compute expectation: Σ_k [∫ u'(σ(...)) φ(η) dη] * Π[j, k]
        expectations = jax.vmap(expected_mu)(jnp.arange(n_z))
        expectation = jnp.sum(expectations * Π[j, :])
        # Invert to get consumption c_ij at (s_i, z_j)
        return u_prime_inv(β * R * expectation)

    # Set up index grids for vmap computation of all c_{ij}
    i_grid = jnp.arange(1, n_a)
    j_grid = jnp.arange(n_z)

    # vmap over j for each i
    compute_c_i = jax.vmap(compute_c, in_axes=(None, 0))
    # vmap over i
    compute_c = jax.vmap(lambda i: compute_c_i(i, j_grid))
    # Compute consumption for i >= 1
    c_out_interior = compute_c(i_grid)  # Shape: (n_a-1, n_z)
    # For i = 0, set consumption to 0
    c_out_boundary = jnp.zeros((1, n_z))

    # Concatenate boundary and interior
    c_out = jnp.concatenate([c_out_boundary, c_out_interior], axis=0)

    # Compute endogenous asset grid: a^e_{ij} = c_{ij} + s_i
    a_out = c_out + s[:, None]

    return c_out, a_out
```

Here's a jit-accelerated iterative routine to solve the model using this operator.

```
@jax.jit
def solve_model(
        ifp: IFP,
        c_init: jnp.ndarray,  # Initial guess of σ on grid endogenous grid
        a_init: jnp.ndarray,  # Initial endogenous grid
        tol: float = 1e-5,
        max_iter: int = 1000
    ) -> jnp.ndarray:
    """
    Solve the model using time iteration with EGM.

    """

    def condition(loop_state):
        c_in, a_in, i, error = loop_state
        return (error > tol) & (i < max_iter)
```

(continues on next page)

(continued from previous page)

```
    def body(loop_state):
        c_in, a_in, i, error = loop_state
        c_out, a_out = K(c_in, a_in, ifp)
        error = jnp.max(jnp.abs(c_out - c_in))
        i += 1
        return c_out, a_out, i, error

    i, error = 0, tol + 1
    initial_state = (c_init, a_init, i, error)
    final_loop_state = jax.lax.while_loop(condition, body, initial_state)
    c_out, a_out, i, error = final_loop_state

    return c_out, a_out
```

62.4.3 Test run

Let's road test the EGM code.

```
ifp = create_ifp()
R, β, γ, Π, z_grid, s, a_y, b_y, η_draws = ifp
# Set initial conditions where the agent consumes everything
a_init = s[:, None] * jnp.ones(len(z_grid))
c_init = a_init
# Solve starting from these initial conditions
c_vec_jax, a_vec_jax = solve_model(ifp, c_init, a_init)
```

To verify the correctness of our JAX implementation, let's compare it with the NumPy version we developed earlier.

```
# Compare the results
max_c_diff = np.max(np.abs(np.array(c_vec) - c_vec_jax))
max_ae_diff = np.max(np.abs(np.array(a_vec) - a_vec_jax))

print(f"Maximum difference in consumption policy: {max_c_diff:.2e}")
print(f"Maximum difference in asset grid:        {max_ae_diff:.2e}")
```

```
Maximum difference in consumption policy: 3.94e-01
Maximum difference in asset grid:        3.94e-01
```

These numbers confirm that we are computing essentially the same policy using the two approaches.

(Remaining differences are mainly due to different Monte Carlo integration outcomes over relatively small samples.)

62.4.4 Timing

Now let's compare the execution time between NumPy and JAX implementations.

```
import time

# Set up initial conditions for NumPy version
s_np = np.array(s)
z_grid_np = np.array(z_grid)
a_init_np = s_np[:, None] * np.ones(len(z_grid_np))
c_init_np = a_init_np.copy()
```

(continues on next page)

(continued from previous page)

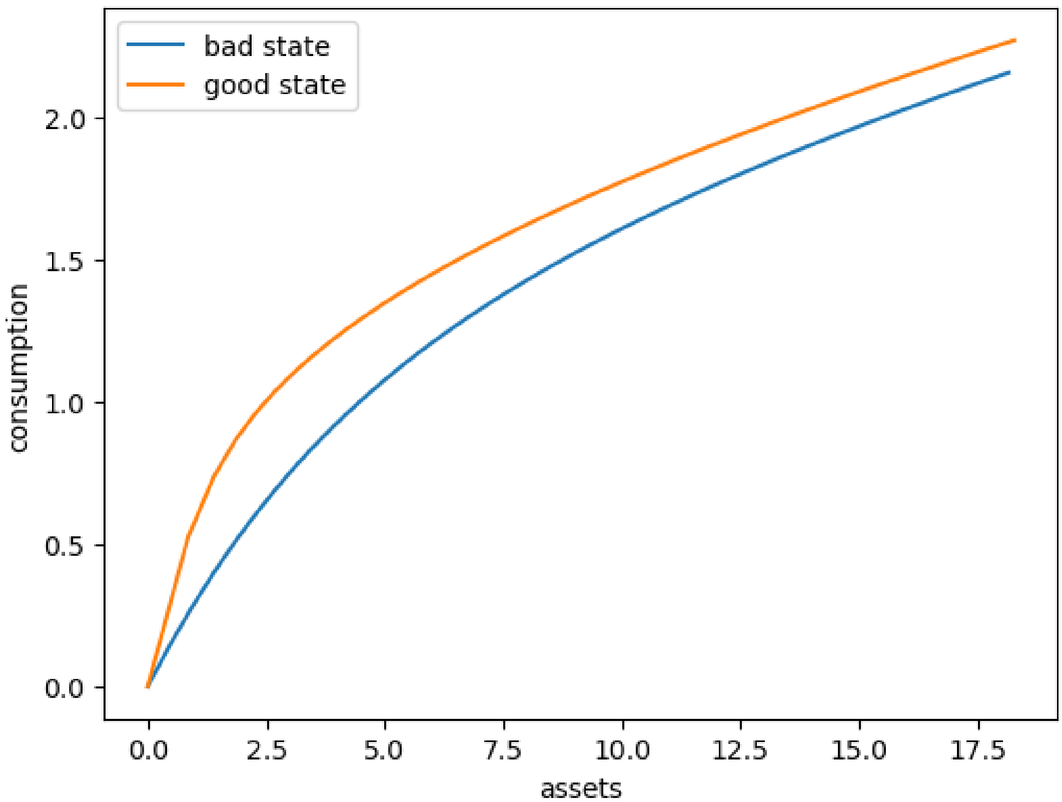
```
# Set up initial conditions for JAX version
a_init_jx = s[:, None] * jnp.ones(len(z_grid))
c_init_jx = a_init_jx

# Time NumPy version
start = time.time()
c_vec_np, a_vec_np = solve_model_numpy(ifp_numpy, c_init_np, a_init_np)
numpy_time = time.time() - start

# Time JAX version (with compilation)
start = time.time()
c_vec_jx, a_vec_jx = solve_model(ifp, c_init_jx, a_init_jx)
c_vec_jx.block_until_ready()
jax_time_with_compile = time.time() - start

# Time JAX version (without compilation - second run)
start = time.time()
c_vec_jx, a_vec_jx = solve_model(ifp, c_init_jx, a_init_jx)
c_vec_jx.block_until_ready()
jax_time = time.time() - start

print(f"NumPy time:                 {numpy_time:.4f} seconds")
print(f"JAX time (with compile):    {jax_time_with_compile:.4f} seconds")
print(f"JAX time (without compile): {jax_time:.4f} seconds")
print(f"Speedup (NumPy/JAX):        {numpy_time/jax_time:.2f}x")
```

```
NumPy time:                 0.3787 seconds
JAX time (with compile):    0.0095 seconds
JAX time (without compile): 0.0093 seconds
Speedup (NumPy/JAX):        40.85x
```

The JAX implementation is significantly faster due to JIT compilation and GPU/TPU acceleration (if available).

Here's a plot of the optimal policy for each z state

```
fig, ax = plt.subplots()
ax.plot(a_vec[:, 0], c_vec[:, 0], label='bad state')
ax.plot(a_vec[:, 1], c_vec[:, 1], label='good state')
ax.set(xlabel='assets', ylabel='consumption')
ax.legend()
plt.show()
```



62.4.5 Dynamics

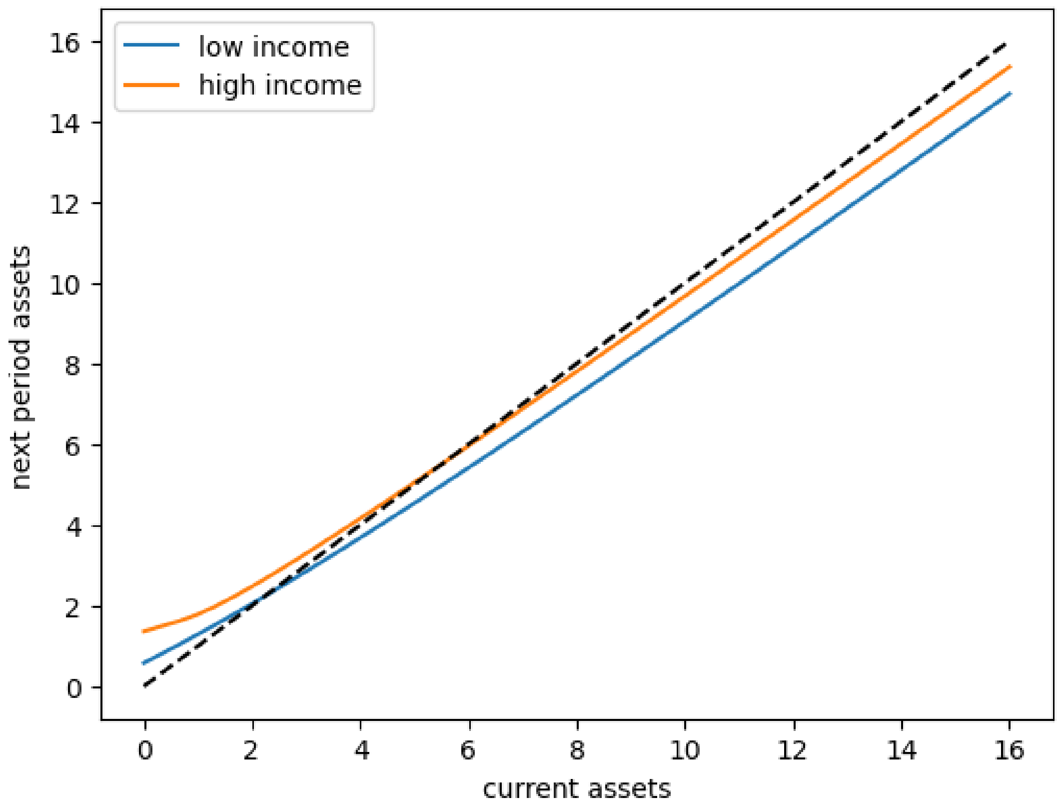
To begin to understand the long run asset levels held by households under the default parameters, let's look at the 45 degree diagram showing the law of motion for assets under the optimal consumption policy.

```
fig, ax = plt.subplots()

def y(z, η):
    return jnp.exp(a_y * η + z * b_y)

def y_bar(k):
    """
    Taking z = z_grid[k], compute an approximation to

            E_z Y' = Σ_{z'} ∫ y(z', η') ψ(η') dη' Π[z, z']

    This is the expectation of Y_{t+1} given Z_t = z.
    """
    # Approximate ∫ y(z', η') ψ(η') dη' at given z'
    def mean_y_at_z(z_prime):
        return jnp.mean(y(z_prime, η_draws))
    # Evaluate this integral across all z'
    y_means = jax.vmap(mean_y_at_z)(z_grid)
    # Weight by transition probabilities and sum
    return jnp.sum(y_means * Π[k, :])

for k, label in zip((0, 1), ('low income', 'high income')):
```

(continues on next page)

(continued from previous page)

```
    # Interpolate consumption policy on the savings grid
    c_on_grid = jnp.interp(s, a_vec[:, k], c_vec[:, k])
    ax.plot(s, R * (s - c_on_grid) + y_bar(k) , label=label)

ax.plot(s, s, 'k--')
ax.set(xlabel='current assets', ylabel='next period assets')

ax.legend()
plt.show()
```



The unbroken lines show the update function for assets at each z, which is

$$a \mapsto R(a - \sigma^*(a, z)) + \bar{y}(z)$$

where

$$\bar{y}(z) := \sum_{z'} \frac{1}{m} \sum_{\ell=1}^{m} y(z', \eta_\ell) \Pi(z, z')$$

is a Monte Carlo approximation to expected labor income conditional on current state z.

The dashed line is the 45 degree line.

The figure suggests that, on average, the dynamics will be stable — assets do not diverge even in the highest state.

This turns out to be true: there is a unique stationary distribution of assets.

- For details see [Ma *et al.*, 2020]

This stationary distribution represents the long run dispersion of assets across households when households have idiosyncratic shocks.

62.5 Simulation

Let's return to the default model and study the stationary distribution of assets.

Our plan is to run a large number of households forward for T periods and then histogram the cross-sectional distribution of assets.

Set `num_households=50_000, T=500`.

First we write a function to run a single household forward in time and record the final value of assets.

The function takes a solution pair `c_vec` and `a_vec`, understanding them as representing an optimal policy associated with a given model `ifp`

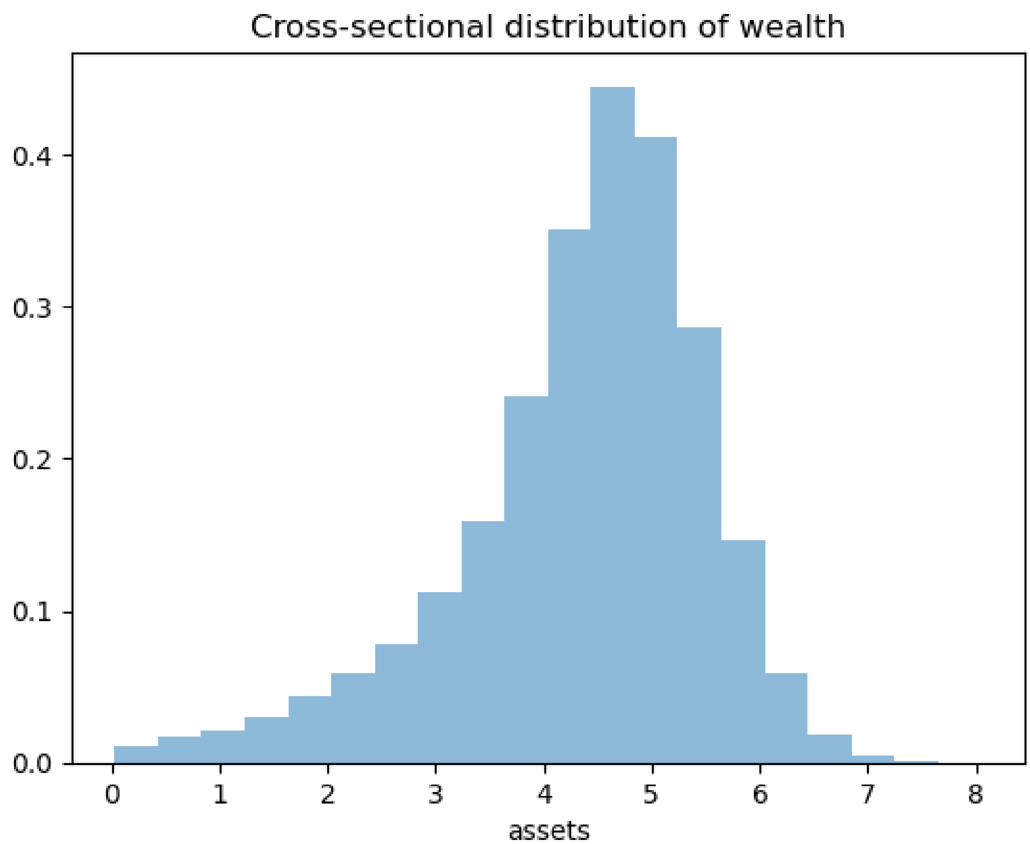
```
@jax.jit
def simulate_household(
        key, a_0, z_idx_0, c_vec, a_vec, ifp, T
    ):
    """
    Simulates a single household for T periods to approximate the stationary
    distribution of assets.

    - key is the state of the random number generator
    - ifp is an instance of IFP
    - c_vec, a_vec are the optimal consumption policy, endogenous grid for ifp

    """
    R, β, γ, Π, z_grid, s, a_y, b_y, η_draws = ifp
    n_z = len(z_grid)

    def y(z, η):
        return jnp.exp(a_y * η + z * b_y)

    # Create interpolation function for consumption policy
    σ = lambda a, z_idx: jnp.interp(a, a_vec[:, z_idx], c_vec[:, z_idx])

    # Simulate forward T periods
    def update(t, state):
        a, z_idx = state
        # Draw next shock z' from Π[z, z']
        current_key = jax.random.fold_in(key, 2*t)
        z_next_idx = jax.random.choice(current_key, n_z, p=Π[z_idx]).astype(jnp.int32)
        z_next = z_grid[z_next_idx]
        # Draw η shock
        η_key = jax.random.fold_in(key, 2*t + 1)
        η = jax.random.normal(η_key)
        # Update assets: a' = R * (a - c) + Y'
        a_next = R * (a - σ(a, z_idx)) + y(z_next, η)
        # Return updated state
        return a_next, z_next_idx

    initial_state = a_0, z_idx_0
    final_state = jax.lax.fori_loop(0, T, update, initial_state)
    a_final, _ = final_state
    return a_final
```

Now we write a function to simulate many households in parallel.

```python
def compute_asset_stationary(
        c_vec, a_vec, ifp, num_households=50_000, T=500, seed=1234
    ):
    """
    Simulates num_households households for T periods to approximate
    the stationary distribution of assets.

    Returns the final cross-section of asset holdings.

    - ifp is an instance of IFP
    - c_vec, a_vec are the optimal consumption policy and endogenous grid.

    """
    R, β, γ, Π, z_grid, s, a_y, b_y, η_draws = ifp
    n_z = len(z_grid)

    # Create interpolation function for consumption policy
    # Interpolate on the endogenous grid
    σ = lambda a, z_idx: jnp.interp(a, a_vec[:, z_idx], c_vec[:, z_idx])

    # Start with assets = savings_grid_max / 2
    a_0_vector = jnp.full(num_households, s[-1] / 2)
    # Initialize the exogenous state of each household
    z_idx_0_vector = jnp.zeros(num_households).astype(jnp.int32)

    # Vectorize over many households
    key = jax.random.PRNGKey(seed)
    keys = jax.random.split(key, num_households)
    # Vectorize simulate_household in (key, a_0, z_idx_0)
    sim_all_households = jax.vmap(
        simulate_household, in_axes=(0, 0, 0, None, None, None, None)
    )
    assets = sim_all_households(keys, a_0_vector, z_idx_0_vector, c_vec, a_vec, ifp,␣
↪T)

    return np.array(assets)
```

Now we call the function, generate the asset distribution and histogram it:

```python
ifp = create_ifp()
R, β, γ, Π, z_grid, s, a_y, b_y, η_draws = ifp
a_init = s[:, None] * jnp.ones(len(z_grid))
c_init = a_init
c_vec, a_vec = solve_model(ifp, c_init, a_init)
assets = compute_asset_stationary(c_vec, a_vec, ifp)

fig, ax = plt.subplots()
ax.hist(assets, bins=20, alpha=0.5, density=True)
ax.set(xlabel='assets', title="Cross-sectional distribution of wealth")
plt.show()
```



As was the case in *The Income Fluctuation Problem III: The Endogenous Grid Method*, the wealth distribution looks implausible.

While we have at least gained a nontrivial right tail, we still have a left skew.

62.6 Wealth Inequality

Lets' look at wealth inequality by computing some standard measures of this phenomenon.

We will also examine how inequality varies with the interest rate.

62.6.1 Measuring Inequality

We'll compute two common measures of wealth inequality:

1. **Gini coefficient**: A measure of inequality ranging from 0 (perfect equality) to 1 (perfect inequality)
2. **Top 1% wealth share**: The fraction of total wealth held by the richest 1% of households

Here are functions to compute these measures:

```
def gini_coefficient(x):
    """
    Compute the Gini coefficient for array x.

    """
```

(continues on next page)

(continued from previous page)

```
    x = jnp.asarray(x)
    n = len(x)
    x_sorted = jnp.sort(x)
    # Compute Gini coefficient
    cumsum = jnp.cumsum(x_sorted)
    a = (2 * jnp.sum((jnp.arange(1, n+1)) * x_sorted)) / (n * cumsum[-1])
    return a - (n + 1) / n


def top_share(
        x: jnp.array,   # array of wealth values
        p: float=0.01   # fraction of top households (default 0.01 for top 1%)
    ):
    """
    Compute the share of total wealth held by the top p fraction of households.

    """
    x = jnp.asarray(x)
    x_sorted = jnp.sort(x)
    # Number of households in top p%
    n_top = int(jnp.ceil(len(x) * p))
    # Wealth held by top p%
    wealth_top = jnp.sum(x_sorted[-n_top:])
    # Total wealth
    wealth_total = jnp.sum(x_sorted)
    return wealth_top / wealth_total
```

Let's compute these measures for our baseline simulation:

```
gini = gini_coefficient(assets)
top1 = top_share(assets, p=0.01)

print(f"Gini coefficient: {gini:.4f}")
print(f"Top 1% wealth share: {top1:.4f}")
```

```
Gini coefficient: 0.1428
Top 1% wealth share: 0.0154
```

These numbers are a long way out, at least for a country such as the US!

Recent numbers suggest that

- the Gini coefficient for wealth in the US is around 0.8
- the top 1% wealth share is over 0.3

Of course we have not made much effort to accurately estimate or calibrate our parameters.

But actually the cause is deeper — a model with this structure will always struggle to replicate the observed wealth distribution.

In a *later lecture* we'll see if we can improve on these numbers.

62.6.2 Interest Rate and Inequality

Let's examine how wealth inequality varies with the interest rate r.

We conjecture that higher interest rates will increase wealth inequality, as wealthier households benefit more from returns on their assets.

Let's investigate empirically:

```
# Test over 8 interest rate values
M = 8
r_vals = np.linspace(0, 0.05, M)

gini_vals = []
top1_vals = []

# Solve and simulate for each r
for r in r_vals:
    print(f'Analyzing inequality at r = {r:.4f}')
    ifp = create_ifp(r=r)
    R, β, γ, Π, z_grid, s, a_y, b_y, η_draws = ifp
    a_init = s[:, None] * jnp.ones(len(z_grid))
    c_init = a_init
    c_vec, a_vec = solve_model(ifp, c_init, a_init)
    assets = compute_asset_stationary(
        c_vec, a_vec, ifp, num_households=50_000, T=500
    )
    gini = gini_coefficient(assets)
    top1 = top_share(assets, p=0.01)
    gini_vals.append(gini)
    top1_vals.append(top1)
    # Use last solution as initial conditions for the policy solver
    c_init = c_vec
    a_init = a_vec
```

```
Analyzing inequality at r = 0.0000
Analyzing inequality at r = 0.0071
Analyzing inequality at r = 0.0143
Analyzing inequality at r = 0.0214
Analyzing inequality at r = 0.0286
Analyzing inequality at r = 0.0357
Analyzing inequality at r = 0.0429
Analyzing inequality at r = 0.0500
```

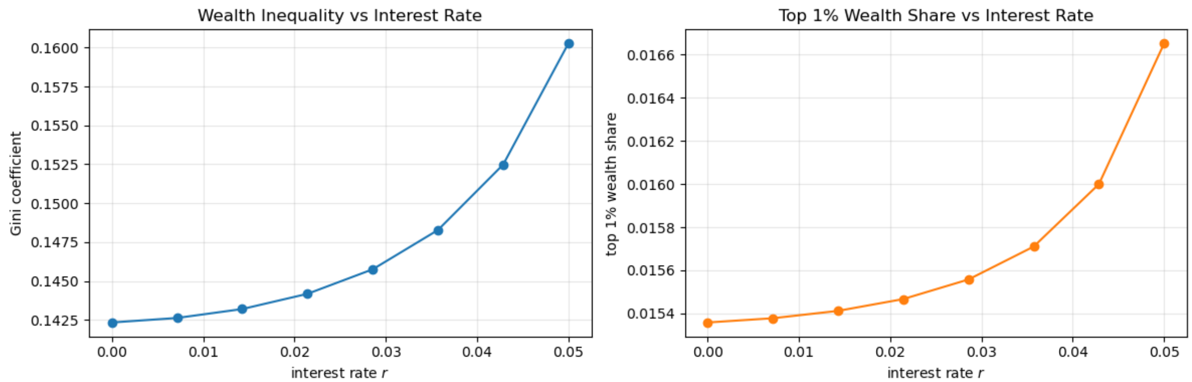
Now let's visualize the results:

```
fig, axes = plt.subplots(1, 2, figsize=(12, 4))

# Plot Gini coefficient vs interest rate
axes[0].plot(r_vals, gini_vals, 'o-')
axes[0].set_xlabel('interest rate $r$')
axes[0].set_ylabel('Gini coefficient')
axes[0].set_title('Wealth Inequality vs Interest Rate')
axes[0].grid(alpha=0.3)

# Plot top 1% share vs interest rate
axes[1].plot(r_vals, top1_vals, 'o-', color='C1')
axes[1].set_xlabel('interest rate $r$')
```

(continues on next page)

(continued from previous page)

```
axes[1].set_ylabel('top 1% wealth share')
axes[1].set_title('Top 1% Wealth Share vs Interest Rate')
axes[1].grid(alpha=0.3)

plt.tight_layout()
plt.show()
```



The results show that these two inequality measures increase with the interest rate.

However the differences are minor and we cannot increase r much more without violating the stability constraint.

Certainly changing the interest rate cannot produce the kinds of numbers that we see in the data.

62.7 Exercises

Exercise 62.7.1

Let's consider how the interest rate affects consumption.

- Step `r` through `np.linspace(0, 0.016, 4)`.
- Other than `r`, hold all parameters at their default values.
- Plot consumption against assets for income shock fixed at the smallest value.

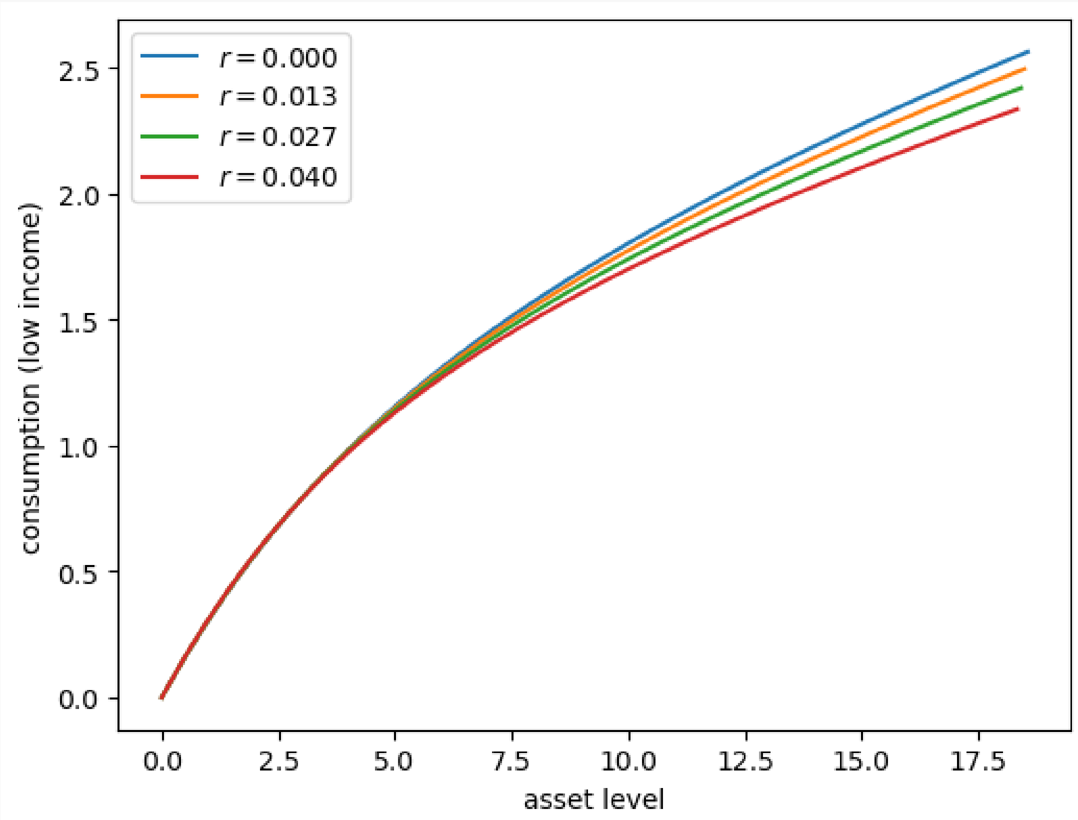
Your figure should show that, for this model, higher interest rates suppress consumption (because they encourage more savings).

Solution

Here's one solution:

```
# With β=0.96, we need R*β < 1, so r < 0.0416
r_vals = np.linspace(0, 0.04, 4)

fig, ax = plt.subplots()
for r_val in r_vals:
    ifp = create_ifp(r=r_val)
    R, β, γ, Π, z_grid, s, a_y, b_y, η_draws = ifp
    a_init = s[:, None] * jnp.ones(len(z_grid))
    c_init = a_init
```

```
    c_vec, a_vec = solve_model(ifp, c_init, a_init)
    # Plot policy
    ax.plot(a_vec[:, 0], c_vec[:, 0], label=f'$r = {r_val:.3f}$')
    # Start next round with last solution
    c_init = c_vec
    a_init = a_vec

ax.set(xlabel='asset level', ylabel='consumption (low income)')
ax.legend()
plt.show()
```



Exercise 62.7.2

Following on from Exercises 1, let's look at how savings and aggregate asset holdings vary with the interest rate

Note

[Ljungqvist and Sargent, 2018] section 18.6 can be consulted for more background on the topic treated in this exercise.

For a given parameterization of the model, the mean of the stationary distribution of assets can be interpreted as aggregate capital in an economy with a unit mass of *ex-ante* identical households facing idiosyncratic shocks.

Your task is to investigate how this measure of aggregate capital varies with the interest rate.

Intuition suggests that a higher interest rate should encourage capital formation — test this.

For the interest rate grid, use

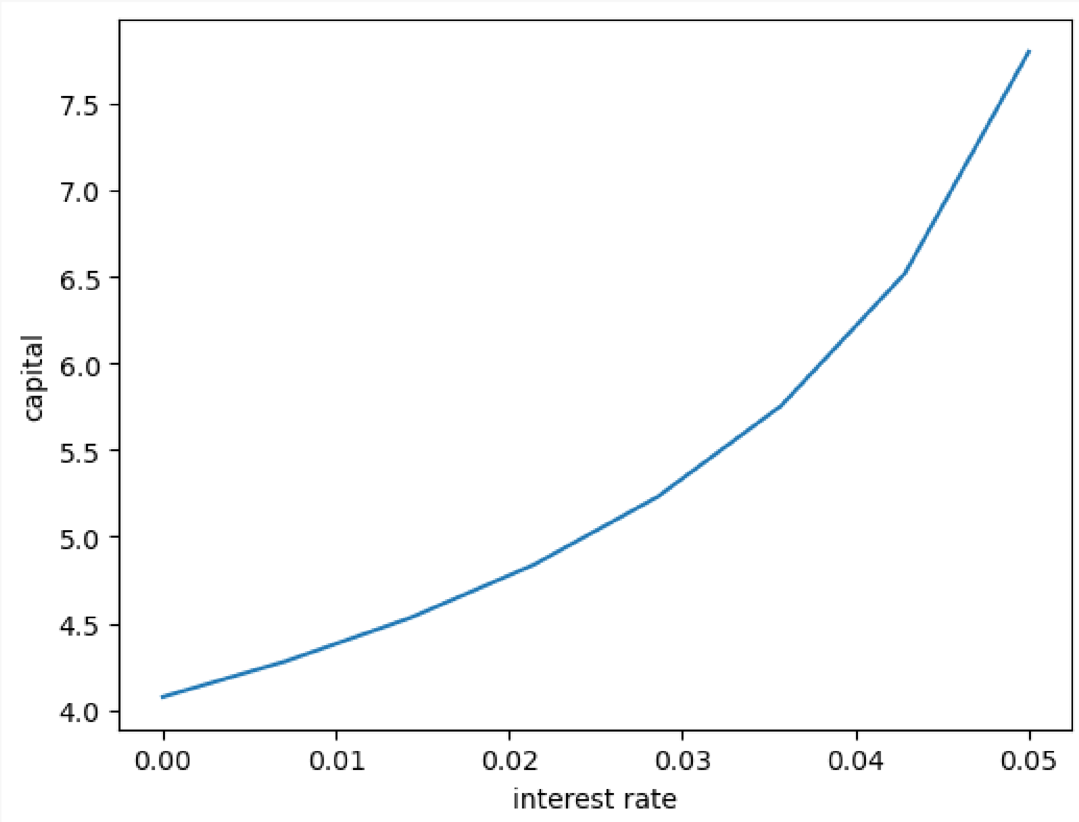
```
M = 8
r_vals = np.linspace(0, 0.05, M)
```

Solution

Here's one solution

```
fig, ax = plt.subplots()

asset_mean = []
for r in r_vals:
    print(f'Solving model at r = {r}')
    ifp = create_ifp(r=r)
    R, β, γ, Π, z_grid, s, a_y, b_y, η_draws = ifp
    a_init = s[:, None] * jnp.ones(len(z_grid))
    c_init = a_init
    c_vec, a_vec = solve_model(ifp, c_init, a_init)
    assets = compute_asset_stationary(
        c_vec, a_vec, ifp, num_households=10_000, T=500
    )
    mean = np.mean(assets)
    asset_mean.append(mean)
    print(f'  Mean assets: {mean:.4f}')
    # Start next round with last solution
    c_init = c_vec
    a_init = a_vec
ax.plot(r_vals, asset_mean)

ax.set(xlabel='interest rate', ylabel='capital')

plt.show()
```

```
Solving model at r = 0.0
  Mean assets: 4.0748
Solving model at r = 0.007142857142857143 5
  Mean assets: 4.2822
Solving model at r = 0.014285714285714287
  Mean assets: 4.5313
Solving model at r = 0.02142857142857143
  Mean assets: 4.8383
Solving model at r = 0.028571428571428574
  Mean assets: 5.2304
Solving model at r = 0.03571428571428572
  Mean assets: 5.7570
Solving model at r = 0.04285714285714286
  Mean assets: 6.5214
Solving model at r = 0.05
  Mean assets: 7.8005
```



As expected, aggregate savings increases with the interest rate.

CHAPTER

SIXTYTHREE

THE INCOME FLUCTUATION PROBLEM V: STOCHASTIC RETURNS ON ASSETS

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

Contents

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
```

63.1 Overview

In this lecture, we continue our study of the income fluctuation problem described in *The Income Fluctuation Problem III: The Endogenous Grid Method*.

While the interest rate was previously taken to be fixed, we now allow returns on assets to be state-dependent.

This matches the fact that most households with a positive level of assets face some capital income risk.

It has been argued that modeling capital income risk is essential for understanding the joint distribution of income and wealth (see, e.g., [Benhabib *et al.*, 2015] or [Stachurski and Toda, 2019]).

Theoretical properties of the household savings model presented here are analyzed in detail in [Ma *et al.*, 2020].

In terms of computation, we use a combination of time iteration and the endogenous grid method to solve the model quickly and accurately.

We require the following imports:

```
import matplotlib.pyplot as plt
import numpy as np
import quantecon as qe
import jax
import jax.numpy as jnp
from jax import vmap
from typing import NamedTuple
from functools import partial
```

63.2 The Model

In this section we review the household problem and optimality results.

63.2.1 Set Up

A household chooses a consumption-asset path $\{(c_t, a_t)\}$ to maximize

$$\mathbb{E}\left\{\sum_{t=0}^{\infty} \beta^t u(c_t)\right\} \tag{63.1}$$

subject to

$$a_{t+1} = R_{t+1}(a_t - c_t) + Y_{t+1} \text{ and } 0 \leq c_t \leq a_t, \tag{63.2}$$

with initial condition $(a_0, Z_0) = (a, z)$ treated as given.

The only difference from *The Income Fluctuation Problem IV: Transient Income Shocks* is that $\{R_t\}_{t\geq 1}$, the gross rate of return on wealth, is allowed to be stochastic.

In particular, we assume that

$$R_t = R(Z_t, \zeta_t) \quad \text{and} \quad Y_t = Y(Z_t, \eta_t), \tag{63.3}$$

where

- R and Y are time-invariant nonnegative functions,
- the innovation processes $\{\zeta_t\}$ and $\{\eta_t\}$ are IID and independent of each other, and

- $\{Z_t\}_{t \geq 0}$ is a Markov chain on a finite set Z

Let P represent the Markov matrix for the chain $\{Z_t\}_{t \geq 0}$.

In what follows, $\mathbb{E}_z \hat{X}$ means expectation of next period value $\hat{X}$ given current value $Z = z$.

63.2.2 Assumptions

We need restrictions to ensure that the objective (63.1) is finite and the solution methods described below converge.

We also need to ensure that the present discounted value of wealth does not grow too quickly.

When $\{R_t\}$ was constant we required that $\beta R < 1$.

Since it is now stochastic, we require (see [Ma *et al.*, 2020]) that

$$\beta G_R < 1, \quad \text{where} \quad G_R := \lim_{n \to \infty} \left(\mathbb{E} \prod_{t=1}^{n} R_t \right)^{1/n} \tag{63.4}$$

The value G_R can be thought of as the long run (geometric) average gross rate of return.

To simplify this lecture, we will *assume that the interest rate process is IID*.

In that case, it is clear from the definition of G_R that G_R is just $\mathbb{E} R_t$.

We test the condition $\beta \mathbb{E} R_t < 1$ in the code below.

Finally, we impose some routine technical restrictions on non-financial income.

$$\mathbb{E}\, Y_t < \infty \text{ and } \mathbb{E}\, u'(Y_t) < \infty$$

One relatively simple setting where all these restrictions are satisfied is the IID and CRRA environment of [Benhabib *et al.*, 2015].

63.2.3 Optimality

Let the class of candidate consumption policies $\mathscr{C}$ be defined as in *The Income Fluctuation Problem III: The Endogenous Grid Method*.

In [Ma *et al.*, 2020] it is shown that, under the stated assumptions,

- any $\sigma \in \mathscr{C}$ satisfying the Euler equation is an optimal policy and
- exactly one such policy exists in $\mathscr{C}$.

In the present setting, the Euler equation takes the form

$$(u' \circ \sigma)(a, z) = \max \left\{ \beta \, \mathbb{E}_z \, \hat{R} \, (u' \circ \sigma)[\hat{R}(a - \sigma(a, z)) + \hat{Y}, \, \hat{Z}], \, u'(a) \right\} \tag{63.5}$$

(Intuition and derivation are similar to *The Income Fluctuation Problem III: The Endogenous Grid Method*.)

We again solve the Euler equation using time iteration, iterating with a Coleman–Reffett operator K defined to match the Euler equation (63.5).

63.3 Solution Algorithm

63.3.1 A Time Iteration Operator

Our definition of the candidate class $\sigma \in \mathscr{C}$ of consumption policies is the same as in *The Income Fluctuation Problem III: The Endogenous Grid Method*.

For fixed $\sigma \in \mathscr{C}$ and $(a, z) \in \mathbf{S}$, the value $K\sigma(a, z)$ of the function $K\sigma$ at (a, z) is defined as the $\xi \in (0, a]$ that solves

$$u'(\xi) = \max \left\{ \beta \, \mathbb{E}_z \, \hat{R} \, (u' \circ \sigma)[\hat{R}(a - \xi) + \hat{Y}, \, \hat{Z}], \, u'(a) \right\} \tag{63.6}$$

The idea behind K is that, as can be seen from the definitions, $\sigma \in \mathscr{C}$ satisfies the Euler equation if and only if $K\sigma(a, z) = \sigma(a, z)$ for all $(a, z) \in \mathbf{S}$.

This means that fixed points of K in $\mathscr{C}$ and optimal consumption policies exactly coincide (see [Ma *et al.*, 2020] for more details).

63.3.2 Convergence Properties

As before, we pair $\mathscr{C}$ with the distance

$$\rho(c, d) := \sup_{(a,z) \in \mathbf{S}} |(u' \circ c)\,(a, z) - (u' \circ d)\,(a, z)|\,,$$

It can be shown that

1. $(\mathscr{C}, \rho)$ is a complete metric space,
2. there exists an integer n such that K^n is a contraction mapping on $(\mathscr{C}, \rho)$, and
3. The unique fixed point of K in $\mathscr{C}$ is the unique optimal policy in $\mathscr{C}$.

We now have a clear path to successfully approximating the optimal policy: choose some $\sigma \in \mathscr{C}$ and then iterate with K until convergence (as measured by the distance ρ).

63.3.3 Using an Endogenous Grid

In the study of that model we found that it was possible to further accelerate time iteration via the *endogenous grid method*.

We will use the same method here.

The methodology is the same as it was for the optimal growth model, with the minor exception that we need to remember that consumption is not always interior.

In particular, optimal consumption can be equal to assets when the level of assets is low.

Finding Optimal Consumption

The endogenous grid method (EGM) calls for us to take a grid of *savings* values s_i, where each such s is interpreted as $s = a - c$.

For the lowest grid point we take $s_0 = 0$.

For the corresponding a_0, c_0 pair we have $a_0 = c_0$.

This happens close to the origin, where assets are low and the household consumes all that it can.

Although there are many solutions, the one we take is $a_0 = c_0 = 0$, which pins down the policy at the origin, aiding interpolation.

For $s > 0$, we have, by definition, $c < a$, and hence consumption is interior.

Hence the max component of (63.5) drops out, and we solve for

$$c_i = (u')^{-1} \left\{ \beta \, \mathbb{E}_z \hat{R} (u' \circ \sigma) \, [\hat{R} s_i + \hat{Y}, \, \hat{Z}] \right\} \tag{63.7}$$

at each s_i.

Iterating

Once we have the pairs $\{s_i, c_i\}$, the endogenous asset grid is obtained by $a_i = c_i + s_i$.

Also, we held $z \in \mathsf{Z}$ in the discussion above so we can pair it with a_i.

An approximation of the policy $(a, z) \mapsto \sigma(a, z)$ can be obtained by interpolating $\{a_i, c_i\}$ at each z.

In what follows, we use linear interpolation.

63.4 Implementation

Here's the model as a `NamedTuple`.

```
class IFP(NamedTuple):
    """
    A NamedTuple that stores primitives for the income fluctuation
    problem, using JAX.
    """
    γ: float
    β: float
    P: jnp.ndarray
    a_r: float
    b_r: float
    a_y: float
    b_y: float
    s_grid: jnp.ndarray
    η_draws: jnp.ndarray
    ζ_draws: jnp.ndarray


def create_ifp(
        γ=1.5,                       # Utility parameter
        β=0.96,                      # Discount factor
        P=jnp.array([(0.9, 0.1),     # Default Markov chain for Z
                     (0.1, 0.9)]),
        a_r=0.16,                    # Volatility term in R shock
        b_r=0.0,                     # Mean shift R shock
        a_y=0.2,                     # Volatility term in Y shock
        b_y=0.5,                     # Mean shift Y shock
        shock_draw_size=100,         # For Monte Carlo
        grid_max=100,                # Exogenous grid max
        grid_size=100,               # Exogenous grid size
        seed=1234                    # Random seed
    ):
```

(continues on next page)

(continued from previous page)

```
    """
    Create an instance of IFP with the given parameters.

    """
    # Test stability assuming {R_t} is IID and ln R ~ N(b_r, a_r)
    ER = np.exp(b_r + a_r**2 / 2)
    assert β * ER < 1, "Stability condition failed."

    # Generate random draws using JAX
    key = jax.random.PRNGKey(seed)
    subkey1, subkey2 = jax.random.split(key)
    η_draws = jax.random.normal(subkey1, (shock_draw_size,))
    ζ_draws = jax.random.normal(subkey2, (shock_draw_size,))
    s_grid = jnp.linspace(0, grid_max, grid_size)

    return IFP(
        γ, β, P, a_r, b_r, a_y, b_y, s_grid, η_draws, ζ_draws
    )


def u_prime(c, γ):
    """Marginal utility"""
    return c**(-γ)

def u_prime_inv(c, γ):
    """Inverse of marginal utility"""
    return c**(-1/γ)

def R(z, ζ, a_r, b_r):
    """Gross return on assets"""
    return jnp.exp(a_r * ζ + b_r)

def Y(z, η, a_y, b_y):
    """Labor income"""
    return jnp.exp(a_y * η + (z * b_y))
```

Here's the Coleman-Reffett operator using JAX:

```
def K(
        a_in: jnp.array,    # a_in[i, z] is an asset grid
        c_in: jnp.array,    # c_in[i, z] = consumption at a_in[i, z]
        ifp: IFP
    ):
    """
    The Coleman--Reffett operator for the income fluctuation problem,
    using the endogenous grid method with JAX.

    """

    # Extract parameters from ifp
    γ, β, P, a_r, b_r, a_y, b_y, s_grid, η_draws, ζ_draws = ifp
    n = len(P)

    def compute_expectation(s, z):
        def inner_expectation(z_hat):
            def compute_term(η, ζ):
                R_hat = R(z_hat, ζ, a_r, b_r)
```

(continues on next page)

(continued from previous page)

```
                Y_hat = Y(z_hat, η, a_y, b_y)
                a_val = R_hat * s + Y_hat
                # Interpolate consumption
                c_interp = jnp.interp(a_val, a_in[:, z_hat], c_in[:, z_hat])
                mu = u_prime(c_interp, γ)
                return R_hat * mu
            # Vectorize over all shock combinations
            η_grid, ζ_grid = jnp.meshgrid(η_draws, ζ_draws, indexing='ij')
            terms = vmap(vmap(compute_term))(η_grid, ζ_grid)
            return P[z, z_hat] * jnp.mean(terms)
        # Sum over z_hat states
        Ez = jnp.sum(vmap(inner_expectation)(jnp.arange(n)))
        return u_prime_inv(β * Ez, γ)

    # Vectorize over s_grid and z
    compute_exp_v1 = vmap(compute_expectation, in_axes=(None, 0))
    compute_exp_v2 = vmap(compute_exp_v1,      in_axes=(0, None))
    c_out = compute_exp_v2(s_grid, jnp.arange(n))
    # Calculate endogenous asset grid
    a_out = s_grid[:, None] + c_out
    # Fix consumption-asset pair at (0, 0)
    c_out = c_out.at[0, :].set(0)
    a_out = a_out.at[0, :].set(0)

    return a_out, c_out
```

The next function solves for an approximation of the optimal consumption policy via time iteration using JAX:

```
@jax.jit
def solve_model(
        ifp: IFP,
        c_init: jnp.ndarray,  # Initial guess of σ on grid endogenous grid
        a_init: jnp.ndarray,  # Initial endogenous grid
        tol: float = 1e-5,
        max_iter: int = 1000
    ) -> jnp.ndarray:
    " Solve the model using time iteration with EGM. "

    def condition(loop_state):
        c_in, a_in, i, error = loop_state
        return (error > tol) & (i < max_iter)

    def body(loop_state):
        c_in, a_in, i, error = loop_state
        c_out, a_out = K(c_in, a_in, ifp)
        error = jnp.max(jnp.abs(c_out - c_in))
        i += 1
        return c_out, a_out, i, error

    i, error = 0, tol + 1
    initial_state = (c_init, a_init, i, error)
    final_loop_state = jax.lax.while_loop(condition, body, initial_state)
    c_out, a_out, i, error = final_loop_state

    return c_out, a_out
```

Now we can create an instance and solve the model using JAX:

```
ifp = create_ifp()
```

Set up the initial condition:

```
# Initial guess of σ = consume all assets
k = len(ifp.s_grid)
n = len(ifp.P)
σ_init = jnp.empty((k, n))
for z in range(n):
    σ_init = σ_init.at[:, z].set(ifp.s_grid)
a_init = σ_init.copy()
```

Let's generate an approximation solution with JAX:

```
a_star, σ_star = solve_model(ifp, a_init, σ_init)
```

Let's try it again with a timer.

```
with qe.Timer(precision=8):
    a_star, σ_star = solve_model(ifp, a_init, σ_init)
    a_star.block_until_ready()
```

```
0.43271589 seconds elapsed
```

63.5 Simulation

Let's return to the default model and study the stationary distribution of assets.

Our plan is to run a large number of households forward for T periods and then histogram the cross-sectional distribution of assets.

Set `num_households=50_000, T=500`.

First we write a function to run a single household forward in time and record the final value of assets.

The function takes a solution pair `c_vec` and `a_vec`, understanding them as representing an optimal policy associated with a given model `ifp`

```
def simulate_household(
        key, a_0, z_idx_0, c_vec, a_vec, ifp, T
    ):
    """
    Simulates a single household for T periods to approximate the stationary
    distribution of assets.

    - key is the state of the random number generator
    - ifp is an instance of IFP
    - c_vec, a_vec are the optimal consumption policy, endogenous grid for ifp

    """
    # Extract parameters from ifp
    γ, β, P, a_r, b_r, a_y, b_y, s_grid, η_draws, ζ_draws = ifp
    n_z = len(P)

    # Create interpolation function for consumption policy
```

(continues on next page)

(continued from previous page)

```
    σ = lambda a, z_idx: jnp.interp(a, a_vec[:, z_idx], c_vec[:, z_idx])

    # Simulate forward T periods
    def update(t, state):
        a, z_idx = state
        # Draw next shock z' from P[z, z']
        current_key = jax.random.fold_in(key, 3*t)
        z_next_idx = jax.random.choice(current_key, n_z, p=P[z_idx]).astype(jnp.int32)
        # Draw η shock for income
        η_key = jax.random.fold_in(key, 3*t + 1)
        η = jax.random.normal(η_key)
        # Draw ζ shock for return
        ζ_key = jax.random.fold_in(key, 3*t + 2)
        ζ = jax.random.normal(ζ_key)
        # Compute stochastic return
        R_next = R(z_next_idx, ζ, a_r, b_r)
        # Compute income
        Y_next = Y(z_next_idx, η, a_y, b_y)
        # Update assets: a' = R' * (a - c) + Y'
        a_next = R_next * (a - σ(a, z_idx)) + Y_next
        # Return updated state
        return a_next, z_next_idx

    initial_state = a_0, z_idx_0
    final_state = jax.lax.fori_loop(0, T, update, initial_state)
    a_final, _ = final_state
    return a_final
```

Now we write a function to simulate many households in parallel.

```
@partial(jax.jit, static_argnums=(3, 4, 5))
def compute_asset_stationary(
        c_vec, a_vec, ifp, num_households=50_000, T=500, seed=1234
    ):
    """
    Simulates num_households households for T periods to approximate
    the stationary distribution of assets.

    Returns the final cross-section of asset holdings.

    - ifp is an instance of IFP
    - c_vec, a_vec are the optimal consumption policy and endogenous grid.

    """
    # Extract parameters from ifp
    γ, β, P, a_r, b_r, a_y, b_y, s_grid, η_draws, ζ_draws = ifp

    # Start with assets = savings_grid_max / 2
    a_0_vector = jnp.full(num_households, s_grid[-1] / 2)
    # Initialize the exogenous state of each household
    z_idx_0_vector = jnp.zeros(num_households).astype(jnp.int32)

    # Vectorize over many households
    key = jax.random.PRNGKey(seed)
    keys = jax.random.split(key, num_households)
    # Vectorize simulate_household in (key, a_0, z_idx_0)
    sim_all_households = jax.vmap(
```

(continues on next page)

(continued from previous page)

```
        simulate_household, in_axes=(0, 0, 0, None, None, None, None)
    )
    assets = sim_all_households(keys, a_0_vector, z_idx_0_vector, c_vec, a_vec, ifp,␣
↪T)

    return jnp.array(assets)
```

We'll need some inequality measures for visualization, so let's define them first:

```
def gini_coefficient(x):
    """
    Compute the Gini coefficient for array x.

    """
    x = jnp.asarray(x)
    n = len(x)
    x_sorted = jnp.sort(x)
    # Compute Gini coefficient
    cumsum = jnp.cumsum(x_sorted)
    a = (2 * jnp.sum((jnp.arange(1, n+1)) * x_sorted)) / (n * cumsum[-1])
    return a - (n + 1) / n


def top_share(
        x: jnp.array,    # array of wealth values
        p: float=0.01    # fraction of top households (default 0.01 for top 1%)
    ):
    """
    Compute the share of total wealth held by the top p fraction of households.

    """
    x = jnp.asarray(x)
    x_sorted = jnp.sort(x)
    # Number of households in top p%
    n_top = int(jnp.ceil(len(x) * p))
    # Wealth held by top p%
    wealth_top = jnp.sum(x_sorted[-n_top:])
    # Total wealth
    wealth_total = jnp.sum(x_sorted)
    return wealth_top / wealth_total
```

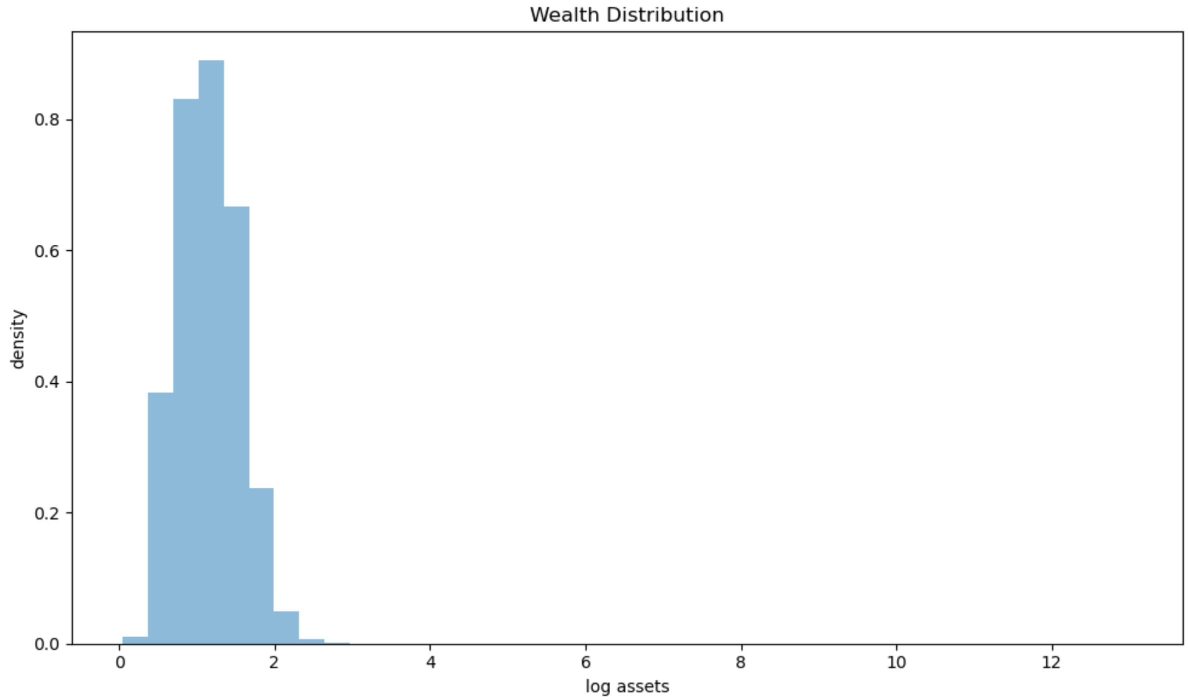
Now we call the function, generate the asset distribution and visualize it:

```
ifp = create_ifp()
# Extract parameters for initialization
s_grid = ifp.s_grid
n_z = len(ifp.P)
a_init = s_grid[:, None] * jnp.ones(n_z)
c_init = a_init
a_vec, c_vec = solve_model(ifp, a_init, c_init)
assets = compute_asset_stationary(c_vec, a_vec, ifp, num_households=200_000)

# Compute Gini coefficient for the plot
gini_plot = gini_coefficient(assets)

# Plot histogram of log wealth
fig, ax = plt.subplots(figsize=(10, 6))
```

(continues on next page)

(continued from previous page)

```
ax.hist(jnp.log(assets), bins=40, alpha=0.5, density=True)
ax.set(xlabel='log assets', ylabel='density', title="Wealth Distribution")
plt.tight_layout()
plt.show()
```



The histogram shows the distribution of log wealth.

Bearing in mind that we are looking at log values, the histogram suggests a long right tail of the distribution.

Below we examine this in more detail.

63.6 Wealth Inequality

Lets' look at wealth inequality by computing some standard measures of this phenomenon.

We will also examine how inequality varies with the interest rate.

63.6.1 Measuring Inequality

Let's print the Gini coefficient and the top 1% wealth share from our simulation:

```
gini = gini_coefficient(assets)
top1 = top_share(assets, p=0.01)

print(f"Gini coefficient: {gini:.4f}")
print(f"Top 1% wealth share: {top1:.4f}")
```

```
Gini coefficient: 0.7847
Top 1% wealth share: 0.7306
```

Recent numbers suggest that

- the Gini coefficient for wealth in the US is around 0.8
- the top 1% wealth share is over 0.3

Our model with stochastic returns generates a Gini coefficient close to the empirical value, demonstrating that capital income risk is an important factor in wealth inequality.

The top 1% wealth share is, however, too large.

Our model needs proper calibration and additional work – we set these tasks aside for now.

63.7 Exercises

Exercise 63.7.1

Plot how the Gini coefficient varies with the volatility of returns on assets.

Specifically, compute the Gini coefficient for values of `a_r` ranging from 0.10 to 0.16 (use at least 5 different values) and plot the results.

What does this tell you about the relationship between capital income risk and wealth inequality?

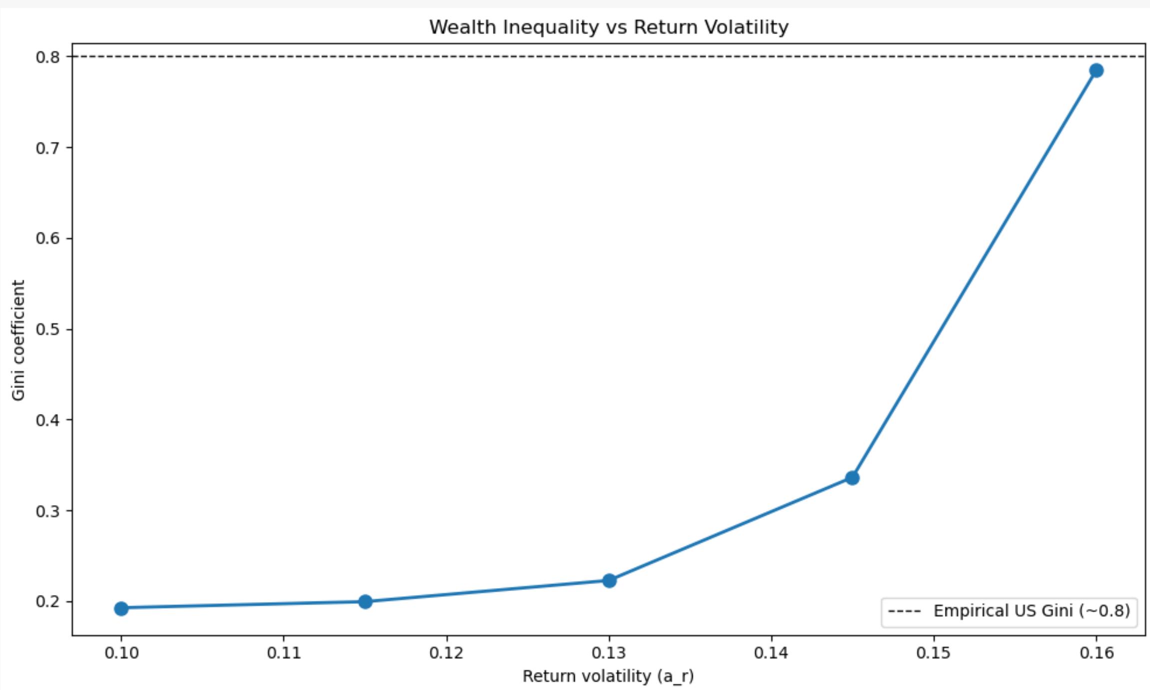
Solution

We loop over different values of `a_r`, solve the model for each, simulate the wealth distribution, and compute the Gini coefficient.

```
# Range of a_r values to explore
a_r_vals = np.linspace(0.10, 0.16, 5)
gini_vals = []

print("Computing Gini coefficients for different return volatilities...\n")

for a_r in a_r_vals:
    print(f"a_r = {a_r:.3f}...", end=" ")

    # Create model with this a_r value
    ifp_temp = create_ifp(a_r=a_r, grid_max=100)

    # Solve the model
    s_grid_temp = ifp_temp.s_grid
    n_z_temp = len(ifp_temp.P)
    a_init_temp = s_grid_temp[:, None] * jnp.ones(n_z_temp)
    c_init_temp = a_init_temp
    a_vec_temp, c_vec_temp = solve_model(
        ifp_temp, a_init_temp, c_init_temp
    )

    # Simulate households
    assets_temp = compute_asset_stationary(
        c_vec_temp, a_vec_temp, ifp_temp, num_households=200_000
    )

    # Compute Gini coefficient
```

```
    gini_temp = gini_coefficient(assets_temp)
    gini_vals.append(gini_temp)
    print(f"Gini = {gini_temp:.4f}")

# Plot the results
fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(a_r_vals, gini_vals, 'o-', linewidth=2, markersize=8)
ax.set(xlabel='Return volatility (a_r)',
       ylabel='Gini coefficient',
       title='Wealth Inequality vs Return Volatility')
ax.axhline(y=0.8, color='k', linestyle='--', linewidth=1,
           label='Empirical US Gini (~0.8)')
ax.legend()
plt.tight_layout()
plt.show()
```

```
Computing Gini coefficients for different return volatilities...

a_r = 0.100...
Gini = 0.1925
a_r = 0.115...
Gini = 0.1992
a_r = 0.130...
Gini = 0.2225
a_r = 0.145...
Gini = 0.3362
a_r = 0.160...
Gini = 0.7847
```



The plot shows that wealth inequality (measured by the Gini coefficient) increases with return volatility.

This demonstrates that capital income risk is a key driver of wealth inequality.

When returns are more volatile, lucky households who experience sequences of high returns accumulate substantially more wealth than unlucky households, leading to greater inequality in the wealth distribution.

Exercise 63.7.2

Plot how the Gini coefficient varies with the volatility of labor income.

Specifically, compute the Gini coefficient for values of `a_y` ranging from 0.125 to 0.20 and plot the results. Set `a_r=0.10` for this exercise.

What does this tell you about the relationship between labor income risk and wealth inequality? Can we achieve the same rise in inequality by varying labor income volatility as we can by varying return volatility?

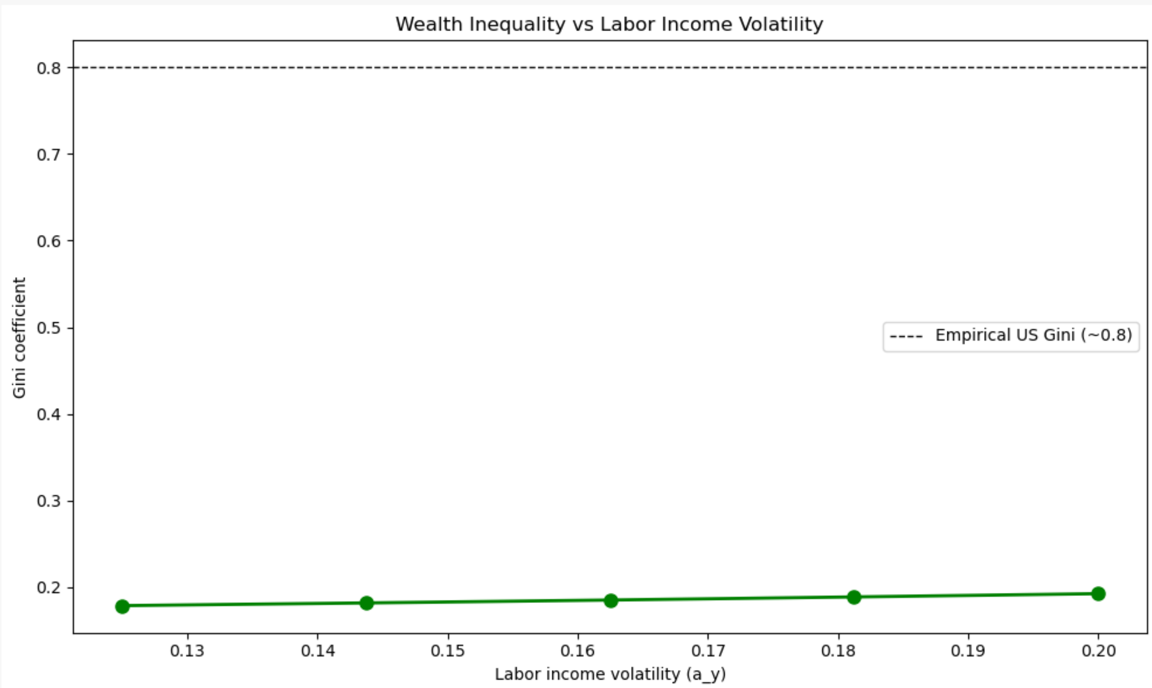
Solution

We loop over different values of `a_y`, solve the model for each, simulate the wealth distribution, and compute the Gini coefficient.

```
# Range of a_y values to explore
a_y_vals = np.linspace(0.125, 0.20, 5)
gini_vals_y = []

print("Computing Gini coefficients for different labor income volatilities...\n")

for a_y in a_y_vals:
    print(f"a_y = {a_y:.3f}...", end=" ")

    # Create model with this a_y value and a_r=0.10
    ifp_temp = create_ifp(a_y=a_y, a_r=0.10, grid_max=100)

    # Solve the model
    s_grid_temp = ifp_temp.s_grid
    n_z_temp = len(ifp_temp.P)
    a_init_temp = s_grid_temp[:, None] * jnp.ones(n_z_temp)
    c_init_temp = a_init_temp
    a_vec_temp, c_vec_temp = solve_model(
        ifp_temp, a_init_temp, c_init_temp
    )

    # Simulate households
    assets_temp = compute_asset_stationary(
        c_vec_temp, a_vec_temp, ifp_temp, num_households=200_000
    )

    # Compute Gini coefficient
    gini_temp = gini_coefficient(assets_temp)
    gini_vals_y.append(gini_temp)
    print(f"Gini = {gini_temp:.4f}")

# Plot the results
fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(a_y_vals, gini_vals_y, 'o-', linewidth=2, markersize=8, color='green')
ax.set(xlabel='Labor income volatility (a_y)',
       ylabel='Gini coefficient',
       title='Wealth Inequality vs Labor Income Volatility')
ax.axhline(y=0.8, color='k', linestyle='--', linewidth=1,
```

```
            label='Empirical US Gini (~0.8)')
ax.legend()
plt.tight_layout()
plt.show()
```

```
Computing Gini coefficients for different labor income volatilities...

a_y = 0.125...
Gini = 0.1787
a_y = 0.144...
Gini = 0.1818
a_y = 0.163...
Gini = 0.1852
a_y = 0.181...
Gini = 0.1887
a_y = 0.200...
Gini = 0.1925
```



The plot shows that wealth inequality increases with labor income volatility, but the effect is much weaker than the effect of return volatility.

Comparing the two exercises:

- When return volatility (`a_r`) varies from 0.10 to 0.16, the Gini coefficient rises dramatically from around 0.20 to 0.79
- When labor income volatility (`a_y`) varies from 0.125 to 0.20, a similar amount in percentage terms, the Gini coefficient increases but by a much smaller amount

This suggests that capital income risk is a more important driver of wealth inequality than labor income risk.

The intuition is that wealth accumulation compounds over time: households who experience favorable returns on their assets can reinvest those returns, leading to exponential growth.

In contrast, labor income shocks, while they affect current consumption and savings, do not have the same compounding effect on wealth accumulation.

Part X

LQ Control

CHAPTER

SIXTYFOUR

LQ CONTROL: FOUNDATIONS

Contents

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
```

64.1 Overview

Linear quadratic (LQ) control refers to a class of dynamic optimization problems that have found applications in almost every scientific field.

This lecture provides an introduction to LQ control and its economic applications.

As we will see, LQ systems have a simple structure that makes them an excellent workhorse for a wide variety of economic problems.

Moreover, while the linear-quadratic structure is restrictive, it is in fact far more flexible than it may appear initially.

These themes appear repeatedly below.

Mathematically, LQ control problems are closely related to *the Kalman filter*

- Recursive formulations of linear-quadratic control problems and Kalman filtering problems both involve matrix **Riccati equations**.
- Classical formulations of linear control and linear filtering problems make use of similar matrix decompositions (see for example this lecture and this lecture).

In reading what follows, it will be useful to have some familiarity with

- matrix manipulations
- vectors of random variables
- dynamic programming and the Bellman equation (see for example this lecture and *Optimal Savings III: Stochastic Returns*)

For additional reading on LQ control, see, for example,

- [Ljungqvist and Sargent, 2018], chapter 5
- [Hansen and Sargent, 2008], chapter 4
- [Hernandez-Lerma and Lasserre, 1996], section 3.5

In order to focus on computation, we leave longer proofs to these sources (while trying to provide as much intuition as possible).

Let's start with some imports:

```
import matplotlib.pyplot as plt
import numpy as np
from quantecon import LQ
```

64.2 Introduction

The "linear" part of LQ is a linear law of motion for the state, while the "quadratic" part refers to preferences.

Let's begin with the former, move on to the latter, and then put them together into an optimization problem.

64.2.1 The Law of Motion

Let x_t be a vector describing the state of some economic system.

Suppose that x_t follows a linear law of motion given by

$$x_{t+1} = Ax_t + Bu_t + Cw_{t+1}, \qquad t = 0, 1, 2, \ldots \tag{64.1}$$

Here

- u_t is a "control" vector, incorporating choices available to a decision-maker confronting the current state x_t
- $\{w_t\}$ is an uncorrelated zero mean shock process satisfying $\mathbb{E}w_t w_t' = I$, where the right-hand side is the identity matrix

Regarding the dimensions

- x_t is $n \times 1$, A is $n \times n$
- u_t is $k \times 1$, B is $n \times k$
- w_t is $j \times 1$, C is $n \times j$

Example 1

Consider a household budget constraint given by

$$a_{t+1} + c_t = (1+r)a_t + y_t$$

Here a_t is assets, r is a fixed interest rate, c_t is current consumption, and y_t is current non-financial income.

If we suppose that $\{y_t\}$ is serially uncorrelated and $N(0, \sigma^2)$, then, taking $\{w_t\}$ to be standard normal, we can write the system as

$$a_{t+1} = (1+r)a_t - c_t + \sigma w_{t+1}$$

This is clearly a special case of (64.1), with assets being the state and consumption being the control.

Example 2

One unrealistic feature of the previous model is that non-financial income has a zero mean and is often negative.

This can easily be overcome by adding a sufficiently large mean.

Hence in this example, we take $y_t = \sigma w_{t+1} + \mu$ for some positive real number μ.

Another alteration that's useful to introduce (we'll see why soon) is to change the control variable from consumption to the deviation of consumption from some "ideal" quantity $\bar{c}$.

(Most parameterizations will be such that $\bar{c}$ is large relative to the amount of consumption that is attainable in each period, and hence the household wants to increase consumption.)

For this reason, we now take our control to be $u_t := c_t - \bar{c}$.

In terms of these variables, the budget constraint $a_{t+1} = (1+r)a_t - c_t + y_t$ becomes

$$a_{t+1} = (1+r)a_t - u_t - \bar{c} + \sigma w_{t+1} + \mu \tag{64.2}$$

How can we write this new system in the form of equation (64.1)?

If, as in the previous example, we take a_t as the state, then we run into a problem: the law of motion contains some constant terms on the right-hand side.

This means that we are dealing with an *affine* function, not a linear one (recall *this discussion*).

Fortunately, we can easily circumvent this problem by adding an extra state variable.

In particular, if we write

$$\begin{pmatrix} a_{t+1} \\ 1 \end{pmatrix} = \begin{pmatrix} 1+r & -\bar{c}+\mu \\ 0 & 1 \end{pmatrix} \begin{pmatrix} a_t \\ 1 \end{pmatrix} + \begin{pmatrix} -1 \\ 0 \end{pmatrix} u_t + \begin{pmatrix} \sigma \\ 0 \end{pmatrix} w_{t+1} \tag{64.3}$$

then the first row is equivalent to (64.2).

Moreover, the model is now linear and can be written in the form of (64.1) by setting

$$x_t := \begin{pmatrix} a_t \\ 1 \end{pmatrix}, \quad A := \begin{pmatrix} 1+r & -\bar{c}+\mu \\ 0 & 1 \end{pmatrix}, \quad B := \begin{pmatrix} -1 \\ 0 \end{pmatrix}, \quad C := \begin{pmatrix} \sigma \\ 0 \end{pmatrix} \tag{64.4}$$

In effect, we've bought ourselves linearity by adding another state.

64.2.2 Preferences

In the LQ model, the aim is to minimize flow of losses, where time-t loss is given by the quadratic expression

$$x_t' R x_t + u_t' Q u_t \tag{64.5}$$

Here

- R is assumed to be $n \times n$, symmetric and nonnegative definite.
- Q is assumed to be $k \times k$, symmetric and positive definite.

> **Note**
>
> In fact, for many economic problems, the definiteness conditions on R and Q can be relaxed. It is sufficient that certain submatrices of R and Q be nonnegative definite. See [Hansen and Sargent, 2008] for details.

Example 1

A very simple example that satisfies these assumptions is to take R and Q to be identity matrices so that current loss is

$$x_t' I x_t + u_t' I u_t = \|x_t\|^2 + \|u_t\|^2$$

Thus, for both the state and the control, loss is measured as squared distance from the origin.

(In fact, the general case (64.5) can also be understood in this way, but with R and Q identifying other – non-Euclidean – notions of "distance" from the zero vector.)

Intuitively, we can often think of the state x_t as representing deviation from a target, such as

- deviation of inflation from some target level
- deviation of a firm's capital stock from some desired quantity

The aim is to put the state close to the target, while using controls parsimoniously.

Example 2

In the household problem *studied above*, setting $R = 0$ and $Q = 1$ yields preferences

$$x_t' R x_t + u_t' Q u_t = u_t^2 = (c_t - \bar c)^2$$

Under this specification, the household's current loss is the squared deviation of consumption from the ideal level $\bar c$.

64.3 Optimality – Finite Horizon

Let's now be precise about the optimization problem we wish to consider, and look at how to solve it.

64.3.1 The Objective

We will begin with the finite horizon case, with terminal time $T \in \mathbb{N}$.

In this case, the aim is to choose a sequence of controls $\{u_0, \dots, u_{T-1}\}$ to minimize the objective

$$\mathbb{E}\left\{\sum_{t=0}^{T-1} \beta^t (x_t' R x_t + u_t' Q u_t) + \beta^T x_T' R_f x_T\right\} \tag{64.6}$$

subject to the law of motion (64.1) and initial state x_0.

The new objects introduced here are β and the matrix R_f.

The scalar β is the discount factor, while $x' R_f x$ gives terminal loss associated with state x.

Comments:

- We assume R_f to be $n \times n$, symmetric and nonnegative definite.
- We allow $\beta = 1$, and hence include the undiscounted case.
- x_0 may itself be random, in which case we require it to be independent of the shock sequence $w_1, \dots, w_T$.

64.3.2 Information

There's one constraint we've neglected to mention so far, which is that the decision-maker who solves this LQ problem knows only the present and the past, not the future.

To clarify this point, consider the sequence of controls $\{u_0, \dots, u_{T-1}\}$.

When choosing these controls, the decision-maker is permitted to take into account the effects of the shocks $\{w_1, \dots, w_T\}$ on the system.

However, it is typically assumed — and will be assumed here — that the time-t control u_t can be made with knowledge of past and present shocks only.

The fancy measure-theoretic way of saying this is that u_t must be measurable with respect to the σ-algebra generated by $x_0, w_1, w_2, \dots, w_t$.

This is in fact equivalent to stating that u_t can be written in the form $u_t = g_t(x_0, w_1, w_2, \dots, w_t)$ for some Borel measurable function g_t.

(Just about every function that's useful for applications is Borel measurable, so, for the purposes of intuition, you can read that last phrase as "for some function g_t")

Now note that x_t will ultimately depend on the realizations of $x_0, w_1, w_2, \dots, w_t$.

In fact, it turns out that x_t summarizes all the information about these historical shocks that the decision-maker needs to set controls optimally.

More precisely, it can be shown that any optimal control u_t can always be written as a function of the current state alone.

Hence in what follows we restrict attention to control policies (i.e., functions) of the form $u_t = g_t(x_t)$.

Actually, the preceding discussion applies to all standard dynamic programming problems.

What's special about the LQ case is that – as we shall soon see — the optimal u_t turns out to be a linear function of x_t.

64.3.3 Solution

To solve the finite horizon LQ problem we can use a dynamic programming strategy based on backward induction that is conceptually similar to the approach adopted in this lecture.

For reasons that will soon become clear, we first introduce the notation $J_T(x) = x' R_f x$.

Now consider the problem of the decision-maker in the second to last period.

In particular, let the time be $T - 1$, and suppose that the state is x_{T-1}.

The decision-maker must trade-off current and (discounted) final losses, and hence solves

$$\min_u \{x'_{T-1} R x_{T-1} + u' Q u + \beta \, \mathbb{E} J_T(A x_{T-1} + B u + C w_T)\}$$

At this stage, it is convenient to define the function

$$J_{T-1}(x) = \min_u \{x' R x + u' Q u + \beta \, \mathbb{E} J_T(A x + B u + C w_T)\} \tag{64.7}$$

The function J_{T-1} will be called the $T-1$ value function, and $J_{T-1}(x)$ can be thought of as representing total "loss-to-go" from state x at time $T - 1$ when the decision-maker behaves optimally.

Now let's step back to $T - 2$.

For a decision-maker at $T-2$, the value $J_{T-1}(x)$ plays a role analogous to that played by the terminal loss $J_T(x) = x' R_f x$ for the decision-maker at $T - 1$.

That is, $J_{T-1}(x)$ summarizes the future loss associated with moving to state x.

The decision-maker chooses her control u to trade off current loss against future loss, where

- the next period state is $x_{T-1} = A x_{T-2} + B u + C w_{T-1}$, and hence depends on the choice of current control.
- the "cost" of landing in state x_{T-1} is $J_{T-1}(x_{T-1})$.

Her problem is therefore

$$\min_u \{x'_{T-2} R x_{T-2} + u' Q u + \beta \, \mathbb{E} J_{T-1}(A x_{T-2} + B u + C w_{T-1})\}$$

Letting

$$J_{T-2}(x) = \min_u \{x' R x + u' Q u + \beta \, \mathbb{E} J_{T-1}(A x + B u + C w_{T-1})\}$$

the pattern for backward induction is now clear.

In particular, we define a sequence of value functions $\{J_0, \dots, J_T\}$ via

$$J_{t-1}(x) = \min_u \{x' R x + u' Q u + \beta \, \mathbb{E} J_t(A x + B u + C w_t)\} \quad \text{and} \quad J_T(x) = x' R_f x$$

The first equality is the Bellman equation from dynamic programming theory specialized to the finite horizon LQ problem.

Now that we have $\{J_0, \dots, J_T\}$, we can obtain the optimal controls.

As a first step, let's find out what the value functions look like.

It turns out that every J_t has the form $J_t(x) = x' P_t x + d_t$ where P_t is a $n \times n$ matrix and d_t is a constant.

We can show this by induction, starting from $P_T := R_f$ and $d_T = 0$.

Using this notation, (64.7) becomes

$$J_{T-1}(x) = \min_u \{x' R x + u' Q u + \beta \, \mathbb{E}(A x + B u + C w_T)' P_T (A x + B u + C w_T)\} \tag{64.8}$$

To obtain the minimizer, we can take the derivative of the r.h.s. with respect to u and set it equal to zero.

Applying the relevant rules of *matrix calculus*, this gives

$$u = -(Q + \beta B' P_T B)^{-1} \beta B' P_T A x \tag{64.9}$$

Plugging this back into (64.8) and rearranging yields

$$J_{T-1}(x) = x' P_{T-1} x + d_{T-1}$$

where

$$P_{T-1} = R - \beta^2 A' P_T B (Q + \beta B' P_T B)^{-1} B' P_T A + \beta A' P_T A \tag{64.10}$$

and

$$d_{T-1} := \beta \operatorname{trace}(C' P_T C) \tag{64.11}$$

(The algebra is a good exercise — we'll leave it up to you.)

If we continue working backwards in this manner, it soon becomes clear that $J_t(x) = x' P_t x + d_t$ as claimed, where $\{P_t\}$ and $\{d_t\}$ satisfy the recursions

$$P_{t-1} = R - \beta^2 A' P_t B (Q + \beta B' P_t B)^{-1} B' P_t A + \beta A' P_t A \quad \text{with} \quad P_T = R_f \tag{64.12}$$

and

$$d_{t-1} = \beta (d_t + \operatorname{trace}(C' P_t C)) \quad \text{with} \quad d_T = 0 \tag{64.13}$$

Recalling (64.9), the minimizers from these backward steps are

$$u_t = -F_t x_t \quad \text{where} \quad F_t := (Q + \beta B' P_{t+1} B)^{-1} \beta B' P_{t+1} A \tag{64.14}$$

These are the linear optimal control policies we *discussed above*.

In particular, the sequence of controls given by (64.14) and (64.1) solves our finite horizon LQ problem.

Rephrasing this more precisely, the sequence $u_0, \ldots, u_{T-1}$ given by

$$u_t = -F_t x_t \quad \text{with} \quad x_{t+1} = (A - BF_t) x_t + C w_{t+1} \tag{64.15}$$

for $t = 0, \ldots, T-1$ attains the minimum of (64.6) subject to our constraints.

64.4 Implementation

We will use code from lqcontrol.py in QuantEcon.py to solve finite and infinite horizon linear quadratic control problems.

In the module, the various updating, simulation and fixed point methods are wrapped in a class called `LQ`, which includes

- Instance data:
  - The required parameters Q, R, A, B and optional parameters C, β, T, R_f, N specifying a given LQ model
    - set T and R_f to `None` in the infinite horizon case
    - set `C = None` (or zero) in the deterministic case
  - the value function and policy data
    - d_t, P_t, F_t in the finite horizon case

    * d, P, F in the infinite horizon case

- Methods:
  - `update_values` — shifts d_t, P_t, F_t to their $t-1$ values via (64.12), (64.13) and (64.14)
  - `stationary_values` — computes P, d, F in the infinite horizon case
  - `compute_sequence` —- simulates the dynamics of x_t, u_t, w_t given x_0 and assuming standard normal shocks

64.4.1 An Application

Early Keynesian models assumed that households have a constant marginal propensity to consume from current income.

Data contradicted the constancy of the marginal propensity to consume.

In response, Milton Friedman, Franco Modigliani and others built models based on a consumer's preference for an intertemporally smooth consumption stream.

(See, for example, [Friedman, 1956] or [Modigliani and Brumberg, 1954].)

One property of those models is that households purchase and sell financial assets to make consumption streams smoother than income streams.

The household savings problem *outlined above* captures these ideas.

The optimization problem for the household is to choose a consumption sequence in order to minimize

$$\mathbb{E}\left\{\sum_{t=0}^{T-1}\beta^t(c_t-\bar{c})^2+\beta^T q a_T^2\right\} \tag{64.16}$$

subject to the sequence of budget constraints $a_{t+1}=(1+r)a_t-c_t+y_t,\ t\geq 0$.

Here q is a large positive constant, the role of which is to induce the consumer to target zero debt at the end of her life.

(Without such a constraint, the optimal choice is to choose $c_t=\bar{c}$ in each period, letting assets adjust accordingly.)

As before we set $y_t=\sigma w_{t+1}+\mu$ and $u_t := c_t-\bar{c}$, after which the constraint can be written as in (64.2).

We saw how this constraint could be manipulated into the LQ formulation $x_{t+1}=Ax_t+Bu_t+Cw_{t+1}$ by setting $x_t=(a_t\ 1)'$ and using the definitions in (64.4).

To match with this state and control, the objective function (64.16) can be written in the form of (64.6) by choosing

$$Q := 1, \quad R := \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}, \quad \text{and} \quad R_f := \begin{pmatrix} q & 0 \\ 0 & 0 \end{pmatrix}$$

Now that the problem is expressed in LQ form, we can proceed to the solution by applying (64.12) and (64.14).

After generating shocks $w_1, \ldots, w_T$, the dynamics for assets and consumption can be simulated via (64.15).

The following figure was computed using $r=0.05, \beta=1/(1+r), \bar{c}=2, \mu=1, \sigma=0.25, T=45$ and $q=10^6$.

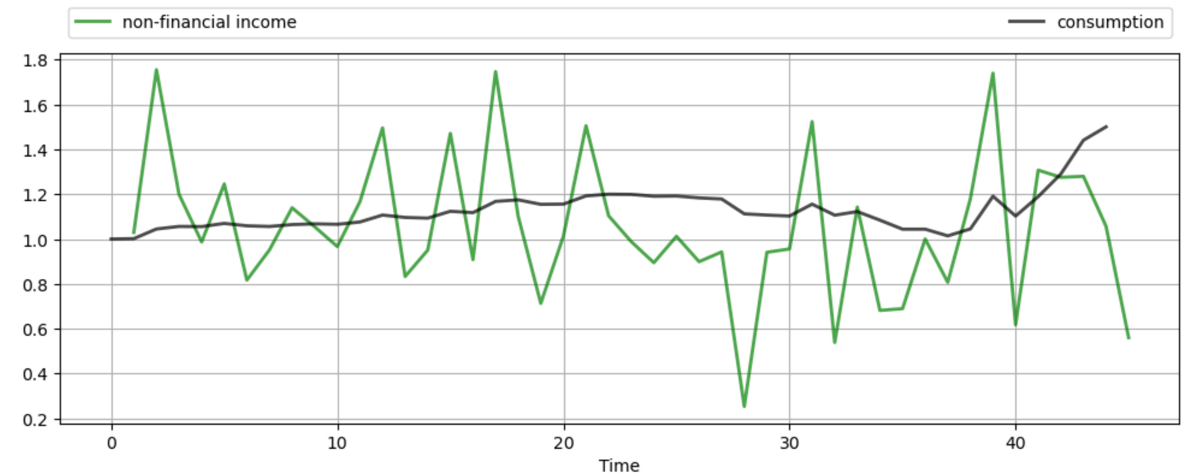
The shocks $\{w_t\}$ were taken to be IID and standard normal.

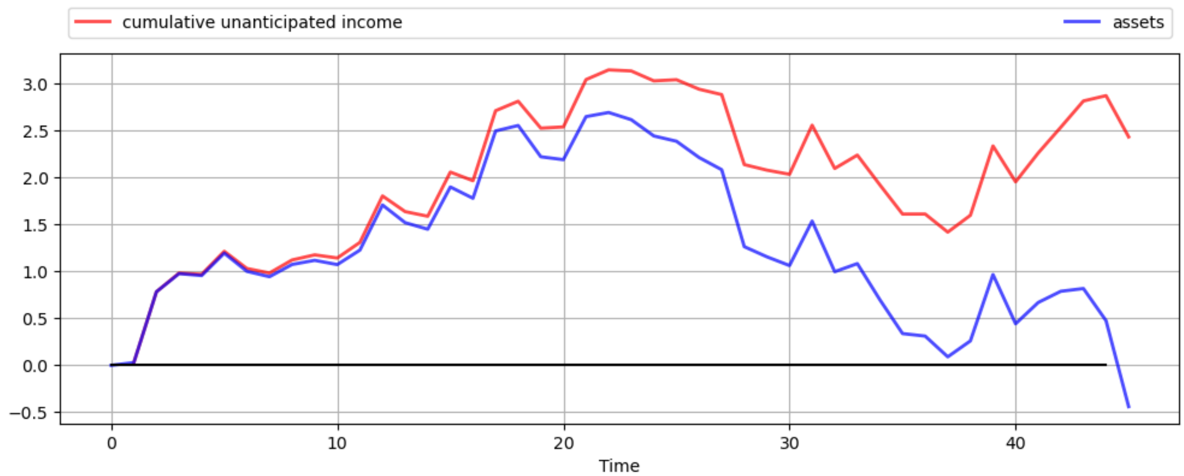
```
# Model parameters
r = 0.05
β = 1/(1 + r)
T = 45
c_bar = 2
σ = 0.25
```

(continues on next page)

(continued from previous page)

```
μ = 1
q = 1e6

# Formulate as an LQ problem
Q = 1
R = np.zeros((2, 2))
Rf = np.zeros((2, 2))
Rf[0, 0] = q
A = [[1 + r, -c_bar + μ],
    [0,              1]]
B = [[-1],
    [ 0]]
C = [[σ],
    [0]]

# Compute solutions and simulate
lq = LQ(Q, R, A, B, C, beta=β, T=T, Rf=Rf)
x0 = (0, 1)
xp, up, wp = lq.compute_sequence(x0)

# Convert back to assets, consumption and income
assets = xp[0, :]           # a_t
c = up.flatten() + c_bar    # c_t
income = σ * wp[0, 1:] + μ  # y_t

# Plot results
n_rows = 2
fig, axes = plt.subplots(n_rows, 1, figsize=(12, 10))

plt.subplots_adjust(hspace=0.5)

bbox = (0., 1.02, 1., .102)
legend_args = {'bbox_to_anchor': bbox, 'loc': 3, 'mode': 'expand'}
p_args = {'lw': 2, 'alpha': 0.7}

axes[0].plot(list(range(1, T+1)), income, 'g-', label="non-financial income",
            **p_args)
axes[0].plot(list(range(T)), c, 'k-', label="consumption", **p_args)

axes[1].plot(list(range(1, T+1)), np.cumsum(income - μ), 'r-',
            label="cumulative unanticipated income", **p_args)
axes[1].plot(list(range(T+1)), assets, 'b-', label="assets", **p_args)
axes[1].plot(list(range(T)), np.zeros(T), 'k-')

for ax in axes:
    ax.grid()
    ax.set_xlabel('Time')
    ax.legend(ncol=2, **legend_args)

plt.show()
```





The top panel shows the time path of consumption c_t and income y_t in the simulation.

As anticipated by the discussion on consumption smoothing, the time path of consumption is much smoother than that for income.

(But note that consumption becomes more irregular towards the end of life, when the zero final asset requirement impinges more on consumption choices.)

The second panel in the figure shows that the time path of assets a_t is closely correlated with cumulative unanticipated income, where the latter is defined as

$$z_t := \sum_{j=0}^{t} \sigma w_t$$

A key message is that unanticipated windfall gains are saved rather than consumed, while unanticipated negative shocks are met by reducing assets.

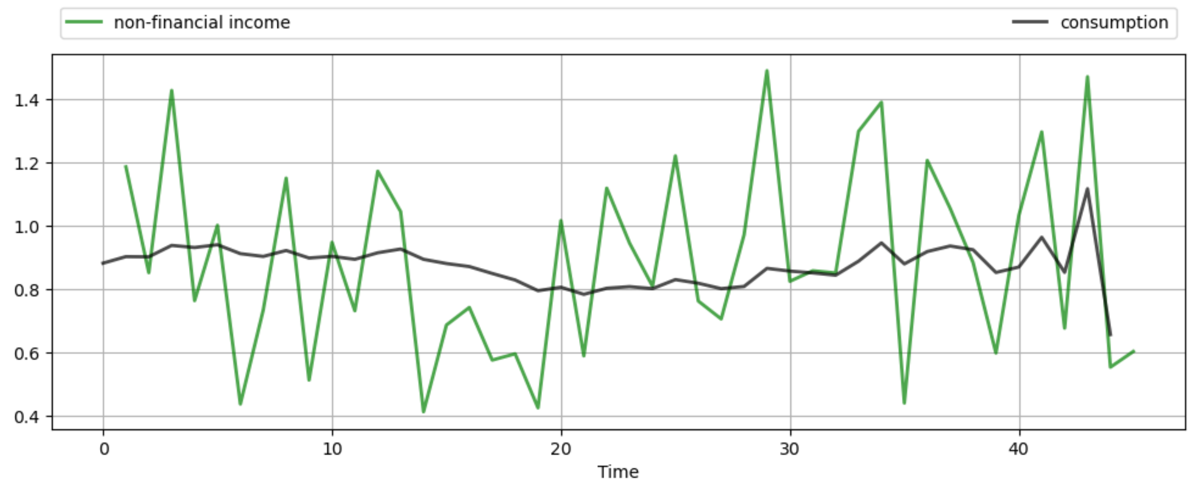
(Again, this relationship breaks down towards the end of life due to the zero final asset requirement.)

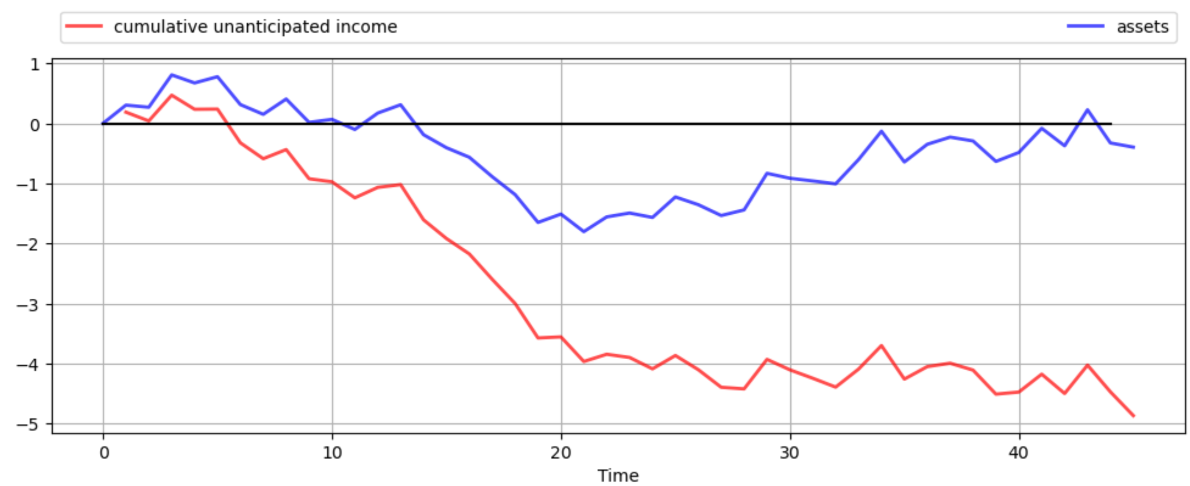
These results are relatively robust to changes in parameters.

For example, let's increase β from $1/(1 + r) \approx 0.952$ to 0.96 while keeping other parameters fixed.

This consumer is slightly more patient than the last one, and hence puts relatively more weight on later consumption values.

```python
# Compute solutions and simulate
lq = LQ(Q, R, A, B, C, beta=0.96, T=T, Rf=Rf)
x0 = (0, 1)
xp, up, wp = lq.compute_sequence(x0)

# Convert back to assets, consumption and income
assets = xp[0, :]           # a_t
c = up.flatten() + c_bar    # c_t
income = σ * wp[0, 1:] + μ  # y_t

# Plot results
n_rows = 2
fig, axes = plt.subplots(n_rows, 1, figsize=(12, 10))

plt.subplots_adjust(hspace=0.5)

bbox = (0., 1.02, 1., .102)
legend_args = {'bbox_to_anchor': bbox, 'loc': 3, 'mode': 'expand'}
p_args = {'lw': 2, 'alpha': 0.7}

axes[0].plot(list(range(1, T+1)), income, 'g-', label="non-financial income",
             **p_args)
axes[0].plot(list(range(T)), c, 'k-', label="consumption", **p_args)

axes[1].plot(list(range(1, T+1)), np.cumsum(income - μ), 'r-',
             label="cumulative unanticipated income", **p_args)
axes[1].plot(list(range(T+1)), assets, 'b-', label="assets", **p_args)
axes[1].plot(list(range(T)), np.zeros(T), 'k-')

for ax in axes:
    ax.grid()
    ax.set_xlabel('Time')
    ax.legend(ncol=2, **legend_args)

plt.show()
```





We now have a slowly rising consumption stream and a hump-shaped build-up of assets in the middle periods to fund rising consumption.

However, the essential features are the same: consumption is smooth relative to income, and assets are strongly positively correlated with cumulative unanticipated income.

64.5 Extensions and Comments

Let's now consider a number of standard extensions to the LQ problem treated above.

64.5.1 Time-Varying Parameters

In some settings, it can be desirable to allow A, B, C, R and Q to depend on t.

For the sake of simplicity, we've chosen not to treat this extension in our implementation given below.

However, the loss of generality is not as large as you might first imagine.

In fact, we can tackle many models with time-varying parameters by suitable choice of state variables.

One illustration is given *below*.

For further examples and a more systematic treatment, see [Hansen and Sargent, 2013], section 2.4.

64.5.2 Adding a Cross-Product Term

In some LQ problems, preferences include a cross-product term $u_t' N x_t$, so that the objective function becomes

$$\mathbb{E}\left\{\sum_{t=0}^{T-1} \beta^t (x_t' R x_t + u_t' Q u_t + 2u_t' N x_t) + \beta^T x_T' R_f x_T\right\} \tag{64.17}$$

Our results extend to this case in a straightforward way.

The sequence $\{P_t\}$ from (64.12) becomes

$$P_{t-1} = R - (\beta B' P_t A + N)' (Q + \beta B' P_t B)^{-1} (\beta B' P_t A + N) + \beta A' P_t A \quad \text{with} \quad P_T = R_f \tag{64.18}$$

The policies in (64.14) are modified to

$$u_t = -F_t x_t \quad \text{where} \quad F_t := (Q + \beta B' P_{t+1} B)^{-1} (\beta B' P_{t+1} A + N) \tag{64.19}$$

The sequence $\{d_t\}$ is unchanged from (64.13).

We leave interested readers to confirm these results (the calculations are long but not overly difficult).

64.5.3 Infinite Horizon

Finally, we consider the infinite horizon case, with *cross-product term*, unchanged dynamics and objective function given by

$$\mathbb{E}\left\{\sum_{t=0}^{\infty} \beta^t (x_t' R x_t + u_t' Q u_t + 2u_t' N x_t)\right\} \tag{64.20}$$

In the infinite horizon case, optimal policies can depend on time only if time itself is a component of the state vector x_t.

In other words, there exists a fixed matrix F such that $u_t = -F x_t$ for all t.

That decision rules are constant over time is intuitive — after all, the decision-maker faces the same infinite horizon at every stage, with only the current state changing.

Not surprisingly, P and d are also constant.

The stationary matrix P is the solution to the discrete-time algebraic Riccati equation.

$$P = R - (\beta B' P A + N)' (Q + \beta B' P B)^{-1} (\beta B' P A + N) + \beta A' P A \tag{64.21}$$

Equation (64.21) is also called the *LQ Bellman equation*, and the map that sends a given P into the right-hand side of (64.21) is called the *LQ Bellman operator*.

The stationary optimal policy for this model is

$$u = -Fx \quad \text{where} \quad F = (Q + \beta B' P B)^{-1} (\beta B' P A + N) \tag{64.22}$$

The sequence $\{d_t\}$ from (64.13) is replaced by the constant value

$$d := \operatorname{trace}(C' P C) \frac{\beta}{1 - \beta} \tag{64.23}$$

The state evolves according to the time-homogeneous process $x_{t+1} = (A - BF)x_t + Cw_{t+1}$.

An example infinite horizon problem is treated *below*.

64.5.4 Certainty Equivalence

Linear quadratic control problems of the class discussed above have the property of *certainty equivalence*.

By this, we mean that the optimal policy F is not affected by the parameters in C, which specify the shock process.

This can be confirmed by inspecting (64.22) or (64.19).

It follows that we can ignore uncertainty when solving for optimal behavior, and plug it back in when examining optimal state dynamics.

64.6 Further Applications

64.6.1 Application 1: Age-Dependent Income Process

Previously we studied a permanent income model that generated consumption smoothing.

One unrealistic feature of that model is the assumption that the mean of the random income process does not depend on the consumer's age.

A more realistic income profile is one that rises in early working life, peaks towards the middle and maybe declines toward the end of working life and falls more during retirement.

In this section, we will model this rise and fall as a symmetric inverted "U" using a polynomial in age.

As before, the consumer seeks to minimize

$$\mathbb{E} \left\{ \sum_{t=0}^{T-1} \beta^t (c_t - \bar{c})^2 + \beta^T q a_T^2 \right\} \tag{64.24}$$

subject to $a_{t+1} = (1 + r)a_t - c_t + y_t, \ t \geq 0$.

For income we now take $y_t = p(t) + \sigma w_{t+1}$ where $p(t) := m_0 + m_1 t + m_2 t^2$.

(In *the next section* we employ some tricks to implement a more sophisticated model.)

The coefficients m_0, m_1, m_2 are chosen such that $p(0) = 0, p(T/2) = \mu$, and $p(T) = 0$.

You can confirm that the specification $m_0 = 0, m_1 = T\mu/(T/2)^2, m_2 = -\mu/(T/2)^2$ satisfies these constraints.

To put this into an LQ setting, consider the budget constraint, which becomes

$$a_{t+1} = (1 + r)a_t - u_t - \bar{c} + m_1 t + m_2 t^2 + \sigma w_{t+1} \tag{64.25}$$

The fact that a_{t+1} is a linear function of $(a_t, 1, t, t^2)$ suggests taking these four variables as the state vector x_t.

Once a good choice of state and control (recall $u_t = c_t - \bar{c}$) has been made, the remaining specifications fall into place relatively easily.

Thus, for the dynamics we set

$$x_t := \begin{pmatrix} a_t \\ 1 \\ t \\ t^2 \end{pmatrix}, \quad A := \begin{pmatrix} 1+r & -\bar{c} & m_1 & m_2 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 2 & 1 \end{pmatrix}, \quad B := \begin{pmatrix} -1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \quad C := \begin{pmatrix} \sigma \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{64.26}$$

If you expand the expression $x_{t+1} = Ax_t + Bu_t + Cw_{t+1}$ using this specification, you will find that assets follow (64.25) as desired and that the other state variables also update appropriately.

To implement preference specification (64.24) we take

$$Q := 1, \quad R := \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad \text{and} \quad R_f := \begin{pmatrix} q & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \tag{64.27}$$

The next figure shows a simulation of consumption and assets computed using the `compute_sequence` method of `lqcontrol.py` with initial assets set to zero.

Once again, smooth consumption is a dominant feature of the sample paths.

The asset path exhibits dynamics consistent with standard life cycle theory.

Exercise 64.7.1 gives the full set of parameters used here and asks you to replicate the figure.

64.6.2 Application 2: A Permanent Income Model with Retirement

In the *previous application*, we generated income dynamics with an inverted U shape using polynomials and placed them in an LQ framework.

It is arguably the case that this income process still contains unrealistic features.

A more common earning profile is where

1. income grows over working life, fluctuating around an increasing trend, with growth flattening off in later years
2. retirement follows, with lower but relatively stable (non-financial) income

Letting K be the retirement date, we can express these income dynamics by

$$y_t = \begin{cases} p(t) + \sigma w_{t+1} & \text{if } t \leq K \\ s & \text{otherwise} \end{cases} \tag{64.28}$$

Here

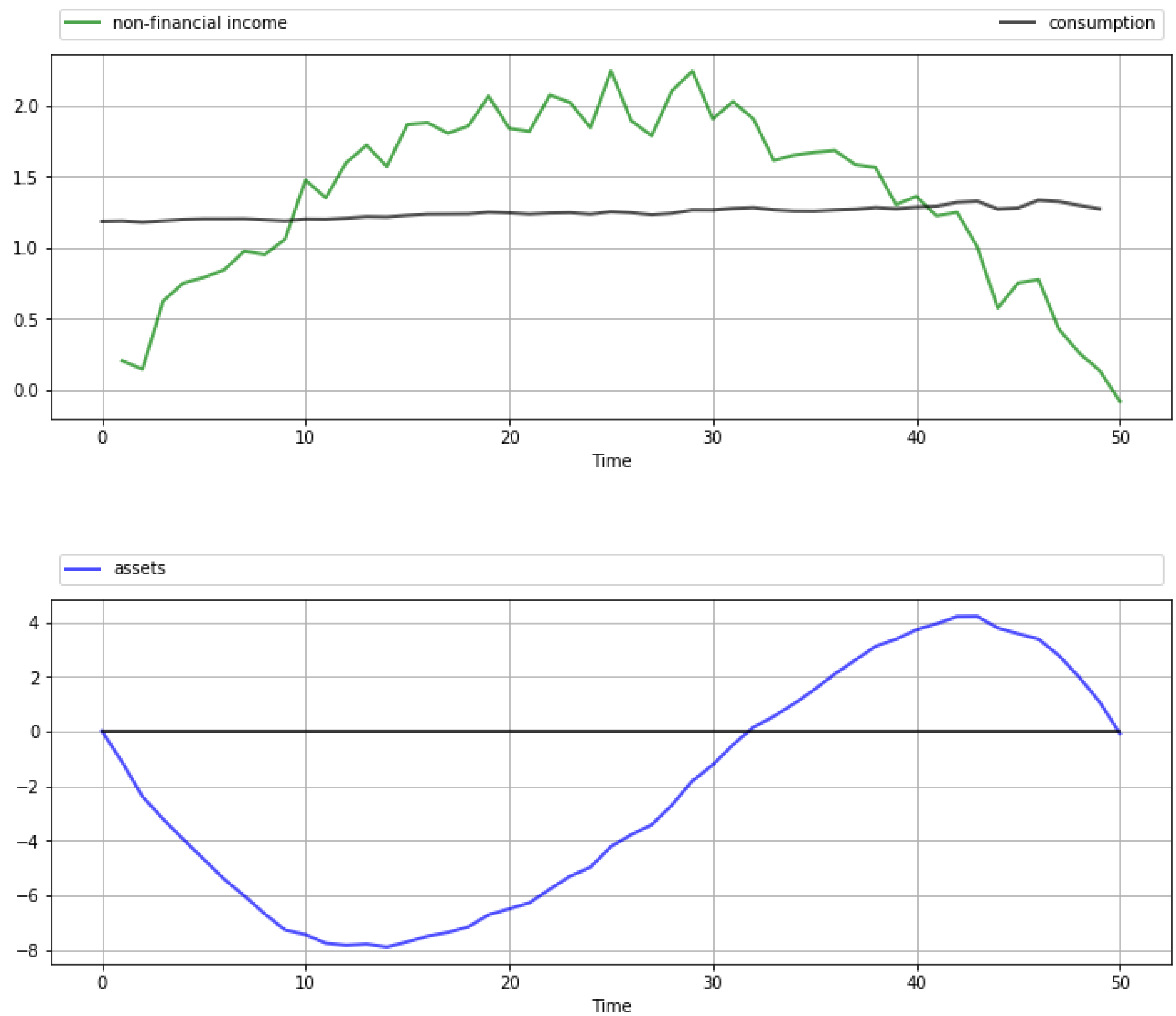
- $p(t) := m_1 t + m_2 t^2$ with the coefficients m_1, m_2 chosen such that $p(K) = \mu$ and $p(0) = p(2K) = 0$
- s is retirement income

We suppose that preferences are unchanged and given by (64.16).

The budget constraint is also unchanged and given by $a_{t+1} = (1+r)a_t - c_t + y_t$.

Our aim is to solve this problem and simulate paths using the LQ techniques described in this lecture.

In fact, this is a nontrivial problem, as the kink in the dynamics (64.28) at K makes it very difficult to express the law of motion as a fixed-coefficient linear system.


non-financial income
consumption
2.0
1.5
1.0
0.5
0.0
0
10
20
30
40
50
Time
assets
4
2
0
−2
−4
−6
−8
0
10
20
30
40
50
Time

However, we can still use our LQ methods here by suitably linking two-component LQ problems.

These two LQ problems describe the consumer's behavior during her working life (`lq_working`) and retirement (`lq_retired`).

(This is possible because, in the two separate periods of life, the respective income processes [polynomial trend and constant] each fit the LQ framework.)

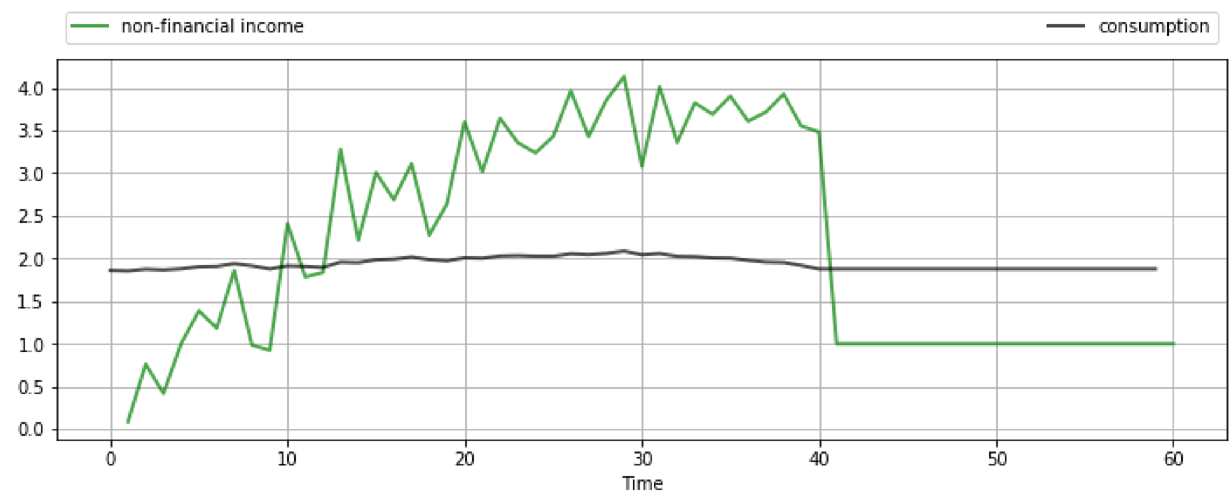
The basic idea is that although the whole problem is not a single time-invariant LQ problem, it is still a dynamic programming problem, and hence we can use appropriate Bellman equations at every stage.

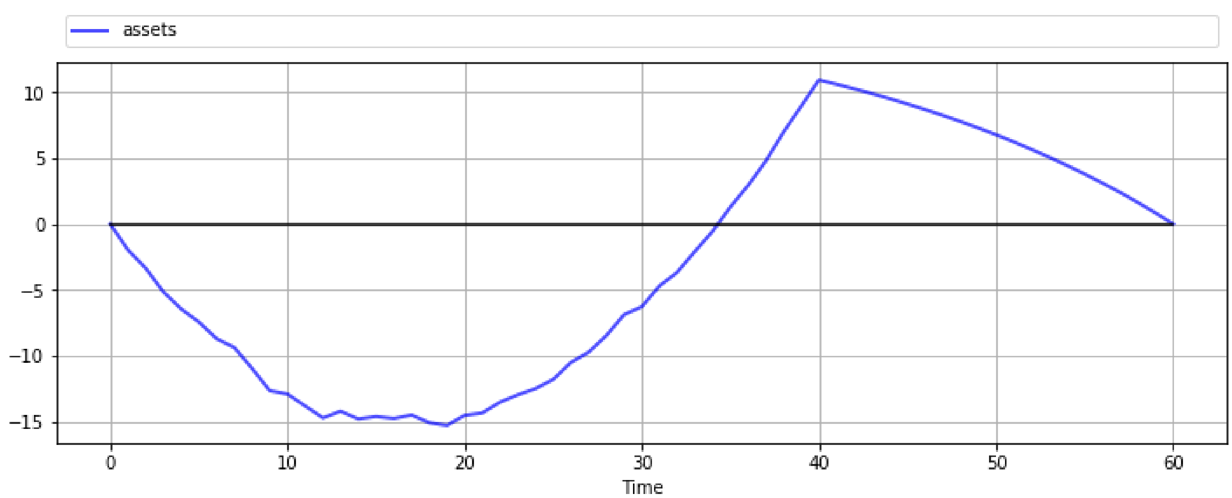
Based on this logic, we can

1. solve `lq_retired` by the usual backward induction procedure, iterating back to the start of retirement.
2. take the start-of-retirement value function generated by this process, and use it as the terminal condition R_f to feed into the `lq_working` specification.
3. solve `lq_working` by backward induction from this choice of R_f, iterating back to the start of working life.

This process gives the entire life-time sequence of value functions and optimal policies.

The next figure shows one simulation based on this procedure.





The full set of parameters used in the simulation is discussed in Exercise 64.7.2, where you are asked to replicate the figure.

Once again, the dominant feature observable in the simulation is consumption smoothing.

The asset path fits well with standard life cycle theory, with dissaving early in life followed by later saving.

Assets peak at retirement and subsequently decline.

64.6.3 Application 3: Monopoly with Adjustment Costs

Consider a monopolist facing stochastic inverse demand function

$$p_t = a_0 - a_1 q_t + d_t$$

Here q_t is output, and the demand shock d_t follows

$$d_{t+1} = \rho d_t + \sigma w_{t+1}$$

where $\{w_t\}$ is IID and standard normal.

The monopolist maximizes the expected discounted sum of present and future profits

$$\mathbb{E} \left\{ \sum_{t=0}^{\infty} \beta^t \pi_t \right\} \quad \text{where} \quad \pi_t := p_t q_t - c q_t - \gamma (q_{t+1} - q_t)^2 \tag{64.29}$$

Here

- $\gamma (q_{t+1} - q_t)^2$ represents adjustment costs
- c is average cost of production

This can be formulated as an LQ problem and then solved and simulated, but first let's study the problem and try to get some intuition.

One way to start thinking about the problem is to consider what would happen if $\gamma = 0$.

Without adjustment costs there is no intertemporal trade-off, so the monopolist will choose output to maximize current profit in each period.

It's not difficult to show that profit-maximizing output is

$$\bar{q}_t := \frac{a_0 - c + d_t}{2 a_1}$$

In light of this discussion, what we might expect for general γ is that

- if γ is close to zero, then q_t will track the time path of $\bar{q}_t$ relatively closely.
- if γ is larger, then q_t will be smoother than $\bar{q}_t$, as the monopolist seeks to avoid adjustment costs.

This intuition turns out to be correct.

The following figures show simulations produced by solving the corresponding LQ problem.

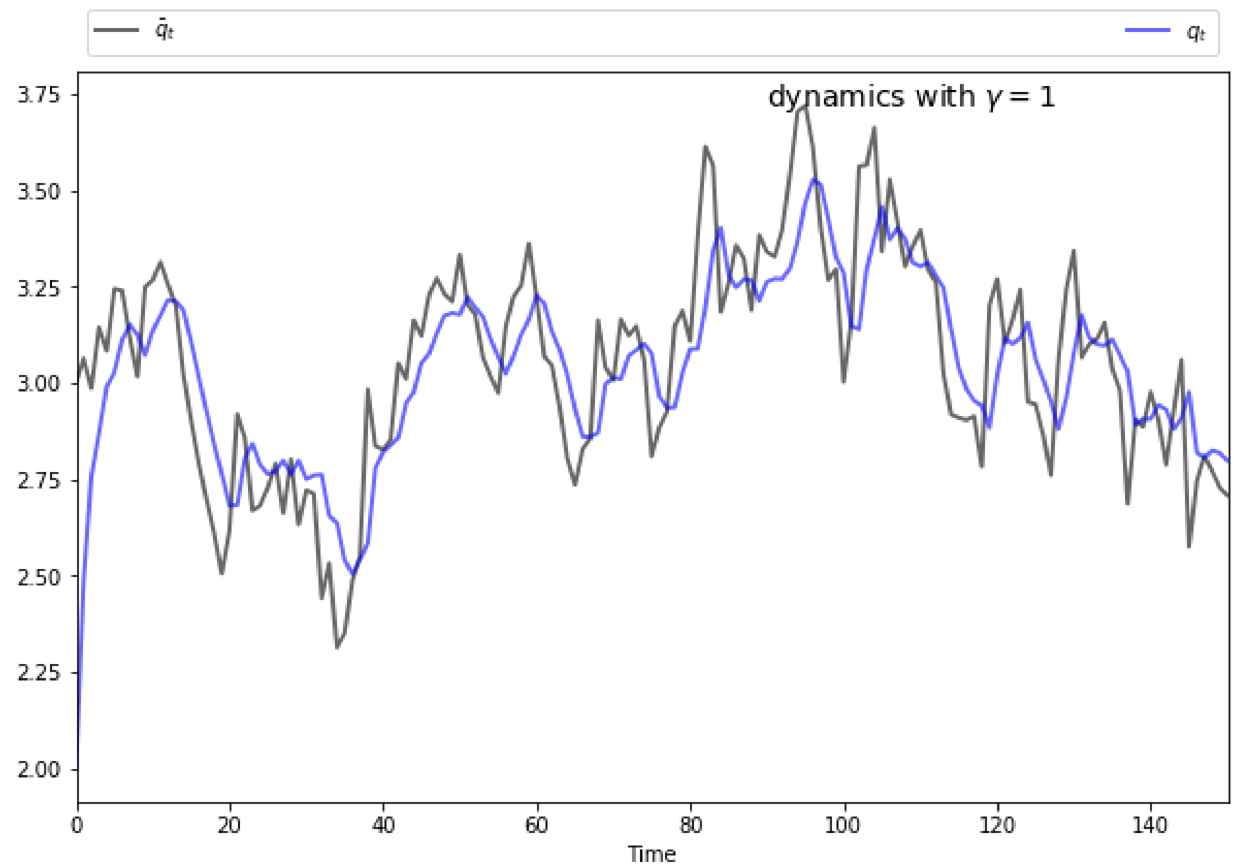
The only difference in parameters across the figures is the size of γ

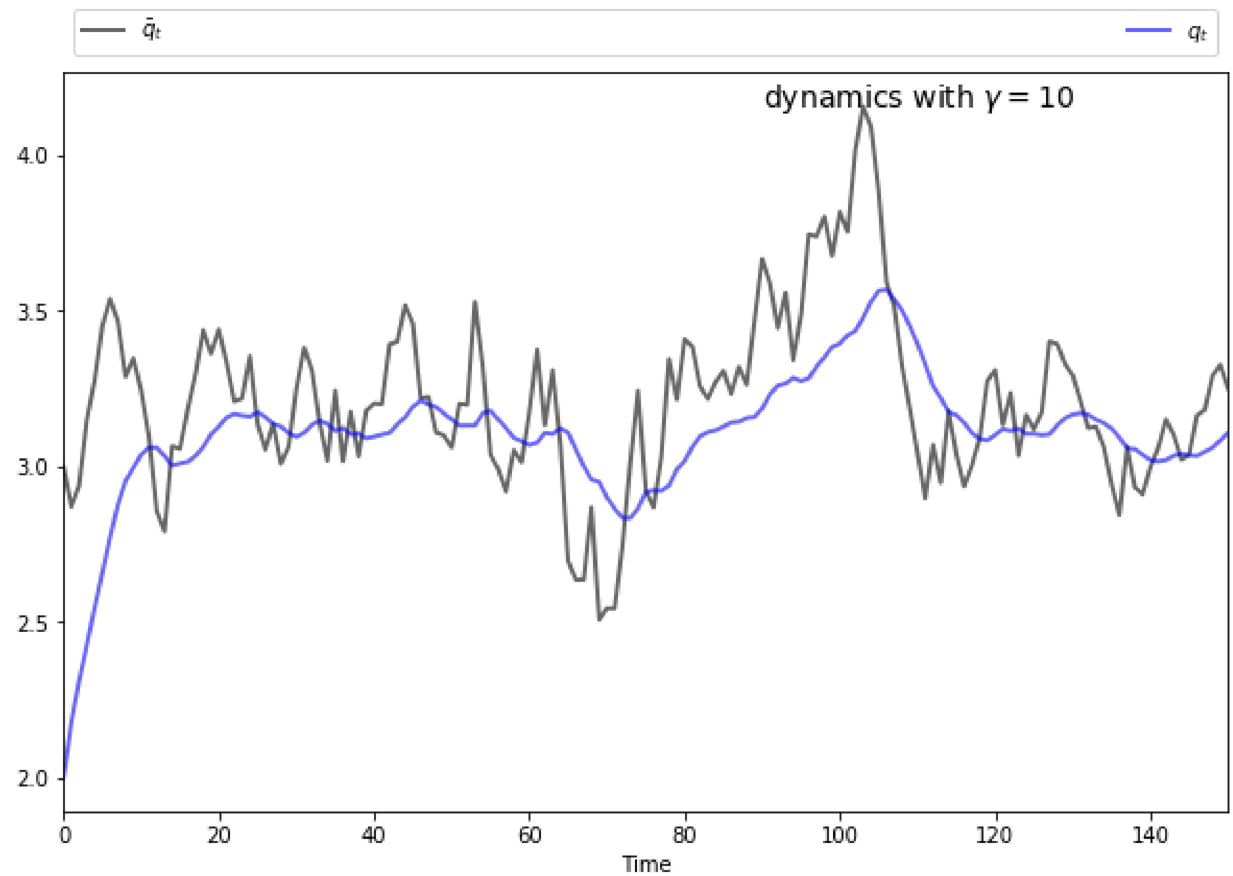
To produce these figures we converted the monopolist problem into an LQ problem.

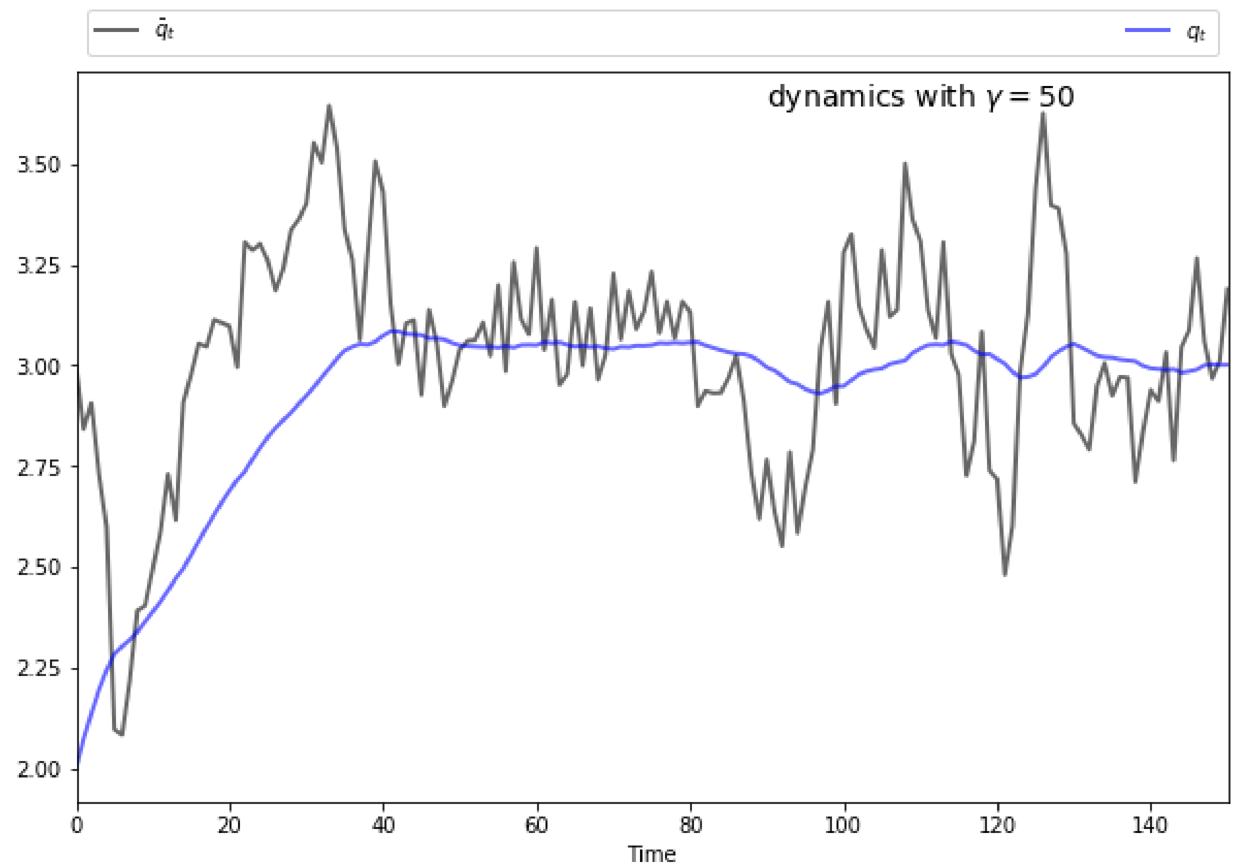
The key to this conversion is to choose the right state — which can be a bit of an art.

Here we take $x_t = (\bar{q}_t \;\; q_t \;\; 1)'$, while the control is chosen as $u_t = q_{t+1} - q_t$.

We also manipulated the profit function slightly.


$\bar{q}_t$
q_t
dynamics with $\gamma = 1$
3.75
3.50
3.25
3.00
2.75
2.50
2.25
2.00
0
20
40
60
80
100
120
140
Time


$\bar{q}_t$
q_t
dynamics with $\gamma = 10$
4.0
3.5
3.0
2.5
2.0
0
20
40
60
80
100
120
140
Time


$\bar{q}_t$
q_t
dynamics with $\gamma = 50$
3.50
3.25
3.00
2.75
2.50
2.25
2.00
0
20
40
60
80
100
120
140
Time

In (64.29), current profits are $\pi_t := p_t q_t - c q_t - \gamma(q_{t+1} - q_t)^2$.

Let's now replace π_t in (64.29) with $\hat{\pi}_t := \pi_t - a_1 \bar{q}_t^2$.

This makes no difference to the solution, since $a_1 \bar{q}_t^2$ does not depend on the controls.

(In fact, we are just adding a constant term to (64.29), and optimizers are not affected by constant terms.)

The reason for making this substitution is that, as you will be able to verify, $\hat{\pi}_t$ reduces to the simple quadratic

$$\hat{\pi}_t = -a_1 (q_t - \bar{q}_t)^2 - \gamma u_t^2$$

After negation to convert to a minimization problem, the objective becomes

$$\min \mathbb{E} \sum_{t=0}^{\infty} \beta^t \left\{ a_1 (q_t - \bar{q}_t)^2 + \gamma u_t^2 \right\} \tag{64.30}$$

It's now relatively straightforward to find R and Q such that (64.30) can be written as (64.20).

Furthermore, the matrices A, B and C from (64.1) can be found by writing down the dynamics of each element of the state.

Exercise 64.7.3 asks you to complete this process, and reproduce the preceding figures.

64.7 Exercises

Exercise 64.7.1

Replicate the figure with polynomial income *shown above*.

The parameters are $r = 0.05, \beta = 1/(1+r), \bar{c} = 1.5, \mu = 2, \sigma = 0.15, T = 50$ and $q = 10^4$.

Solution

Here's one solution.

We use some fancy plot commands to get a certain style — feel free to use simpler ones.

The model is an LQ permanent income / life-cycle model with hump-shaped income

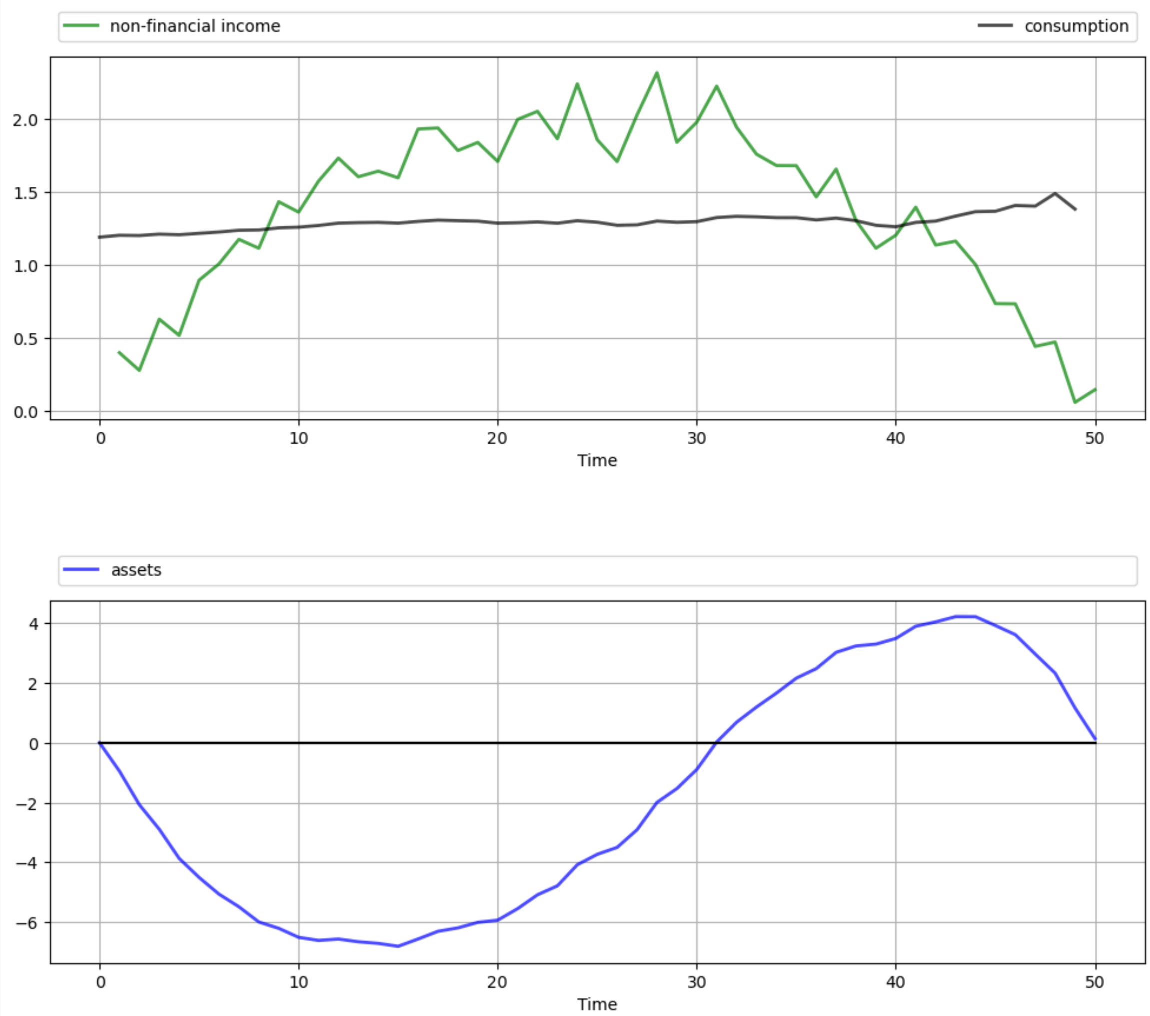
$$y_t = m_1 t + m_2 t^2 + \sigma w_{t+1}$$

where $\{w_t\}$ is IID $N(0,1)$ and the coefficients m_1 and m_2 are chosen so that $p(t) = m_1 t + m_2 t^2$ has an inverted U shape with

- $p(0) = 0, p(T/2) = \mu$, and
- $p(T) = 0$

```
# Model parameters
r = 0.05
β = 1/(1 + r)
T = 50
c_bar = 1.5
σ = 0.15
μ = 2
q = 1e4
```

```
m1 = T * (μ/(T/2)**2)
m2 = -(μ/(T/2)**2)

# Formulate as an LQ problem
Q = 1
R = np.zeros((4, 4))
Rf = np.zeros((4, 4))
Rf[0, 0] = q
A = [[1 + r, -c_bar, m1, m2],
     [0,          1,  0,  0],
     [0,          1,  1,  0],
     [0,          1,  2,  1]]
B = [[-1],
     [ 0],
     [ 0],
     [ 0]]
C = [[σ],
     [0],
     [0],
     [0]]

# Compute solutions and simulate
lq = LQ(Q, R, A, B, C, beta=β, T=T, Rf=Rf)
x0 = (0, 1, 0, 0)
xp, up, wp = lq.compute_sequence(x0)

# Convert results back to assets, consumption and income
ap = xp[0, :]               # Assets
c = up.flatten() + c_bar    # Consumption
time = np.arange(1, T+1)
income = σ * wp[0, 1:] + m1 * time + m2 * time**2  # Income


# Plot results
n_rows = 2
fig, axes = plt.subplots(n_rows, 1, figsize=(12, 10))

plt.subplots_adjust(hspace=0.5)

bbox = (0., 1.02, 1., .102)
legend_args = {'bbox_to_anchor': bbox, 'loc': 3, 'mode': 'expand'}
p_args = {'lw': 2, 'alpha': 0.7}

axes[0].plot(range(1, T+1), income, 'g-', label="non-financial income",
            **p_args)
axes[0].plot(range(T), c, 'k-', label="consumption", **p_args)

axes[1].plot(range(T+1), ap.flatten(), 'b-', label="assets", **p_args)
axes[1].plot(range(T+1), np.zeros(T+1), 'k-')

for ax in axes:
    ax.grid()
    ax.set_xlabel('Time')
    ax.legend(ncol=2, **legend_args)

plt.show()
```



Exercise 64.7.2

Replicate the figure on work and retirement *shown above*.

The parameters are $r = 0.05, \beta = 1/(1 + r), \bar{c} = 4, \mu = 4, \sigma = 0.35, K = 40, T = 60, s = 1$ and $q = 10^4$.

To understand the overall procedure, carefully read the section containing that figure.

Hint

First, in order to make our approach work, we must ensure that both LQ problems have the same state variables and control.
As with previous applications, the control can be set to $u_t = c_t - \bar{c}$.
For `lq_working`, x_t, A, B, C can be chosen as in (64.26).

- Recall that m_1, m_2 are chosen so that $p(K) = \mu$ and $p(2K) = 0$.

For `lq_retired`, use the same definition of x_t and u_t, but modify A, B, C to correspond to constant income $y_t = s$.
For `lq_retired`, set preferences as in (64.27).
For `lq_working`, preferences are the same, except that R_f should be replaced by the final value function that emerges from iterating `lq_retired` back to the start of retirement.
With some careful footwork, the simulation can be generated by patching together the simulations from these two separate models.

Solution

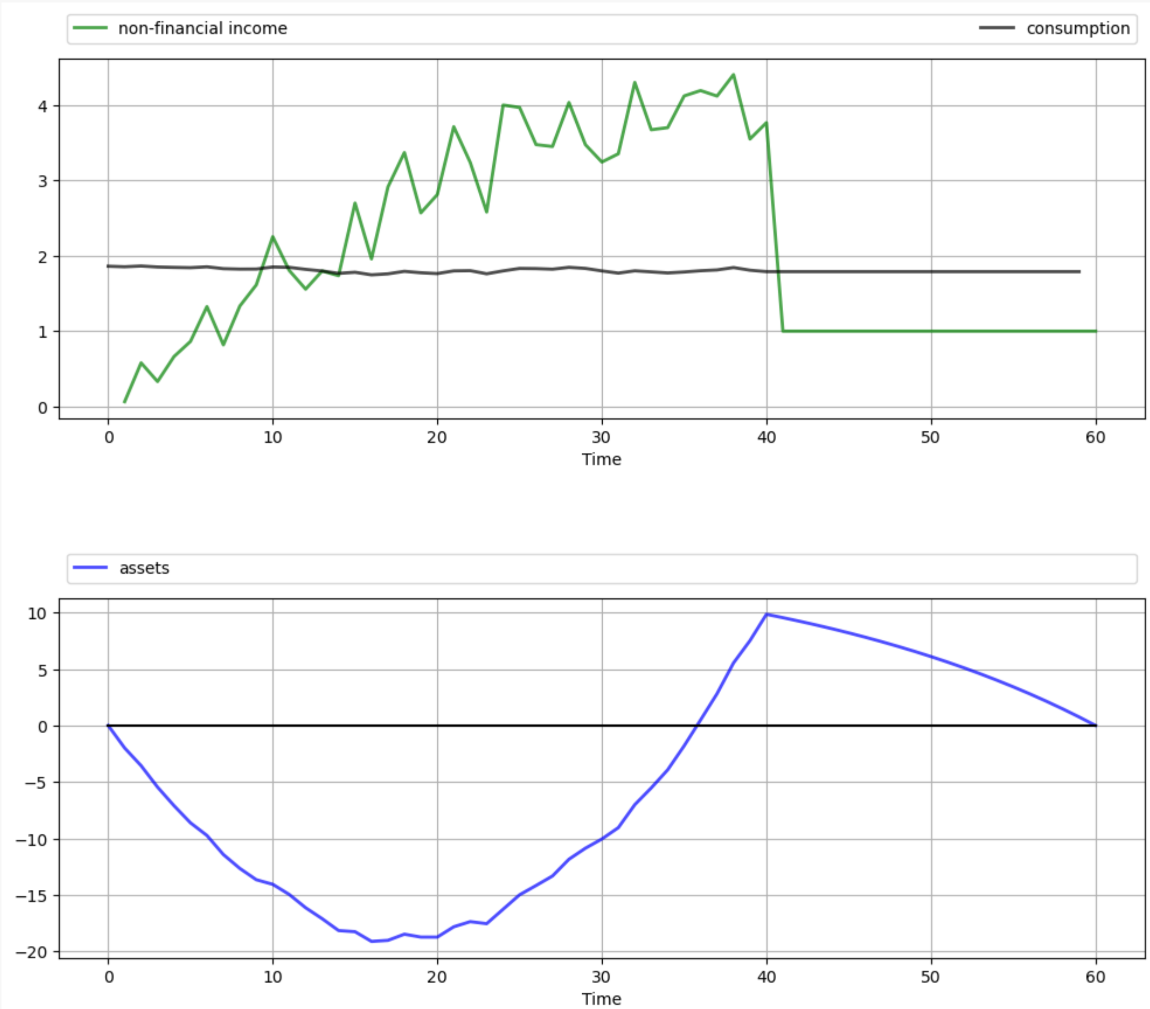
This is a permanent income / life-cycle model with polynomial growth in income over working life followed by a fixed retirement income.

The model is solved by combining two LQ programming problems as described in the lecture.

```
# Model parameters
r = 0.05
β = 1/(1 + r)
T = 60
K = 40
c_bar = 4
σ = 0.35
μ = 4
q = 1e4
s = 1
m1 = 2 * μ/K
m2 = -μ/K**2

# Formulate LQ problem 1 (retirement)
Q = 1
R = np.zeros((4, 4))
Rf = np.zeros((4, 4))
Rf[0, 0] = q
A = [[1 + r, s - c_bar, 0, 0],
     [0,              1, 0, 0],
     [0,              1, 1, 0],
     [0,              1, 2, 1]]
B = [[-1],
     [ 0],
     [ 0],
     [ 0]]
C = [[0],
     [0],
     [0],
     [0]]

# Initialize LQ instance for retired agent
lq_retired = LQ(Q, R, A, B, C, beta=β, T=T-K, Rf=Rf)
```

```
# Iterate back to start of retirement, record final value function
for i in range(T-K):
    lq_retired.update_values()
Rf2 = lq_retired.P

# Formulate LQ problem 2 (working life)
R = np.zeros((4, 4))
A = [[1 + r, -c_bar, m1, m2],
     [0,          1,  0,  0],
     [0,          1,  1,  0],
     [0,          1,  2,  1]]
B = [[-1],
     [ 0],
     [ 0],
     [ 0]]
C = [[σ],
     [0],
     [0],
     [0]]

# Set up working life LQ instance with terminal Rf from lq_retired
lq_working = LQ(Q, R, A, B, C, beta=β, T=K, Rf=Rf2)

# Simulate working state / control paths
x0 = (0, 1, 0, 0)
xp_w, up_w, wp_w = lq_working.compute_sequence(x0)
# Simulate retirement paths (note the initial condition)
xp_r, up_r, wp_r = lq_retired.compute_sequence(xp_w[:, K])

# Convert results back to assets, consumption and income
xp = np.column_stack((xp_w, xp_r[:, 1:]))
assets = xp[0, :]                    # Assets

up = np.column_stack((up_w, up_r))
c = up.flatten() + c_bar             # Consumption

time = np.arange(1, K+1)
income_w = σ * wp_w[0, 1:K+1] + m1 * time + m2 * time**2  # Income
income_r = np.full(T-K, s)
income = np.concatenate((income_w, income_r))

# Plot results
n_rows = 2
fig, axes = plt.subplots(n_rows, 1, figsize=(12, 10))

plt.subplots_adjust(hspace=0.5)

bbox = (0., 1.02, 1., .102)
legend_args = {'bbox_to_anchor': bbox, 'loc': 3, 'mode': 'expand'}
p_args = {'lw': 2, 'alpha': 0.7}

axes[0].plot(range(1, T+1), income, 'g-', label="non-financial income",
            **p_args)
axes[0].plot(range(T), c, 'k-', label="consumption", **p_args)

axes[1].plot(range(T+1), assets, 'b-', label="assets", **p_args)
axes[1].plot(range(T+1), np.zeros(T+1), 'k-')
```

```
for ax in axes:
    ax.grid()
    ax.set_xlabel('Time')
    ax.legend(ncol=2, **legend_args)

plt.show()
```



Exercise 64.7.3

Reproduce the figures from the monopolist application *given above*.

For parameters, use $a_0 = 5, a_1 = 0.5, \sigma = 0.15, \rho = 0.9, \beta = 0.95$ and $c = 2$, while γ varies between 1 and 50 (see figures).

Solution

The first task is to find the matrices A, B, C, Q, R that define the LQ problem.

Recall that $x_t = (\bar{q}_t \;\; q_t \;\; 1)'$, while $u_t = q_{t+1} - q_t$.

Letting $m_0 := (a_0 - c)/2a_1$ and $m_1 := 1/2a_1$, we can write $\bar{q}_t = m_0 + m_1 d_t$, and then, with some manipulation

$$\bar{q}_{t+1} = m_0(1-\rho) + \rho \bar{q}_t + m_1 \sigma w_{t+1}$$

By our definition of u_t, the dynamics of q_t are $q_{t+1} = q_t + u_t$.

Using these facts you should be able to build the correct A, B, C matrices (and then check them against those found in the solution code below).

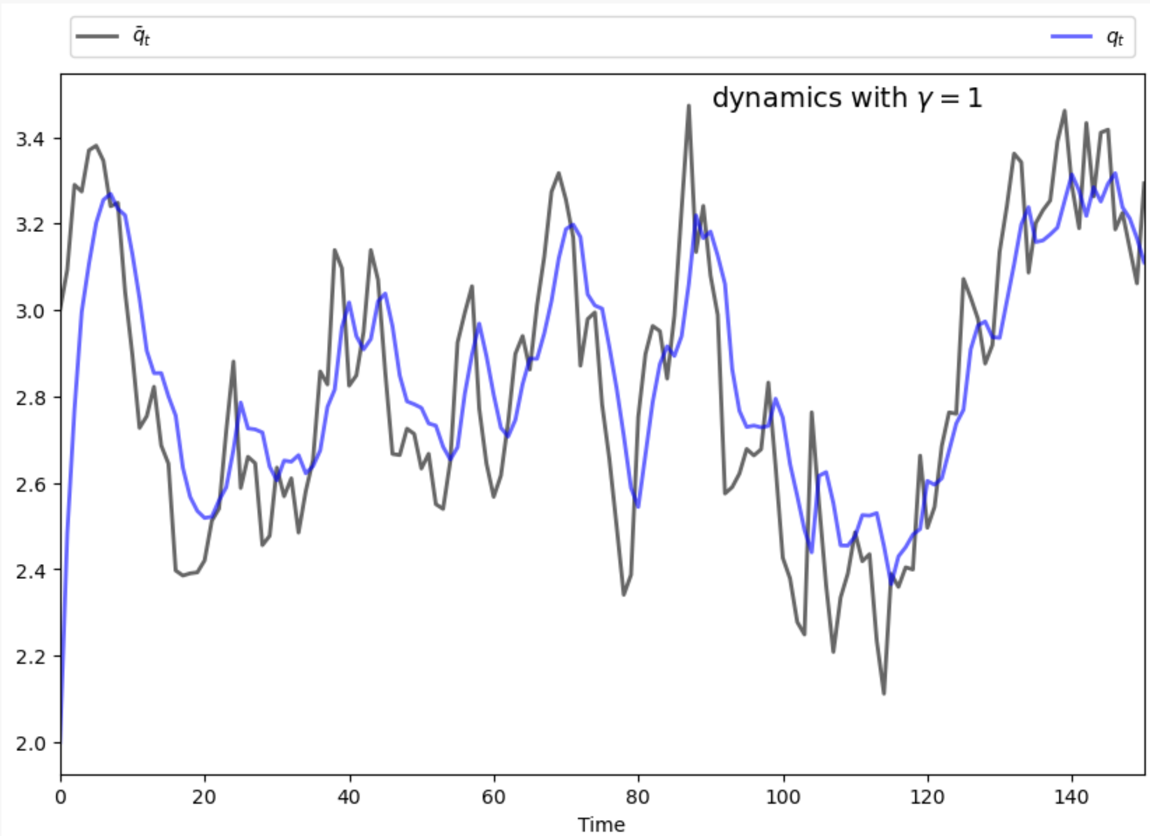
Suitable R, Q matrices can be found by inspecting the objective function, which we repeat here for convenience:

$$\min \mathbb{E} \left\{ \sum_{t=0}^{\infty} \beta^t a_1 (q_t - \bar{q}_t)^2 + \gamma u_t^2 \right\}$$

Our solution code is

```
# Model parameters
a0 = 5
a1 = 0.5
σ = 0.15
ρ = 0.9
γ = 1
β = 0.95
c = 2
T = 120

# Useful constants
m0 = (a0-c)/(2 * a1)
m1 = 1/(2 * a1)

# Formulate LQ problem
Q = γ
R = [[ a1, -a1,  0],
     [-a1,  a1,  0],
     [  0,   0,  0]]
A = [[ρ, 0, m0 * (1 - ρ)],
     [0, 1,            0],
     [0, 0,            1]]

B = [[0],
     [1],
     [0]]
C = [[m1 * σ],
     [     0],
     [     0]]

lq = LQ(Q, R, A, B, C=C, beta=β)

# Simulate state / control paths
x0 = (m0, 2, 1)
xp, up, wp = lq.compute_sequence(x0, ts_length=150)
q_bar = xp[0, :]
q = xp[1, :]

# Plot simulation results
fig, ax = plt.subplots(figsize=(10, 6.5))
```

```
# Some fancy plotting stuff -- simplify if you prefer
bbox = (0., 1.01, 1., .101)
legend_args = {'bbox_to_anchor': bbox, 'loc': 3, 'mode': 'expand'}
p_args = {'lw': 2, 'alpha': 0.6}

time = range(len(q))
ax.set(xlabel='Time', xlim=(0, max(time)))
ax.plot(time, q_bar, 'k-', lw=2, alpha=0.6, label=r'$\bar q_t$')
ax.plot(time, q, 'b-', lw=2, alpha=0.6, label='$q_t$')
ax.legend(ncol=2, **legend_args)
s = fr'dynamics with $\gamma = {γ}$'
ax.text(max(time) * 0.6, 1 * q_bar.max(), s, fontsize=14)
plt.show()
```



CHAPTER

SIXTYFIVE

LAGRANGIAN FOR LQ CONTROL

```
!pip install quantecon
```

```
import numpy as np
from quantecon import LQ
from scipy.linalg import schur
```

65.1 Overview

This is a sequel to this lecture *linear quadratic dynamic programming*

It can also be regarded as presenting **invariant subspace** techniques that extend ones that we encountered earlier in this lecture *stability in linear rational expectations models*

We present a Lagrangian formulation of an infinite horizon linear quadratic undiscounted dynamic programming problem.

Such a problem is also sometimes called an optimal linear regulator problem.

A Lagrangian formulation

- carries insights about connections between stability and optimality
- is the basis for fast algorithms for solving Riccati equations
- opens the way to constructing solutions of dynamic systems that don't come directly from an intertemporal optimization problem

A key tool in this lecture is the concept of an $n \times n$ **symplectic** matrix.

A symplectic matrix has eigenvalues that occur in **reciprocal pairs**, meaning that if $\lambda_i \in (-1, 1)$ is an eigenvalue, then so is λ_i^{-1}.

This reciprocal pairs property of the eigenvalues of a matrix is a tell-tale sign that the matrix describes the joint dynamics of a system of equations describing the **states** and **costates** that constitute first-order necessary conditions for solving an undiscounted linear-quadratic infinite-horizon optimization problem.

The symplectic matrix that will interest us describes the first-order dynamics of **state** and **co-state** vectors of an optimally controlled system.

In focusing on eigenvalues and eigenvectors of this matrix, we capitalize on an analysis of **invariant subspaces.**

These invariant subspace formulations of LQ dynamic programming problems provide a bridge between recursive (i.e., dynamic programming) formulations and classical formulations of linear control and linear filtering problems that make use of related matrix decompositions (see for example this lecture and this lecture).

While most of this lecture focuses on undiscounted problems, later sections describe handy ways of transforming discounted problems to undiscounted ones.

The techniques in this lecture will prove useful when we study Stackelberg and Ramsey problem in this lecture.

65.2 Undiscounted LQ DP Problem

The problem is to choose a sequence of controls $\{u_t\}_{t=0}^{\infty}$ to maximize the criterion

$$-\sum_{t=0}^{\infty}\{x_t'Rx_t + u_t'Qu_t\}$$

subject to $x_{t+1} = Ax_t + Bu_t$, where x_0 is a given initial state vector.

Here x_t is an $(n \times 1)$ vector of state variables, u_t is a $(k \times 1)$ vector of controls, R is a positive semidefinite symmetric matrix, Q is a positive definite symmetric matrix, A is an $(n \times n)$ matrix, and B is an $(n \times k)$ matrix.

The optimal value function turns out to be quadratic, $V(x) = -x'Px$, where P is a positive semidefinite symmetric matrix.

Using the transition law to eliminate next period's state, the Bellman equation becomes

$$-x'Px = \max_u\{-x'Rx - u'Qu - (Ax + Bu)'P(Ax + Bu)\} \tag{65.1}$$

The first-order necessary conditions for the maximum problem on the right side of equation (65.1) are

> **Note**
>
> We use the following rules for differentiating quadratic and bilinear matrix forms: $\frac{\partial x'Ax}{\partial x} = (A + A')x$; $\frac{\partial y'Bz}{\partial y} = Bz$, $\frac{\partial y'Bz}{\partial z} = B'y$.

$$(Q + B'PB)u = -B'PAx,$$

which implies that an optimal decision rule for u is

$$u = -(Q + B'PB)^{-1}B'PAx$$

or

$$u = -Fx,$$

where

$$F = (Q + B'PB)^{-1}B'PA.$$

Substituting $u = -(Q + B'PB)^{-1}B'PAx$ into the right side of equation (65.1) and rearranging gives

$$P = R + A'PA - A'PB(Q + B'PB)^{-1}B'PA. \tag{65.2}$$

Equation (65.2) is called an **algebraic matrix Riccati** equation.

There are multiple solutions of equation (65.2).

But only one of them is positive definite.

The positive define solution is associated with the maximum of our problem.

It expresses the matrix P as an implicit function of the matrices R, Q, A, B.

Notice that the **gradient of the value function** is

$$\frac{\partial V(x)}{\partial x} = -2Px \tag{65.3}$$

We shall use fact (65.3) later.

65.3 Lagrangian

For the undiscounted optimal linear regulator problem, form the Lagrangian

$$L = -\sum_{t=0}^{\infty} \left\{ x_t' R x_t + u_t' Q u_t + 2\mu_{t+1}' [A x_t + B u_t - x_{t+1}] \right\} \tag{65.4}$$

where $2\mu_{t+1}$ is a vector of Lagrange multipliers on the time t transition law $x_{t+1} = Ax_t + Bu_t$.

(We put the 2 in front of μ_{t+1} to make things match up nicely with equation (65.3).)

First-order conditions for maximization with respect to $\{u_t, x_{t+1}\}_{t=0}^{\infty}$ are

$$\begin{aligned} 2Qu_t + 2B'\mu_{t+1} &= 0 \ , \ t \geq 0 \\ \mu_t &= Rx_t + A'\mu_{t+1} \ , \ t \geq 1. \end{aligned} \tag{65.5}$$

Define μ_0 to be a vector of shadow prices of x_0 and apply an envelope condition to (65.4) to deduce that

$$\mu_0 = Rx_0 + A'\mu_1,$$

which is a time $t = 0$ counterpart to the second equation of system (65.5).

An important fact is that

$$\mu_{t+1} = P x_{t+1} \tag{65.6}$$

where P is a positive define matrix that solves the algebraic Riccati equation (65.2).

Thus, from equations (65.3) and (65.6), $-2\mu_t$ is the gradient of the value function with respect to x_t.

The Lagrange multiplier vector μ_t is often called the **costate** vector that corresponds to the **state** vector x_t.

It is useful to proceed with the following steps:

- solve the first equation of (65.5) for u_t in terms of μ_{t+1}.
- substitute the result into the law of motion $x_{t+1} = Ax_t + Bu_t$.
- arrange the resulting equation and the second equation of (65.5) into the form

$$L \begin{bmatrix} x_{t+1} \\ \mu_{t+1} \end{bmatrix} = N \begin{bmatrix} x_t \\ \mu_t \end{bmatrix} \ , \ t \geq 0, \tag{65.7}$$

where

$$L = \begin{bmatrix} I & BQ^{-1}B' \\ 0 & A' \end{bmatrix}, \quad N = \begin{bmatrix} A & 0 \\ -R & I \end{bmatrix}.$$

When L is of full rank (i.e., when A is of full rank), we can write system (65.7) as

$$\begin{bmatrix} x_{t+1} \\ \mu_{t+1} \end{bmatrix} = M \begin{bmatrix} x_t \\ \mu_t \end{bmatrix} \tag{65.8}$$

where

$$M \equiv L^{-1} N = \begin{bmatrix} A + BQ^{-1}B'A'^{-1}R & -BQ^{-1}B'A'^{-1} \\ -A'^{-1}R & A'^{-1} \end{bmatrix}. \tag{65.9}$$

65.4 State-Costate Dynamics

We seek to solve the difference equation system (65.8) for a sequence $\{x_t\}_{t=0}^{\infty}$ that satisfies

- an initial condition for x_0
- a terminal condition $\lim_{t \rightarrow +\infty} x_t = 0$

This terminal condition reflects our desire for a **stable** solution, one that does not diverge as $t \rightarrow \infty$.

We inherit our wish for stability of the $\{x_t\}$ sequence from a desire to maximize

$$-\sum_{t=0}^{\infty} [x_t' R x_t + u_t' Q u_t],$$

which requires that $x_t' R x_t$ converge to zero as $t \rightarrow +\infty$.

65.5 Reciprocal Pairs Property

To proceed, we study properties of the $(2n \times 2n)$ matrix M defined in (65.9).

It helps to introduce a $(2n \times 2n)$ matrix

$$J = \begin{bmatrix} 0 & -I_n \\ I_n & 0 \end{bmatrix}.$$

The rank of J is $2n$.

Definition: A matrix M is called **symplectic** if

$$MJM' = J. \tag{65.10}$$

Salient properties of symplectic matrices that are readily verified include:

- If M is symplectic, then M^2 is symplectic
- The determinant of a symplectic, then $\det(M) = 1$

It can be verified directly that M in equation (65.9) is symplectic.

It follows from equation (65.10) and from the fact $J^{-1} = J' = -J$ that for any symplectic matrix M,

$$M' = J^{-1} M^{-1} J. \tag{65.11}$$

Equation (65.11) states that M' is related to the inverse of M by a **similarity transformation**.

For square matrices, recall that

- similar matrices share eigenvalues
- eigenvalues of the inverse of a matrix are inverses of eigenvalues of the matrix
- a matrix and its transpose share eigenvalues

It then follows from equation (65.11) that the eigenvalues of M occur in reciprocal pairs: if λ is an eigenvalue of M, so is λ^{-1}.

Write equation (65.8) as

$$y_{t+1} = M y_t \tag{65.12}$$

where $y_t = \begin{bmatrix} x_t \\ \mu_t \end{bmatrix}$.

Consider a **triangularization** of M

$$V^{-1} M V = \begin{bmatrix} W_{11} & W_{12} \\ 0 & W_{22} \end{bmatrix} \tag{65.13}$$

where

- each block on the right side is $(n \times n)$
- V is nonsingular
- all eigenvalues of W_{22} exceed 1 in modulus
- all eigenvalues of W_{11} are less than 1 in modulus

65.6 Schur decomposition

The **Schur decomposition** and the **eigenvalue decomposition** are two decompositions of the form (65.13).

Write equation (65.12) as

$$y_{t+1} = V W V^{-1} y_t. \tag{65.14}$$

A solution of equation (65.14) for arbitrary initial condition y_0 is evidently

$$y_t = V \begin{bmatrix} W_{11}^t & W_{12,t} \\ 0 & W_{22}^t \end{bmatrix} V^{-1} y_0 \tag{65.15}$$

where $W_{12,t} = W_{12}$ for $t = 1$ and for $t \geq 2$ obeys the recursion

$$W_{12,t} = W_{11}^{t-1} W_{12,t-1} + W_{12,t-1} W_{22}^{t-1}$$

and where W_{ii}^t is W_{ii} raised to the tth power.

Write equation (65.15) as

$$\begin{bmatrix} y_{1t}^* \\ y_{2t}^* \end{bmatrix} = \begin{bmatrix} W_{11}^t & W_{12,t} \\ 0 & W_{22}^t \end{bmatrix} \begin{bmatrix} y_{10}^* \\ y_{20}^* \end{bmatrix}$$

where $y_t^* = V^{-1} y_t$, and in particular where

$$y_{2t}^* = V^{21} x_t + V^{22} \mu_t, \tag{65.16}$$

and where V^{ij} denotes the (i, j) piece of the partitioned V^{-1} matrix.

Because W_{22} is an unstable matrix, y_t^* will diverge unless $y_{20}^* = 0$.

Let V^{ij} denote the (i, j) piece of the partitioned V^{-1} matrix.

To attain stability, we must impose $y_{20}^* = 0$, which from equation (65.16) implies

$$V^{21} x_0 + V^{22} \mu_0 = 0$$

or

$$\mu_0 = -(V^{22})^{-1} V^{21} x_0.$$

This equation replicates itself over time in the sense that it implies

$$\mu_t = -(V^{22})^{-1} V^{21} x_t.$$

But notice that because $(V^{21}\ V^{22})$ is the second row block of the inverse of V, it follows that

$$(V^{21}\ V^{22}) \quad \begin{bmatrix} V_{11} \\ V_{21} \end{bmatrix} = 0$$

which implies

$$V^{21} V_{11} + V^{22} V_{21} = 0.$$

Therefore,

$$-(V^{22})^{-1} V^{21} = V_{21} V_{11}^{-1}.$$

So we can write

$$\mu_0 = V_{21} V_{11}^{-1} x_0$$

and

$$\mu_t = V_{21} V_{11}^{-1} x_t.$$

However, we know that $\mu_t = P x_t$, where P occurs in the matrix that solves the Riccati equation.

Thus, the preceding argument establishes that

$$P = V_{21} V_{11}^{-1}. \tag{65.17}$$

Remarkably, formula (65.17) provides us with a computationally efficient way of computing the positive definite matrix P that solves the algebraic Riccati equation (65.2) that emerges from dynamic programming.

This same method can be applied to compute the solution of any system of the form (65.8) if a solution exists, even if eigenvalues of M fail to occur in reciprocal pairs.

The method will typically work so long as the eigenvalues of M split half inside and half outside the unit circle.

Systems in which eigenvalues (properly adjusted for discounting) fail to occur in reciprocal pairs arise when the system being solved is an equilibrium of a model in which there are distortions that prevent there being any optimum problem that the equilibrium solves. See [Ljungqvist and Sargent, 2018], ch 12.

65.7 Application

Here we demonstrate the computation with an example which is the deterministic version of an example borrowed from this quantecon lecture.

```
# Model parameters
r = 0.05
c_bar = 2
μ = 1

# Formulate as an LQ problem
Q = np.array([[1]])
R = np.zeros((2, 2))
```

(continues on next page)

(continued from previous page)

```
A = [[1 + r, -c_bar + μ],
     [0,              1]]
B = [[-1],
     [0]]

# Construct an LQ instance
lq = LQ(Q, R, A, B)
```

Given matrices A, B, Q, R, we can then compute L, N, and $M = L^{-1}N$.

```
def construct_LNM(A, B, Q, R):

    n, k = lq.n, lq.k

    # construct L and N
    L = np.zeros((2*n, 2*n))
    L[:n, :n] = np.eye(n)
    L[:n, n:] = B @ np.linalg.inv(Q) @ B.T
    L[n:, n:] = A.T

    N = np.zeros((2*n, 2*n))
    N[:n, :n] = A
    N[n:, :n] = -R
    N[n:, n:] = np.eye(n)

    # compute M
    M = np.linalg.inv(L) @ N

    return L, N, M
```

```
L, N, M = construct_LNM(lq.A, lq.B, lq.Q, lq.R)
```

```
M
```

```
array([[ 1.05      , -1.        , -0.95238095,  0.        ],
       [ 0.        ,  1.        ,  0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.95238095,  0.        ],
       [ 0.        ,  0.        ,  0.95238095,  1.        ]])
```

Let's verify that M is symplectic.

```
n = lq.n
J = np.zeros((2*n, 2*n))
J[n:, :n] = np.eye(n)
J[:n, n:] = -np.eye(n)

M @ J @ M.T - J
```

```
array([[-1.32169408e-17,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
```

(continues on next page)

(continued from previous page)

```
          0.00000000e+00]])
```

We can compute the eigenvalues of M using `np.linalg.eigvals`, arranged in ascending order.

```
eigvals = sorted(np.linalg.eigvals(M))
eigvals
```

```
[np.float64(0.9523809523809523),
 np.float64(1.0),
 np.float64(1.0),
 np.float64(1.05)]
```

When we apply Schur decomposition such that $M = VWV^{-1}$, we want

- the upper left block of W, W_{11}, to have all of its eigenvalues less than 1 in modulus, and
- the lower right block W_{22} to have eigenvalues that exceed 1 in modulus.

To get what we want, let's define a sorting function that tells `scipy.schur` to sort the corresponding eigenvalues with modulus smaller than 1 to the upper left.

```
stable_eigvals = eigvals[:n]

def sort_fun(x):
    "Sort the eigenvalues with modules smaller than 1 to the top-left."

    if x in stable_eigvals:
        stable_eigvals.pop(stable_eigvals.index(x))
        return True
    else:
        return False

W, V, _ = schur(M, sort=sort_fun)
```

```
W
```

```
array([[ 1.        ,  0.02316402,  1.00085948, -0.95000594],
       [ 0.        ,  0.95238095, -0.00237501,  0.95325452],
       [ 0.        ,  0.        ,  1.05      , -0.02432222],
       [ 0.        ,  0.        ,  0.        ,  1.        ]])
```

```
V
```

```
array([[ 0.99875234, -0.00121459,  0.04992284,  0.        ],
       [ 0.04993762,  0.02429188, -0.99845688,  0.        ],
       [ 0.        , -0.04992284, -0.00121459,  0.99875234],
       [ 0.        ,  0.99845688,  0.02429188,  0.04993762]])
```

We can check the modulus of eigenvalues of W_{11} and W_{22}.

Since they are both triangular matrices, eigenvalues are the diagonal elements.

```
# W11
np.diag(W[:n, :n])
```

```
array([1.        , 0.95238095])
```

```
# W22
np.diag(W[n:, n:])
```

```
array([1.05, 1.  ])
```

The following functions wrap M matrix construction, Schur decomposition, and stability-imposing computation of P.

```
def stable_solution(M, verbose=True):
    """
    Given a system of linear difference equations

        y' = |a b| y
        x' = |c d| x

    which is potentially unstable, find the solution
    by imposing stability.

    Parameter
    ---------
    M : np.ndarray(float)
        The matrix represents the linear difference equations system.
    """
    n = M.shape[0] // 2
    stable_eigvals = list(sorted(np.linalg.eigvals(M))[:n])

    def sort_fun(x):
        "Sort the eigenvalues with modules smaller than 1 to the top-left."

        if x in stable_eigvals:
            stable_eigvals.pop(stable_eigvals.index(x))
            return True
        else:
            return False

    W, V, _ = schur(M, sort=sort_fun)
    if verbose:
        print('eigenvalues:\n')
        print('    W11: {}'.format(np.diag(W[:n, :n])))
        print('    W22: {}'.format(np.diag(W[n:, n:])))

    # compute V21 V11^{-1}
    P = V[n:, :n] @ np.linalg.inv(V[:n, :n])

    return W, V, P

def stationary_P(lq, verbose=True):
    """
    Computes the matrix :math:`P` that represent the value function

         V(x) = x' P x

    in the infinite horizon case. Computation is via imposing stability
    on the solution path and using Schur decomposition.

    Parameters
```

(continues on next page)

(continued from previous page)

```
    ----------
    lq : qe.LQ
        QuantEcon class for analyzing linear quadratic optimal control
        problems of infinite horizon form.

    Returns
    -------
    P : array_like(float)
        P matrix in the value function representation.
    """

    Q = lq.Q
    R = lq.R
    A = lq.A * lq.beta ** (1/2)
    B = lq.B * lq.beta ** (1/2)

    n, k = lq.n, lq.k

    L, N, M = construct_LNM(A, B, Q, R)
    W, V, P = stable_solution(M, verbose=verbose)

    return P
```

```
# compute P
stationary_P(lq)
```

```
eigenvalues:

    W11: [1.         0.95238095]
    W22: [1.05 1.  ]
```

```
array([[ 0.1025, -2.05  ],
       [-2.05  , 41.    ]])
```

Note that the matrix P computed in this way is close to what we get from the routine in quantecon that solves an algebraic Riccati equation by iterating to convergence on a Riccati difference equation.

The small difference comes from computational errors and will decrease as we increase the maximum number of iterations or decrease the tolerance for convergence.

```
lq.stationary_values()
```

```
(array([[ 0.1025, -2.05  ],
        [-2.05  , 41.01  ]]),
 array([[-0.09761905,  1.95238095]]),
 0)
```

Using a Schur decomposition is much more efficient.

```
%%timeit
stationary_P(lq, verbose=False)
```

```
123 µs ± 246 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

```
%%timeit
lq.stationary_values()
```

```
1.81 ms ± 1.49 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

65.8 Other Applications

The preceding approach to imposing stability on a system of potentially unstable linear difference equations is not limited to linear quadratic dynamic optimization problems.

For example, the same method is used in our Stability in Linear Rational Expectations Models lecture.

Let's try to solve the model described in that lecture by applying the `stable_solution` function defined in this lecture above.

```
def construct_H(ρ, λ, δ):
    "contruct matrix H given parameters."

    H = np.empty((2, 2))
    H[0, :] = ρ,δ
    H[1, :] = - (1 - λ) / λ, 1 / λ

    return H

H = construct_H(ρ=.9, λ=.5, δ=0)
```

```
W, V, P = stable_solution(H)
P
```

```
eigenvalues:

    W11: [0.9]
    W22: [2.]
```

```
array([[0.90909091]])
```

65.9 Discounted Problems

65.9.1 Transforming States and Controls to Eliminate Discounting

A pair of useful transformations allows us to convert a discounted problem into an undiscounted one.

Thus, suppose that we have a discounted problem with objective

$$-\sum_{t=0}^{\infty} \beta^t \left\{ x_t' R x_t + u_t' Q u_t \right\}$$

and that the state transition equation is again $x_{t+1} = Ax_t + Bu_t$.

Define the transformed state and control variables

- $\hat{x}_t = \beta^{\frac{t}{2}} x_t$

- $\hat{u}_t = \beta^{\frac{t}{2}} u_t$

and the transformed transition equation matrices

- $\hat{A} = \beta^{\frac{1}{2}} A$
- $\hat{B} = \beta^{\frac{1}{2}} B$

so that the adjusted state and control variables obey the transition law

$$\hat{x}_{t+1} = \hat{A}\hat{x}_t + \hat{B}\hat{u}_t.$$

Then a discounted optimal control problem defined by A, B, R, Q, β having optimal policy characterized by P, F is associated with an equivalent undiscounted problem defined by $\hat{A}, \hat{B}, Q, R$ having optimal policy characterized by $\hat{F}, \hat{P}$ that satisfy the following equations:

$$\hat{F} = (Q + B'\hat{P}B)^{-1}\hat{B}'P\hat{A}$$

and

$$\hat{P} = R + \hat{A}'P\hat{A} - \hat{A}'P\hat{B}(Q + B'\hat{P}\hat{B})^{-1}\hat{B}'P\hat{A}$$

It follows immediately from the definitions of $\hat{A}, \hat{B}$ that $\hat{F} = F$ and $\hat{P} = P$.

By exploiting these transformations, we can solve a discounted problem by solving an associated undiscounted problem.

In particular, we can first transform a discounted LQ problem to an undiscounted one and then solve that discounted optimal regulator problem using the Lagrangian and invariant subspace methods described above.

For example, when $\beta = \frac{1}{1+r}$, we can solve for P with $\hat{A} = \beta^{1/2}A$ and $\hat{B} = \beta^{1/2}B$.

These settings are adopted by default in the function `stationary_P` defined above.

```
β = 1 / (1 + r)
lq.beta = β
```

```
stationary_P(lq)
```

```
eigenvalues:

    W11: [0.97590007 0.97590007]
    W22: [1.02469508 1.02469508]
```

```
array([[ 0.0525, -1.05  ],
       [-1.05  , 21.    ]])
```

We can verify that the solution agrees with one that comes from applying the routine `LQ.stationary_values` in the quantecon package.

```
lq.stationary_values()
```

```
(array([[ 0.0525, -1.05  ],
        [-1.05  , 21.    ]]),
 array([[-0.05,  1.  ]]),
 np.float64(0.0))
```

65.9.2 Lagrangian for Discounted Problem

For several purposes, it is useful explicitly briefly to describe a Lagrangian for a discounted problem.

Thus, for the discounted optimal linear regulator problem, form the Lagrangian

$$L = -\sum_{t=0}^{\infty} \beta^t \left\{ x_t' R x_t + u_t' Q u_t + 2\beta \mu_{t+1}' [A x_t + B u_t - x_{t+1}] \right\} \tag{65.18}$$

where $2\mu_{t+1}$ is a vector of Lagrange multipliers on the state vector x_{t+1}.

First-order conditions for maximization with respect to $\{u_t, x_{t+1}\}_{t=0}^{\infty}$ are

$$\begin{aligned} 2Qu_t + 2\beta B' \mu_{t+1} &= 0 \ , \ t \geq 0 \\ \mu_t &= R x_t + \beta A' \mu_{t+1} \ , \ t \geq 1. \end{aligned} \tag{65.19}$$

Define $2\mu_0$ to be the vector of shadow prices of x_0 and apply an envelope condition to (65.18) to deduce that

$$\mu_0 = R x_0 + \beta A' \mu_1,$$

which is a time $t = 0$ counterpart to the second equation of system (65.19).

Proceeding as we did above with the undiscounted system (65.5), we can rearrange the first-order conditions into the system

$$\begin{bmatrix} I & \beta B Q^{-1} B' \\ 0 & \beta A' \end{bmatrix} \begin{bmatrix} x_{t+1} \\ \mu_{t+1} \end{bmatrix} = \begin{bmatrix} A & 0 \\ -R & I \end{bmatrix} \begin{bmatrix} x_t \\ \mu_t \end{bmatrix} \tag{65.20}$$

which in the special case that $\beta = 1$ agrees with equation (65.5), as expected.

By staring at system (65.20), we can infer identities that shed light on the structure of optimal linear regulator problems, some of which will be useful in this lecture when we apply and extend the methods of this lecture to study Stackelberg and Ramsey problems.

First, note that the first block of equation system (65.20) asserts that when $\mu_{t+1} = P x_{t+1}$, then

$$(I + \beta Q^{-1} B' P B P) x_{t+1} = A x_t,$$

which can be rearranged to sbe

$$x_{t+1} = (I + \beta B Q^{-1} B' P)^{-1} A x_t.$$

This expression for the optimal closed loop dynamics of the state must agree with an alternative expression that we had derived with dynamic programming, namely,

$$x_{t+1} = (A - BF) x_t.$$

But using

$$F = \beta (Q + \beta B' P B)^{-1} B' P A \tag{65.21}$$

it follows that

$$A - BF = (I - \beta B (Q + \beta B' P B)^{-1} B' P) A.$$

Thus, our two expressions for the closed loop dynamics agree if and only if

$$(I + \beta B Q^{-1} B' P)^{-1} = (I - \beta B (Q + \beta B' P B)^{-1} B' P). \tag{65.22}$$

Matrix equation (65.22) can be verified by applying a partitioned inverse formula.

Note

Just use the formula $(a-bd^{-1}c)^{-1} = a^{-1}+a^{-1}b(d-ca^{-1}b)^{-1}ca^{-1}$ for appropriate choices of the matrices a, b, c, d.

Next, note that for *any* fixed F for which eigenvalues of $A-BF$ are less than $\frac{1}{\beta}$ in modulus, the value function associated with using this rule forever is $-x_0\tilde{P}x_0$ where $\tilde{P}$ obeys the following matrix equation:

$$\tilde{P} = (R + F'QF) + \beta(A - BF)'P(A - BF). \tag{65.23}$$

Evidently, $\tilde{P} = P$ only when F obeys formula (65.21).

Next, note that the second equation of system (65.20) implies the "forward looking" equation for the Lagrange multiplier

$$\mu_t = Rx_t + \beta A'\mu_{t+1}$$

whose solution is

$$\mu_t = Px_t,$$

where

$$P = R + \beta A'P(A - BF) \tag{65.24}$$

where we must require that F obeys equation (65.21).

Equations (65.23) and (65.24) provide different perspectives on the optimal value function.

CHAPTER

SIXTYSIX

ELIMINATING CROSS PRODUCTS

66.1 Overview

This lecture describes formulas for eliminating

- cross products between states and control in linear-quadratic dynamic programming problems
- covariances between state and measurement noises in Kalman filtering problems

For a linear-quadratic dynamic programming problem, the idea involves these steps

- transform states and controls in a way that leads to an equivalent problem with no cross-products between transformed states and controls
- solve the transformed problem using standard formulas for problems with no cross-products between states and controls presented in this lecture *Linear Control: Foundations*
- transform the optimal decision rule for the altered problem into the optimal decision rule for the original problem with cross-products between states and controls

66.2 Undiscounted Dynamic Programming Problem

Here is a nonstochastic undiscounted LQ dynamic programming with cross products between states and controls in the objective function.

The problem is defined by the 5-tuple of matrices (A, B, R, Q, H) where R and Q are positive definite symmetric matrices and $A \sim m \times m, B \sim m \times k, Q \sim k \times k, R \sim m \times m$ and $H \sim k \times m$.

The problem is to choose $\{x_{t+1}, u_t\}_{t=0}^{\infty}$ to maximize

$$-\sum_{t=0}^{\infty}(x_t' R x_t + u_t' Q u_t + 2 u_t H x_t)$$

subject to the linear constraints

$$x_{t+1} = A x_t + B u_t, \quad t \geq 0$$

where x_0 is a given initial condition.

The solution to this undiscounted infinite-horizon problem is a time-invariant feedback rule

$$u_t = -F x_t$$

where

$$F = -(Q + B'PB)^{-1}B'PA$$

and $P \sim m \times m$ is a positive definite solution of the algebraic matrix Riccati equation

$$P = R + A'PA - (A'PB + H')(Q + B'PB)^{-1}(B'PA + H).$$

It can be verified that an **equivalent** problem without cross-products between states and controls is defined by a 4-tuple of matrices : (A^*, B, R^*, Q).

That the omitted matrix $H = 0$ indicates that there are no cross products between states and controls in the equivalent problem.

The matrices (A^*, B, R^*, Q) defining the equivalent problem and the value function, policy function matrices P, F^* that solve it are related to the matrices (A, B, R, Q, H) defining the original problem and the value function, policy function matrices P, F that solve the original problem by

$$\begin{aligned}
A^* &= A - BQ^{-1}H, \\
R^* &= R - H'Q^{-1}H, \\
P &= R^* + {A^*}'PA - ({A^*}'PB)(Q + B'PB)^{-1}B'PA^*, \\
F^* &= (Q + B'PB)^{-1}B'PA^*, \\
F &= F^* + Q^{-1}H.
\end{aligned}$$

66.3 Kalman Filter

The **duality** that prevails between a linear-quadratic optimal control and a Kalman filtering problem means that there is an analogous transformation that allows us to transform a Kalman filtering problem with non-zero covariance matrix between between shocks to states and shocks to measurements to an equivalent Kalman filtering problem with zero covariance between shocks to states and measurments.

Let's look at the appropriate transformations.

First, let's recall the Kalman filter with covariance between noises to states and measurements.

The hidden Markov model is

$$\begin{aligned}
x_{t+1} &= Ax_t + Bw_{t+1}, \\
z_{t+1} &= Dx_t + Fw_{t+1},
\end{aligned}$$

where $A \sim m \times m, B \sim m \times p$ and $D \sim k \times m, F \sim k \times p$, and w_{t+1} is the time $t + 1$ component of a sequence of i.i.d. $p \times 1$ normally distibuted random vectors with mean vector zero and covariance matrix equal to a $p \times p$ identity matrix.

Thus, x_t is $m \times 1$ and z_t is $k \times 1$.

The Kalman filtering formulas are

$$\begin{aligned}
K(\Sigma_t) &= (A\Sigma_t D' + BF')(D\Sigma_t D' + FF')^{-1}, \\
\Sigma_{t+1} &= A\Sigma_t A' + BB' - (A\Sigma_t D' + BF')(D\Sigma_t D' + FF')^{-1}(D\Sigma_t A' + FB').
\end{aligned}$$

(eq:Kalman102)

Define tranformed matrices

$$\begin{aligned}
A^* &= A - BF'(FF')^{-1}D, \\
B^*{B^*}' &= BB' - BF'(FF')^{-1}FB'.
\end{aligned}$$

66.3.1 Algorithm

A consequence of formulas {eq}`eq:Kalman102} is that we can use the following algorithm to solve Kalman filtering problems that involve non zero covariances between state and signal noises.

First, compute Σ, K^* using the ordinary Kalman filtering formula with $BF' = 0$, i.e., with zero covariance matrix between random shocks to states and random shocks to measurements.

That is, compute K^* and Σ that satisfy

$$\begin{aligned} K^* &= (A^* \Sigma D')(D \Sigma D' + FF')^{-1} \\ \Sigma &= A^* \Sigma A^{*'} + B^* B^{*'} - (A^* \Sigma D')(D \Sigma D' + FF')^{-1}(D \Sigma A^{*'}). \end{aligned}$$

The Kalman gain for the original problem **with non-zero covariance** between shocks to states and measurements is then

$$K = K^* + BF'(FF')^{-1},$$

The state reconstruction covariance matrix Σ for the original problem equals the state reconstrution covariance matrix for the transformed problem.

66.4 Duality table

Here is a handy table to remember how the Kalman filter and dynamic program are related.

Dynamic Program	Kalman Filter
A	A'
B	D'
H	FB'
Q	FF'
R	BB'
F	K'
P	Σ

CHAPTER

SIXTYSEVEN

THE PERMANENT INCOME MODEL

Contents

67.1 Overview

This lecture describes a rational expectations version of the famous permanent income model of Milton Friedman [Friedman, 1956].

Robert Hall cast Friedman's model within a linear-quadratic setting [Hall, 1978].

Like Hall, we formulate an infinite-horizon linear-quadratic savings problem.

We use the model as a vehicle for illustrating

- alternative formulations of the *state* of a dynamic system
- the idea of *cointegration*
- impulse response functions
- the idea that changes in consumption are useful as predictors of movements in income

Background readings on the linear-quadratic-Gaussian permanent income model are Hall's [Hall, 1978] and chapter 2 of [Ljungqvist and Sargent, 2018].

Let's start with some imports

```
import matplotlib.pyplot as plt
import numpy as np
import random
from numba import jit
```

67.2 The Savings Problem

In this section, we state and solve the savings and consumption problem faced by the consumer.

67.2.1 Preliminaries

We use a class of stochastic processes called martingales.

A discrete-time martingale is a stochastic process (i.e., a sequence of random variables) $\{X_t\}$ with finite mean at each t and satisfying

$$\mathbb{E}_t[X_{t+1}] = X_t, \qquad t = 0, 1, 2, \ldots$$

Here $\mathbb{E}_t := \mathbb{E}[\cdot \mid \mathcal{F}_t]$ is a conditional mathematical expectation conditional on the time t *information set* $\mathcal{F}_t$.

The latter is just a collection of random variables that the modeler declares to be visible at t.

- When not explicitly defined, it is usually understood that $\mathcal{F}_t = \{X_t, X_{t-1}, \ldots, X_0\}$.

Martingales have the feature that the history of past outcomes provides no predictive power for changes between current and future outcomes.

For example, the current wealth of a gambler engaged in a "fair game" has this property.

One common class of martingales is the family of *random walks*.

A **random walk** is a stochastic process $\{X_t\}$ that satisfies

$$X_{t+1} = X_t + w_{t+1}$$

for some IID zero mean *innovation* sequence $\{w_t\}$.

Evidently, X_t can also be expressed as

$$X_t = \sum_{j=1}^{t} w_j + X_0$$

Not every martingale arises as a random walk (see, for example, Wald's martingale).

67.2.2 The Decision Problem

A consumer has preferences over consumption streams that are ordered by the utility functional

$$\mathbb{E}_0 \left[\sum_{t=0}^{\infty} \beta^t u(c_t) \right] \tag{67.1}$$

where

- $\mathbb{E}_t$ is the mathematical expectation conditioned on the consumer's time t information
- c_t is time t consumption
- u is a strictly concave one-period utility function
- $\beta \in (0, 1)$ is a discount factor

The consumer maximizes (67.1) by choosing a consumption, borrowing plan $\{c_t, b_{t+1}\}_{t=0}^{\infty}$ subject to the sequence of budget constraints

$$c_t + b_t = \frac{1}{1+r} b_{t+1} + y_t \quad t \geq 0 \tag{67.2}$$

Here

- y_t is an exogenous endowment process.
- $r > 0$ is a time-invariant risk-free net interest rate.
- b_t is one-period risk-free debt maturing at t.

The consumer also faces initial conditions b_0 and y_0, which can be fixed or random.

67.2.3 Assumptions

For the remainder of this lecture, we follow Friedman and Hall in assuming that $(1+r)^{-1} = \beta$.

Regarding the endowment process, we assume it has the *state-space representation*

$$\begin{aligned} z_{t+1} &= A z_t + C w_{t+1} \\ y_t &= U z_t \end{aligned} \tag{67.3}$$

where

- $\{w_t\}$ is an IID vector process with $\mathbb{E} w_t = 0$ and $\mathbb{E} w_t w_t' = I$.
- The *spectral radius* of A satisfies $\rho(A) < \sqrt{1/\beta}$.
- U is a selection vector that pins down y_t as a particular linear combination of components of z_t.

The restriction on $\rho(A)$ prevents income from growing so fast that discounted geometric sums of some quadratic forms to be described below become infinite.

Regarding preferences, we assume the quadratic utility function

$$u(c_t) = -(c_t - \gamma)^2$$

where γ is a bliss level of consumption.

Note

Along with this quadratic utility specification, we allow consumption to be negative. However, by choosing parameters appropriately, we can make the probability that the model generates negative consumption paths over finite time horizons as low as desired.

Finally, we impose the *no Ponzi scheme* condition

$$\mathbb{E}_0 \left[\sum_{t=0}^{\infty} \beta^t b_t^2 \right] < \infty \tag{67.4}$$

This condition rules out an always-borrow scheme that would allow the consumer to enjoy bliss consumption forever.

67.2.4 First-Order Conditions

First-order conditions for maximizing (67.1) subject to (67.2) are

$$\mathbb{E}_t[u'(c_{t+1})] = u'(c_t), \qquad t = 0, 1, \ldots \tag{67.5}$$

These optimality conditions are also known as *Euler equations*.

If you're not sure where they come from, you can find a proof sketch in the *appendix*.

With our quadratic preference specification, (67.5) has the striking implication that consumption follows a martingale:

$$\mathbb{E}_t[c_{t+1}] = c_t \tag{67.6}$$

(In fact, quadratic preferences are *necessary* for this conclusion[1].)

One way to interpret (67.6) is that consumption will change only when "new information" about permanent income is revealed.

These ideas will be clarified below.

67.2.5 The Optimal Decision Rule

Now let's deduce the optimal decision rule[2].

> **Note**
>
> One way to solve the consumer's problem is to apply *dynamic programming* as in *this lecture*. We do this later. But first we use an alternative approach that is revealing and shows the work that dynamic programming does for us behind the scenes.

In doing so, we need to combine

1. the optimality condition (67.6)
2. the period-by-period budget constraint (67.2), and
3. the boundary condition (67.4)

To accomplish this, observe first that (67.4) implies $\lim_{t\to\infty} \beta^{\frac{t}{2}} b_{t+1} = 0$.

Using this restriction on the debt path and solving (67.2) forward yields

$$b_t = \sum_{j=0}^{\infty} \beta^j (y_{t+j} - c_{t+j}) \tag{67.7}$$

Take conditional expectations on both sides of (67.7) and use the martingale property of consumption and the *law of iterated expectations* to deduce

$$b_t = \sum_{j=0}^{\infty} \beta^j \mathbb{E}_t[y_{t+j}] - \frac{c_t}{1-\beta} \tag{67.8}$$

[1] A linear marginal utility is essential for deriving (67.6) from (67.5). Suppose instead that we had imposed the following more standard assumptions on the utility function: $u'(c) > 0, u''(c) < 0, u'''(c) > 0$ and required that $c \geq 0$. The Euler equation remains (67.5). But the fact that $u''' < 0$ implies via Jensen's inequality that $\mathbb{E}_t[u'(c_{t+1})] > u'(\mathbb{E}_t[c_{t+1}])$. This inequality together with (67.5) implies that $\mathbb{E}_t[c_{t+1}] > c_t$ (consumption is said to be a 'submartingale'), so that consumption stochastically diverges to $+\infty$. The consumer's savings also diverge to $+\infty$.

[2] An optimal decision rule is a map from the current state into current actions—in this case, consumption.

Expressed in terms of c_t we get

$$c_t = (1-\beta)\left[\sum_{j=0}^{\infty}\beta^j \mathbb{E}_t[y_{t+j}] - b_t\right] = \frac{r}{1+r}\left[\sum_{j=0}^{\infty}\beta^j \mathbb{E}_t[y_{t+j}] - b_t\right] \tag{67.9}$$

where the last equality uses $(1+r)\beta = 1$.

These last two equations assert that consumption equals *economic income*

- **financial wealth** equals $-b_t$
- **non-financial wealth** equals $\sum_{j=0}^{\infty}\beta^j \mathbb{E}_t[y_{t+j}]$
- **total wealth** equals the sum of financial and non-financial wealth
- a **marginal propensity to consume out of total wealth** equals the interest factor $\frac{r}{1+r}$
- **economic income** equals
  - a constant marginal propensity to consume times the sum of non-financial wealth and financial wealth
  - the amount the consumer can consume while leaving its wealth intact

Responding to the State

The *state* vector confronting the consumer at t is $\begin{bmatrix} b_t & z_t \end{bmatrix}$.

Here

- z_t is an *exogenous* component, unaffected by consumer behavior.
- b_t is an *endogenous* component (since it depends on the decision rule).

Note that z_t contains all variables useful for forecasting the consumer's future endowment.

It is plausible that current decisions c_t and b_{t+1} should be expressible as functions of z_t and b_t.

This is indeed the case.

In fact, from *this discussion*, we see that

$$\sum_{j=0}^{\infty}\beta^j \mathbb{E}_t[y_{t+j}] = \mathbb{E}_t\left[\sum_{j=0}^{\infty}\beta^j y_{t+j}\right] = U(I-\beta A)^{-1}z_t$$

Combining this with (67.9) gives

$$c_t = \frac{r}{1+r}\left[U(I-\beta A)^{-1}z_t - b_t\right] \tag{67.10}$$

Using this equality to eliminate c_t in the budget constraint (67.2) gives

$$\begin{aligned}
b_{t+1} &= (1+r)(b_t + c_t - y_t) \\
&= (1+r)b_t + r[U(I-\beta A)^{-1}z_t - b_t] - (1+r)Uz_t \\
&= b_t + U[r(I-\beta A)^{-1} - (1+r)I]z_t \\
&= b_t + U(I-\beta A)^{-1}(A-I)z_t
\end{aligned}$$

To get from the second last to the last expression in this chain of equalities is not trivial.

A key is to use the fact that $(1+r)\beta = 1$ and $(I-\beta A)^{-1} = \sum_{j=0}^{\infty}\beta^j A^j$.

We've now successfully written c_t and b_{t+1} as functions of b_t and z_t.

A State-Space Representation

We can summarize our dynamics in the form of a linear state-space system governing consumption, debt and income:

$$
\begin{aligned}
z_{t+1} &= A z_t + C w_{t+1} \\
b_{t+1} &= b_t + U[(I - \beta A)^{-1}(A - I)] z_t \\
y_t &= U z_t \\
c_t &= (1 - \beta)[U(I - \beta A)^{-1} z_t - b_t]
\end{aligned} \tag{67.11}
$$

To write this more succinctly, let

$$
x_t = \begin{bmatrix} z_t \\ b_t \end{bmatrix}, \quad \tilde A = \begin{bmatrix} A & 0 \\ U(I - \beta A)^{-1}(A - I) & 1 \end{bmatrix}, \quad \tilde C = \begin{bmatrix} C \\ 0 \end{bmatrix}
$$

and

$$
\tilde U = \begin{bmatrix} U & 0 \\ (1 - \beta) U (I - \beta A)^{-1} & -(1 - \beta) \end{bmatrix}, \quad \tilde y_t = \begin{bmatrix} y_t \\ c_t \end{bmatrix}
$$

Then we can express equation (67.11) as

$$
\begin{aligned}
x_{t+1} &= \tilde A x_t + \tilde C w_{t+1} \\
\tilde y_t &= \tilde U x_t
\end{aligned} \tag{67.12}
$$

We can use the following formulas from *linear state space models* to compute population mean $\mu_t = \mathbb{E} x_t$ and covariance $\Sigma_t := \mathbb{E}[(x_t - \mu_t)(x_t - \mu_t)']$

$$
\mu_{t+1} = \tilde A \mu_t \quad \text{with} \quad \mu_0 \text{ given} \tag{67.13}
$$

$$
\Sigma_{t+1} = \tilde A \Sigma_t \tilde A' + \tilde C \tilde C' \quad \text{with} \quad \Sigma_0 \text{ given} \tag{67.14}
$$

We can then compute the mean and covariance of $\tilde y_t$ from

$$
\begin{aligned}
\mu_{y,t} &= \tilde U \mu_t \\
\Sigma_{y,t} &= \tilde U \Sigma_t \tilde U'
\end{aligned} \tag{67.15}
$$

A Simple Example with IID Income

To gain some preliminary intuition on the implications of (67.11), let's look at a highly stylized example where income is just IID.

(Later examples will investigate more realistic income streams.)

In particular, let $\{w_t\}_{t=1}^\infty$ be IID and scalar standard normal, and let

$$
z_t = \begin{bmatrix} z_t^1 \\ 1 \end{bmatrix}, \quad A = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 1 & \mu \end{bmatrix}, \quad C = \begin{bmatrix} \sigma \\ 0 \end{bmatrix}
$$

Finally, let $b_0 = z_0^1 = 0$.

Under these assumptions, we have $y_t = \mu + \sigma w_t \sim N(\mu, \sigma^2)$.

Further, if you work through the state space representation, you will see that

$$
b_t = -\sigma \sum_{j=1}^{t-1} w_j
$$

$$
c_t = \mu + (1 - \beta) \sigma \sum_{j=1}^{t} w_j
$$

Thus, income is IID and debt and consumption are both Gaussian random walks.

Defining assets as $-b_t$, we see that assets are just the cumulative sum of unanticipated incomes prior to the present date.

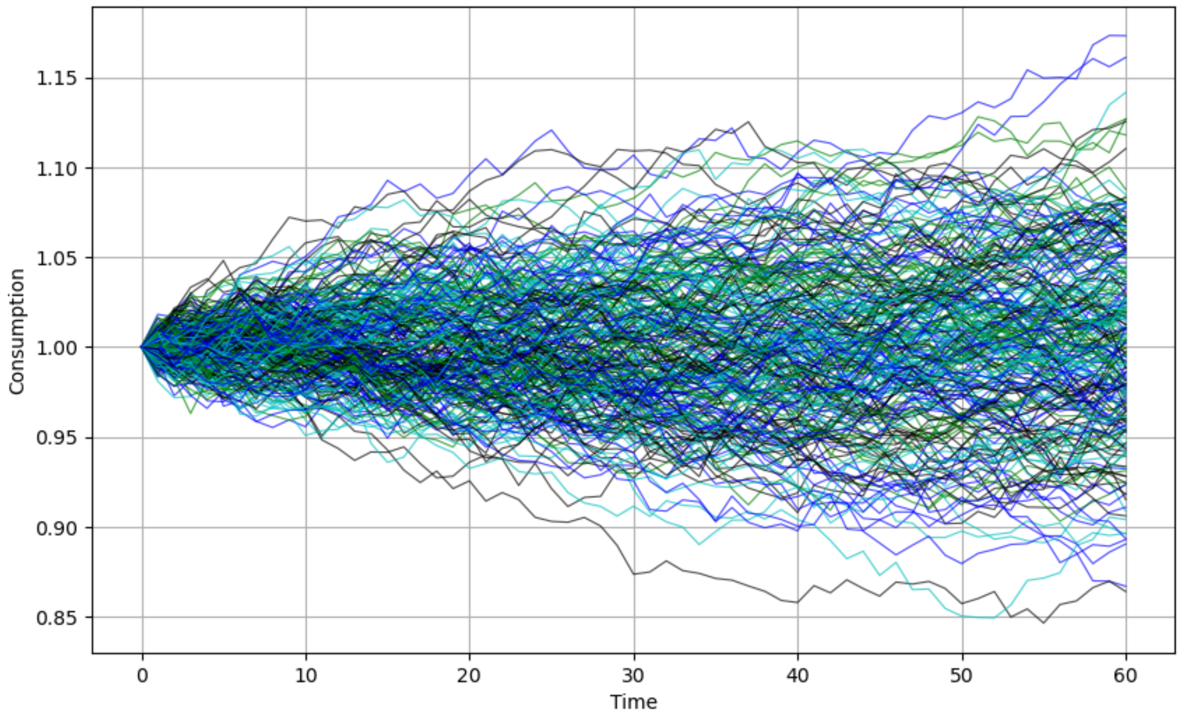
The next figure shows a typical realization with $r = 0.05$, $\mu = 1$, and $\sigma = 0.15$

```
r = 0.05
β = 1 / (1 + r)
σ = 0.15
μ = 1
T = 60

@jit
def time_path(T):
    w = np.random.randn(T+1)  # w_0, w_1, ..., w_T
    w[0] = 0
    b = np.zeros(T+1)
    for t in range(1, T+1):
        b[t] = w[1:t].sum()
    b = -σ * b
    c = μ + (1 - β) * (σ * w - b)
    return w, b, c

w, b, c = time_path(T)

fig, ax = plt.subplots(figsize=(10, 6))

ax.plot(μ + σ * w, 'g-', label="Non-financial income")
ax.plot(c, 'k-', label="Consumption")
ax.plot( b, 'b-', label="Debt")
ax.legend(ncol=3, mode='expand', bbox_to_anchor=(0., 1.02, 1., .102))
ax.grid()
ax.set_xlabel('Time')

plt.show()
```

Observe that consumption is considerably smoother than income.

The figure below shows the consumption paths of 250 consumers with independent income streams

```
fig, ax = plt.subplots(figsize=(10, 6))

b_sum = np.zeros(T+1)
for i in range(250):
    w, b, c = time_path(T)  # Generate new time path
    rcolor = random.choice(('c', 'g', 'b', 'k'))
    ax.plot(c, color=rcolor, lw=0.8, alpha=0.7)

ax.grid()
ax.set(xlabel='Time', ylabel='Consumption')

plt.show()
```



67.3 Alternative Representations

In this section, we shed more light on the evolution of savings, debt and consumption by representing their dynamics in several different ways.

67.3.1 Hall's Representation

Hall [Hall, 1978] suggested an insightful way to summarize the implications of LQ permanent income theory.

First, to represent the solution for b_t, shift (67.9) forward one period and eliminate b_{t+1} by using (67.2) to obtain

$$c_{t+1} = (1-\beta)\sum_{j=0}^{\infty}\beta^j \mathbb{E}_{t+1}[y_{t+j+1}] - (1-\beta)\left[\beta^{-1}(c_t + b_t - y_t)\right]$$

If we add and subtract $\beta^{-1}(1-\beta)\sum_{j=0}^{\infty}\beta^j \mathbb{E}_t y_{t+j}$ from the right side of the preceding equation and rearrange, we obtain

$$c_{t+1} - c_t = (1-\beta)\sum_{j=0}^{\infty}\beta^j \left\{\mathbb{E}_{t+1}[y_{t+j+1}] - \mathbb{E}_t[y_{t+j+1}]\right\} \tag{67.16}$$

The right side is the time $t+1$ *innovation to the expected present value* of the endowment process $\{y_t\}$.

We can represent the optimal decision rule for (c_t, b_{t+1}) in the form of (67.16) and (67.8), which we repeat:

$$b_t = \sum_{j=0}^{\infty}\beta^j \mathbb{E}_t[y_{t+j}] - \frac{1}{1-\beta}c_t \tag{67.17}$$

Equation (67.17) asserts that the consumer's debt due at t equals the expected present value of its endowment minus the expected present value of its consumption stream.

A high debt thus indicates a large expected present value of surpluses $y_t - c_t$.

Recalling again our discussion on *forecasting geometric sums*, we have

$$\begin{aligned} \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j} &= U(I - \beta A)^{-1} z_t \\ \mathbb{E}_{t+1} \sum_{j=0}^{\infty} \beta^j y_{t+j+1} &= U(I - \beta A)^{-1} z_{t+1} \\ \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j+1} &= U(I - \beta A)^{-1} A z_t \end{aligned}$$

Using these formulas together with (67.3) and substituting into (67.16) and (67.17) gives the following representation for the consumer's optimum decision rule:

$$\begin{aligned} c_{t+1} &= c_t + (1 - \beta) U (I - \beta A)^{-1} C w_{t+1} \\ b_t &= U (I - \beta A)^{-1} z_t - \frac{1}{1 - \beta} c_t \\ y_t &= U z_t \\ z_{t+1} &= A z_t + C w_{t+1} \end{aligned} \tag{67.18}$$

Representation (67.18) makes clear that

- The state can be taken as (c_t, z_t).
  - The endogenous part is c_t and the exogenous part is z_t.
  - Debt b_t has disappeared as a component of the state because it is encoded in c_t.
- Consumption is a random walk with innovation $(1 - \beta) U (I - \beta A)^{-1} C w_{t+1}$.
  - This is a more explicit representation of the martingale result in (67.6).

67.3.2 Cointegration

Representation (67.18) reveals that the joint process $\{c_t, b_t\}$ possesses the property that Engle and Granger [Engle and Granger, 1987] called cointegration.

Cointegration is a tool that allows us to apply powerful results from the theory of stationary stochastic processes to (certain transformations of) nonstationary models.

To apply cointegration in the present context, suppose that z_t is asymptotically stationary[3].

Despite this, both c_t and b_t will be non-stationary because they have unit roots (see (67.11) for b_t).

Nevertheless, there is a linear combination of c_t, b_t that *is* asymptotically stationary.

In particular, from the second equality in (67.18) we have

$$(1 - \beta) b_t + c_t = (1 - \beta) U (I - \beta A)^{-1} z_t \tag{67.19}$$

Hence the linear combination $(1 - \beta) b_t + c_t$ is asymptotically stationary.

Accordingly, Granger and Engle would call $\begin{bmatrix} (1 - \beta) & 1 \end{bmatrix}$ a **cointegrating vector** for the state.

When applied to the nonstationary vector process $\begin{bmatrix} b_t & c_t \end{bmatrix}'$, it yields a process that is asymptotically stationary.

[3] This would be the case if, for example, the *spectral radius* of A is strictly less than one.

Equation (67.19) can be rearranged to take the form

$$(1-\beta)b_t + c_t = (1-\beta)\mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j} \tag{67.20}$$

Equation (67.20) asserts that the *cointegrating residual* on the left side equals the conditional expectation of the geometric sum of future incomes on the right[4].

67.3.3 Cross-Sectional Implications

Consider again (67.18), this time in light of our discussion of distribution dynamics in the *lecture on linear systems*.

The dynamics of c_t are given by

$$c_{t+1} = c_t + (1-\beta)U(I-\beta A)^{-1}Cw_{t+1} \tag{67.21}$$

or

$$c_t = c_0 + \sum_{j=1}^{t} \hat{w}_j \quad \text{for} \quad \hat{w}_{t+1} := (1-\beta)U(I-\beta A)^{-1}Cw_{t+1}$$

The unit root affecting c_t causes the time t variance of c_t to grow linearly with t.

In particular, since $\{\hat{w}_t\}$ is IID, we have

$$\text{Var}[c_t] = \text{Var}[c_0] + t\,\hat{\sigma}^2 \tag{67.22}$$

where

$$\hat{\sigma}^2 := (1-\beta)^2 U(I-\beta A)^{-1}CC'(I-\beta A')^{-1}U'$$

When $\hat{\sigma} > 0$, $\{c_t\}$ has no asymptotic distribution.

Let's consider what this means for a cross-section of ex-ante identical consumers born at time 0.

Let the distribution of c_0 represent the cross-section of initial consumption values.

Equation (67.22) tells us that the variance of c_t increases over time at a rate proportional to t.

A number of different studies have investigated this prediction and found some support for it (see, e.g., [Deaton and Paxson, 1994], [Storesletten *et al.*, 2004]).

67.3.4 Impulse Response Functions

Impulse response functions measure responses to various impulses (i.e., temporary shocks).

The impulse response function of $\{c_t\}$ to the innovation $\{w_t\}$ is a box.

In particular, the response of c_{t+j} to a unit increase in the innovation w_{t+1} is $(1-\beta)U(I-\beta A)^{-1}C$ for all $j \geq 1$.

[4] See [John Y. Campbell, 1988], [Lettau and Ludvigson, 2001], [Lettau and Ludvigson, 2004] for interesting applications of related ideas.

67.3.5 Moving Average Representation

It's useful to express the innovation to the expected present value of the endowment process in terms of a moving average representation for income y_t.

The endowment process defined by (67.3) has the moving average representation

$$y_{t+1} = d(L)w_{t+1} \tag{67.23}$$

where

- $d(L) = \sum_{j=0}^{\infty} d_j L^j$ for some sequence d_j, where L is the lag operator[5]
- at time t, the consumer has an information set[6] $w^t = [w_t, w_{t-1}, ...]$

Notice that

$$y_{t+j} - \mathbb{E}_t[y_{t+j}] = d_0 w_{t+j} + d_1 w_{t+j-1} + \cdots + d_{j-1} w_{t+1}$$

It follows that

$$\mathbb{E}_{t+1}[y_{t+j}] - \mathbb{E}_t[y_{t+j}] = d_{j-1} w_{t+1} \tag{67.24}$$

Using (67.24) in (67.16) gives

$$c_{t+1} - c_t = (1-\beta)d(\beta)w_{t+1} \tag{67.25}$$

The object $d(\beta)$ is the **present value of the moving average coefficients** in the representation for the endowment process y_t.

67.4 Two Classic Examples

We illustrate some of the preceding ideas with two examples.

In both examples, the endowment follows the process $y_t = z_{1t} + z_{2t}$ where

$$\begin{bmatrix} z_{1t+1} \\ z_{2t+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} z_{1t} \\ z_{2t} \end{bmatrix} + \begin{bmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{bmatrix} \begin{bmatrix} w_{1t+1} \\ w_{2t+1} \end{bmatrix}$$

Here

- w_{t+1} is an IID 2×1 process distributed as $N(0, I)$.
- z_{1t} is a permanent component of y_t.
- z_{2t} is a purely transitory component of y_t.

67.4.1 Example 1

Assume as before that the consumer observes the state z_t at time t.

In view of (67.18) we have

$$c_{t+1} - c_t = \sigma_1 w_{1t+1} + (1-\beta)\sigma_2 w_{2t+1} \tag{67.26}$$

Formula (67.26) shows how an increment $\sigma_1 w_{1t+1}$ to the permanent component of income z_{1t+1} leads to

[5] Representation (67.3) implies that $d(L) = U(I - AL)^{-1}C$.

[6] A moving average representation for a process y_t is said to be **fundamental** if the linear space spanned by y^t is equal to the linear space spanned by w^t. A time-invariant innovations representation, attained via the Kalman filter, is by construction fundamental.

- a permanent one-for-one increase in consumption and
- no increase in savings $-b_{t+1}$

But the purely transitory component of income $\sigma_2 w_{2t+1}$ leads to a permanent increment in consumption by a fraction $1-\beta$ of transitory income.

The remaining fraction β is saved, leading to a permanent increment in $-b_{t+1}$.

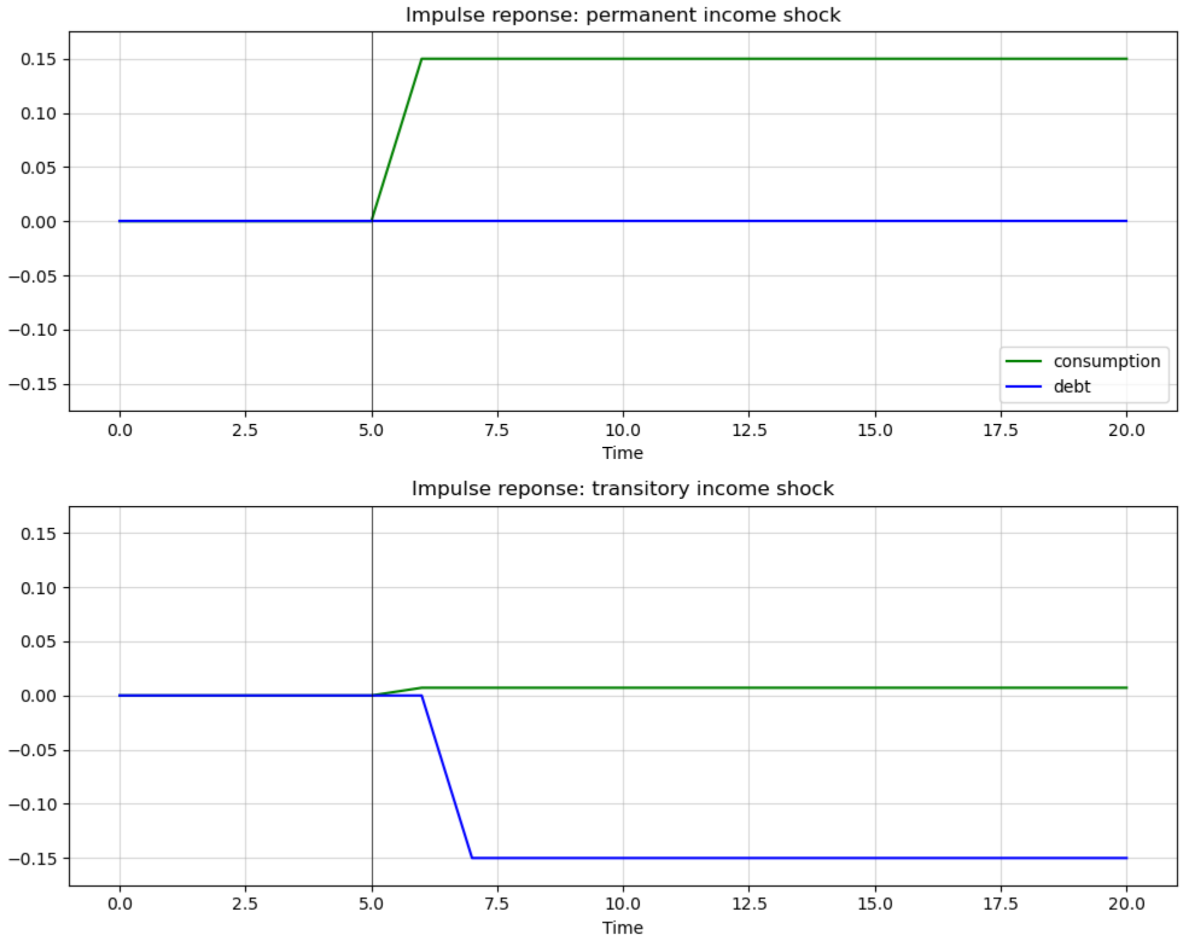
Application of the formula for debt in (67.11) to this example shows that

$$b_{t+1} - b_t = -z_{2t} = -\sigma_2 w_{2t} \tag{67.27}$$

This confirms that none of $\sigma_1 w_{1t}$ is saved, while all of $\sigma_2 w_{2t}$ is saved.

The next figure displays impulse-response functions that illustrates these very different reactions to transitory and permanent income shocks.

```
r = 0.05
β = 1 / (1 + r)
S = 5   # Impulse date
σ1 = σ2 = 0.15

@jit
def time_path(T, permanent=False):
    "Time path of consumption and debt given shock sequence"
    w1 = np.zeros(T+1)
    w2 = np.zeros(T+1)
    b = np.zeros(T+1)
    c = np.zeros(T+1)
    if permanent:
        w1[S+1] = 1.0
    else:
        w2[S+1] = 1.0
    for t in range(1, T):
        b[t+1] = b[t] - σ2 * w2[t]
        c[t+1] = c[t] + σ1 * w1[t+1] + (1 - β) * σ2 * w2[t+1]
    return b, c


fig, axes = plt.subplots(2, 1, figsize=(10, 8))
titles = ['permanent', 'transitory']

L = 0.175

for ax, truefalse, title in zip(axes, (True, False), titles):
    b, c = time_path(T=20, permanent=truefalse)
    ax.set_title(f'Impulse reponse: {title} income shock')
    ax.plot(c, 'g-', label="consumption")
    ax.plot(b, 'b-', label="debt")
    ax.plot((S, S), (-L, L), 'k-', lw=0.5)
    ax.grid(alpha=0.5)
    ax.set(xlabel=r'Time', ylim=(-L, L))

axes[0].legend(loc='lower right')

plt.tight_layout()
plt.show()
```



Notice how the permanent income shock provokes no change in assets $-b_{t+1}$ and an immediate permanent change in consumption equal to the permanent increment in non-financial income.

In contrast, notice how most of a transitory income shock is saved and only a small amount is saved.

The box-like impulse responses of consumption to both types of shock reflect the random walk property of the optimal consumption decision.

67.4.2 Example 2

Assume now that at time t the consumer observes y_t, and its history up to t, but not z_t.

Under this assumption, it is appropriate to use an *innovation representation* to form A, C, U in (67.18).

The discussion in sections 2.9.1 and 2.11.3 of [Ljungqvist and Sargent, 2018] shows that the pertinent state space representation for y_t is

$$\begin{aligned}
\begin{bmatrix} y_{t+1} \\ a_{t+1} \end{bmatrix} &= \begin{bmatrix} 1 & -(1-K) \\ 0 & 0 \end{bmatrix} \begin{bmatrix} y_t \\ a_t \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix} a_{t+1} \\
y_t &= \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} y_t \\ a_t \end{bmatrix}
\end{aligned}$$

where

- $K :=$ the stationary Kalman gain

- $a_t := y_t - E[y_t \mid y_{t-1}, \dots, y_0]$

In the same discussion in [Ljungqvist and Sargent, 2018] it is shown that $K \in [0, 1]$ and that K increases as σ_1/σ_2 does.

In other words, K increases as the ratio of the standard deviation of the permanent shock to that of the transitory shock increases.

Please see *first look at the Kalman filter*.

Applying formulas (67.18) implies

$$c_{t+1} - c_t = [1 - \beta(1 - K)]a_{t+1} \tag{67.28}$$

where the endowment process can now be represented in terms of the univariate innovation to y_t as

$$y_{t+1} - y_t = a_{t+1} - (1 - K)a_t \tag{67.29}$$

Equation (67.29) indicates that the consumer regards

- fraction K of an innovation a_{t+1} to y_{t+1} as *permanent*
- fraction $1 - K$ as purely transitory

The consumer permanently increases his consumption by the full amount of his estimate of the permanent part of a_{t+1}, but by only $(1 - \beta)$ times his estimate of the purely transitory part of a_{t+1}.

Therefore, in total, he permanently increments his consumption by a fraction $K + (1 - \beta)(1 - K) = 1 - \beta(1 - K)$ of a_{t+1}.

He saves the remaining fraction $\beta(1 - K)$.

According to equation (67.29), the first difference of income is a first-order moving average.

Equation (67.28) asserts that the first difference of consumption is IID.

Application of formula to this example shows that

$$b_{t+1} - b_t = (K - 1)a_t \tag{67.30}$$

This indicates how the fraction K of the innovation to y_t that is regarded as permanent influences the fraction of the innovation that is saved.

67.5 Further Reading

The model described above significantly changed how economists think about consumption.

While Hall's model does a remarkably good job as a first approximation to consumption data, it's widely believed that it doesn't capture important aspects of some consumption/savings data.

For example, liquidity constraints and precautionary savings appear to be present sometimes.

Further discussion can be found in, e.g., [Hall and Mishkin, 1982], [Parker, 1999], [Deaton, 1991], [Carroll, 2001].

67.6 Appendix: The Euler Equation

Where does the first-order condition (67.5) come from?

Here we'll give a proof for the two-period case, which is representative of the general argument.

The finite horizon equivalent of the no-Ponzi condition is that the agent cannot end her life in debt, so $b_2 = 0$.

From the budget constraint (67.2) we then have

$$c_0 = \frac{b_1}{1+r} - b_0 + y_0 \quad \text{and} \quad c_1 = y_1 - b_1$$

Here b_0 and y_0 are given constants.

Substituting these constraints into our two-period objective $u(c_0) + \beta \mathbb{E}_0[u(c_1)]$ gives

$$\max_{b_1} \left\{ u\left(\frac{b_1}{R} - b_0 + y_0\right) + \beta \, \mathbb{E}_0[u(y_1 - b_1)] \right\}$$

You will be able to verify that the first-order condition is

$$u'(c_0) = \beta R \, \mathbb{E}_0[u'(c_1)]$$

Using $\beta R = 1$ gives (67.5) in the two-period case.

The proof for the general case is similar.

CHAPTER

SIXTYEIGHT

PERMANENT INCOME II: LQ TECHNIQUES

Contents

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
```

68.1 Overview

This lecture continues our analysis of the linear-quadratic (LQ) permanent income model of savings and consumption.

As we saw in our *previous lecture* on this topic, Robert Hall [Hall, 1978] used the LQ permanent income model to restrict and interpret intertemporal comovements of nondurable consumption, nonfinancial income, and financial wealth.

For example, we saw how the model asserts that for any covariance stationary process for nonfinancial income

- consumption is a random walk
- financial wealth has a unit root and is cointegrated with consumption

Other applications use the same LQ framework.

For example, a model isomorphic to the LQ permanent income model has been used by Robert Barro [Barro, 1979] to interpret intertemporal comovements of a government's tax collections, its expenditures net of debt service, and its public debt.

This isomorphism means that in analyzing the LQ permanent income model, we are in effect also analyzing the Barro tax smoothing model.

It is just a matter of appropriately relabeling the variables in Hall's model.

In this lecture, we'll

- show how the solution to the LQ permanent income model can be obtained using LQ control methods.
- represent the model as a linear state space system as in *this lecture*.
- apply QuantEcon's LinearStateSpace class to characterize statistical features of the consumer's optimal consumption and borrowing plans.

We'll then use these characterizations to construct a simple model of cross-section wealth and consumption dynamics in the spirit of Truman Bewley [Bewley, 1986].

(Later we'll study other Bewley models—see *this lecture*.)

The model will prove useful for illustrating concepts such as

- stationarity
- ergodicity
- ensemble moments and cross-section observations

Let's start with some imports:

```
import matplotlib.pyplot as plt
import quantecon as qe
import numpy as np
import scipy.linalg as la
```

68.2 Setup

Let's recall the basic features of the model discussed in the *permanent income model*.

Consumer preferences are ordered by

$$E_0 \sum_{t=0}^{\infty} \beta^t u(c_t) \tag{68.1}$$

where $u(c) = -(c - \gamma)^2$.

The consumer maximizes (68.1) by choosing a consumption, borrowing plan $\{c_t, b_{t+1}\}_{t=0}^{\infty}$ subject to the sequence of budget constraints

$$c_t + b_t = \frac{1}{1+r} b_{t+1} + y_t, \quad t \geq 0 \tag{68.2}$$

and the no-Ponzi condition

$$E_0 \sum_{t=0}^{\infty} \beta^t b_t^2 < \infty \tag{68.3}$$

The interpretation of all variables and parameters are the same as in the *previous lecture*.

We continue to assume that $(1 + r)\beta = 1$.

The dynamics of $\{y_t\}$ again follow the linear state space model

$$\begin{aligned} z_{t+1} &= A z_t + C w_{t+1} \\ y_t &= U z_t \end{aligned} \tag{68.4}$$

The restrictions on the shock process and parameters are the same as in our *previous lecture*.

68.2.1 Digression on a Useful Isomorphism

The LQ permanent income model of consumption is mathematically isomorphic with a version of Barro's [Barro, 1979] model of tax smoothing.

In the LQ permanent income model

- the household faces an exogenous process of nonfinancial income
- the household wants to smooth consumption across states and time

In the Barro tax smoothing model

- a government faces an exogenous sequence of government purchases (net of interest payments on its debt)
- a government wants to smooth tax collections across states and time

If we set

- T_t, total tax collections in Barro's model to consumption c_t in the LQ permanent income model.
- G_t, exogenous government expenditures in Barro's model to nonfinancial income y_t in the permanent income model.
- B_t, government risk-free one-period assets falling due in Barro's model to risk-free one-period consumer debt b_t falling due in the LQ permanent income model.
- R, the gross rate of return on risk-free one-period government debt in Barro's model to the gross rate of return $1 + r$ on financial assets in the permanent income model of consumption.

then the two models are mathematically equivalent.

All characterizations of a $\{c_t, y_t, b_t\}$ in the LQ permanent income model automatically apply to a $\{T_t, G_t, B_t\}$ process in the Barro model of tax smoothing.

See consumption and tax smoothing models for further exploitation of an isomorphism between consumption and tax smoothing models.

68.2.2 A Specification of the Nonfinancial Income Process

For the purposes of this lecture, let's assume $\{y_t\}$ is a second-order univariate autoregressive process:

$$y_{t+1} = \alpha + \rho_1 y_t + \rho_2 y_{t-1} + \sigma w_{t+1}$$

We can map this into the linear state space framework in (68.4), as discussed in our lecture on *linear models*.

To do so we take

$$z_t = \begin{bmatrix} 1 \\ y_t \\ y_{t-1} \end{bmatrix}, \quad A = \begin{bmatrix} 1 & 0 & 0 \\ \alpha & \rho_1 & \rho_2 \\ 0 & 1 & 0 \end{bmatrix}, \quad C = \begin{bmatrix} 0 \\ \sigma \\ 0 \end{bmatrix}, \quad \text{and} \quad U = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}$$

68.3 The LQ Approach

Previously we solved the permanent income model by solving a system of linear expectational difference equations subject to two boundary conditions.

Here we solve the same model using *LQ methods* based on dynamic programming.

After confirming that answers produced by the two methods agree, we apply QuantEcon's LinearStateSpace class to illustrate features of the model.

Why solve a model in two distinct ways?

Because by doing so we gather insights about the structure of the model.

Our earlier approach based on solving a system of expectational difference equations brought to the fore the role of the consumer's expectations about future nonfinancial income.

On the other hand, formulating the model in terms of an LQ dynamic programming problem reminds us that

- finding the state (of a dynamic programming problem) is an art, and
- iterations on a Bellman equation implicitly jointly solve both a forecasting problem and a control problem

68.3.1 The LQ Problem

Recall from our *lecture on LQ theory* that the optimal linear regulator problem is to choose a decision rule for u_t to minimize

$$\mathbb{E}\sum_{t=0}^{\infty}\beta^t\{x_t'Rx_t + u_t'Qu_t\},$$

subject to x_0 given and the law of motion

$$x_{t+1} = \tilde{A}x_t + \tilde{B}u_t + \tilde{C}w_{t+1}, \qquad t \geq 0, \tag{68.5}$$

where w_{t+1} is IID with mean vector zero and $\mathbb{E}w_t w_t' = I$.

The tildes in $\tilde{A}, \tilde{B}, \tilde{C}$ are to avoid clashing with notation in (68.4).

The value function for this problem is $v(x) = -x'Px - d$, where

- P is the unique positive semidefinite solution of the *corresponding matrix Riccati equation*.
- The scalar d is given by $d = \beta(1-\beta)^{-1}\text{trace}(P\tilde{C}\tilde{C}')$.

The optimal policy is $u_t = -Fx_t$, where $F := \beta(Q + \beta\tilde{B}'P\tilde{B})^{-1}\tilde{B}'P\tilde{A}$.

Under an optimal decision rule F, the state vector x_t evolves according to $x_{t+1} = (\tilde{A} - \tilde{B}F)x_t + \tilde{C}w_{t+1}$.

68.3.2 Mapping into the LQ Framework

To map into the LQ framework, we'll use

$$x_t := \begin{bmatrix} z_t \\ b_t \end{bmatrix} = \begin{bmatrix} 1 \\ y_t \\ y_{t-1} \\ b_t \end{bmatrix}$$

as the state vector and $u_t := c_t - \gamma$ as the control.

With this notation and $U_\gamma := [\gamma \quad 0 \quad 0]$, we can write the state dynamics as in (68.5) when

$$\tilde{A} := \begin{bmatrix} A & 0 \\ (1+r)(U_\gamma - U) & 1+r \end{bmatrix} \quad \tilde{B} := \begin{bmatrix} 0 \\ 1+r \end{bmatrix} \quad \text{and} \quad \tilde{C} := \begin{bmatrix} C \\ 0 \end{bmatrix} w_{t+1}$$

Please confirm for yourself that, with these definitions, the LQ dynamics (68.5) match the dynamics of z_t and b_t described above.

To map utility into the quadratic form $x_t' R x_t + u_t' Q u_t$ we can set

- $Q := 1$ (remember that we are minimizing) and
- $R :=$ a 4×4 matrix of zeros

However, there is one problem remaining.

We have no direct way to capture the non-recursive restriction (68.3) on the debt sequence $\{b_t\}$ from within the LQ framework.

To try to enforce it, we're going to use a trick: put a small penalty on b_t^2 in the criterion function.

In the present setting, this means adding a small entry $\epsilon > 0$ in the $(4, 4)$ position of R.

That will induce a (hopefully) small approximation error in the decision rule.

We'll check whether it really is small numerically soon.

68.4 Implementation

Let's write some code to solve the model.

One comment before we start is that the bliss level of consumption γ in the utility function has no effect on the optimal decision rule.

We saw this in the previous lecture *permanent income*.

The reason is that it drops out of the Euler equation for consumption.

In what follows we set it equal to unity.

68.4.1 The Exogenous Nonfinancial Income Process

First, we create the objects for the optimal linear regulator

```
# Set parameters
α, β, ρ1, ρ2, σ = 10.0, 0.95, 0.9, 0.0, 1.0

R = 1 / β
A = np.array([[1., 0., 0.],
              [α,  ρ1, ρ2],
              [0., 1., 0.]])
C = np.array([[0.], [σ], [0.]])
G = np.array([[0., 1., 0.]])

# Form LinearStateSpace system and pull off steady state moments
μ_z0 = np.array([[1.0], [0.0], [0.0]])
Σ_z0 = np.zeros((3, 3))
Lz = qe.LinearStateSpace(A, C, G, mu_0=μ_z0, Sigma_0=Σ_z0)
μ_z, μ_y, Σ_z, Σ_y, Σ_yx = Lz.stationary_distributions()
```

(continues on next page)

(continued from previous page)

```
# Mean vector of state for the savings problem
mxo = np.vstack([μ_z, 0.0])

# Create stationary covariance matrix of x -- start everyone off at b=0
a1 = np.zeros((3, 1))
aa = np.hstack([Σ_z, a1])
bb = np.zeros((1, 4))
sxo = np.vstack([aa, bb])

# These choices will initialize the state vector of an individual at zero
# debt and the ergodic distribution of the endowment process. Use these to
# create the Bewley economy.
mxbewley = mxo
sxbewley = sxo
```

The next step is to create the matrices for the LQ system

```
A12 = np.zeros((3,1))
ALQ_l = np.hstack([A, A12])
ALQ_r = np.array([[0, -R, 0, R]])
ALQ = np.vstack([ALQ_l, ALQ_r])

RLQ = np.array([[0., 0., 0., 0.],
                [0., 0., 0., 0.],
                [0., 0., 0., 0.],
                [0., 0., 0., 1e-9]])

QLQ = np.array([1.0])
BLQ = np.array([0., 0., 0., R]).reshape(4,1)
CLQ = np.array([0., σ, 0., 0.]).reshape(4,1)
β_LQ = β
```

Let's print these out and have a look at them

```
print(f"A = \n {ALQ}")
print(f"B = \n {BLQ}")
print(f"R = \n {RLQ}")
print(f"Q = \n {QLQ}")
```

```
A =
 [[ 1.          0.          0.          0.        ]
 [10.          0.9         0.          0.        ]
 [ 0.          1.          0.          0.        ]
 [ 0.         -1.05263158  0.          1.05263158]]
B =
 [[0.        ]
 [0.        ]
 [0.        ]
 [1.05263158]]
R =
 [[0.e+00 0.e+00 0.e+00 0.e+00]
 [0.e+00 0.e+00 0.e+00 0.e+00]
 [0.e+00 0.e+00 0.e+00 0.e+00]
 [0.e+00 0.e+00 0.e+00 1.e-09]]
Q =
 [1.]
```

Now create the appropriate instance of an LQ model

```
lqpi = qe.LQ(QLQ, RLQ, ALQ, BLQ, C=CLQ, beta=β_LQ)
```

We'll save the implied optimal policy function soon compare them with what we get by employing an alternative solution method

```
P, F, d = lqpi.stationary_values()  # Compute value function and decision rule
ABF = ALQ - BLQ @ F  #  Form closed loop system
```

68.4.2 Comparison with the Difference Equation Approach

In our *first lecture* on the infinite horizon permanent income problem we used a different solution method.

The method was based around

- deducing the Euler equations that are the first-order conditions with respect to consumption and savings.
- using the budget constraints and boundary condition to complete a system of expectational linear difference equations.
- solving those equations to obtain the solution.

Expressed in state space notation, the solution took the form

$$
\begin{aligned}
z_{t+1} &= Az_t + Cw_{t+1} \\
b_{t+1} &= b_t + U[(I - \beta A)^{-1}(A - I)]z_t \\
y_t &= Uz_t \\
c_t &= (1 - \beta)[U(I - \beta A)^{-1}z_t - b_t]
\end{aligned}
$$

Now we'll apply the formulas in this system

```
# Use the above formulas to create the optimal policies for b_{t+1} and c_t
b_pol = G @ la.inv(np.eye(3, 3) - β * A) @ (A - np.eye(3, 3))
c_pol = (1 - β) * G @ la.inv(np.eye(3, 3) - β * A)

# Create the A matrix for a LinearStateSpace instance
A_LSS1 = np.vstack([A, b_pol])
A_LSS2 = np.eye(4, 1, -3)
A_LSS = np.hstack([A_LSS1, A_LSS2])

# Create the C matrix for LSS methods
C_LSS = np.vstack([C, np.zeros(1)])

# Create the G matrix for LSS methods
G_LSS1 = np.vstack([G, c_pol])
G_LSS2 = np.vstack([np.zeros(1), -(1 - β)])
G_LSS = np.hstack([G_LSS1, G_LSS2])

# Use the following values to start everyone off at b=0, initial incomes zero
μ_0 = np.array([1., 0., 0., 0.])
Σ_0 = np.zeros((4, 4))
```

`A_LSS` calculated as we have here should equal `ABF` calculated above using the LQ model

```
ABF - A_LSS
```

```
array([[ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00],
       [-9.51248364e-06,  9.51247767e-08,  0.00000000e+00,
        -1.99999900e-08]])
```

Now compare pertinent elements of `c_pol` and `F`

```
print(c_pol, "\n", -F)
```

```
[[65.51724138  0.34482759  0.        ]]
 [[ 6.55172323e+01  3.44827677e-01 -0.00000000e+00 -5.00000190e-02]]
```

We have verified that the two methods give the same solution.

Now let's create instances of the LinearStateSpace class and use it to do some interesting experiments.

To do this, we'll use the outcomes from our second method.

68.5 Two Example Economies

In the spirit of Bewley models [Bewley, 1986], we'll generate panels of consumers.

The examples differ only in the initial states with which we endow the consumers.

All other parameter values are kept the same in the two examples

- In the first example, all consumers begin with zero nonfinancial income and zero debt.
  - The consumers are thus *ex-ante* identical.
- In the second example, while all begin with zero debt, we draw their initial income levels from the invariant distribution of financial income.
  - Consumers are *ex-ante* heterogeneous.

In the first example, consumers' nonfinancial income paths display pronounced transients early in the sample

- these will affect outcomes in striking ways

Those transient effects will not be present in the second example.

We use methods affiliated with the LinearStateSpace class to simulate the model.

68.5.1 First Set of Initial Conditions

We generate 25 paths of the exogenous non-financial income process and the associated optimal consumption and debt paths.

In the first set of graphs, darker lines depict a particular sample path, while the lighter lines describe 24 other paths.

A second graph plots a collection of simulations against the population distribution that we extract from the `LinearStateSpace` instance `LSS`.

Comparing sample paths with population distributions at each date t is a useful exercise—see *our discussion* of the laws of large numbers

```
lss = qe.LinearStateSpace(A_LSS, C_LSS, G_LSS, mu_0=μ_0, Sigma_0=Σ_0)
```

68.5.2 Population and Sample Panels

In the code below, we use the LinearStateSpace class to

- compute and plot population quantiles of the distributions of consumption and debt for a population of consumers.
- simulate a group of 25 consumers and plot sample paths on the same graph as the population distribution.

```
def income_consumption_debt_series(A, C, G, μ_0, Σ_0, T=150, npaths=25):
    """
    This function takes initial conditions (μ_0, Σ_0) and uses the
    LinearStateSpace class from QuantEcon to  simulate an economy
    npaths times for T periods. It then uses that information to
    generate some graphs related to the discussion below.
    """
    lss = qe.LinearStateSpace(A, C, G, mu_0=μ_0, Sigma_0=Σ_0)

    # Simulation/Moment Parameters
    moment_generator = lss.moment_sequence()

    # Simulate various paths
    bsim = np.empty((npaths, T))
    csim = np.empty((npaths, T))
    ysim = np.empty((npaths, T))

    for i in range(npaths):
        sims = lss.simulate(T)
        bsim[i, :] = sims[0][-1, :]
        csim[i, :] = sims[1][1, :]
        ysim[i, :] = sims[1][0, :]

    # Get the moments
    cons_mean = np.empty(T)
    cons_var = np.empty(T)
    debt_mean = np.empty(T)
    debt_var = np.empty(T)
    for t in range(T):
        μ_x, μ_y, Σ_x, Σ_y = next(moment_generator)
        cons_mean[t], cons_var[t] = μ_y[1,0], Σ_y[1, 1]
        debt_mean[t], debt_var[t] = μ_x[3,0], Σ_x[3, 3]

    return bsim, csim, ysim, cons_mean, cons_var, debt_mean, debt_var

def consumption_income_debt_figure(bsim, csim, ysim):

    # Get T
    T =  bsim.shape[1]

    # Create the first figure
    fig, ax = plt.subplots(2, 1, figsize=(10, 8))
    xvals = np.arange(T)

    # Plot consumption and income
    ax[0].plot(csim[0, :], label="c", color="b")
```

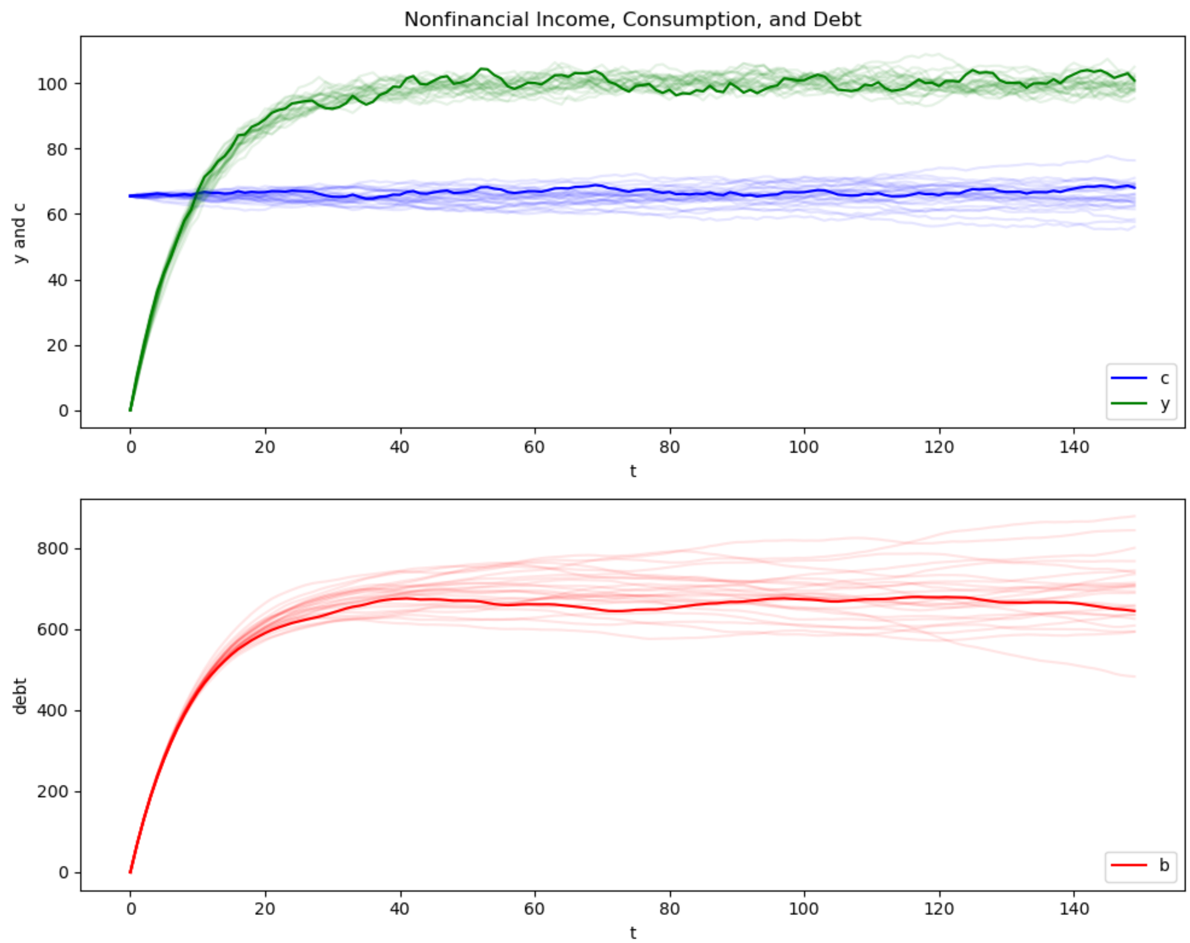
(continues on next page)

(continued from previous page)

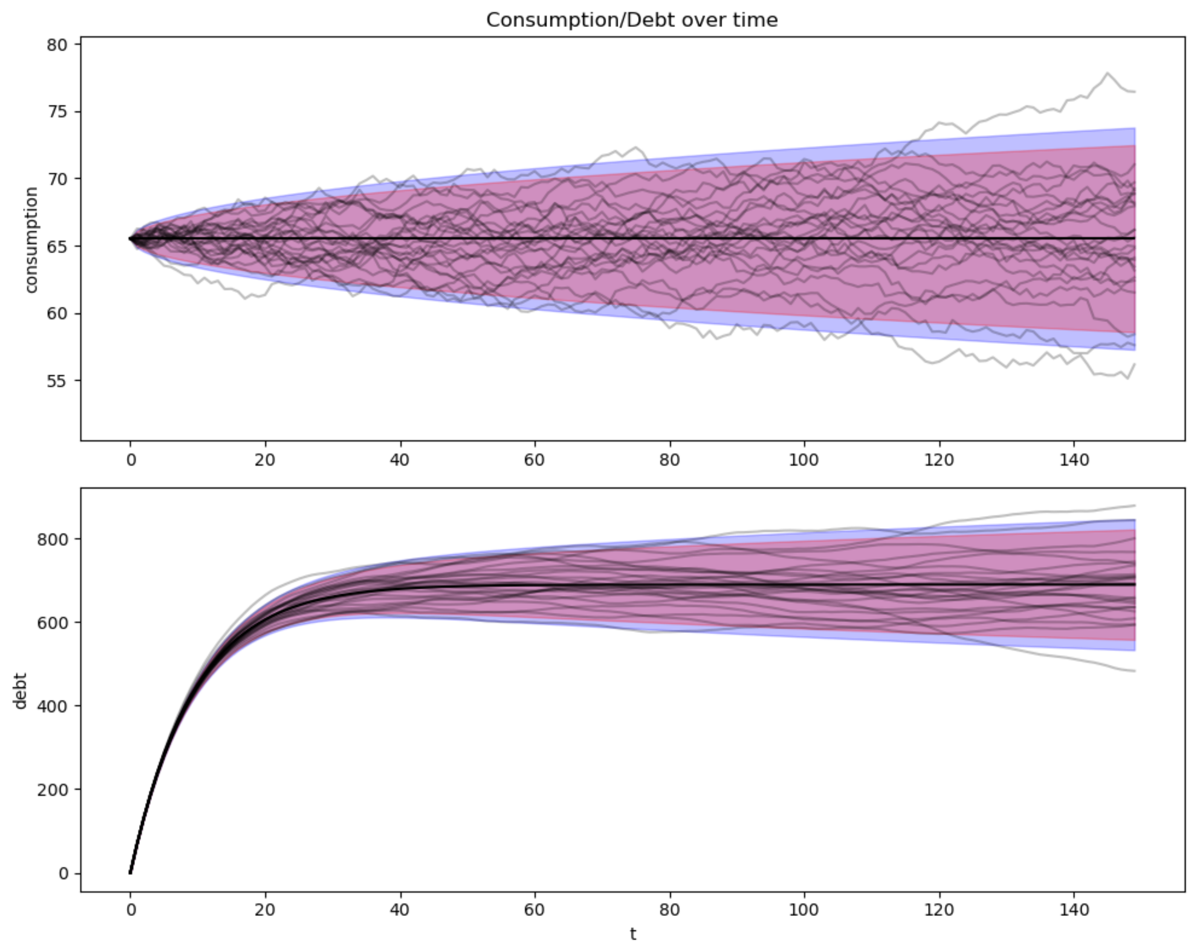
```
    ax[0].plot(ysim[0, :], label="y", color="g")
    ax[0].plot(csim.T, alpha=.1, color="b")
    ax[0].plot(ysim.T, alpha=.1, color="g")
    ax[0].legend(loc=4)
    ax[0].set(title="Nonfinancial Income, Consumption, and Debt",
              xlabel="t", ylabel="y and c")

    # Plot debt
    ax[1].plot(bsim[0, :], label="b", color="r")
    ax[1].plot(bsim.T, alpha=.1, color="r")
    ax[1].legend(loc=4)
    ax[1].set(xlabel="t", ylabel="debt")

    fig.tight_layout()
    return fig

def consumption_debt_fanchart(csim, cons_mean, cons_var,
                              bsim, debt_mean, debt_var):
    # Get T
    T =  bsim.shape[1]

    # Create percentiles of cross-section distributions
    cmean = np.mean(cons_mean)
    c90 = 1.65 * np.sqrt(cons_var)
    c95 = 1.96 * np.sqrt(cons_var)
    c_perc_95p, c_perc_95m = cons_mean + c95, cons_mean - c95
    c_perc_90p, c_perc_90m = cons_mean + c90, cons_mean - c90

    # Create percentiles of cross-section distributions
    dmean = np.mean(debt_mean)
    d90 = 1.65 * np.sqrt(debt_var)
    d95 = 1.96 * np.sqrt(debt_var)
    d_perc_95p, d_perc_95m = debt_mean + d95, debt_mean - d95
    d_perc_90p, d_perc_90m = debt_mean + d90, debt_mean - d90


    # Create second figure
    fig, ax = plt.subplots(2, 1, figsize=(10, 8))
    xvals = np.arange(T)

    # Consumption fan
    ax[0].plot(xvals, cons_mean, color="k")
    ax[0].plot(csim.T, color="k", alpha=.25)
    ax[0].fill_between(xvals, c_perc_95m, c_perc_95p, alpha=.25, color="b")
    ax[0].fill_between(xvals, c_perc_90m, c_perc_90p, alpha=.25, color="r")
    ax[0].set(title="Consumption/Debt over time",
              ylim=(cmean-15, cmean+15), ylabel="consumption")

    # Debt fan
    ax[1].plot(xvals, debt_mean, color="k")
    ax[1].plot(bsim.T, color="k", alpha=.25)
    ax[1].fill_between(xvals, d_perc_95m, d_perc_95p, alpha=.25, color="b")
    ax[1].fill_between(xvals, d_perc_90m, d_perc_90p, alpha=.25, color="r")
    ax[1].set(xlabel="t", ylabel="debt")

    fig.tight_layout()
    return fig
```

Now let's create figures with initial conditions of zero for y_0 and b_0

```
out = income_consumption_debt_series(A_LSS, C_LSS, G_LSS, μ_0, Σ_0)
bsim0, csim0, ysim0 = out[:3]
cons_mean0, cons_var0, debt_mean0, debt_var0 = out[3:]

consumption_income_debt_figure(bsim0, csim0, ysim0)

plt.show()
```




```
consumption_debt_fanchart(csim0, cons_mean0, cons_var0,
                          bsim0, debt_mean0, debt_var0)

plt.show()
```



Here is what is going on in the above graphs.

For our simulation, we have set initial conditions $b_0 = y_{-1} = y_{-2} = 0$.

Because $y_{-1} = y_{-2} = 0$, nonfinancial income y_t starts far below its stationary mean $\mu_{y,\infty}$ and rises early in each simulation.

Recall from the *previous lecture* that we can represent the optimal decision rule for consumption in terms of the **co-integrating relationship**

$$(1-\beta)b_t + c_t = (1-\beta)E_t \sum_{j=0}^{\infty} \beta^j y_{t+j} \tag{68.6}$$

So at time 0 we have

$$c_0 = (1-\beta)E_0 \sum_{t=0}^{\infty} \beta^j y_t$$

This tells us that consumption starts at the income that would be paid by an annuity whose value equals the expected discounted value of nonfinancial income at time $t = 0$.

To support that level of consumption, the consumer borrows a lot early and consequently builds up substantial debt.

In fact, he or she incurs so much debt that eventually, in the stochastic steady state, he consumes less each period than his nonfinancial income.

He uses the gap between consumption and nonfinancial income mostly to service the interest payments due on his debt.

Thus, when we look at the panel of debt in the accompanying graph, we see that this is a group of *ex-ante* identical people each of whom starts with zero debt.

All of them accumulate debt in anticipation of rising nonfinancial income.

They expect their nonfinancial income to rise toward the invariant distribution of income, a consequence of our having started them at $y_{-1} = y_{-2} = 0$.

Cointegration Residual

The following figure plots realizations of the left side of (68.6), which, *as discussed in our last lecture*, is called the **cointegrating residual**.

As mentioned above, the right side can be thought of as an annuity payment on the expected present value of future income $E_t \sum_{j=0}^{\infty} \beta^j y_{t+j}$.

Early along a realization, c_t is approximately constant while $(1-\beta)b_t$ and $(1-\beta)E_t \sum_{j=0}^{\infty} \beta^j y_{t+j}$ both rise markedly as the household's present value of income and borrowing rise pretty much together.

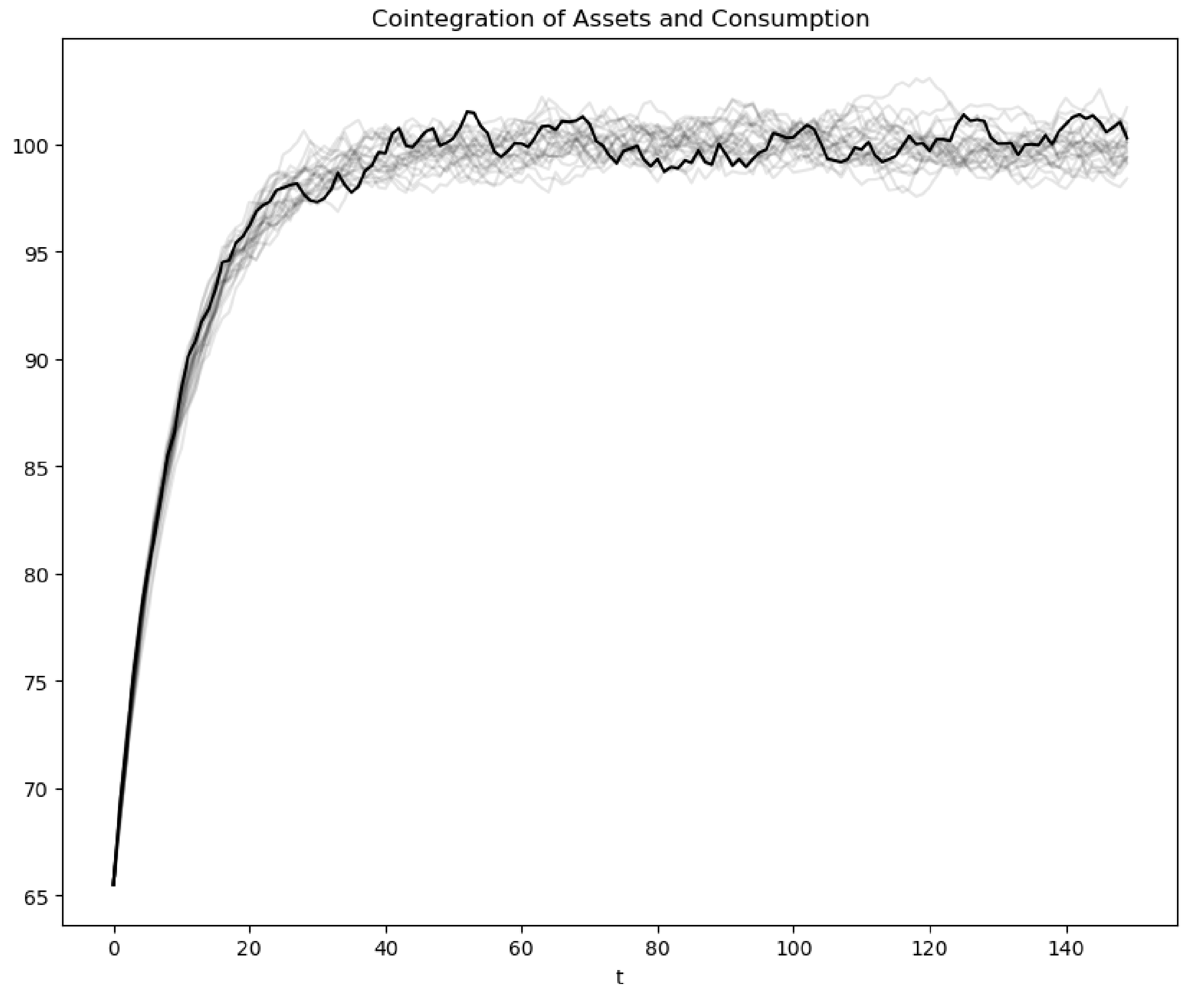
This example illustrates the following point: the definition of cointegration implies that the cointegrating residual is *asymptotically* covariance stationary, not *covariance stationary*.

The cointegrating residual for the specification with zero income and zero debt initially has a notable transient component that dominates its behavior early in the sample.

By altering initial conditions, we shall remove this transient in our second example to be presented below

```
def cointegration_figure(bsim, csim):
    """
    Plots the cointegration
    """
    # Create figure
    fig, ax = plt.subplots(figsize=(10, 8))
    ax.plot((1 - β) * bsim[0, :] + csim[0, :], color="k")
    ax.plot((1 - β) * bsim.T + csim.T, color="k", alpha=.1)

    ax.set(title="Cointegration of Assets and Consumption", xlabel="t")

    return fig
```

```
cointegration_figure(bsim0, csim0)
plt.show()
```



68.5.3 A "Borrowers and Lenders" Closed Economy

When we set $y_{-1} = y_{-2} = 0$ and $b_0 = 0$ in the preceding exercise, we make debt "head north" early in the sample.

Average debt in the cross-section rises and approaches the asymptote.

We can regard these as outcomes of a "small open economy" that borrows from abroad at the fixed gross interest rate $R = r + 1$ in anticipation of rising incomes.

So with the economic primitives set as above, the economy converges to a steady state in which there is an excess aggregate supply of risk-free loans at a gross interest rate of R.

This excess supply is filled by "foreigner lenders" willing to make those loans.

We can use virtually the same code to rig a "poor man's Bewley [Bewley, 1986] model" in the following way

- as before, we start everyone at $b_0 = 0$.
- But instead of starting everyone at $y_{-1} = y_{-2} = 0$, we draw $\begin{bmatrix} y_{-1} \\ y_{-2} \end{bmatrix}$ from the invariant distribution of the $\{y_t\}$ process.

This rigs a closed economy in which people are borrowing and lending with each other at a gross risk-free interest rate of $R = \beta^{-1}$.

Across the group of people being analyzed, risk-free loans are in zero excess supply.

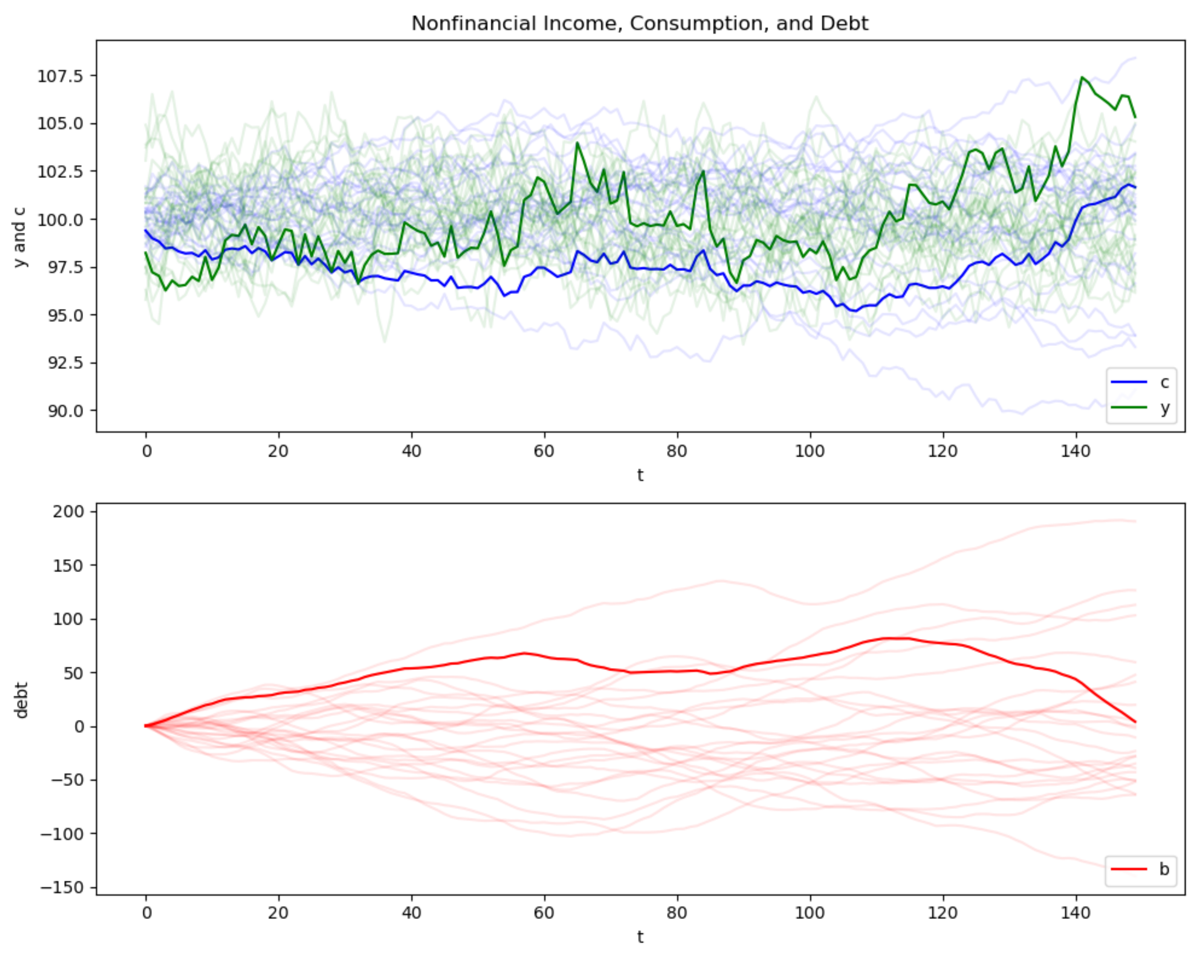
We have arranged primitives so that $R = \beta^{-1}$ clears the market for risk-free loans at zero aggregate excess supply.

So the risk-free loans are being made from one person to another within our closed set of agents.

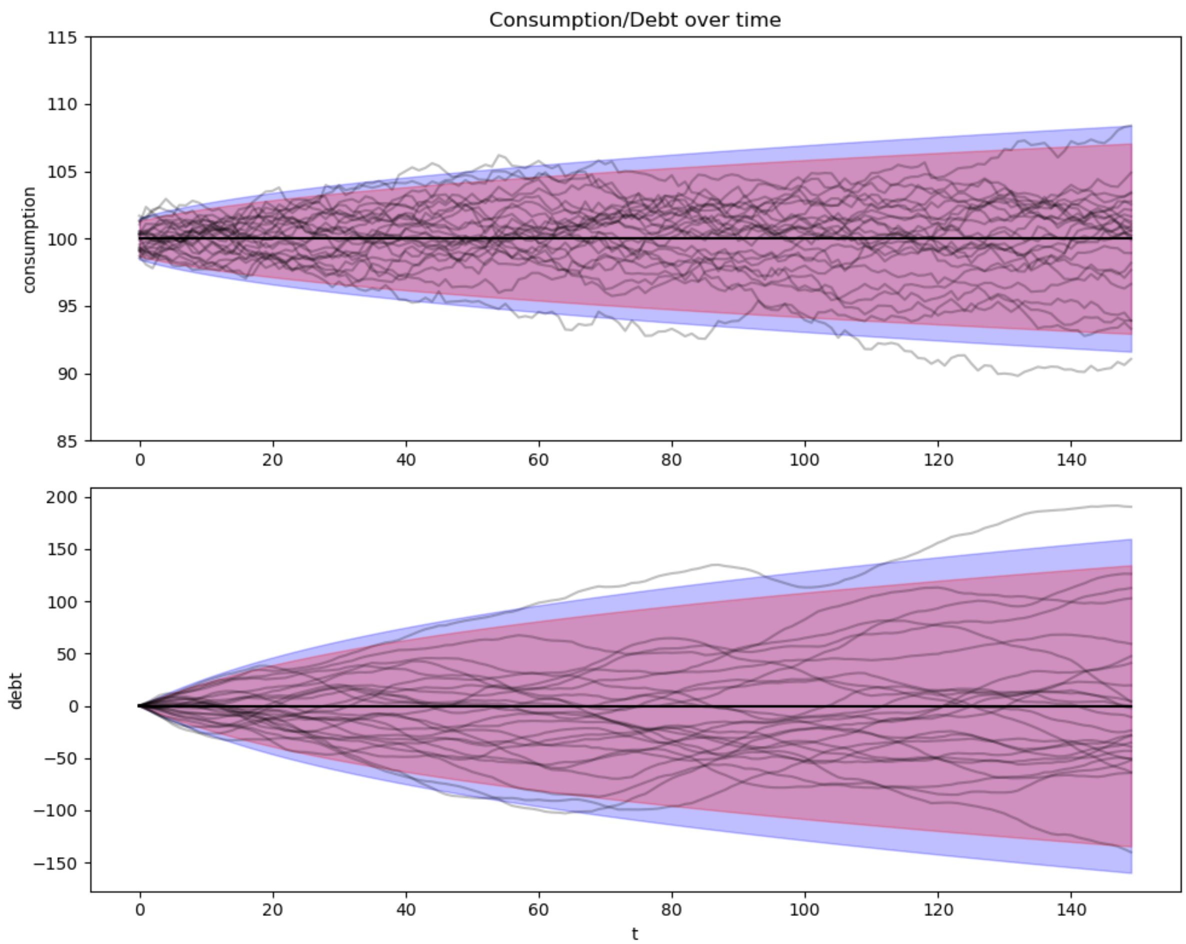
There is no need for foreigners to lend to our group.

Let's have a look at the corresponding figures

```
out = income_consumption_debt_series(A_LSS, C_LSS, G_LSS, mxbewley, sxbewley)
bsimb, csimb, ysimb = out[:3]
cons_meanb, cons_varb, debt_meanb, debt_varb = out[3:]

consumption_income_debt_figure(bsimb, csimb, ysimb)

plt.show()
```




```
consumption_debt_fanchart(csimb, cons_meanb, cons_varb,
                          bsimb, debt_meanb, debt_varb)

plt.show()
```



The graphs confirm the following outcomes:

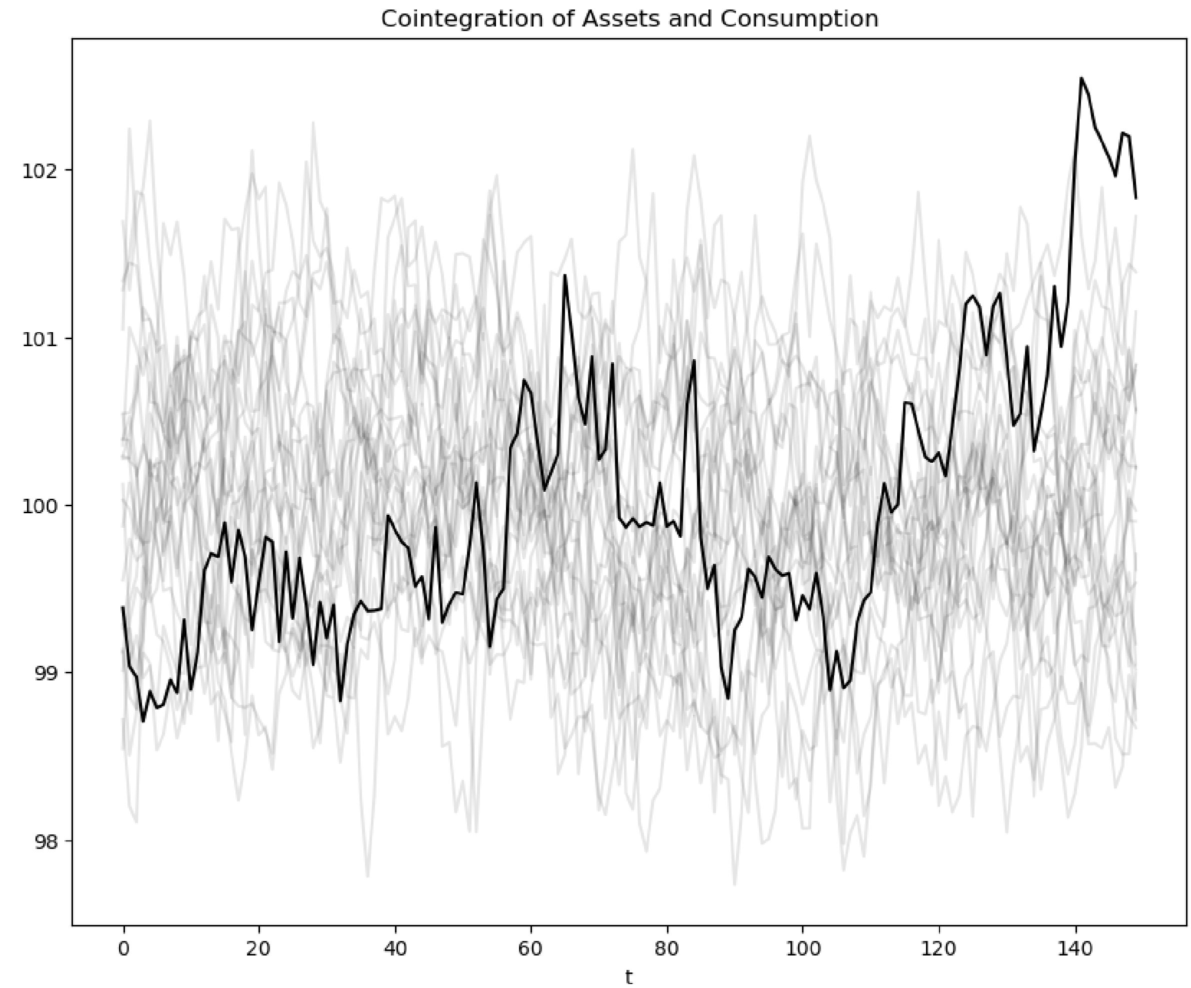
- As before, the consumption distribution spreads out over time.

But now there is some initial dispersion because there is *ex-ante* heterogeneity in the initial draws of $\begin{bmatrix} y_{-1} \\ y_{-2} \end{bmatrix}$.

- As before, the cross-section distribution of debt spreads out over time.
- Unlike before, the average level of debt stays at zero, confirming that this is a closed borrower-and-lender economy.
- Now the cointegrating residual seems stationary, and not just asymptotically stationary.

Let's have a look at the cointegration figure

```
cointegration_figure(bsimb, csimb)
plt.show()
```


Cointegration of Assets and Consumption
102
101
100
99
98
0
20
40
60
80
100
120
140
t

CHAPTER

SIXTYNINE

PRODUCTION SMOOTHING VIA INVENTORIES

Contents

In addition to what's in Anaconda, this lecture employs the following library:

```
!pip install quantecon
```

69.1 Overview

This lecture can be viewed as an application of this *quantecon lecture* about linear quadratic control theory.

It formulates a discounted dynamic program for a firm that chooses a production schedule to balance

- minimizing costs of production across time, against
- keeping costs of holding inventories low

In the tradition of a classic book by Holt, Modigliani, Muth, and Simon [Holt *et al.*, 1960], we simplify the firm's problem by formulating it as a linear quadratic discounted dynamic programming problem of the type studied in this *quantecon lecture*.

Because its costs of production are increasing and quadratic in production, the firm holds inventories as a buffer stock in order to smooth production across time, provided that holding inventories is not too costly.

But the firm also wants to make its sales out of existing inventories, a preference that we represent by a cost that is quadratic in the difference between sales in a period and the firm's beginning of period inventories.

We compute examples designed to indicate how the firm optimally smooths production while keeping inventories close to sales.

To introduce components of the model, let

- S_t be sales at time t
- Q_t be production at time t
- I_t be inventories at the beginning of time t
- $\beta \in (0, 1)$ be a discount factor
- $c(Q_t) = c_1 Q_t + c_2 Q_t^2$, be a cost of production function, where $c_1 > 0, c_2 > 0$, be an inventory cost function
- $d(I_t, S_t) = d_1 I_t + d_2 (S_t - I_t)^2$, where $d_1 > 0, d_2 > 0$, be a cost-of-holding-inventories function, consisting of two components:
  - a cost $d_1 I_t$ of carrying inventories, and
  - a cost $d_2 (S_t - I_t)^2$ of having inventories deviate from sales
- $p_t = a_0 - a_1 S_t + v_t$ be an inverse demand function for a firm's product, where $a_0 > 0, a_1 > 0$ and v_t is a demand shock at time t
- $\pi\_t = p_t S_t - c(Q_t) - d(I_t, S_t)$ be the firm's profits at time t
- $\sum_{t=0}^{\infty} \beta^t \pi_t$ be the present value of the firm's profits at time 0
- $I_{t+1} = I_t + Q_t - S_t$ be the law of motion of inventories
- $z_{t+1} = A_{22} z_t + C_2 \epsilon_{t+1}$ be a law of motion for an exogenous state vector z_t that contains time t information useful for predicting the demand shock v_t
- $v_t = G z_t$ link the demand shock to the information set z_t
- the constant 1 be the first component of z_t

To map our problem into a linear-quadratic discounted dynamic programming problem (also known as an optimal linear regulator), we define the **state** vector at time t as

$$x_t = \begin{bmatrix} I_t \\ z_t \end{bmatrix}$$

and the **control** vector as

$$u_t = \begin{bmatrix} Q_t \\ S_t \end{bmatrix}$$

The law of motion for the state vector x_t is evidently

$$\begin{bmatrix} I_{t+1} \\ z_t \end{bmatrix} = \left[\begin{array}{cc} 1 & 0 \\ 0 & A_{22} \end{array} \right] \begin{bmatrix} I_t \\ z_t \end{bmatrix} + \begin{bmatrix} 1 & -1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} Q_t \\ S_t \end{bmatrix} + \begin{bmatrix} 0 \\ C_2 \end{bmatrix} \epsilon_{t+1}$$

or

$$x_{t+1} = A x_t + B u_t + C \epsilon_{t+1}$$

(At this point, we ask that you please forgive us for using Q_t to be the firm's production at time t, while below we use Q as the matrix in the quadratic form $u_t' Q u_t$ that appears in the firm's one-period profit function)

We can express the firm's profit as a function of states and controls as

$$\pi_t = -(x_t' R x_t + u_t' Q u_t + 2 u_t' N x_t)$$

To form the matrices R, Q, N in an LQ dynamic programming problem, we note that the firm's profits at time t function can be expressed

$$
\begin{aligned}
\pi_t =& p_t S_t - c\left(Q_t\right) - d\left(I_t, S_t\right) \\
=& \left(a_0 - a_1 S_t + v_t\right) S_t - c_1 Q_t - c_2 Q_t^2 - d_1 I_t - d_2 \left(S_t - I_t\right)^2 \\
=& a_0 S_t - a_1 S_t^2 + G z_t S_t - c_1 Q_t - c_2 Q_t^2 - d_1 I_t - d_2 S_t^2 - d_2 I_t^2 + 2 d_2 S_t I_t \\
=& -\left(\underbrace{d_1 I_t + d_2 I_t^2}_{x_t' R x_t} \underbrace{+ a_1 S_t^2 + d_2 S_t^2 + c_2 Q_t^2}_{u_t' Q u_t} \underbrace{- a_0 S_t - G z_t S_t + c_1 Q_t - 2 d_2 S_t I_t}_{2 u_t' N x_t}\right) \\
=& -\left(\left[\begin{array}{cc} I_t & z_t' \end{array}\right] \underbrace{\left[\begin{array}{cc} d_2 & \frac{d_1}{2} S_c \\ \frac{d_1}{2} S_c' & 0 \end{array}\right]}_{\equiv R} \left[\begin{array}{c} I_t \\ z_t \end{array}\right] + \left[\begin{array}{cc} Q_t & S_t \end{array}\right] \underbrace{\left[\begin{array}{cc} c_2 & 0 \\ 0 & a_1 + d_2 \end{array}\right]}_{\equiv Q} \left[\begin{array}{c} Q_t \\ S_t \end{array}\right] + 2 \left[\begin{array}{cc} Q_t & S_t \end{array}\right] \underbrace{\left[\begin{array}{cc} 0 & \frac{c_1}{2} S_c \\ -d_2 & -\frac{a_0}{2} S_c - \frac{G}{2} \end{array}\right]}_{\equiv N} \left[\begin{array}{c} I \\ z \end{array}\right.
\end{aligned}
$$

where $S_c = [1, 0]$.

Remark on notation: The notation for cross product term in the QuantEcon library is N.

The firms' optimum decision rule takes the form

$$
u_t = -F x_t
$$

and the evolution of the state under the optimal decision rule is

$$
x_{t+1} = (A - BF) x_t + C \epsilon_{t+1}
$$

The firm chooses a decision rule for u_t that maximizes

$$
E_0 \sum_{t=0}^{\infty} \beta^t \pi_t
$$

subject to a given x_0.

This is a stochastic discounted LQ dynamic program.

Here is code for computing an optimal decision rule and for analyzing its consequences.

```
import matplotlib.pyplot as plt
import numpy as np
import quantecon as qe
```

```
class SmoothingExample:
    """
    Class for constructing, solving, and plotting results for
    inventories and sales smoothing problem.
    """

    def __init__(self,
                 β=0.96,             # Discount factor
                 c1=1,               # Cost-of-production
                 c2=1,
                 d1=1,               # Cost-of-holding inventories
                 d2=1,
                 a0=10,              # Inverse demand function
                 a1=1,
                 A22=[[1,    0],     # z process
```

(continues on next page)

(continued from previous page)

```
                 [1, 0.9]],
             C2=[[0], [1]],
             G=[0, 1]):

        self.β = β
        self.c1, self.c2 = c1, c2
        self.d1, self.d2 = d1, d2
        self.a0, self.a1 = a0, a1
        self.A22 = np.atleast_2d(A22)
        self.C2 = np.atleast_2d(C2)
        self.G = np.atleast_2d(G)

        # Dimensions
        k, j = self.C2.shape        # Dimensions for randomness part
        n = k + 1                   # Number of states
        m = 2                       # Number of controls

        Sc = np.zeros(k)
        Sc[0] = 1

        # Construct matrices of transition law
        A = np.zeros((n, n))
        A[0, 0] = 1
        A[1:, 1:] = self.A22

        B = np.zeros((n, m))
        B[0, :] = 1, -1

        C = np.zeros((n, j))
        C[1:, :] = self.C2

        self.A, self.B, self.C = A, B, C

        # Construct matrices of one period profit function
        R = np.zeros((n, n))
        R[0, 0] = d2
        R[1:, 0] = d1 / 2 * Sc
        R[0, 1:] = d1 / 2 * Sc

        Q = np.zeros((m, m))
        Q[0, 0] = c2
        Q[1, 1] = a1 + d2

        N = np.zeros((m, n))
        N[1, 0] = - d2
        N[0, 1:] = c1 / 2 * Sc
        N[1, 1:] = - a0 / 2 * Sc - self.G / 2

        self.R, self.Q, self.N = R, Q, N

        # Construct LQ instance
        self.LQ = qe.LQ(Q, R, A, B, C, N, beta=β)
        self.LQ.stationary_values()

    def simulate(self, x0, T=100):

        c1, c2 = self.c1, self.c2
```

(continues on next page)

(continued from previous page)

```
        d1, d2 = self.d1, self.d2
        a0, a1 = self.a0, self.a1
        G = self.G

        x_path, u_path, w_path = self.LQ.compute_sequence(x0, ts_length=T)

        I_path = x_path[0, :-1]
        z_path = x_path[1:, :-1]
        𝜈_path = (G @ z_path)[0, :]

        Q_path = u_path[0, :]
        S_path = u_path[1, :]

        revenue = (a0 - a1 * S_path + 𝜈_path) * S_path
        cost_production = c1 * Q_path + c2 * Q_path ** 2
        cost_inventories = d1 * I_path + d2 * (S_path - I_path) ** 2

        Q_no_inventory = (a0 + 𝜈_path - c1) / (2 * (a1 + c2))
        Q_hardwired = (a0 + 𝜈_path - c1) / (2 * (a1 + c2 + d2))

        fig, ax = plt.subplots(2, 2, figsize=(15, 10))

        ax[0, 0].plot(range(T), I_path, label="inventories")
        ax[0, 0].plot(range(T), S_path, label="sales")
        ax[0, 0].plot(range(T), Q_path, label="production")
        ax[0, 0].legend(loc=1)
        ax[0, 0].set_title("inventories, sales, and production")

        ax[0, 1].plot(range(T), (Q_path - S_path), color='b')
        ax[0, 1].set_ylabel("change in inventories", color='b')
        span = max(abs(Q_path - S_path))
        ax[0, 1].set_ylim(0-span*1.1, 0+span*1.1)
        ax[0, 1].set_title("demand shock and change in inventories")

        ax1_ = ax[0, 1].twinx()
        ax1_.plot(range(T), 𝜈_path, color='r')
        ax1_.set_ylabel("demand shock", color='r')
        span = max(abs(𝜈_path))
        ax1_.set_ylim(0-span*1.1, 0+span*1.1)

        ax1_.plot([0, T], [0, 0], '--', color='k')

        ax[1, 0].plot(range(T), revenue, label="revenue")
        ax[1, 0].plot(range(T), cost_production, label="cost_production")
        ax[1, 0].plot(range(T), cost_inventories, label="cost_inventories")
        ax[1, 0].legend(loc=1)
        ax[1, 0].set_title("profits decomposition")

        ax[1, 1].plot(range(T), Q_path, label="production")
        ax[1, 1].plot(range(T), Q_hardwired, label='production when  $I_t$ \
            forced to be zero')
        ax[1, 1].plot(range(T), Q_no_inventory, label='production when \
            inventories not useful')
        ax[1, 1].legend(loc=1)
        ax[1, 1].set_title('three production concepts')

        plt.show()
```

Notice that the above code sets parameters at the following default values

- discount factor $\beta = 0.96$,
- inverse demand function: $a0 = 10, a1 = 1$
- cost of production $c1 = 1, c2 = 1$
- costs of holding inventories $d1 = 1, d2 = 1$

In the examples below, we alter some or all of these parameter values.

69.2 Example 1

In this example, the demand shock follows AR(1) process:

$$\nu_t = \alpha + \rho \nu_{t-1} + \epsilon_t,$$

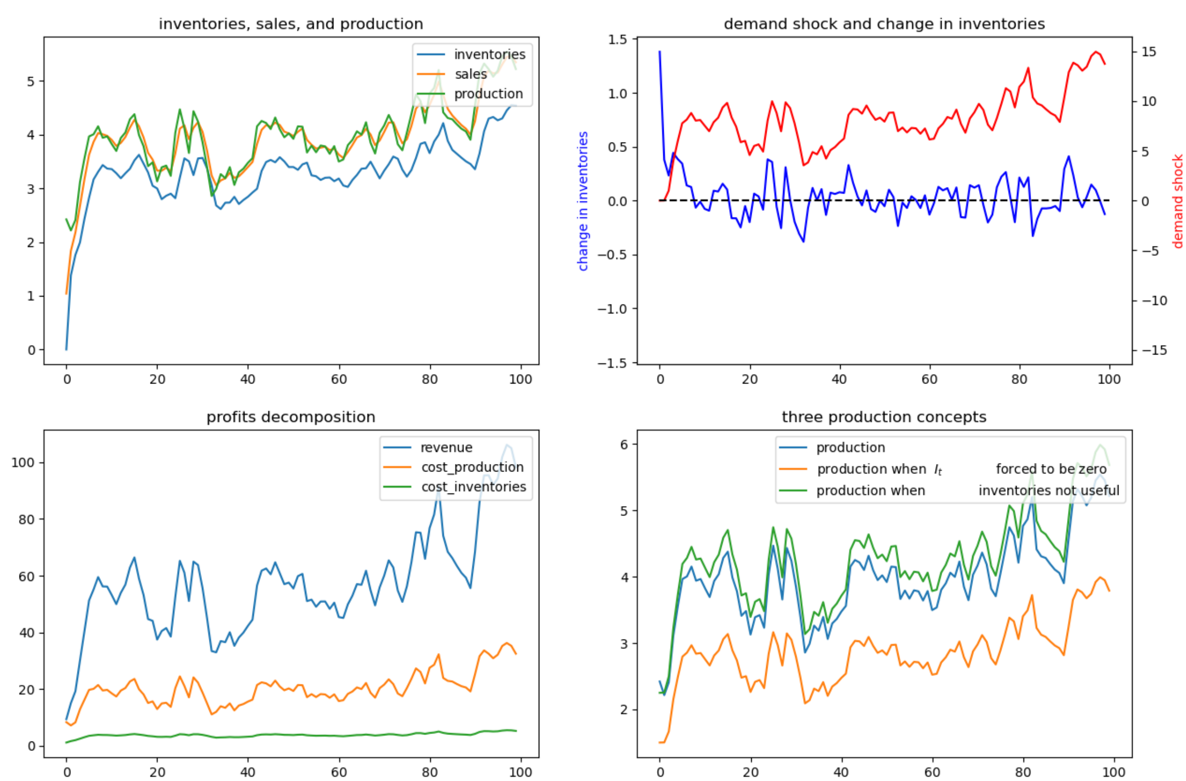
which implies

$$z_{t+1} = \begin{bmatrix} 1 \\ v_{t+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ \alpha & \rho \end{bmatrix} \underbrace{\begin{bmatrix} 1 \\ v_t \end{bmatrix}}_{z_t} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} \epsilon_{t+1}.$$

We set $\alpha = 1$ and $\rho = 0.9$, their default values.

We'll calculate and display outcomes, then discuss them below the pertinent figures.

```
ex1 = SmoothingExample()

x0 = [0, 1, 0]
ex1.simulate(x0)
```



The figures above illustrate various features of an optimal production plan.

Starting from zero inventories, the firm builds up a stock of inventories and uses them to smooth costly production in the face of demand shocks.

Optimal decisions evidently respond to demand shocks.

Inventories are always less than sales, so some sales come from current production, a consequence of the cost, $d_1 I_t$ of holding inventories.

The lower right panel shows differences between optimal production and two alternative production concepts that come from altering the firm's cost structure – i.e., its technology.

These two concepts correspond to these distinct altered firm problems.

- a setting in which inventories are not needed
- a setting in which they are needed but we arbitrarily prevent the firm from holding inventories by forcing it to set $I_t = 0$ always

We use these two alternative production concepts in order to shed light on the baseline model.

69.3 Inventories Not Useful

Let's turn first to the setting in which inventories aren't needed.

In this problem, the firm forms an output plan that maximizes the expected value of

$$\sum_{t=0}^{\infty} \beta^t \{p_t Q_t - C(Q_t)\}$$

It turns out that the optimal plan for Q_t for this problem also solves a sequence of static problems $\max_{Q_t} \{p_t Q_t - c(Q_t)\}$.

When inventories aren't required or used, sales always equal production.

This simplifies the problem and the optimal no-inventory production maximizes the expected value of

$$\sum_{t=0}^{\infty} \beta^t \left\{p_t Q_t - C\left(Q_t\right)\right\}.$$

The optimum decision rule is

$$Q_t^{ni} = \frac{a_0 + \nu_t - c_1}{c_2 + a_1}.$$

69.4 Inventories Useful but are Hardwired to be Zero Always

Next, we turn to a distinct problem in which inventories are useful – meaning that there are costs of $d_2(I_t - S_t)^2$ associated with having sales not equal to inventories – but we arbitrarily impose on the firm the costly restriction that it never hold inventories.

Here the firm's maximization problem is

$$\max_{\{I_t, Q_t, S_t\}} \sum_{t=0}^{\infty} \beta^t \left\{p_t S_t - C\left(Q_t\right) - d\left(I_t, S_t\right)\right\}$$

subject to the restrictions that $I_t = 0$ for all t and that $I_{t+1} = I_t + Q_t - S_t$.

The restriction that $I_t = 0$ implies that $Q_t = S_t$ and that the maximization problem reduces to

$$\max_{Q_t} \sum_{t=0}^{\infty} \beta^t \left\{ p_t Q_t - C\left(Q_t\right) - d\left(0, Q_t\right) \right\}$$

Here the optimal production plan is

$$Q_t^h = \frac{a_0 + \nu_t - c_1}{c_2 + a_1 + d_2}.$$

We introduce this I_t **is hardwired to zero** specification in order to shed light on the role that inventories play by comparing outcomes with those under our two other versions of the problem.

The bottom right panel displays a production path for the original problem that we are interested in (the blue line) as well with an optimal production path for the model in which inventories are not useful (the green path) and also for the model in which, although inventories are useful, they are hardwired to zero and the firm pays cost $d(0, Q_t)$ for not setting sales $S_t = Q_t$ equal to zero (the orange line).

Notice that it is typically optimal for the firm to produce more when inventories aren't useful. Here there is no requirement to sell out of inventories and no costs from having sales deviate from inventories.

But "typical" does not mean "always".

Thus, if we look closely, we notice that for small t, the green "production when inventories aren't useful" line in the lower right panel is below optimal production in the original model.

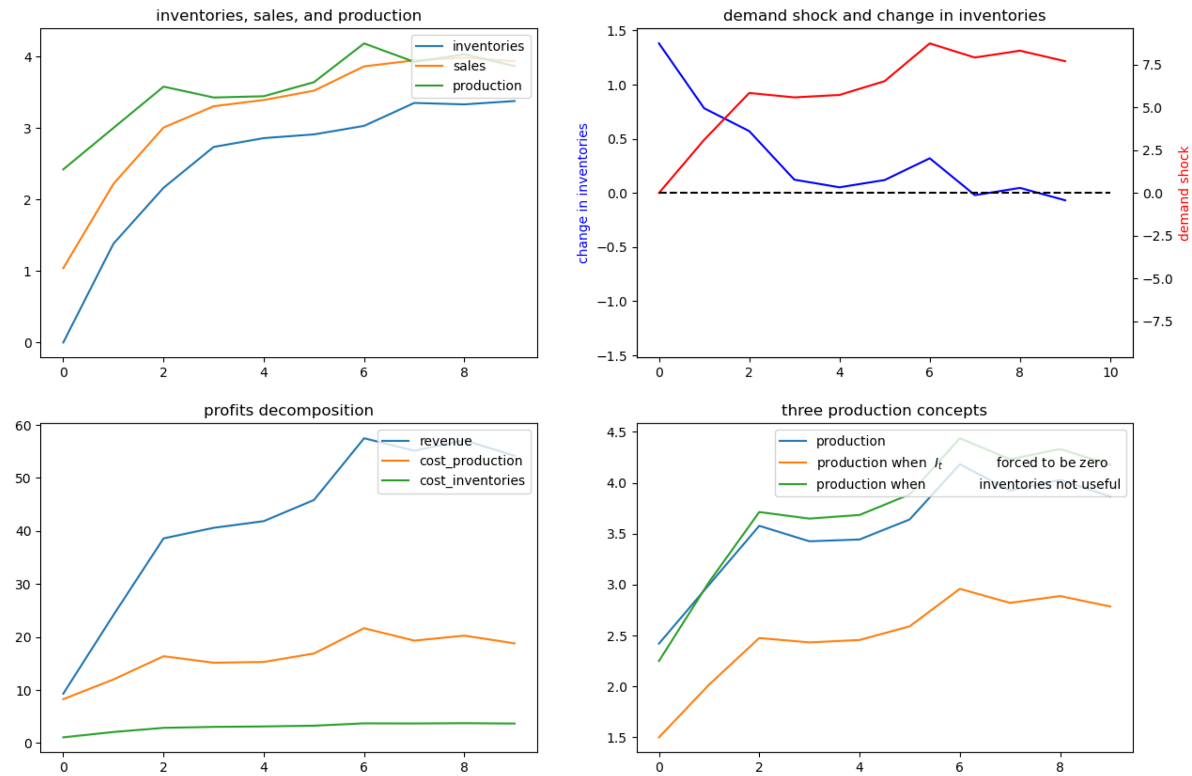
High optimal production in the original model early on occurs because the firm wants to accumulate inventories quickly in order to acquire high inventories for use in later periods.

But how the green line compares to the blue line early on depends on the evolution of the demand shock, as we will see in a deterministically seasonal demand shock example to be analyzed below.

In that example, the original firm optimally accumulates inventories slowly because the next positive demand shock is in the distant future.

To make the green-blue model production comparison easier to see, let's confine the graphs to the first 10 periods:

```
ex1.simulate(x0, T=10)
```



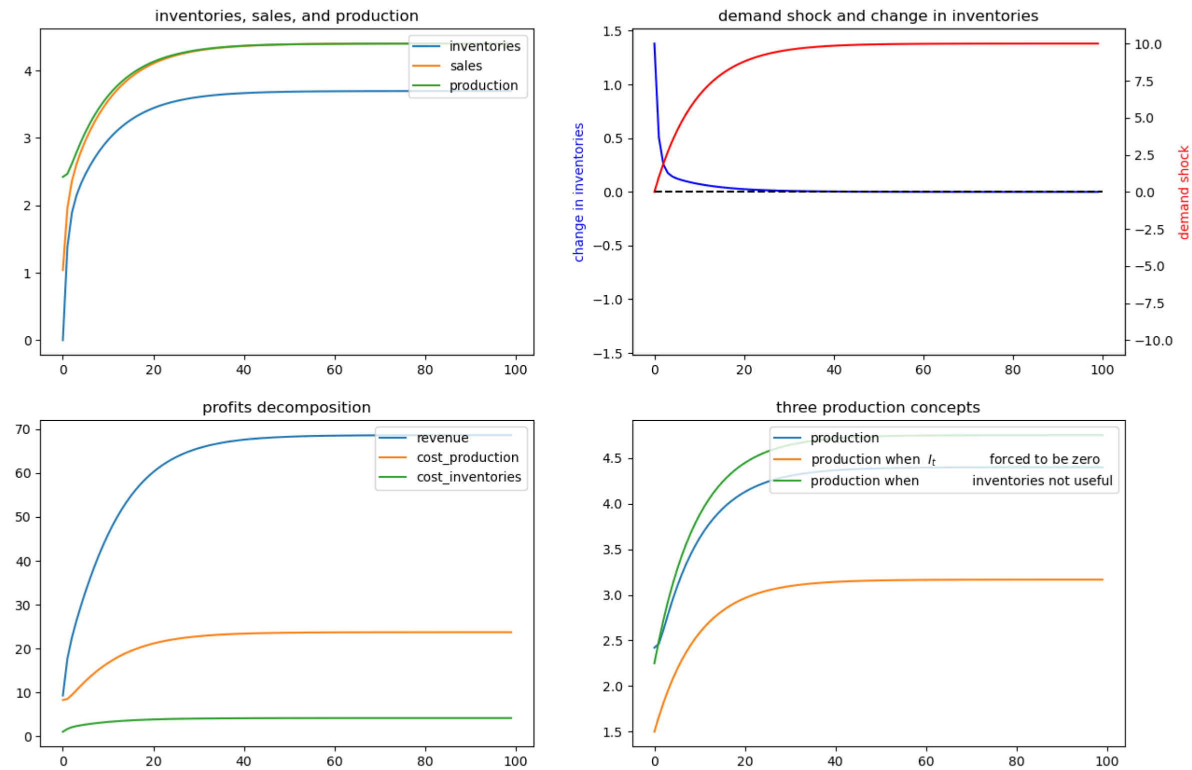
69.5 Example 2

Next, we shut down randomness in demand and assume that the demand shock ν_t follows a deterministic path:

$$\nu_t = \alpha + \rho \nu_{t-1}$$

Again, we'll compute and display outcomes in some figures

```
ex2 = SmoothingExample(C2=[[0], [0]])

x0 = [0, 1, 0]
ex2.simulate(x0)
```



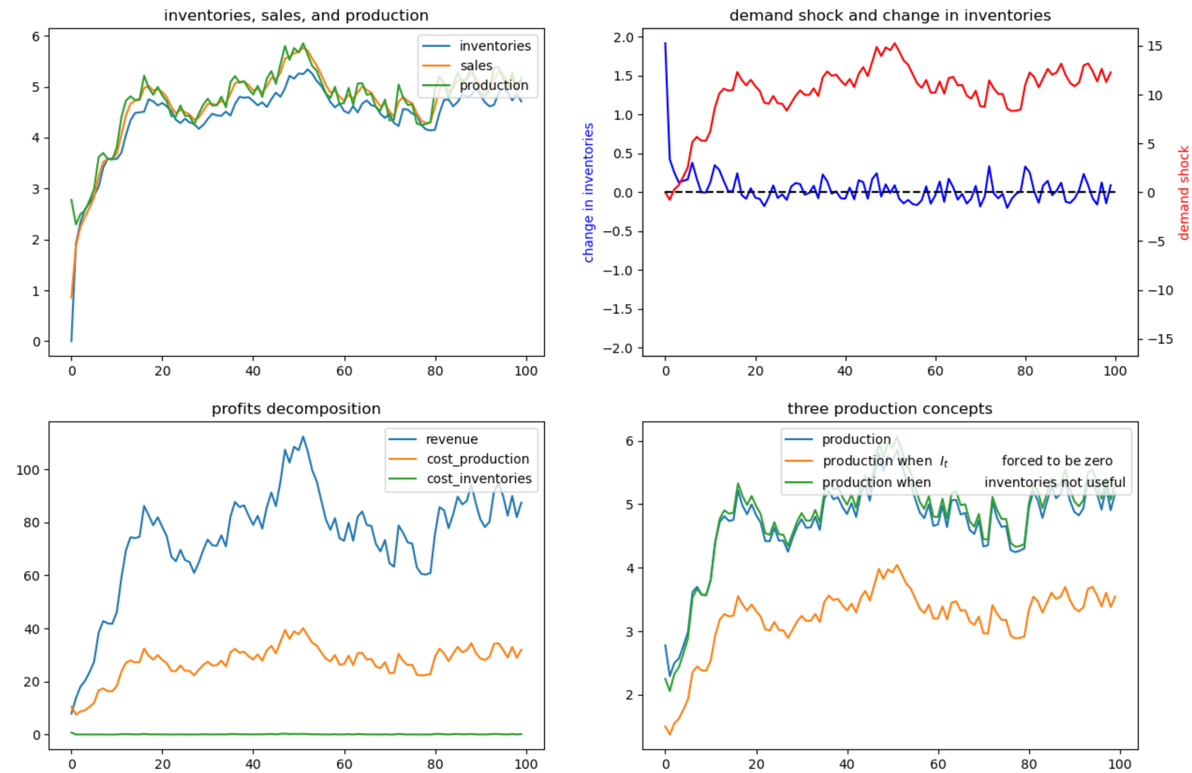
69.6 Example 3

Now we'll put randomness back into the demand shock process and also assume that there are zero costs of holding inventories.

In particular, we'll look at a situation in which $d_1 = 0$ but $d_2 > 0$.

Now it becomes optimal to set sales approximately equal to inventories and to use inventories to smooth production quite well, as the following figures confirm

```
ex3 = SmoothingExample(d1=0)

x0 = [0, 1, 0]
ex3.simulate(x0)
```



69.7 Example 4

To bring out some features of the optimal policy that are related to some technical issues in linear control theory, we'll now temporarily assume that it is costless to hold inventories.

When we completely shut down the cost of holding inventories by setting $d_1 = 0$ and $d_2 = 0$, something absurd happens (because the Bellman equation is opportunistic and very smart).

(Technically, we have set parameters that end up violating conditions needed to assure **stability** of the optimally controlled state.)

The firm finds it optimal to set $Q_t \equiv Q^* = \frac{-c_1}{2c_2}$, an output level that sets the costs of production to zero (when $c_1 > 0$, as it is with our default settings, then it is optimal to set production negative, whatever that means!).

Recall the law of motion for inventories

$$I_{t+1} = I_t + Q_t - S_t$$

So when $d_1 = d_2 = 0$ so that the firm finds it optimal to set $Q_t = \frac{-c_1}{2c_2}$ for all t, then

$$I_{t+1} - I_t = \frac{-c_1}{2c_2} - S_t < 0$$

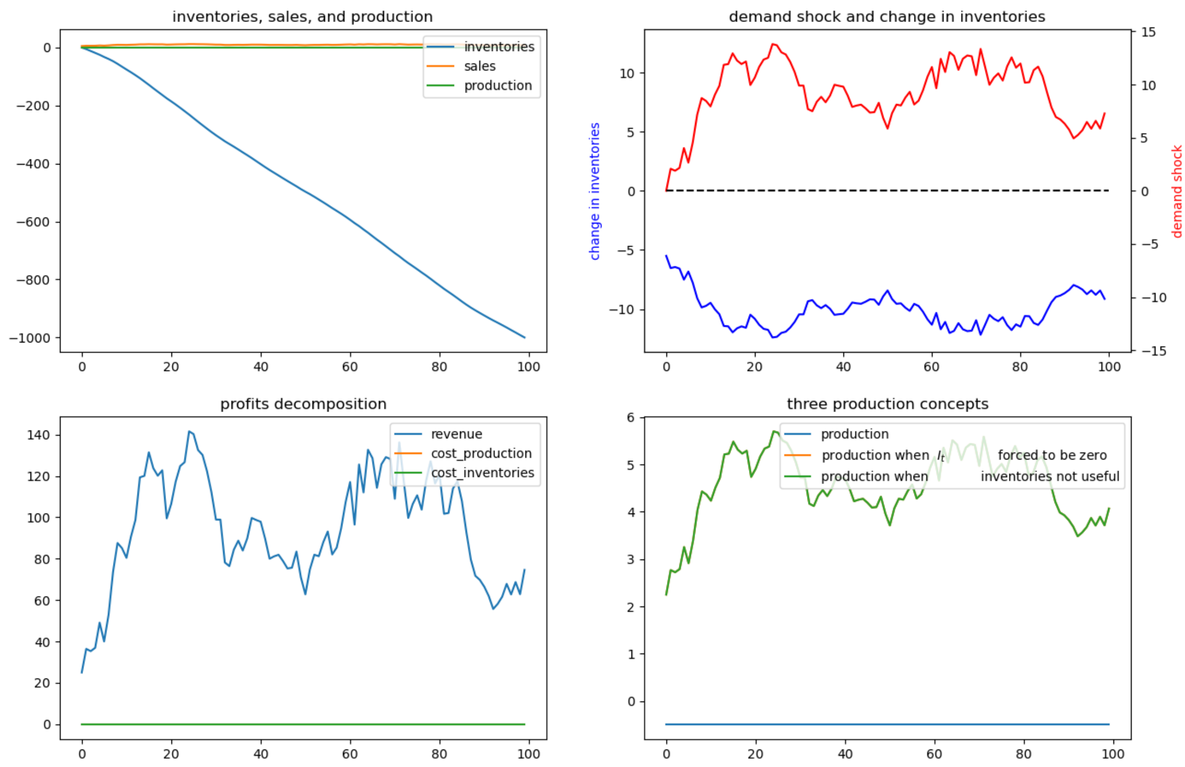
for almost all values of S_t under our default parameters that keep demand positive almost all of the time.

The dynamic program instructs the firm to set production costs to zero and to **run a Ponzi scheme** by running inventories down forever.

(We can interpret this as the firm somehow **going short in** or **borrowing** inventories)

The following figures confirm that inventories head south without limit

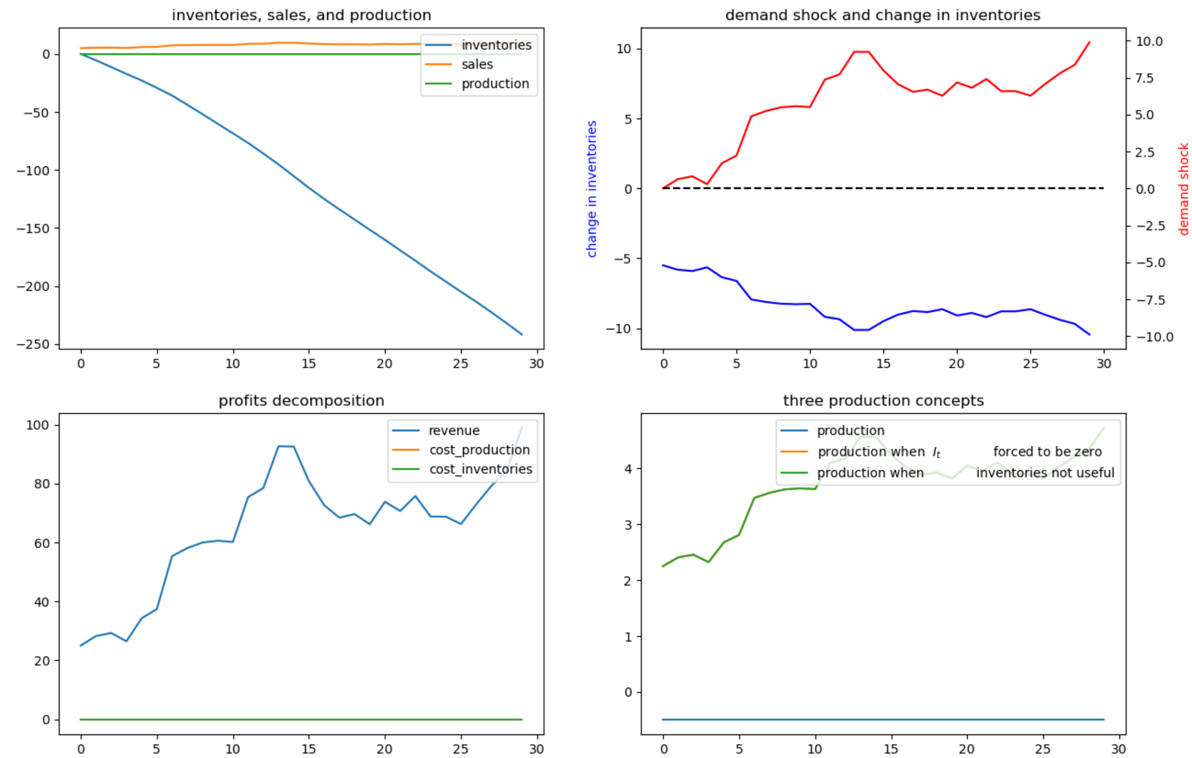
```
ex4 = SmoothingExample(d1=0, d2=0)

x0 = [0, 1, 0]
ex4.simulate(x0)
```



Let's shorten the time span displayed in order to highlight what is going on.

We'll set the horizon $T = 30$ with the following code

```
# shorter period
ex4.simulate(x0, T=30)
```



69.8 Example 5

Now we'll assume that the demand shock that follows a linear time trend

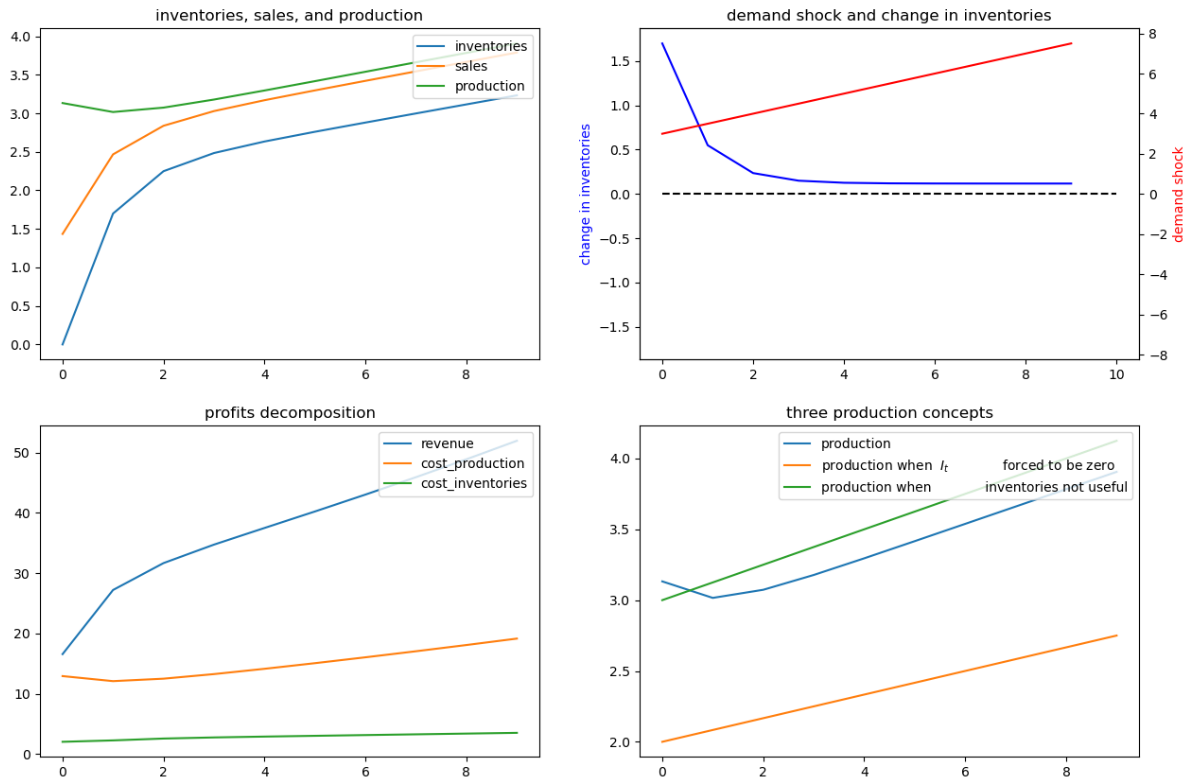
$$v_t = b + at, a > 0, b > 0$$

To represent this, we set $C_2 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$ and

$$A_{22} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}, x_0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, G = \begin{bmatrix} b & a \end{bmatrix}$$

```
# Set parameters
a = 0.5
b = 3.
```

```
ex5 = SmoothingExample(A22=[[1, 0], [1, 1]], C2=[[0], [0]], G=[b, a])

x0 = [0, 1, 0] # set the initial inventory as 0
ex5.simulate(x0, T=10)
```



69.9 Example 6

Now we'll assume a deterministically seasonal demand shock.

To represent this we'll set

$$A_{22} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}, C_2 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, G' = \begin{bmatrix} b \\ a \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

where $a > 0, b > 0$ and

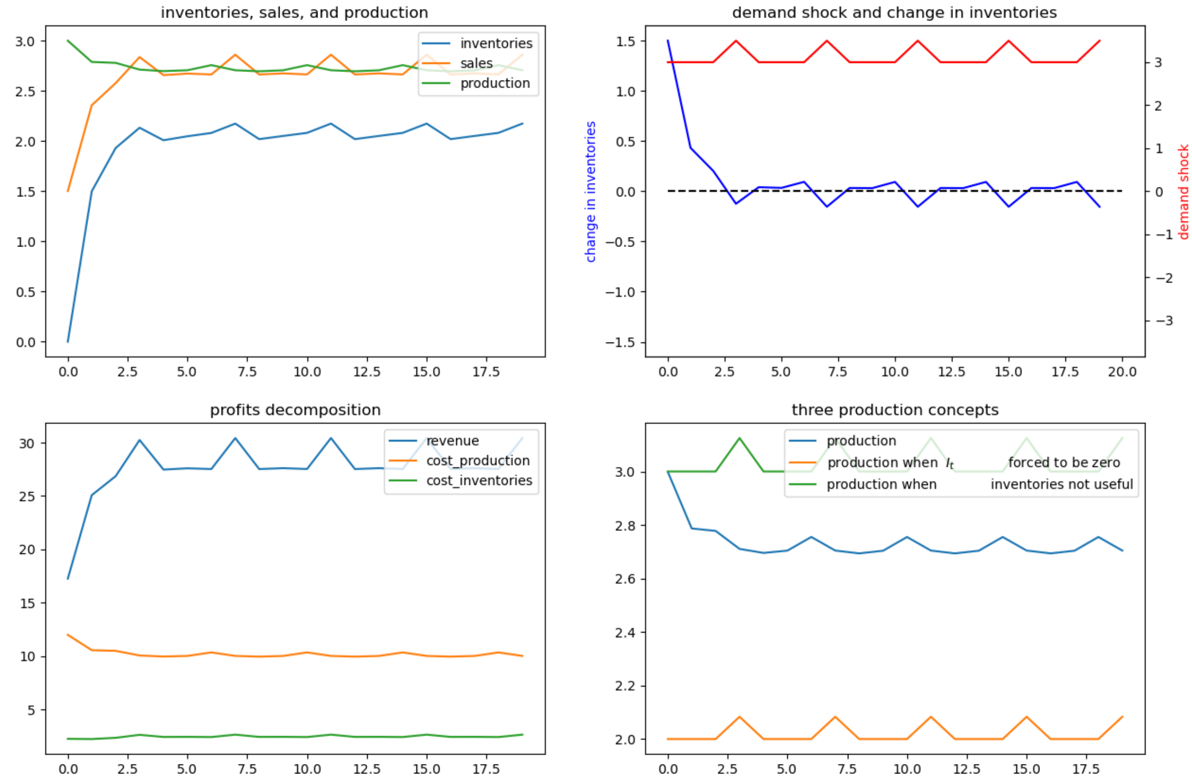
$$x_0 = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

```
ex6 = SmoothingExample(A22=[[1, 0, 0, 0, 0],
                            [0, 0, 0, 0, 1],
                            [0, 1, 0, 0, 0],
                            [0, 0, 1, 0, 0],
                            [0, 0, 0, 1, 0]],
                       C2=[[0], [0], [0], [0], [0]],
                       G=[b, a, 0, 0, 0])
```

(continues on next page)

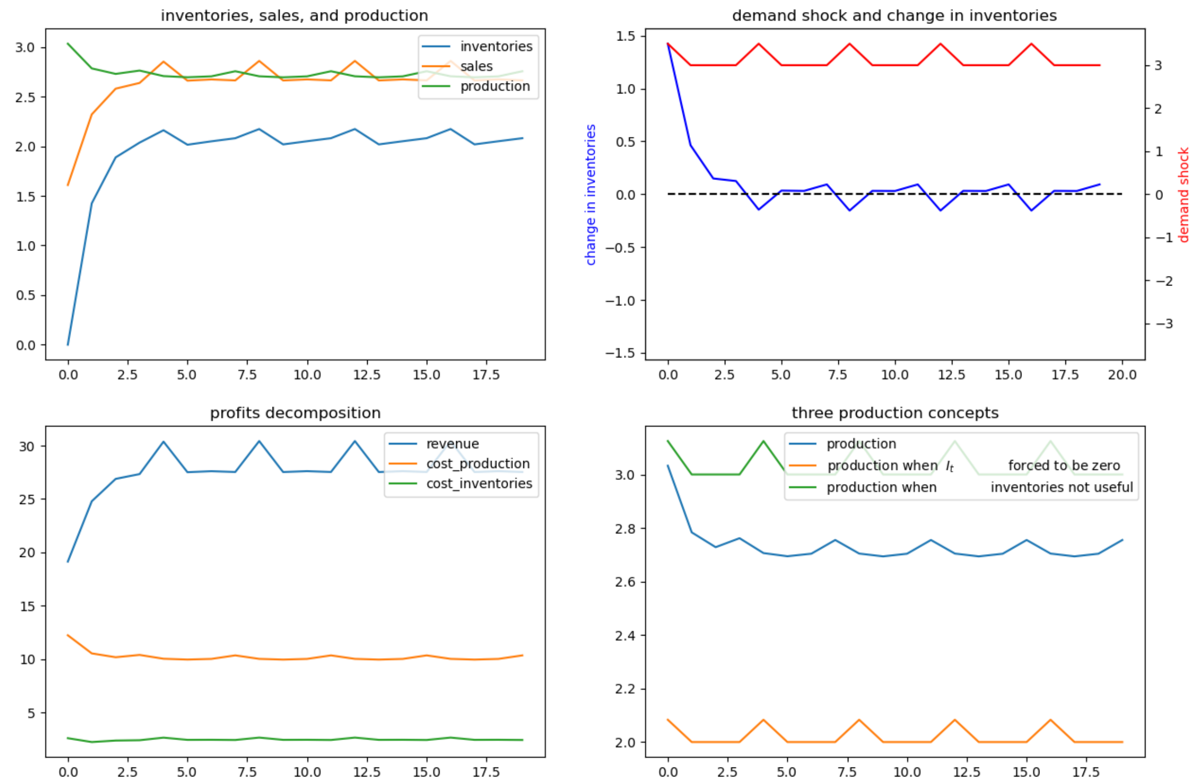
(continued from previous page)

```
x00 = [0, 1, 0, 1, 0, 0] # Set the initial inventory as 0
ex6.simulate(x00, T=20)
```

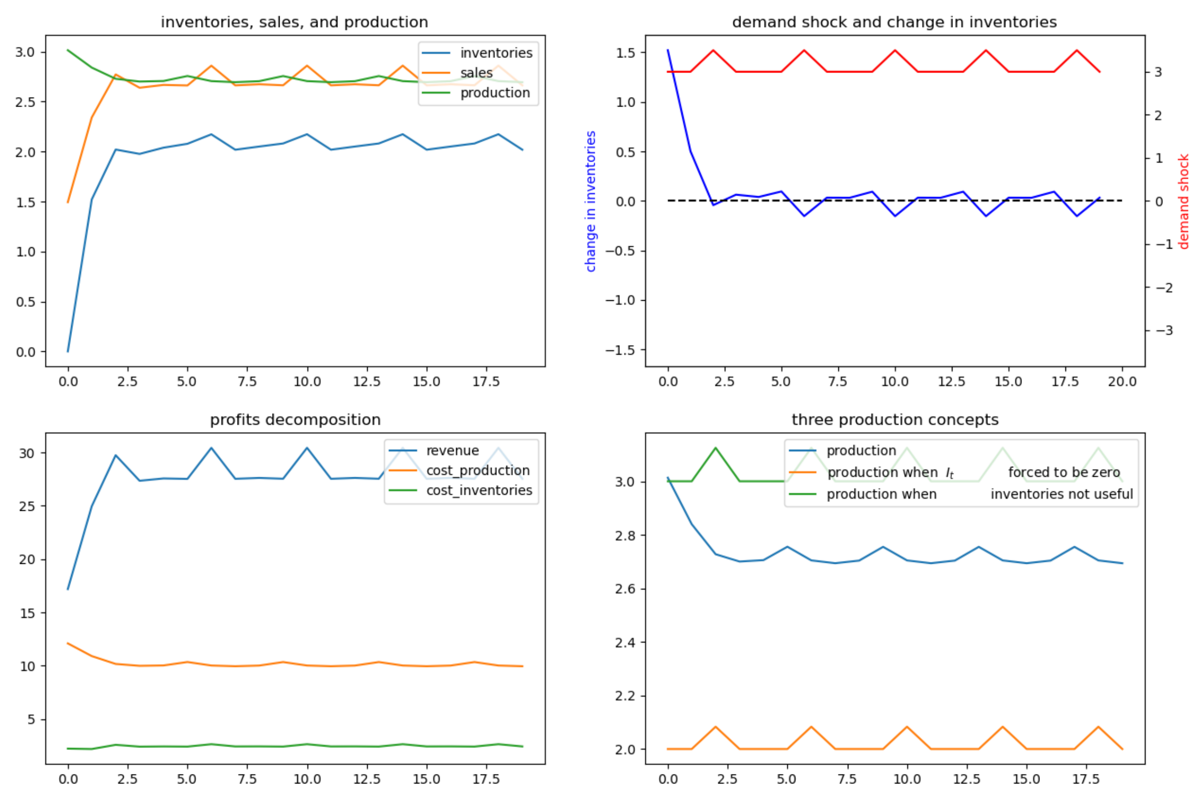


Now we'll generate some more examples that differ simply from the initial **season** of the year in which we begin the demand shock

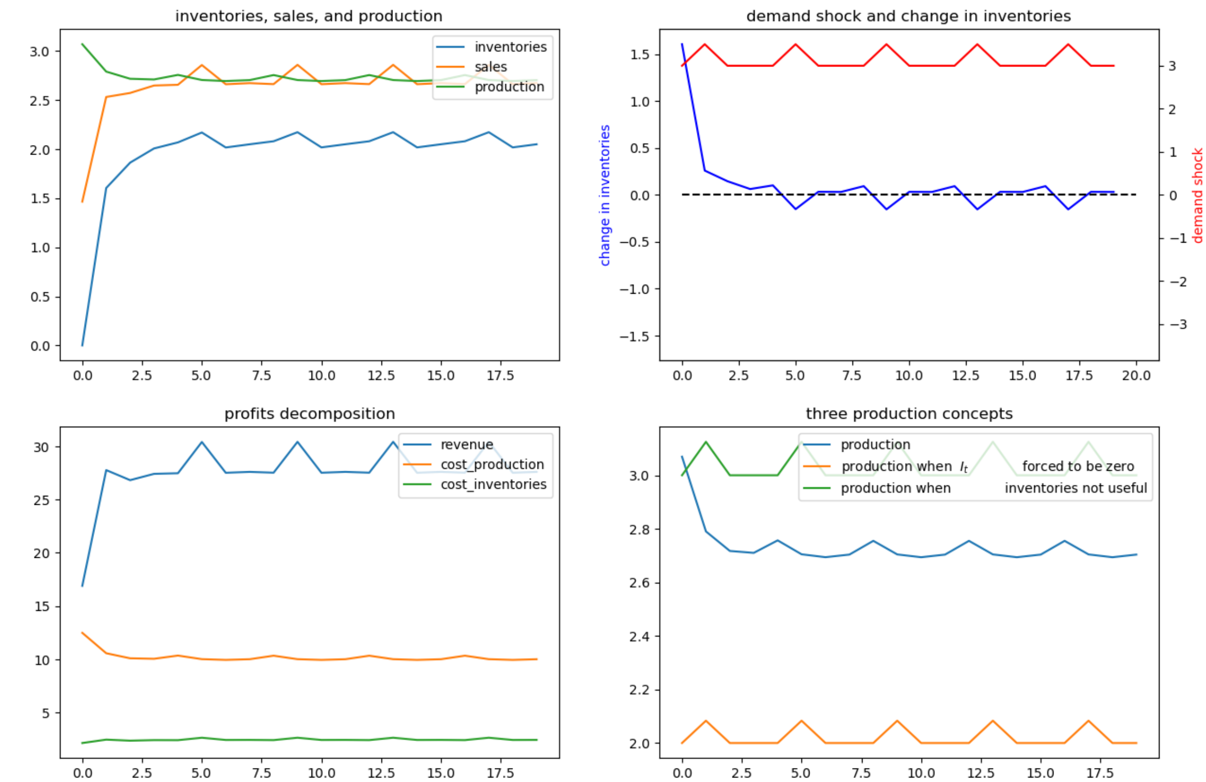
```
x01 = [0, 1, 1, 0, 0, 0]
ex6.simulate(x01, T=20)
```




```
x02 = [0, 1, 0, 0, 1, 0]
ex6.simulate(x02, T=20)
```



```
x03 = [0, 1, 0, 0, 0, 1]
ex6.simulate(x03, T=20)
```



69.10 Exercises

Please try to analyze some inventory sales smoothing problems using the `SmoothingExample` class.

Exercise 69.10.1

Assume that the demand shock follows AR(2) process below:

$$\nu_t = \alpha + \rho_1 \nu_{t-1} + \rho_2 \nu_{t-2} + \epsilon_t.$$

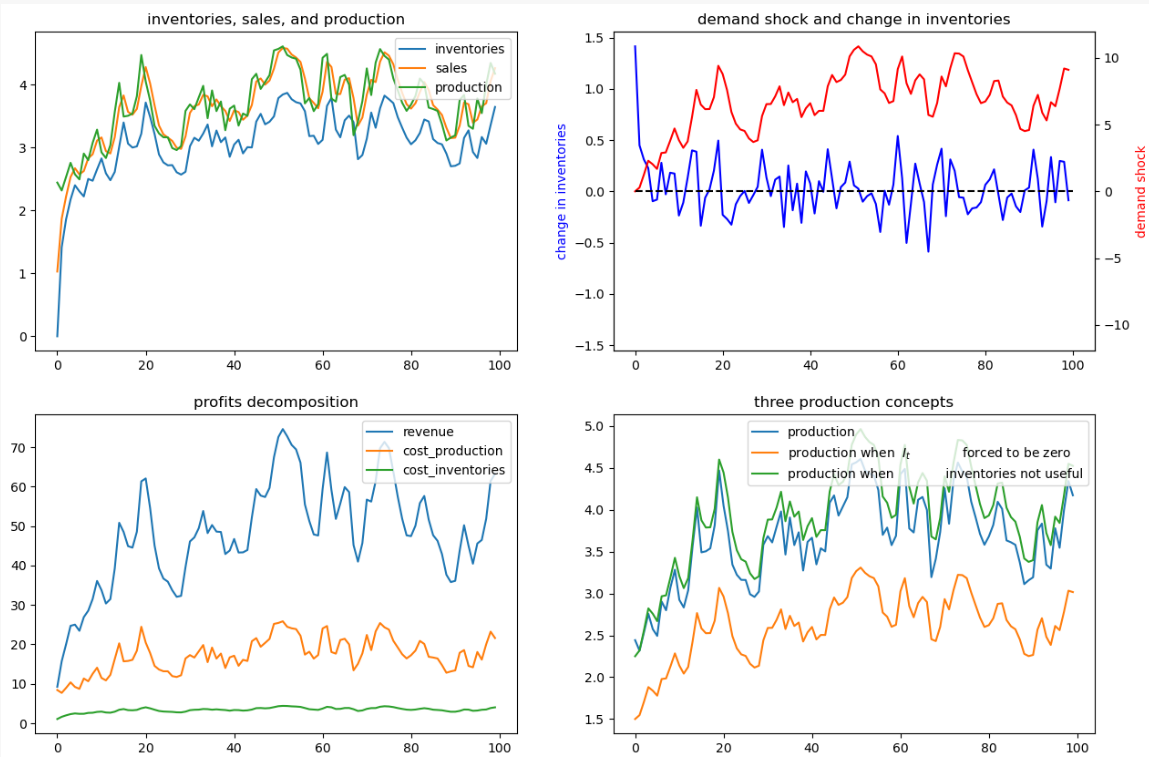
where $\alpha = 1$, $\rho_1 = 1.2$, and $\rho_2 = -0.3$. You need to construct $A22$, C, and G matrices properly and then to input them as the keyword arguments of `SmoothingExample` class. Simulate paths starting from the initial condition $x_0 = [0, 1, 0, 0]'$.

After this, try to construct a very similar `SmoothingExample` with the same demand shock process but exclude the randomness ϵ_t. Compute the stationary states $\bar{x}$ by simulating for a long period. Then try to add shocks with different magnitude to $\bar{\nu}_t$ and simulate paths. You should see how firms respond differently by staring at the production plans.

Solution

```
# set parameters
α = 1
```

```
ρ1 = 1.2
ρ2 = -.3
# construct matrices
A22 =[[1,  0,  0],
          [1, ρ1, ρ2],
          [0,  1, 0]]
C2 = [[0], [1], [0]]
G = [0, 1, 0]
ex1 = SmoothingExample(A22=A22, C2=C2, G=G)

x0 = [0, 1, 0, 0] # initial condition
ex1.simulate(x0)
```

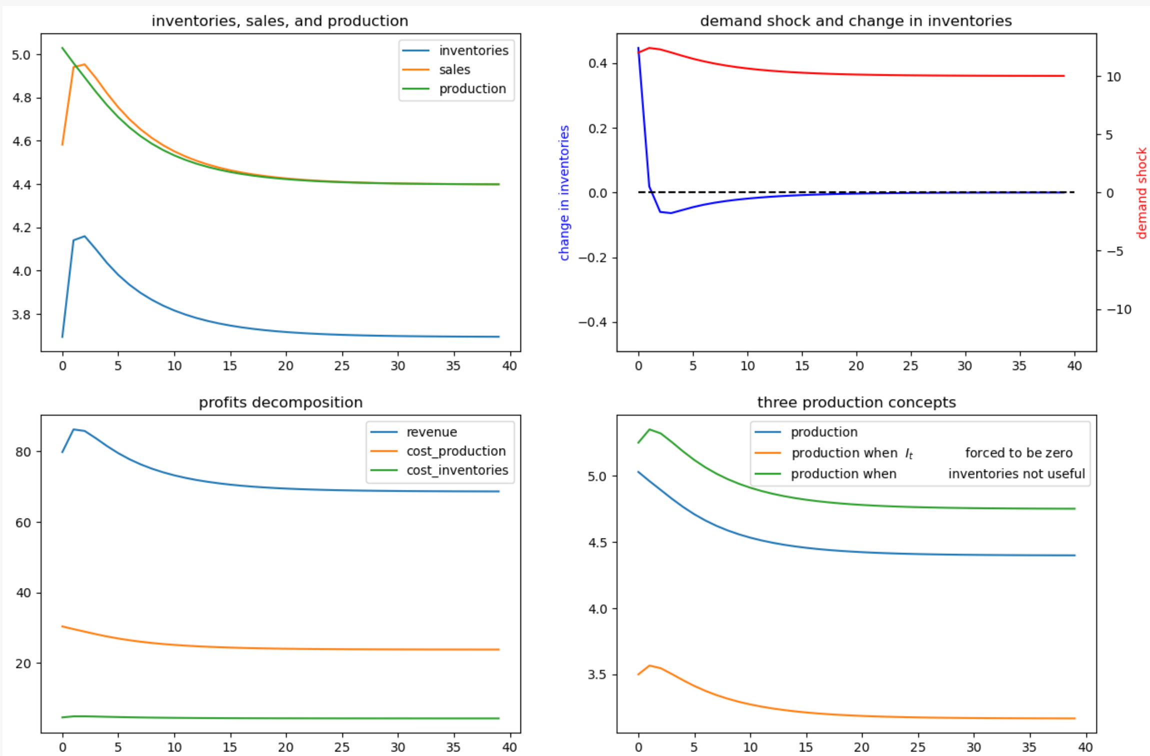



```
# now silence the noise
ex1_no_noise = SmoothingExample(A22=A22, C2=[[0], [0], [0]], G=G)

# initial condition
x0 = [0, 1, 0, 0]

# compute stationary states
x_bar = ex1_no_noise.LQ.compute_sequence(x0, ts_length=250)[0][:, -1]
x_bar
```

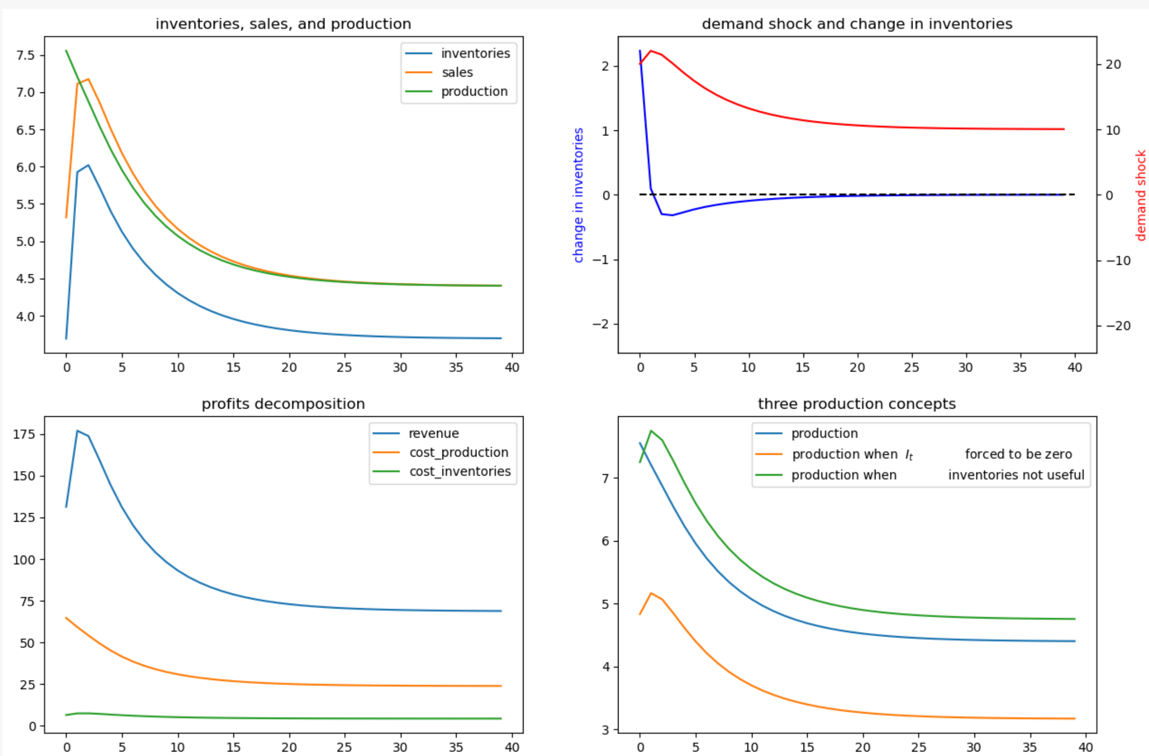
```
array([ 3.69387755,  1.        , 10.        , 10.        ])
```

In the following, we add small and large shocks to $\bar{\nu}_t$ and compare how firm responds differently in quantity. As the shock is not very persistent under the parameterization we are using, we focus on a short period response.

```
T = 40
# small shock
x_bar1 = x_bar.copy()
x_bar1[2] += 2
ex1_no_noise.simulate(x_bar1, T=T)
```




```
# large shock
x_bar1 = x_bar.copy()
x_bar1[2] += 10
ex1_no_noise.simulate(x_bar1, T=T)
```



Exercise 69.10.2

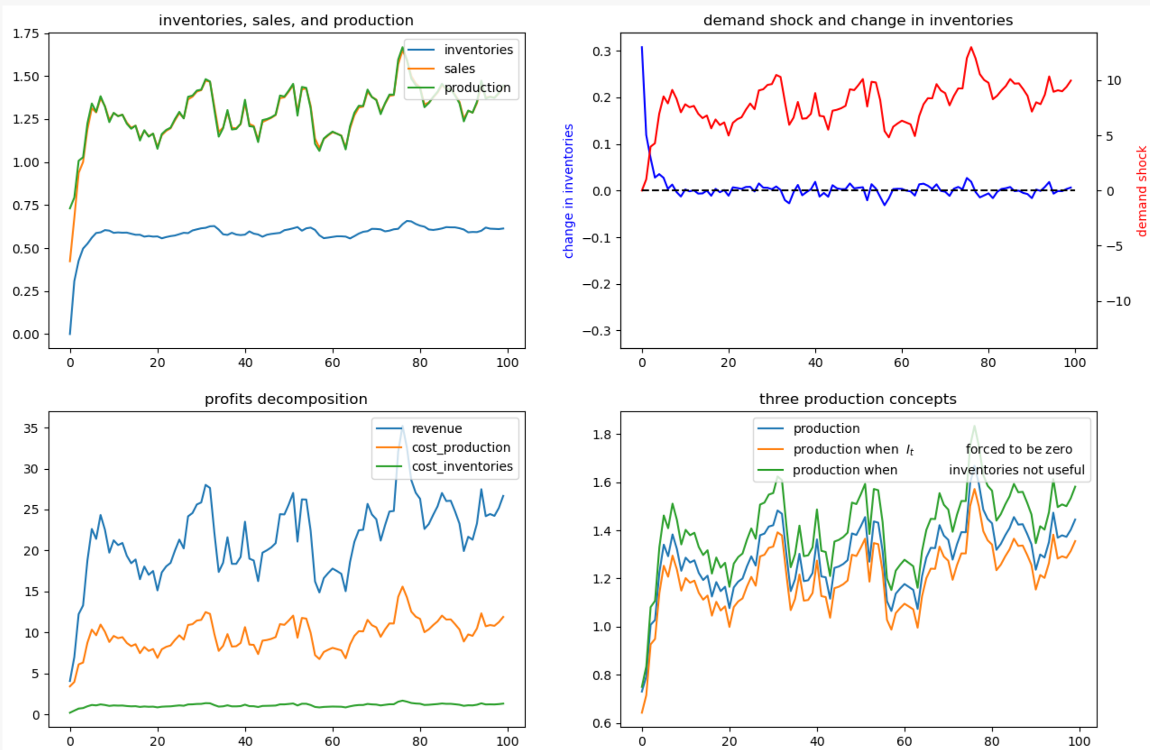
Change parameters of $C(Q_t)$ and $d(I_t, S_t)$.

1. Make production more costly, by setting $c_2 = 5$.
2. Increase the cost of having inventories deviate from sales, by setting $d_2 = 5$.

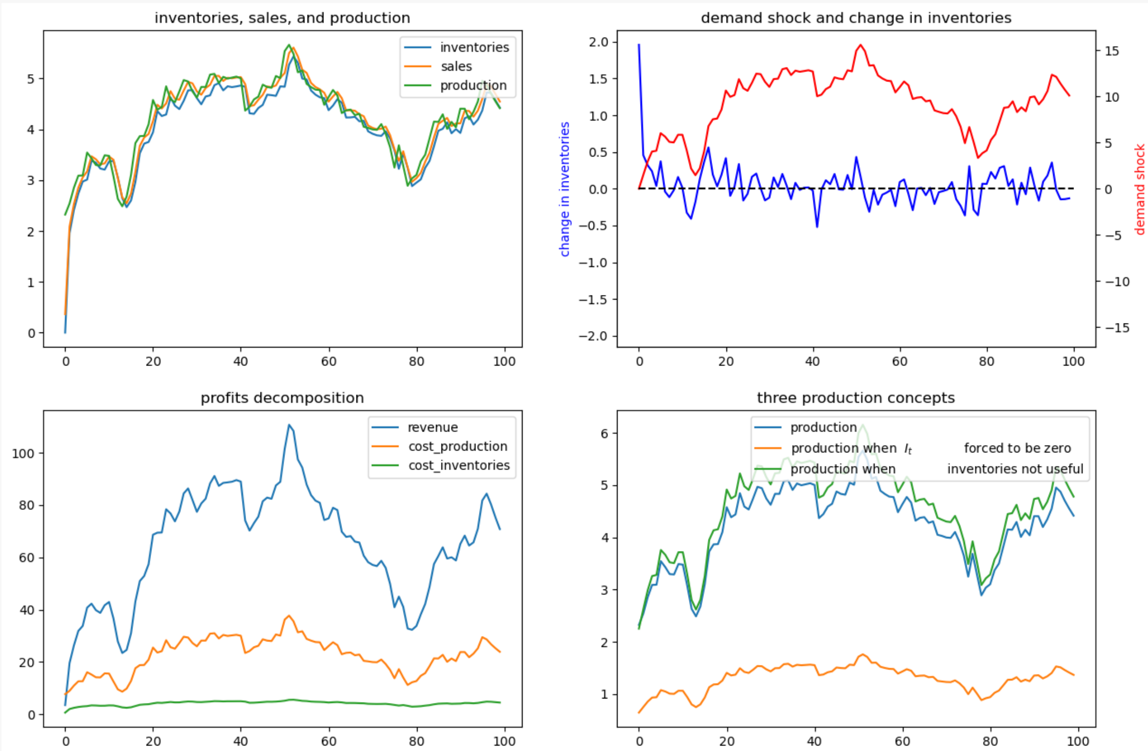
Solution

```
x0 = [0, 1, 0]
```

```
SmoothingExample(c2=5).simulate(x0)
```




```
SmoothingExample(d2=5).simulate(x0)
```


inventories, sales, and production
inventories
sales
production
demand shock and change in inventories
change in inventories
demand shock
profits decomposition
revenue
cost_production
cost_inventories
three production concepts
production
production when I_t forced to be zero
production when inventories not useful

Part XI

Optimal Growth

CHAPTER

SEVENTY

CASS-KOOPMANS MODEL

70.1 Overview

This lecture and *Cass-Koopmans Competitive Equilibrium* describe a model that Tjalling Koopmans [Koopmans, 1965] and David Cass [Cass, 1965] used to analyze optimal growth.

The model extends the model of Robert Solow described in an earlier lecture.

It does so by making saving rate be a decision, instead of a hard-wired constant.

(Solow assumed a constant saving rate determined outside the model.)

We describe two versions of the model, a planning problem in this lecture, and a competitive equilibrium in this lecture *Cass-Koopmans Competitive Equilibrium*.

Together, the two lectures illustrate what is, in fact, a more general connection between a **planned economy** and a decentralized economy organized as a **competitive equilibrium**.

This lecture is devoted to the planned economy version.

In the planned economy, there are

- no prices
- no budget constraints

Instead there is a dictator that tells people

- what to produce
- what to invest in physical capital
- who is to consume what and when

The lecture uses important ideas including

- A min-max problem for solving a planning problem.
- A **shooting algorithm** for solving difference equations subject to initial and terminal conditions.
- A **turnpike** property of optimal paths for long but finite-horizon economies.
- A **stable manifold** and a **phase plane**

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
```

Let's start with some standard imports:

```
import matplotlib.pyplot as plt
from numba import jit, float64
from numba.experimental import jitclass
import numpy as np
from quantecon.optimize import brentq
```

70.2 The Model

Time is discrete and takes values $t = 0, 1, \ldots, T$ where T is finite.

(We'll eventually study a limiting case in which $T = +\infty$)

A single good can either be consumed or invested in physical capital.

The consumption good is not durable and depreciates completely if not consumed immediately.

The capital good is durable but depreciates.

We let C_t be the total consumption of a nondurable consumption good at time t.

Let K_t be the stock of physical capital at time t.

Let $\vec{C}$ = $\{C_0, \ldots, C_T\}$ and $\vec{K}$ = $\{K_0, \ldots, K_{T+1}\}$.

70.2.1 Digression: Aggregation Theory

We use a concept of a representative consumer to be thought of as follows.

There is a unit mass of identical consumers indexed by $\omega \in [0, 1]$.

Consumption of consumer ω is $c(\omega)$.

Aggregate consumption is

$$C = \int_0^1 c(\omega) d\omega$$

Consider a welfare problem that chooses an allocation $\{c(\omega)\}$ across consumers to maximize

$$\int_0^1 u(c(\omega)) d\omega$$

where $u(\cdot)$ is a concave utility function with $u' > 0, u'' < 0$ and maximization is subject to

$$C = \int_0^1 c(\omega) d\omega. \tag{70.1}$$

Form a Lagrangian $L = \int_0^1 u(c(\omega)) d\omega + \lambda[C - \int_0^1 c(\omega) d\omega]$.

Differentiate under the integral signs with respect to each ω to obtain the first-order necessary conditions

$$u'(c(\omega)) = \lambda.$$

These conditions imply that $c(\omega)$ equals a constant c that is independent of ω.

To find c, use feasibility constraint (70.1) to conclude that

$$c(\omega) = c = C.$$

This line of argument indicates the special *aggregation theory* that lies beneath outcomes in which a representative consumer consumes amount C.

It appears often in aggregate economics.

We shall use this aggregation theory here and also in this lecture *Cass-Koopmans Competitive Equilibrium*.

An Economy

A representative household is endowed with one unit of labor at each t and likes the consumption good at each t.

The representative household inelastically supplies a single unit of labor N_t at each t, so that $N_t = 1$ for all $t \in \{0, 1, \dots, T\}$.

The representative household has preferences over consumption bundles ordered by the utility functional:

$$U(\vec{C}) = \sum_{t=0}^{T} \beta^t \frac{C_t^{1-\gamma}}{1-\gamma} \tag{70.2}$$

where $\beta \in (0, 1)$ is a discount factor and $\gamma > 0$ governs the curvature of the one-period utility function.

Larger γ's imply more curvature.

Note that

$$u(C_t) = \frac{C_t^{1-\gamma}}{1-\gamma} \tag{70.3}$$

satisfies $u' > 0, u'' < 0$.

$u' > 0$ asserts that the consumer prefers more to less.

$u'' < 0$ asserts that marginal utility declines with increases in C_t.

We assume that $K_0 > 0$ is an exogenous initial capital stock.

There is an economy-wide production function

$$F(K_t, N_t) = A K_t^{\alpha} N_t^{1-\alpha} \tag{70.4}$$

with $0 < \alpha < 1$, $A > 0$.

A feasible allocation $\vec{C}, \vec{K}$ satisfies

$$C_t + K_{t+1} \leq F(K_t, N_t) + (1-\delta) K_t \quad \text{for all } t \in \{0, 1, \dots, T\} \tag{70.5}$$

where $\delta \in (0, 1)$ is a depreciation rate of capital.

70.3 Planning Problem

A planner chooses an allocation $\{\vec{C}, \vec{K}\}$ to maximize (70.2) subject to (70.5).

Let $\vec{\mu} = \{\mu_0, \dots, \mu_T\}$ be a sequence of nonnegative **Lagrange multipliers**.

To find an optimal allocation, form a Lagrangian

$$\mathcal{L}(\vec{C}, \vec{K}, \vec{\mu}) = \sum_{t=0}^{T} \beta^t \left\{ u(C_t) + \mu_t \left(F(K_t, 1) + (1-\delta) K_t - C_t - K_{t+1} \right) \right\} \tag{70.6}$$

and pose the following min-max problem:

$$\min_{\vec{\mu}} \max_{\vec{C}, \vec{K}} \mathcal{L}(\vec{C}, \vec{K}, \vec{\mu}) \tag{70.7}$$

- **Extremization** means maximization with respect to $\vec{C}, \vec{K}$ and minimization with respect to $\vec{\mu}$.
- Our problem satisfies conditions that assure that second-order conditions are satisfied at an allocation that satisfies the first-order necessary conditions that we are about to compute.

Before computing first-order conditions, we present some handy formulas.

70.3.1 Useful Properties of Linearly Homogeneous Production Function

The following technicalities will help us.

Notice that

$$F(K_t, N_t) = AK_t^{\alpha} N_t^{1-\alpha} = N_t A \left(\frac{K_t}{N_t}\right)^{\alpha}$$

Define the **output per-capita production function**

$$\frac{F(K_t, N_t)}{N_t} \equiv f\left(\frac{K_t}{N_t}\right) = A\left(\frac{K_t}{N_t}\right)^{\alpha}$$

whose argument is **capital per-capita**.

It is useful to recall the following calculations for the marginal product of capital

$$\begin{aligned} \frac{\partial F(K_t, N_t)}{\partial K_t} &= \frac{\partial N_t f\left(\frac{K_t}{N_t}\right)}{\partial K_t} \\ &= N_t f'\left(\frac{K_t}{N_t}\right)\frac{1}{N_t} \quad \text{(Chain rule)} \\ &= f'\left(\frac{K_t}{N_t}\right)\Big|_{N_t=1} \\ &= f'(K_t) \end{aligned} \tag{70.8}$$

and the marginal product of labor

$$\begin{aligned} \frac{\partial F(K_t, N_t)}{\partial N_t} &= \frac{\partial N_t f\left(\frac{K_t}{N_t}\right)}{\partial N_t} \quad \text{(Product rule)} \\ &= f\left(\frac{K_t}{N_t}\right) + N_t f'\left(\frac{K_t}{N_t}\right)\frac{-K_t}{N_t^2} \quad \text{(Chain rule)} \\ &= f\left(\frac{K_t}{N_t}\right) - \frac{K_t}{N_t} f'\left(\frac{K_t}{N_t}\right)\Big|_{N_t=1} \\ &= f(K_t) - f'(K_t)K_t \end{aligned}$$

(Here we are using that $N_t = 1$ for all t, so that $K_t = \frac{K_t}{N_t}$.)

70.3.2 First-order necessary conditions

We now compute **first-order necessary conditions** for extremization of Lagrangian (70.6):

$$C_t: \qquad u'(C_t) - \mu_t = 0 \qquad \text{for all} \quad t = 0, 1, \dots, T \tag{70.9}$$

$$K_t: \qquad \beta \mu_t \left[(1-\delta) + f'(K_t)\right] - \mu_{t-1} = 0 \qquad \text{for all} \quad t = 1, 2, \dots, T \tag{70.10}$$

$$\mu_t: \qquad F(K_t, 1) + (1-\delta) K_t - C_t - K_{t+1} = 0 \qquad \text{for all} \quad t = 0, 1, \dots, T \tag{70.11}$$

$$K_{T+1}: \qquad -\mu_T \leq 0, \ \leq 0 \text{ if } K_{T+1} = 0; \ = 0 \text{ if } K_{T+1} > 0 \tag{70.12}$$

In computing (70.10) we recognize that K_t appears in both the time t and time $t-1$ feasibility constraints (70.5).

Restrictions (70.12) come from differentiating with respect to K_{T+1} and applying the following **Karush-Kuhn-Tucker condition** (KKT) (see Karush-Kuhn-Tucker conditions):

$$\mu_T K_{T+1} = 0 \tag{70.13}$$

Combining (70.9) and (70.10) gives

$$\beta u'\left(C_t\right)\left[(1-\delta) + f'\left(K_t\right)\right] - u'\left(C_{t-1}\right) = 0 \quad \text{for all } t = 1, 2, \dots, T+1$$

which can be rearranged to become

$$\beta u'\left(C_{t+1}\right)\left[(1-\delta) + f'\left(K_{t+1}\right)\right] = u'\left(C_t\right) \quad \text{for all } t = 0, 1, \dots, T \tag{70.14}$$

Applying the inverse marginal utility of consumption function on both sides of the above equation gives

$$C_{t+1} = u'^{-1}\left(\left(\frac{\beta}{u'(C_t)}[f'(K_{t+1}) + (1-\delta)]\right)^{-1}\right)$$

which for our utility function (70.3) becomes the consumption **Euler equation**

$$C_{t+1} = \left(\beta C_t^{\gamma}[f'(K_{t+1}) + (1-\delta)]\right)^{1/\gamma} \tag{70.15}$$

which we can combine with the feasibility constraint (70.5) to get

$$\begin{aligned} C_{t+1} &= C_t \left(\beta[f'(F(K_t, 1) + (1-\delta) K_t - C_t) + (1-\delta)]\right)^{1/\gamma} \\ K_{t+1} &= F(K_t, 1) + (1-\delta) K_t - C_t. \end{aligned} \tag{70.16}$$

This is a pair of non-linear first-order difference equations that map C_t, K_t into C_{t+1}, K_{t+1} and that an optimal sequence $\vec{C}, \vec{K}$ must satisfy.

It must also satisfy the initial condition that K_0 is given and $K_{T+1} = 0$.

Below we define a `jitclass` that stores parameters and functions that define our economy.

```
planning_data = [
    ('γ', float64),    # Coefficient of relative risk aversion
    ('β', float64),    # Discount factor
    ('δ', float64),    # Depreciation rate on capital
    ('α', float64),    # Return to capital per capita
    ('A', float64)     # Technology
]
```

```python
@jitclass(planning_data)
class PlanningProblem():

    def __init__(self, γ=2, β=0.95, δ=0.02, α=0.33, A=1):

        self.γ, self.β = γ, β
        self.δ, self.α, self.A = δ, α, A

    def u(self, c):
        '''
        Utility function
        ASIDE: If you have a utility function that is hard to solve by hand
        you can use automatic or symbolic differentiation
        See https://github.com/HIPS/autograd
        '''
        γ = self.γ

        return c ** (1 - γ) / (1 - γ) if γ!= 1 else np.log(c)

    def u_prime(self, c):
        'Derivative of utility'
        γ = self.γ

        return c ** (-γ)

    def u_prime_inv(self, c):
        'Inverse of derivative of utility'
        γ = self.γ

        return c ** (-1 / γ)

    def f(self, k):
        'Production function'
        α, A = self.α, self.A

        return A * k ** α

    def f_prime(self, k):
        'Derivative of production function'
        α, A = self.α, self.A

        return α * A * k ** (α - 1)

    def f_prime_inv(self, k):
        'Inverse of derivative of production function'
        α, A = self.α, self.A

        return (k / (A * α)) ** (1 / (α - 1))

    def next_k_c(self, k, c):
        ''''
        Given the current capital Kt and an arbitrary feasible
        consumption choice Ct, computes Kt+1 by state transition law
        and optimal Ct+1 by Euler equation.
        '''
        β, δ = self.β, self.δ
        u_prime, u_prime_inv = self.u_prime, self.u_prime_inv
```

(continues on next page)

(continued from previous page)

```
        f, f_prime = self.f, self.f_prime

        k_next = f(k) + (1 - δ) * k - c
        c_next = u_prime_inv(u_prime(c) / (β * (f_prime(k_next) + (1 - δ))))

        return k_next, c_next
```

We can construct an economy with the Python code:

```
pp = PlanningProblem()
```

70.4 Shooting Algorithm

We use **shooting** to compute an optimal allocation $\vec{C}, \vec{K}$ and an associated Lagrange multiplier sequence $\vec{\mu}$.

First-order necessary conditions (70.9), (70.10), and (70.11) for the planning problem form a system of **difference equations** with two boundary conditions:

- K_0 is a given **initial condition** for capital
- $K_{T+1} = 0$ is a **terminal condition** for capital that we deduced from the first-order necessary condition for K_{T+1} the KKT condition (70.13)

We have no initial condition for the Lagrange multiplier μ_0.

If we did, our job would be easy:

- Given μ_0 and k_0, we could compute c_0 from equation (70.9) and then k_1 from equation (70.11) and μ_1 from equation (70.10).
- We could continue in this way to compute the remaining elements of $\vec{C}, \vec{K}, \vec{\mu}$.

However, we woujld not be assured that the Kuhn-Tucker condition (70.13) would be satisfied.

Furthermore, we don't have an initial condition for μ_0.

So this won't work.

Indeed, part of our task is to compute the **optimal** value of μ_0.

To compute μ_0 and the other objects we want, a simple modification of the above procedure will work.

It is called the **shooting algorithm**.

It is an instance of a **guess and verify** algorithm that consists of the following steps:

- Guess an initial Lagrange multiplier μ_0.
- Apply the **simple algorithm** described above.
- Compute K_{T+1} and check whether it equals zero.
- If $K_{T+1} = 0$, we have solved the problem.
- If $K_{T+1} > 0$, lower μ_0 and try again.
- If $K_{T+1} < 0$, raise μ_0 and try again.

The following Python code implements the shooting algorithm for the planning problem.

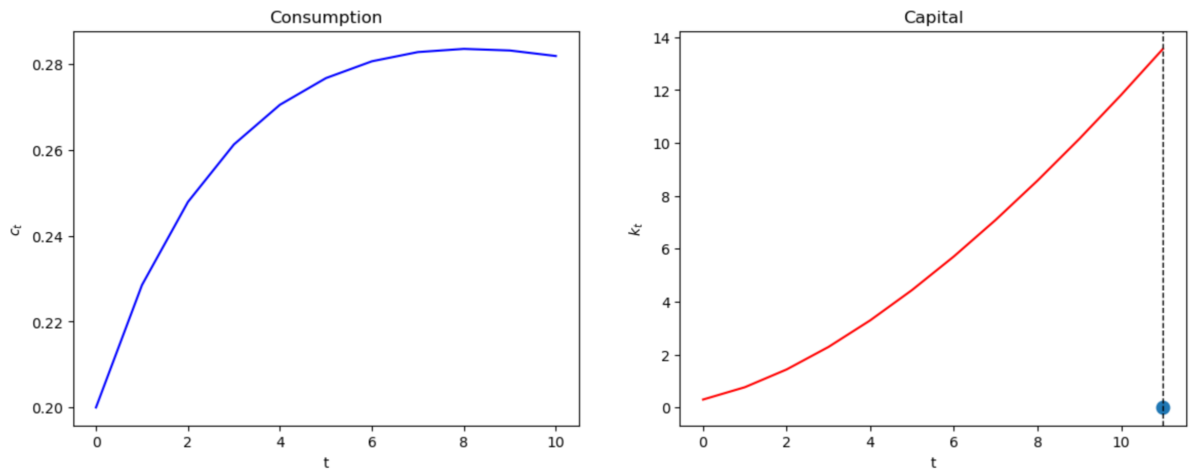
(Actually, we modified the preceding algorithm slightly by starting with a guess for c_0 instead of μ_0 in the following code.)

```
@jit
def shooting(pp, c0, k0, T=10):
    '''
    Given the initial condition of capital k0 and an initial guess
    of consumption c0, computes the whole paths of c and k
    using the state transition law and Euler equation for T periods.
    '''
    if c0 > pp.f(k0) + (1 - pp.δ) * k0:
        print("initial consumption is not feasible")

        return None

    # initialize vectors of c and k
    c_vec = np.empty(T+1)
    k_vec = np.empty(T+2)

    c_vec[0] = c0
    k_vec[0] = k0

    for t in range(T):
        k_vec[t+1], c_vec[t+1] = pp.next_k_c(k_vec[t], c_vec[t])

    k_vec[T+1] = pp.f(k_vec[T]) + (1 - pp.δ) * k_vec[T] - c_vec[T]

    return c_vec, k_vec
```

We'll start with an incorrect guess.

```
paths = shooting(pp, 0.2, 0.3, T=10)
```

```
fig, axs = plt.subplots(1, 2, figsize=(14, 5))

colors = ['blue', 'red']
titles = ['Consumption', 'Capital']
ylabels = ['$c_t$', '$k_t$']

T = paths[0].size - 1
for i in range(2):
    axs[i].plot(paths[i], c=colors[i])
    axs[i].set(xlabel='t', ylabel=ylabels[i], title=titles[i])

axs[1].scatter(T+1, 0, s=80)
axs[1].axvline(T+1, color='k', ls='--', lw=1)

plt.show()
```



Evidently, our initial guess for μ_0 is too high, so initial consumption too low.

We know this because we miss our $K_{T+1} = 0$ target on the high side.

Now we automate things with a search-for-a-good μ_0 algorithm that stops when we hit the target $K_{t+1} = 0$.

We use a **bisection method**.

We make an initial guess for C_0 (we can eliminate μ_0 because C_0 is an exact function of μ_0).

We know that the lowest C_0 can ever be is 0 and that the largest it can be is initial output $f(K_0)$.

Guess C_0 and shoot forward to $T + 1$.

If $K_{T+1} > 0$, we take it to be our new **lower** bound on C_0.

If $K_{T+1} < 0$, we take it to be our new **upper** bound.

Make a new guess for C_0 that is halfway between our new upper and lower bounds.

Shoot forward again, iterating on these steps until we converge.

When K_{T+1} gets close enough to 0 (i.e., within an error tolerance bounds), we stop.

```
@jit
def bisection(pp, c0, k0, T=10, tol=1e-4, max_iter=500, k_ter=0, verbose=True):

    # initial boundaries for guess c0
    c0_upper = pp.f(k0) + (1 - pp.δ) * k0
    c0_lower = 0

    i = 0
    while True:
        c_vec, k_vec = shooting(pp, c0, k0, T)
        error = k_vec[-1] - k_ter

        # check if the terminal condition is satisfied
        if np.abs(error) < tol:
            if verbose:
                print('Converged successfully on iteration ', i+1)
            return c_vec, k_vec

        i += 1
        if i == max_iter:
            if verbose:
```

(continues on next page)

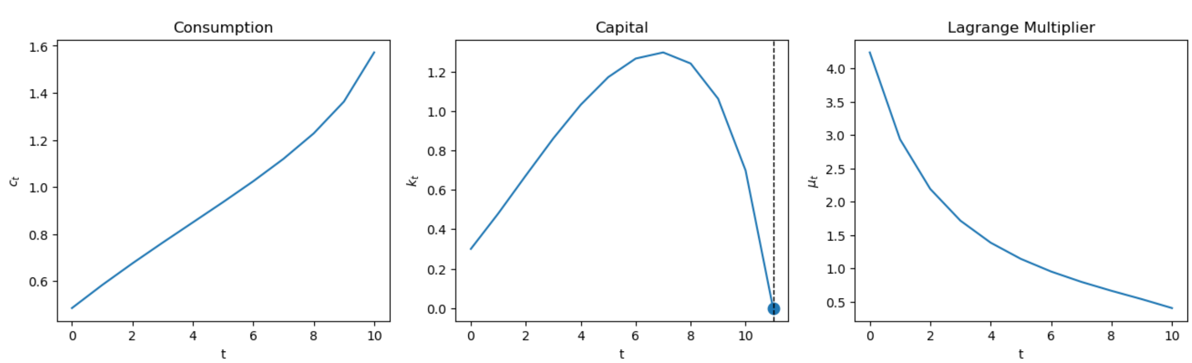
(continued from previous page)

```
            print('Convergence failed.')
        return c_vec, k_vec

    # if iteration continues, updates boundaries and guess of c0
    if error > 0:
        c0_lower = c0
    else:
        c0_upper = c0

    c0 = (c0_lower + c0_upper) / 2
```

```
def plot_paths(pp, c0, k0, T_arr, k_ter=0, k_ss=None, axs=None):

    if axs is None:
        fix, axs = plt.subplots(1, 3, figsize=(16, 4))
    ylabels = ['$c_t$', '$k_t$', r'$\mu_t$']
    titles = ['Consumption', 'Capital', 'Lagrange Multiplier']

    c_paths = []
    k_paths = []
    for T in T_arr:
        c_vec, k_vec = bisection(pp, c0, k0, T, k_ter=k_ter, verbose=False)
        c_paths.append(c_vec)
        k_paths.append(k_vec)

        μ_vec = pp.u_prime(c_vec)
        paths = [c_vec, k_vec, μ_vec]

        for i in range(3):
            axs[i].plot(paths[i])
            axs[i].set(xlabel='t', ylabel=ylabels[i], title=titles[i])

        # Plot steady state value of capital
        if k_ss is not None:
            axs[1].axhline(k_ss, c='k', ls='--', lw=1)

        axs[1].axvline(T+1, c='k', ls='--', lw=1)
        axs[1].scatter(T+1, paths[1][-1], s=80)

    return c_paths, k_paths
```

Now we can solve the model and plot the paths of consumption, capital, and Lagrange multiplier.

```
plot_paths(pp, 0.3, 0.3, [10]);
```



70.5 Setting Initial Capital to Steady State Capital

When $T \rightarrow +\infty$, the optimal allocation converges to steady state values of C_t and K_t.

It is instructive to set K_0 equal to the $\lim_{T\rightarrow+\infty} K_t$, which we'll call steady state capital.

In a steady state $K_{t+1} = K_t = \bar{K}$ for all very large t.

Evalauating feasibility constraint (70.5) at $\bar{K}$ gives

$$f(\bar{K}) - \delta\bar{K} = \bar{C} \tag{70.17}$$

Substituting $K_t = \bar{K}$ and $C_t = \bar{C}$ for all t into (70.14) gives

$$1 = \beta \frac{u'(\bar{C})}{u'(\bar{C})}[f'(\bar{K}) + (1-\delta)]$$

Defining $\beta = \frac{1}{1+\rho}$, and cancelling gives

$$1 + \rho = 1[f'(\bar{K}) + (1-\delta)]$$

Simplifying gives

$$f'(\bar{K}) = \rho + \delta$$

and

$$\bar{K} = f'^{-1}(\rho + \delta)$$

For production function (70.4), this becomes

$$\alpha\bar{K}^{\alpha-1} = \rho + \delta$$

As an example, after setting $\alpha = .33$, $\rho = 1/\beta - 1 = 1/(19/20) - 1 = 20/19 - 19/19 = 1/19$, $\delta = 1/50$, we get

$$\bar{K} = \left(\frac{\frac{33}{100}}{\frac{1}{50} + \frac{1}{19}}\right)^{\frac{67}{100}} \approx 9.57583$$

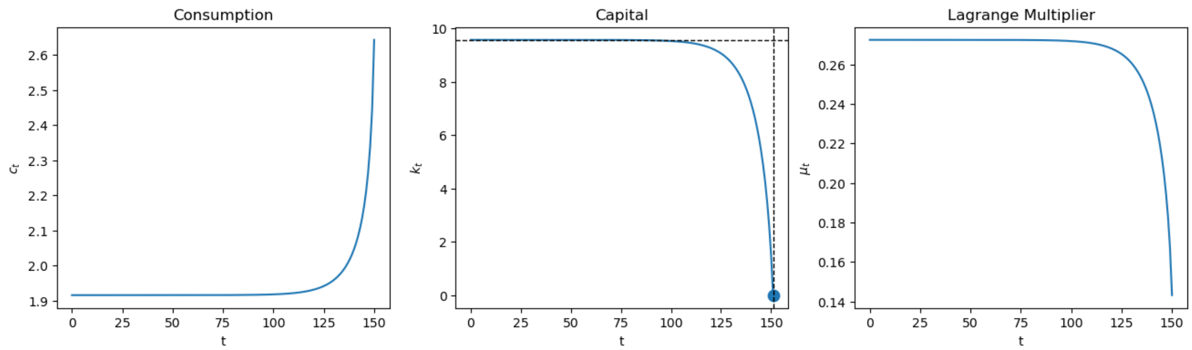
Let's verify this with Python and then use this steady state $\bar{K}$ as our initial capital stock K_0.

```python
ρ = 1 / pp.β - 1
k_ss = pp.f_prime_inv(ρ+pp.δ)

print(f'steady state for capital is: {k_ss}')
```

```
steady state for capital is: 9.57583816331462
```

Now we plot

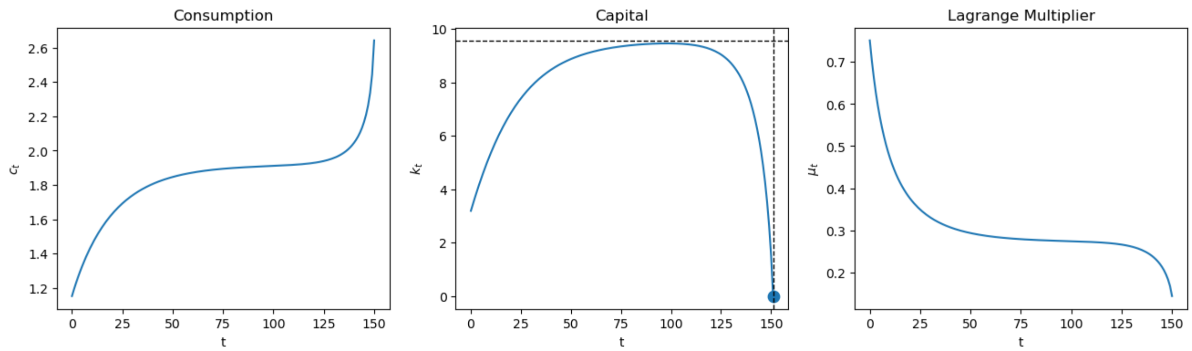
```python
plot_paths(pp, 0.3, k_ss, [150], k_ss=k_ss);
```



Evidently, with a large value of T, K_t stays near K_0 until t approaches T closely.

Let's see what the planner does when we set K_0 below $\bar{K}$.

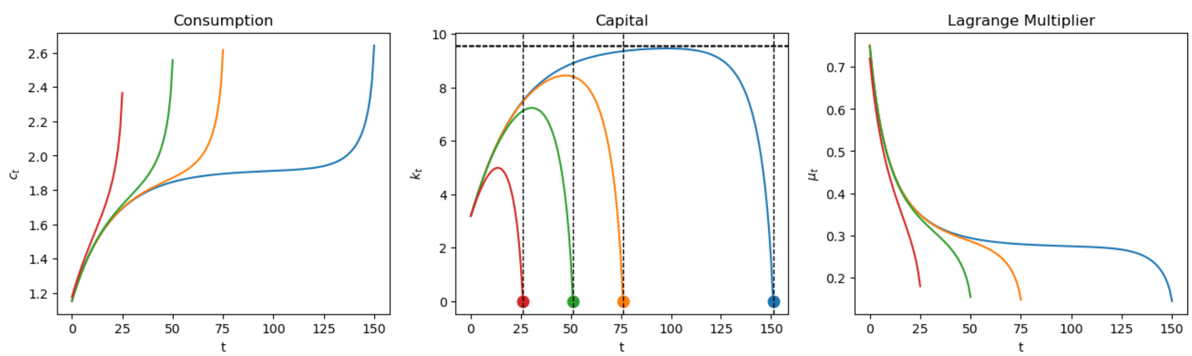
```
plot_paths(pp, 0.3, k_ss/3, [150], k_ss=k_ss);
```



Notice how the planner pushes capital toward the steady state, stays near there for a while, then pushes K_t toward the terminal value $K_{T+1} = 0$ when t closely approaches T.

The following graphs compare optimal outcomes as we vary T.

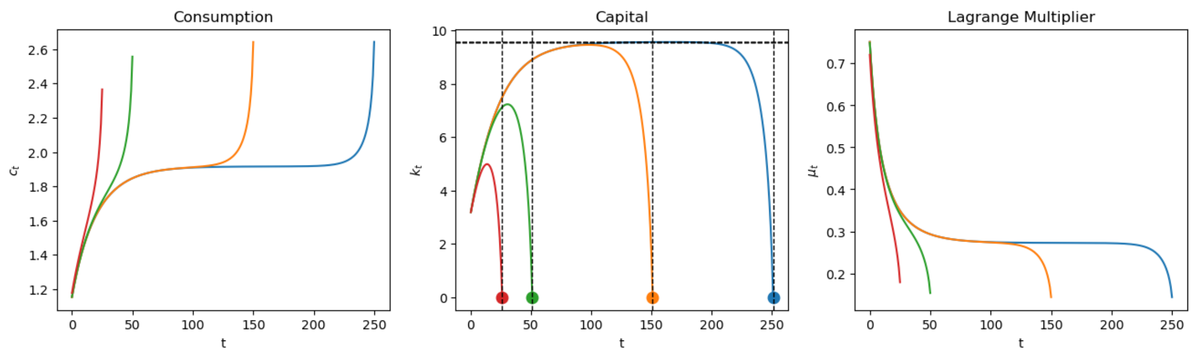
```
plot_paths(pp, 0.3, k_ss/3, [150, 75, 50, 25], k_ss=k_ss);
```



70.6 A Turnpike Property

The following calculation indicates that when T is very large, the optimal capital stock stays close to its steady state value most of the time.

```
plot_paths(pp, 0.3, k_ss/3, [250, 150, 50, 25], k_ss=k_ss);
```



In the above graphs, different colors are associated with different horizons T.

Notice that as the horizon increases, the planner keeps K_t closer to the steady state value $\bar{K}$ for longer.

This pattern reflects a **turnpike** property of the steady state.

A rule of thumb for the planner is

- from K_0, push K_t toward the steady state and stay close to the steady state until time approaches T.

The planner accomplishes this by adjusting the saving rate $\frac{f(K_t)-C_t}{f(K_t)}$ over time.

Exercise 70.6.1

The turnpike property is independent of the initial condition K_0 provided that T is sufficiently large.

Expand the `plot_paths` function so that it plots trajectories for multiple initial points using `k0s = [k_ss*2, k_ss*3, k_ss/3]`.

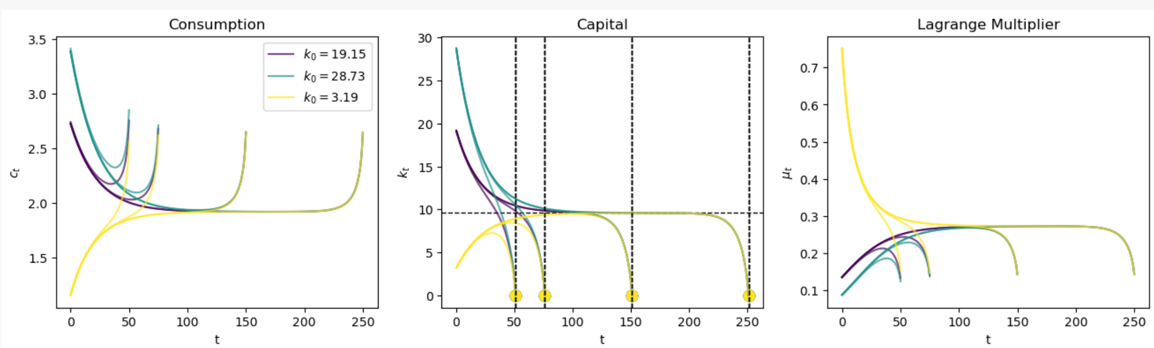
Solution

Here is one solution

```
def plot_multiple_paths(pp, c0, k0s, T_arr, k_ter=0, k_ss=None, axs=None):
    if axs is None:
        fig, axs = plt.subplots(1, 3, figsize=(16, 4))

    ylabels = ['$c_t$', '$k_t$', r'$\mu_t$']
    titles = ['Consumption', 'Capital', 'Lagrange Multiplier']

    colors = plt.cm.viridis(np.linspace(0, 1, len(k0s)))

    all_c_paths = []
    all_k_paths = []

    for i, k0 in enumerate(k0s):
        k0_c_paths = []
        k0_k_paths = []
```

```
        for T in T_arr:
            c_vec, k_vec = bisection(pp, c0, k0, T, k_ter=k_ter, verbose=False)
            k0_c_paths.append(c_vec)
            k0_k_paths.append(k_vec)

            μ_vec = pp.u_prime(c_vec)
            paths = [c_vec, k_vec, μ_vec]

            for j in range(3):
                axs[j].plot(paths[j], color=colors[i],
                            label=f'$k_0 = {k0:.2f}$' if j == 0 and T == T_arr[0]␣
↪else "", alpha=0.7)
                axs[j].set(xlabel='t', ylabel=ylabels[j], title=titles[j])

            if k_ss is not None and i == 0 and T == T_arr[0]:
                axs[1].axhline(k_ss, c='k', ls='--', lw=1)

            axs[1].axvline(T+1, c='k', ls='--', lw=1)
            axs[1].scatter(T+1, paths[1][-1], s=80, color=colors[i])

        all_c_paths.append(k0_c_paths)
        all_k_paths.append(k0_k_paths)

    # Add legend if multiple initial points
    if len(k0s) > 1:
        axs[0].legend()

    return all_c_paths, all_k_paths
```

```
_ = plot_multiple_paths(pp, 0.3, [k_ss*2, k_ss*3, k_ss/3], [250, 150, 75, 50], k_
↪ss=k_ss)
```



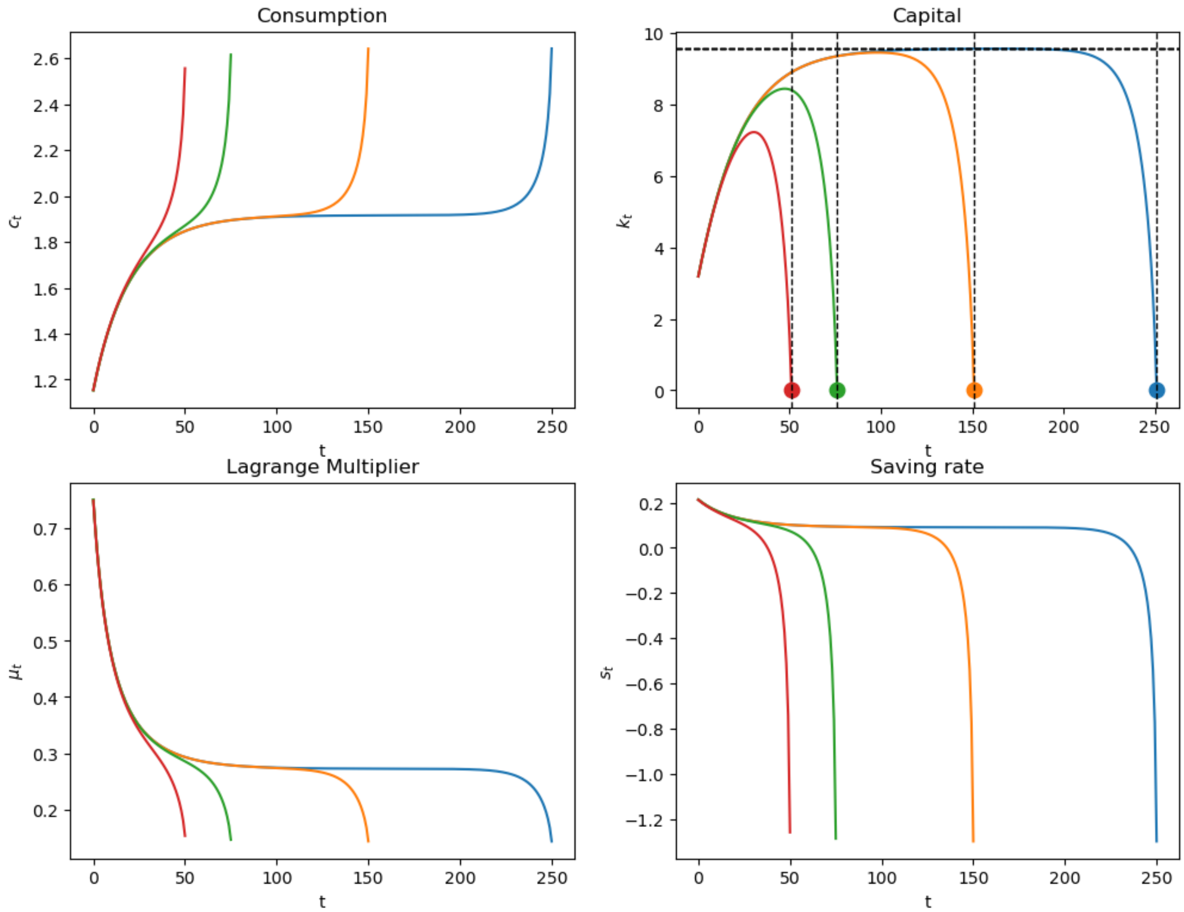
We see that the turnpike property holds for various initial values of K_0.

Let's calculate and plot the saving rate.

```
@jit
def saving_rate(pp, c_path, k_path):
    'Given paths of c and k, computes the path of saving rate.'
    production = pp.f(k_path[:-1])

    return (production - c_path) / production
```

```python
def plot_saving_rate(pp, c0, k0, T_arr, k_ter=0, k_ss=None, s_ss=None):

    fix, axs = plt.subplots(2, 2, figsize=(12, 9))

    c_paths, k_paths = plot_paths(pp, c0, k0, T_arr, k_ter=k_ter, k_ss=k_ss, axs=axs.
↪flatten())

    for i, T in enumerate(T_arr):
        s_path = saving_rate(pp, c_paths[i], k_paths[i])
        axs[1, 1].plot(s_path)

    axs[1, 1].set(xlabel='t', ylabel='$s_t$', title='Saving rate')

    if s_ss is not None:
        axs[1, 1].hlines(s_ss, 0, np.max(T_arr), linestyle='--')
```

```python
plot_saving_rate(pp, 0.3, k_ss/3, [250, 150, 75, 50], k_ss=k_ss)
```



70.7 A Limiting Infinite Horizon Economy

We want to set $T = +\infty$.

The appropriate thing to do is to replace terminal condition (70.12) with

$$\lim_{T \to +\infty} \beta^T u'(C_T) K_{T+1} = 0,$$

a condition that will be satisfied by a path that converges to an optimal steady state.

We can approximate the optimal path by starting from an arbitrary initial K_0 and shooting towards the optimal steady state K at a large but finite $T + 1$.

In the following code, we do this for a large T and plot consumption, capital, and the saving rate.

We know that in the steady state that the saving rate is constant and that $\bar{s} = \frac{f(\bar{K}) - \bar{C}}{f(\bar{K})}$.

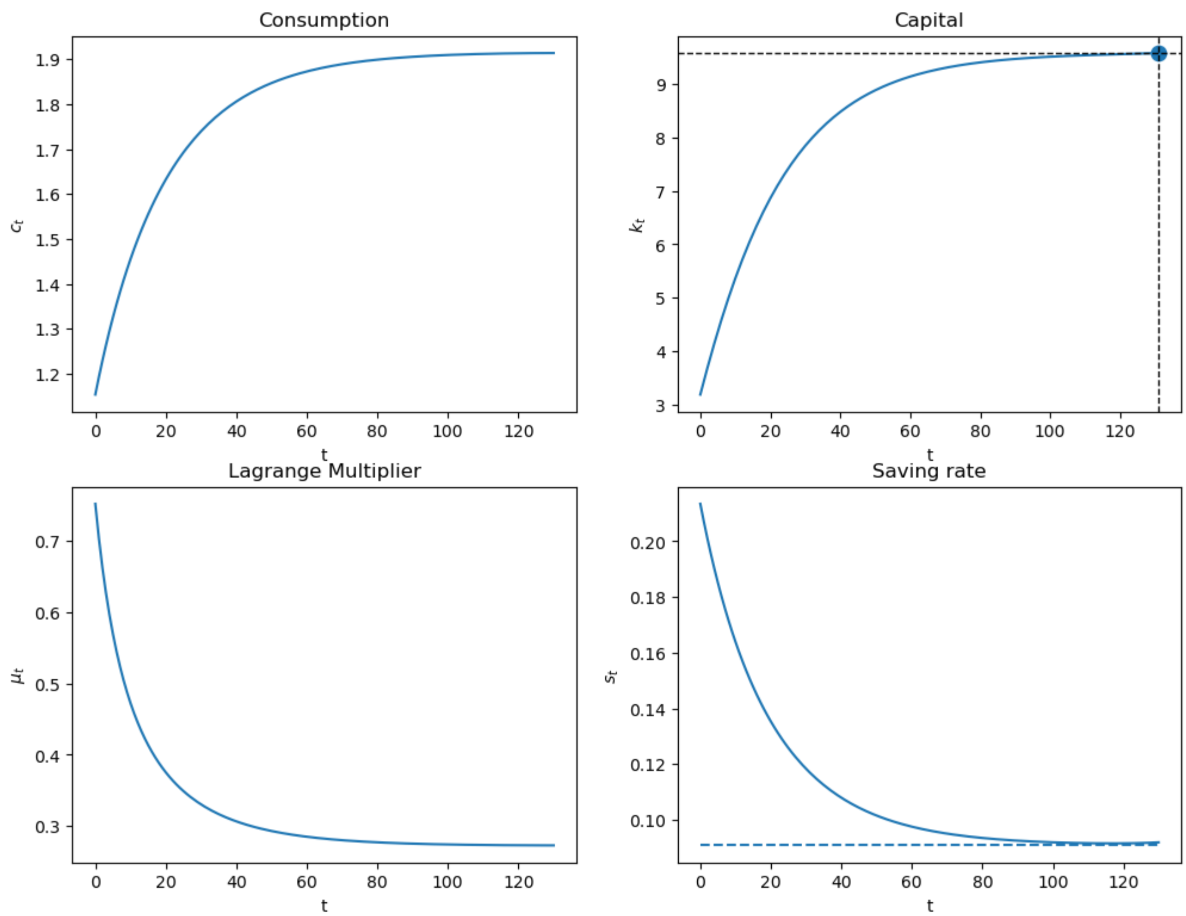
From (70.17) the steady state saving rate equals

$$\bar{s} = \frac{\delta \bar{K}}{f(\bar{K})}$$

The steady state saving rate $\bar{S} = \bar{s} f(\bar{K})$ is the amount required to offset capital depreciation each period.

We first study optimal capital paths that start below the steady state.

```
# steady state of saving rate
s_ss = pp.δ * k_ss / pp.f(k_ss)

plot_saving_rate(pp, 0.3, k_ss/3, [130], k_ter=k_ss, k_ss=k_ss, s_ss=s_ss)
```



Since $K_0 < \bar{K}$, $f'(K_0) > \rho + \delta$.

The planner chooses a positive saving rate that is higher than the steady state saving rate.

Note that $f''(K) < 0$, so as K rises, $f'(K)$ declines.

The planner slowly lowers the saving rate until reaching a steady state in which $f'(K) = \rho + \delta$.

70.8 Stable Manifold and Phase Diagram

We now describe a classic diagram that describes an optimal (K_{t+1}, C_t) path.

The diagram has K on the ordinate axis and C on the coordinate axis.

Given an arbitrary and fixed K, a fixed point C of the consumption Euler equation (70.15) satisfies

$$C = C\left(\beta\left[f'\left(f\left(K\right) + \left(1-\delta\right)K - C\right) + \left(1-\delta\right)\right]\right)^{1/\gamma}$$

which implies

$$\begin{aligned} C &= f\left(K\right) + \left(1-\delta\right)K - f'^{-1}\left(\frac{1}{\beta} - \left(1-\delta\right)\right) \\ &\equiv \tilde{C}\left(K\right) \end{aligned} \tag{70.18}$$

A positive fixed point $C = \tilde{C}(K)$ exists only if $f\left(K\right) + \left(1-\delta\right)K - f'^{-1}\left(\frac{1}{\beta} - \left(1-\delta\right)\right) > 0$

```
@jit
def C_tilde(K, pp):

    return pp.f(K) + (1 - pp.δ) * K - pp.f_prime_inv(1 / pp.β - 1 + pp.δ)
```

Next note that given a time-invariant arbitrary C, a fixed point K of the feasibility condition (70.5) solves the following equation

$$K = f(K) + (1 - \delta K) - C.$$

A fixed point of the above equation is described by a function

$$K = \tilde{K}(C) \tag{70.19}$$

```
@jit
def K_diff(K, C, pp):
    return pp.f(K) - pp.δ * K - C

@jit
def K_tilde(C, pp):

    res = brentq(K_diff, 1e-6, 100, args=(C, pp))

    return res.root
```

A steady state (K_s, C_s) is a pair (K, C) that satisfies both equations (70.18) and (70.19).

It is thus the intersection of the two curves $\tilde{C}$ and $\tilde{K}$ that we'll plot in Figure Fig. 70.1 below.

We can compute K_s by solving the equation $K_s = \tilde{K}\left(\tilde{C}\left(K_s\right)\right)$

```
@jit
def K_tilde_diff(K, pp):

    K_out = K_tilde(C_tilde(K, pp), pp)

    return K - K_out
```

```
res = brentq(K_tilde_diff, 8, 10, args=(pp,))

Ks = res.root
Cs = C_tilde(Ks, pp)

Ks, Cs
```

```
(9.575838163314447, 1.9160839808123402)
```

We can use the shooting algorithm to compute trajectories that approach (K_s, C_s).

For a given K, let's compute $\vec{C}$ and $\vec{K}$ for a large T , e.g., $= 200$.

We compute C_0 by the bisection algorithm that assures that $K_T = K_s$.

Let's compute two trajectories towards (K_s, C_s) that start from different sides of K_s: $\bar{K}_0 = 1e - 3 < K_s < \bar{K}_1 = 15$.

```
c_vec1, k_vec1 = bisection(pp, 5, 15, T=200, k_ter=Ks)
c_vec2, k_vec2 = bisection(pp, 1e-3, 1e-3, T=200, k_ter=Ks)
```

```
Converged successfully on iteration  46
Converged successfully on iteration  51
```

The following code generates Figure Fig. 70.1, which is patterned on a graph that appears on page 411 of [Intriligator, 2002].

Figure Fig. 70.1 is a classic "phase plane" with "state" variable K on the ordinate axis and "co-state" variable C on the coordinate axis.

Figure Fig. 70.1 plots three curves:

- the blue line graphs $C = \tilde{C}(K)$ of fixed points described by equation (70.18).
- the red line graphs $K = \tilde{K}(C)$ of fixed points described by equation (70.19)
- the green line graphs the stable traced out by paths that converge to the steady state starting from an arbitrary K_0 at time 0.
  - for a given K_0, the shooting algorithm sets C_0 to the coordinate on the green line in order to initiate a path that converges to the optimal steady state
  - the arrows on the green line show the direction in which dynamics (70.16) push successive (K_{t+1}, C_t) pairs.

In addition to the three curves, Figure Fig. 70.1 plots arrows that point where the dynamics (70.16) drive the system when, for a given K_0, C_0 is not on the stable manifold depicted in the green line.

- If C_0 is set below the green line for a given K_0, too much capital is accumulated
- If C_0 is set above the green line for a given K_0, too little capital is accumulated

70.9 Concluding Remarks

In *Cass-Koopmans Competitive Equilibrium*, we study a decentralized version of an economy with exactly the same technology and preference structure as deployed here.

In that lecture, we replace the planner of this lecture with Adam Smith's **invisible hand**.

In place of quantity choices made by the planner, there are market prices that are set by a *deus ex machina* from outside the model, a so-called invisible hand.

Equilibrium market prices must reconcile distinct decisions that are made independently by a representative household and a representative firm.

The relationship between a command economy like the one studied in this lecture and a market economy like that studied in *Cass-Koopmans Competitive Equilibrium* is a foundational topic in general equilibrium theory and welfare economics.

70.9.1 Exercise

Exercise 70.9.1

- Plot the optimal consumption, capital, and saving paths when the initial capital level begins at 1.5 times the steady state level as we shoot towards the steady state at $T = 130$.
- Why does the saving rate respond as it does?

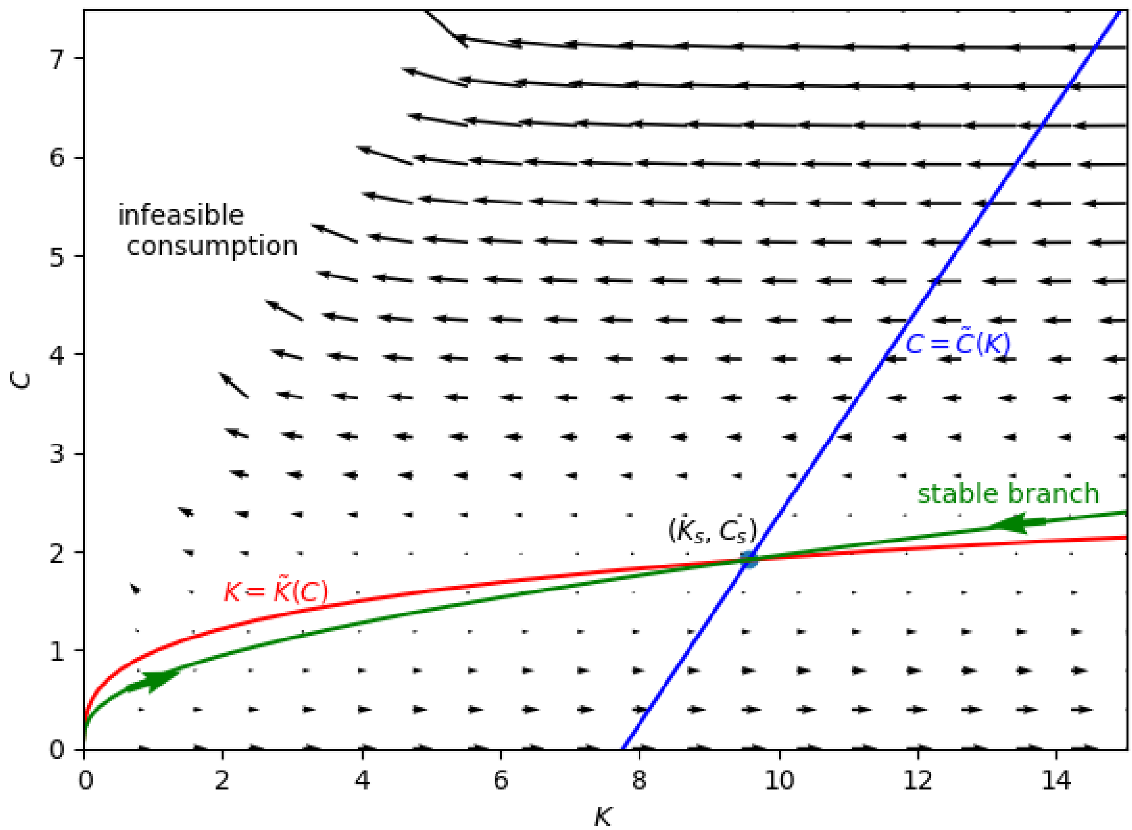
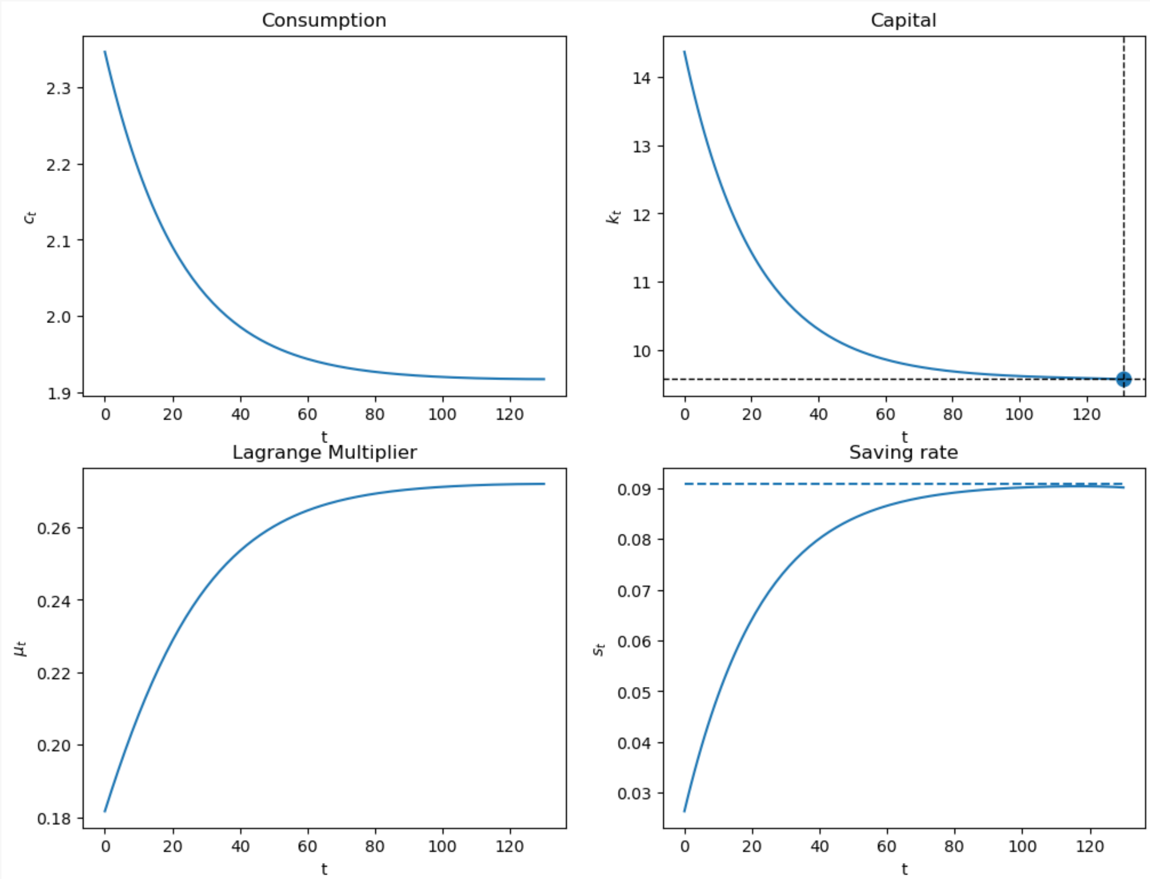


Fig. 70.1: Stable Manifold and Phase Plane

Solution

```
plot_saving_rate(pp, 0.3, k_ss*1.5, [130], k_ter=k_ss, k_ss=k_ss, s_ss=s_ss)
```



CHAPTER

SEVENTYONE

CASS-KOOPMANS COMPETITIVE EQUILIBRIUM

Contents

71.1 Overview

This lecture continues our analysis in this lecture *Cass-Koopmans Planning Model* about the model that Tjalling Koopmans [Koopmans, 1965] and David Cass [Cass, 1965] used to study optimal capital accumulation.

This lecture illustrates what is, in fact, a more general connection between a **planned economy** and an economy organized as a competitive equilibrium or a **market economy**.

The earlier lecture *Cass-Koopmans Planning Model* studied a planning problem and used ideas including

- A Lagrangian formulation of the planning problem that leads to a system of difference equations.
- A **shooting algorithm** for solving difference equations subject to initial and terminal conditions.
- A **turnpike** property that describes optimal paths for long-but-finite horizon economies.

The present lecture uses additional ideas including

- Hicks-Arrow prices, named after John R. Hicks and Kenneth Arrow.
- A connection between some Lagrange multipliers from the planning problem and the Hicks-Arrow prices.
- A **Big** K **, little** k trick widely used in macroeconomic dynamics.
  - We shall encounter this trick in this lecture and also in this lecture.

- A non-stochastic version of a theory of the **term structure of interest rates**.
- An intimate connection between two ways to organize an economy, namely:
  - **socialism** in which a central planner commands the allocation of resources, and
  - **competitive markets** in which competitive equilibrium **prices** induce individual consumers and producers to choose a socially optimal allocation as unintended consequences of their selfish decisions

Let's start with some standard imports:

```
import matplotlib.pyplot as plt
from numba import jit, float64
from numba.experimental import jitclass
import numpy as np
```

71.2 Review of Cass-Koopmans Model

The physical setting is identical with that in *Cass-Koopmans Planning Model*.

Time is discrete and takes values $t = 0, 1, \ldots, T$.

Output of a single good can either be consumed or invested in physical capital.

The capital good is durable but partially depreciates each period at a constant rate.

We let C_t be a nondurable consumption good at time t.

Let K_t be the stock of physical capital at time t.

Let $\vec{C}$ = $\{C_0, \ldots, C_T\}$ and $\vec{K}$ = $\{K_0, \ldots, K_{T+1}\}$.

A representative household is endowed with one unit of labor at each t and likes the consumption good at each t.

The representative household inelastically supplies a single unit of labor N_t at each t, so that $N_t = 1$ for all $t \in \{0, 1, \ldots, T\}$.

The representative household has preferences over consumption bundles ordered by the utility functional:

$$U(\vec{C}) = \sum_{t=0}^{T} \beta^t \frac{C_t^{1-\gamma}}{1-\gamma}$$

where $\beta \in (0, 1)$ is a discount factor and $\gamma > 0$ governs the curvature of the one-period utility function.

We assume that $K_0 > 0$.

There is an economy-wide production function

$$F(K_t, N_t) = A K_t^{\alpha} N_t^{1-\alpha}$$

with $0 < \alpha < 1$, $A > 0$.

A feasible allocation $\vec{C}, \vec{K}$ satisfies

$$C_t + K_{t+1} \leq F(K_t, N_t) + (1 - \delta) K_t \quad \text{for all } t \in \{0, 1, \ldots, T\}$$

where $\delta \in (0, 1)$ is a depreciation rate of capital.

71.2.1 Planning Problem

In this lecture *Cass-Koopmans Planning Model*, we studied a problem in which a planner chooses an allocation $\{\vec{C}, \vec{K}\}$ to maximize (70.2) subject to (70.5).

The allocation that solves the planning problem reappears in a competitive equilibrium, as we shall see below.

71.3 Competitive Equilibrium

We now study a decentralized version of the economy.

It shares the same technology and preference structure as the planned economy studied in this lecture *Cass-Koopmans Planning Model*.

But now there is no planner.

There are (unit masses of) price-taking consumers and firms.

Market prices are set to reconcile distinct decisions that are made separately by a representative consumer and a representative firm.

There is a representative consumer who has the same preferences over consumption plans as did a consumer in the planned economy.

Instead of being told what to consume and save by a planner, a consumer (also known as a *household*) chooses for itself subject to a budget constraint.

- At each time t, the consumer receives wages and rentals of capital from a firm – these comprise its **income** at time t.
- The consumer decides how much income to allocate to consumption or to savings.
- The household can save either by acquiring additional physical capital (it trades one for one with time t consumption) or by acquiring claims on consumption at dates other than t.
- The household owns physical capital and labor and rents them to the firm.
- The household consumes, supplies labor, and invests in physical capital.
- A profit-maximizing representative firm operates the production technology.
- The firm rents labor and capital each period from the representative household and sells its output each period to the household.
- The representative household and the representative firm are both **price takers** who believe that prices are not affected by their choices

Note

Again, we can think of there being unit measures of identical representative consumers and identical representative firms.

71.4 Market Structure

The representative household and the representative firm are both price takers.

The household owns both factors of production, namely, labor and physical capital.

Each period, the firm rents both factors from the household.

There is a **single** grand competitive market in which a household trades date 0 goods for goods at all other dates $t = 1, 2, \ldots, T$.

71.4.1 Prices

There are sequences of prices $\{w_t, \eta_t\}_{t=0}^T = \{\vec{w}, \vec{\eta}\}$ where

- w_t is a wage, i.e., a rental rate, for labor at time t
- η_t is a rental rate for capital at time t

In addition there is a vector $\{q_t^0\}$ of intertemporal prices where

- q_t^0 is the price at time 0 of one unit of the good at date t.

We call $\{q_t^0\}_{t=0}^T$ a vector of **Hicks-Arrow prices**, named after the 1972 economics Nobel prize winners.

Because is a **relative price**. the unit of account in terms of which the prices q_t^0 are stated is; we are free to re-normalize them by multiplying all of them by a positive scalar, say $\lambda > 0$.

Units of q_t^0 could be set so that they are

$$\frac{\text{number of time 0 goods}}{\text{number of time t goods}}$$

In this case, we would be taking the time 0 consumption good to be the **numeraire**.

71.5 Firm Problem

At time t a representative firm hires labor $\tilde{n}_t$ and capital $\tilde{k}_t$.

The firm's profits at time t are

$$F(\tilde{k}_t, \tilde{n}_t) - w_t \tilde{n}_t - \eta_t \tilde{k}_t$$

where w_t is a wage rate at t and η_t is the rental rate on capital at t.

As in the planned economy model

$$F(\tilde{k}_t, \tilde{n}_t) = A \tilde{k}_t^{\alpha} \tilde{n}_t^{1-\alpha}$$

71.5.1 Zero Profit Conditions

Zero-profits conditions for capital and labor are

$$F_k(\tilde{k}_t, \tilde{n}_t) = \eta_t$$

and

$$F_n(\tilde{k}_t, \tilde{n}_t) = w_t \tag{71.1}$$

These conditions emerge from a no-arbitrage requirement.

To describe this no-arbitrage profits reasoning, we begin by applying a theorem of Euler about linearly homogenous functions.

The theorem applies to the Cobb-Douglas production function because we it displays constant returns to scale:

$$\alpha F(\tilde{k}_t, \tilde{n}_t) = F(\alpha \tilde{k}_t, \alpha \tilde{n}_t)$$

for $\alpha \in (0, 1)$.

Taking partial derivatives $\frac{\partial}{\partial \alpha}$ on both sides of the above equation gives

$$F(\tilde{k}_t, \tilde{n}_t) = \frac{\partial F}{\partial \tilde{k}_t} \tilde{k}_t + \frac{\partial F}{\partial \tilde{n}_t} \tilde{n}_t$$

Rewrite the firm's profits as

$$\frac{\partial F}{\partial \tilde{k}_t} \tilde{k}_t + \frac{\partial F}{\partial \tilde{n}_t} \tilde{n}_t - w_t \tilde{n}_t - \eta_t k_t$$

or

$$\left(\frac{\partial F}{\partial \tilde{k}_t} - \eta_t \right) \tilde{k}_t + \left(\frac{\partial F}{\partial \tilde{n}_t} - w_t \right) \tilde{n}_t$$

Because F is homogeneous of degree 1, it follows that $\frac{\partial F}{\partial \tilde{k}_t}$ and $\frac{\partial F}{\partial \tilde{n}_t}$ are homogeneous of degree 0 and therefore fixed with respect to $\tilde{k}_t$ and $\tilde{n}_t$.

If $\frac{\partial F}{\partial \tilde{k}_t} > \eta_t$, then the firm makes positive profits on each additional unit of $\tilde{k}_t$, so it would want to make $\tilde{k}_t$ arbitrarily large.

But setting $\tilde{k}_t = +\infty$ is not physically feasible, so **equilibrium** prices must take values that present the firm with no such arbitrage opportunity.

A similar argument applies if $\frac{\partial F}{\partial \tilde{n}_t} > w_t$.

If $\frac{\partial \tilde{k}_t}{\partial \tilde{k}_t} < \eta_t$, the firm would want to set $\tilde{k}_t$ to zero, which is not feasible.

It is convenient to define $\vec{w} = \{w_0, \dots, w_T\}$ and $\vec{\eta} = \{\eta_0, \dots, \eta_T\}$.

71.6 Household Problem

A representative household lives at $t = 0, 1, \dots, T$.

At t, the household rents 1 unit of labor and k_t units of capital to a firm and receives income

$$w_t 1 + \eta_t k_t$$

At t the household allocates its income to the following purchases between the following two categories:

- consumption c_t
- net investment $k_{t+1} - (1-\delta)k_t$

Here $(k_{t+1} - (1-\delta)k_t)$ is the household's net investment in physical capital and $\delta \in (0,1)$ is again a depreciation rate of capital.

In period t, the consumer is free to purchase more goods to be consumed and invested in physical capital than its income from supplying capital and labor to the firm, provided that in some other periods its income exceeds its purchases.

A consumer's net excess demand for time t consumption goods is the gap

$$e_t \equiv (c_t + (k_{t+1} - (1-\delta)k_t)) - (w_t 1 + \eta_t k_t)$$

Let $\vec{c} = \{c_0, \dots, c_T\}$ and let $\vec{k} = \{k_1, \dots, k_{T+1}\}$.

k_0 is given to the household.

The household faces a **single** budget constraint that requires that the present value of the household's net excess demands must be zero:

$$\sum_{t=0}^{T} q_t^0 e_t \le 0$$

or

$$\sum_{t=0}^{T} q_t^0 \left(c_t + (k_{t+1} - (1-\delta)k_t)\right) \le \sum_{t=0}^{T} q_t^0 (w_t 1 + \eta_t k_t)$$

The household faces price system $\{q_t^0, w_t, \eta_t\}$ as a price-taker and chooses an allocation to solve the constrained optimization problem:

$$\begin{aligned} & \max_{\vec{c}, \vec{k}} \sum_{t=0}^{T} \beta^t u(c_t) \\ \text{subject to} \ & \sum_{t=0}^{T} q_t^0 \left(c_t + (k_{t+1} - (1-\delta)k_t) - (w_t - \eta_t k_t)\right) \le 0 \end{aligned}$$

Components of a **price system** have the following units:

- w_t is measured in units of the time t good per unit of time t labor hired
- η_t is measured in units of the time t good per unit of time t capital hired
- q_t^0 is measured in units of a numeraire per unit of the time t good

71.6.1 Definitions

- A **price system** is a sequence $\{q_t^0, \eta_t, w_t\}_{t=0}^T = \{\vec{q}, \vec{\eta}, \vec{w}\}$.
- An **allocation** is a sequence $\{c_t, k_{t+1}, n_t = 1\}_{t=0}^T = \{\vec{c}, \vec{k}, \vec{n}\}$.
- A **competitive equilibrium** is a price system and an allocation with the following properties:
  - Given the price system, the allocation solves the household's problem.
  - Given the price system, the allocation solves the firm's problem.

The vision here is that an equilibrium price system and allocation are determined once and for all.

In effect, we imagine that all trades occur just before time 0.

71.7 Computing a Competitive Equilibrium

We compute a competitive equilibrium by using a **guess and verify** approach.

- We **guess** equilibrium price sequences $\{\vec{q}, \vec{\eta}, \vec{w}\}$.
- We then **verify** that at those prices, the household and the firm choose the same allocation.

71.7.1 Guess for Price System

In this lecture *Cass-Koopmans Planning Model*, we computed an allocation $\{\vec{C}, \vec{K}, \vec{N}\}$ that solves a planning problem.

We use that allocation to construct a guess for the equilibrium price system.

> **Note**
>
> This allocation will constitute the **Big** K to be in the present instance of the **Big** K **, little** k trick that we'll apply to a competitive equilibrium in the spirit of this lecture and this lecture.

In particular, we shall use the following procedure:

- obtain first-order conditions for the representative firm and the representative consumer.
- from these equations, obtain a new set of equations by replacing the firm's choice variables $\tilde{k}, \tilde{n}$ and the consumer's choice variables with the quantities $\vec{C}, \vec{K}$ that solve the planning problem.
- solve the resulting equations for $\{\vec{q}, \vec{\eta}, \vec{w}\}$ as functions of $\vec{C}, \vec{K}$.
- verify that at these prices, $c_t = C_t, k_t = \tilde{k}_t = K_t, \tilde{n}_t = 1$ for $t = 0, 1, \dots, T$.

Thus, we guess that for $t = 0, \dots, T$:

$$q_t^0 = \beta^t u'(C_t) \tag{71.2}$$

$$w_t = f(K_t) - K_t f'(K_t) \tag{71.3}$$

$$\eta_t = f'(K_t) \tag{71.4}$$

At these prices, let capital chosen by the household be

$$k_t^*(\vec{q}, \vec{w}, \vec{\eta}), \quad t \geq 0 \tag{71.5}$$

and let the allocation chosen by the firm be

$$\tilde{k}_t^*(\vec{q}, \vec{w}, \vec{\eta}), \quad t \geq 0$$

and so on.

If our guess for the equilibrium price system is correct, then it must occur that

$$k_t^* = \tilde{k}_t^* \tag{71.6}$$

$$1 = \tilde{n}_t^* \tag{71.7}$$

$$c_t^* + k_{t+1}^* - (1-\delta)k_t^* = F(\tilde{k}_t^*, \tilde{n}_t^*)$$

We shall verify that for $t = 0, \dots, T$ allocations chosen by the household and the firm both equal the allocation that solves the planning problem:

$$k_t^* = \tilde{k}_t^* = K_t, \tilde{n}_t = 1, c_t^* = C_t \tag{71.8}$$

71.7.2 Verification Procedure

Our approach is firsts to stare at first-order necessary conditions for optimization problems of the household and the firm.

At the price system we have guessed, we'll then verify that both sets of first-order conditions are satisfied at the allocation that solves the planning problem.

71.7.3 Household's Lagrangian

To solve the household's problem, we formulate the Lagrangian

$$\mathcal{L}(\vec{c},\vec{k},\lambda) = \sum_{t=0}^{T} \beta^t u(c_t) + \lambda \left(\sum_{t=0}^{T} q_t^0 \left(((1-\delta)k_t - w_t) + \eta_t k_t - c_t - k_{t+1} \right) \right)$$

and attack the min-max problem:

$$\min_{\lambda} \max_{\vec{c},\vec{k}} \mathcal{L}(\vec{c},\vec{k},\lambda)$$

First-order conditions are

$$c_t: \quad \beta^t u'(c_t) - \lambda q_t^0 = 0 \quad t = 0, 1, \dots, T \tag{71.9}$$

$$k_t: \quad -\lambda q_t^0 \left[(1-\delta) + \eta_t \right] + \lambda q_{t-1}^0 = 0 \quad t = 1, 2, \dots, T+1 \tag{71.10}$$

$$\lambda: \quad \left(\sum_{t=0}^{T} q_t^0 \left(c_t + (k_{t+1} - (1-\delta)k_t) - w_t - \eta_t k_t \right) \right) \leq 0 \tag{71.11}$$

$$k_{T+1}: \quad -\lambda q_0^{T+1} \leq 0, \ \leq 0 \text{ if } k_{T+1} = 0; \ = 0 \text{ if } k_{T+1} > 0 \tag{71.12}$$

Now we plug in our guesses of prices and do some algebra in the hope of recovering all first-order necessary conditions (70.9)-(70.12) for the planning problem from this lecture *Cass-Koopmans Planning Model*.

Combining (71.9) and (71.2), we get:

$$u'(C_t) = \mu_t$$

which is (70.9).

Combining (71.10), (71.2), and (71.4), we get:

$$-\lambda \beta^t \mu_t \left[(1-\delta) + f'(K_t) \right] + \lambda \beta^{t-1} \mu_{t-1} = 0 \tag{71.13}$$

Rewriting (71.13) by dividing by λ on both sides (which is nonzero since u'>0) we get:

$$\beta^t \mu_t [(1-\delta + f'(K_t)] = \beta^{t-1} \mu_{t-1}$$

or

$$\beta \mu_t [(1-\delta + f'(K_t)] = \mu_{t-1}$$

which is (70.10).

Combining (71.11), (71.2), (71.3) and (71.4) after multiplying both sides of (71.11) by λ, we get

$$\sum_{t=0}^{T} \beta^t \mu_t \left(C_t + (K_{t+1} - (1-\delta)K_t) - f(K_t) + K_t f'(K_t) - f'(K_t)K_t \right) \leq 0$$

which simplifies to

$$\sum_{t=0}^{T} \beta^t \mu_t \left(C_t + K_{t+1} - (1-\delta)K_t - F(K_t, 1)\right) \leq 0$$

Since $\beta^t \mu_t > 0$ for $t = 0, \ldots, T$, it follows that

$$C_t + K_{t+1} - (1-\delta)K_t - F(K_t, 1) = 0 \quad \text{for all } t \text{ in } \{0, 1, \ldots, T\}$$

which is (70.11).

Combining (71.12) and (71.2), we get:

$$-\beta^{T+1}\mu_{T+1} \leq 0$$

Dividing both sides by β^{T+1} gives

$$-\mu_{T+1} \leq 0$$

which is (70.12) for the planning problem.

Thus, at our guess of the equilibrium price system, the allocation that solves the planning problem also solves the problem faced by a representative household living in a competitive equilibrium.

71.7.4 Representative Firm's Problem

We now turn to the problem faced by a firm in a competitive equilibrium:

If we plug (71.8) into (71.1) for all t, we get

$$\frac{\partial F(K_t, 1)}{\partial K_t} = f'(K_t) = \eta_t$$

which is (71.4).

If we now plug (71.8) into (71.1) for all t, we get:

$$\frac{\partial F(\tilde{K}_t, 1)}{\partial \tilde{L}_t} = f(K_t) - f'(K_t)K_t = w_t$$

which is exactly (71.5).

Thus, at our guess for the equilibrium price system, the allocation that solves the planning problem also solves the problem faced by a firm within a competitive equilibrium.

By (71.6) and (71.7) this allocation is identical to the one that solves the consumer's problem.

Note

Because budget sets are affected only by relative prices, $\{q_t^0\}$ is determined only up to multiplication by a positive constant.

Normalization: We are free to choose a $\{q_t^0\}$ that makes $\lambda = 1$ so that we are measuring q_t^0 in units of the marginal utility of time 0 goods.

We will plot q, w, η below to show these equilibrium prices induce the same aggregate movements that we saw earlier in the planning problem.

To proceed, we bring in Python code that *Cass-Koopmans Planning Model* used to solve the planning problem

First let's define a `jitclass` that stores parameters and functions the characterize an economy.

```
planning_data = [
    ('γ', float64),    # Coefficient of relative risk aversion
    ('β', float64),    # Discount factor
    ('δ', float64),    # Depreciation rate on capital
    ('α', float64),    # Return to capital per capita
    ('A', float64)     # Technology
]
```

```
@jitclass(planning_data)
class PlanningProblem():

    def __init__(self, γ=2, β=0.95, δ=0.02, α=0.33, A=1):

        self.γ, self.β = γ, β
        self.δ, self.α, self.A = δ, α, A

    def u(self, c):
        '''
        Utility function
        ASIDE: If you have a utility function that is hard to solve by hand
        you can use automatic or symbolic differentiation
        See https://github.com/HIPS/autograd
        '''
        γ = self.γ

        return c ** (1 - γ) / (1 - γ) if γ!= 1 else np.log(c)

    def u_prime(self, c):
        'Derivative of utility'
        γ = self.γ

        return c ** (-γ)

    def u_prime_inv(self, c):
        'Inverse of derivative of utility'
        γ = self.γ

        return c ** (-1 / γ)

    def f(self, k):
        'Production function'
        α, A = self.α, self.A

        return A * k ** α

    def f_prime(self, k):
        'Derivative of production function'
        α, A = self.α, self.A

        return α * A * k ** (α - 1)

    def f_prime_inv(self, k):
        'Inverse of derivative of production function'
        α, A = self.α, self.A

        return (k / (A * α)) ** (1 / (α - 1))
```

(continues on next page)

(continued from previous page)

```
    def next_k_c(self, k, c):
        ''''
        Given the current capital Kt and an arbitrary feasible
        consumption choice Ct, computes Kt+1 by state transition law
        and optimal Ct+1 by Euler equation.
        '''
        β, δ = self.β, self.δ
        u_prime, u_prime_inv = self.u_prime, self.u_prime_inv
        f, f_prime = self.f, self.f_prime

        k_next = f(k) + (1 - δ) * k - c
        c_next = u_prime_inv(u_prime(c) / (β * (f_prime(k_next) + (1 - δ))))

        return k_next, c_next
```

```
@jit
def shooting(pp, c0, k0, T=10):
    '''
    Given the initial condition of capital k0 and an initial guess
    of consumption c0, computes the whole paths of c and k
    using the state transition law and Euler equation for T periods.
    '''
    if c0 > pp.f(k0):
        print("initial consumption is not feasible")

        return None

    # initialize vectors of c and k
    c_vec = np.empty(T+1)
    k_vec = np.empty(T+2)

    c_vec[0] = c0
    k_vec[0] = k0

    for t in range(T):
        k_vec[t+1], c_vec[t+1] = pp.next_k_c(k_vec[t], c_vec[t])

    k_vec[T+1] = pp.f(k_vec[T]) + (1 - pp.δ) * k_vec[T] - c_vec[T]

    return c_vec, k_vec
```

```
@jit
def bisection(pp, c0, k0, T=10, tol=1e-4, max_iter=500, k_ter=0, verbose=True):

    # initial boundaries for guess c0
    c0_upper = pp.f(k0)
    c0_lower = 0

    i = 0
    while True:
        c_vec, k_vec = shooting(pp, c0, k0, T)
        error = k_vec[-1] - k_ter

        # check if the terminal condition is satisfied
        if np.abs(error) < tol:
            if verbose:
```

(continues on next page)

(continued from previous page)

```
                print('Converged successfully on iteration ', i+1)
            return c_vec, k_vec

        i += 1
        if i == max_iter:
            if verbose:
                print('Convergence failed.')
            return c_vec, k_vec

        # if iteration continues, updates boundaries and guess of c0
        if error > 0:
            c0_lower = c0
        else:
            c0_upper = c0

        c0 = (c0_lower + c0_upper) / 2
```

```
pp = PlanningProblem()

# Steady states
ρ = 1 / pp.β - 1
k_ss = pp.f_prime_inv(ρ+pp.δ)
c_ss = pp.f(k_ss) - pp.δ * k_ss
```

The above code from this lecture *Cass-Koopmans Planning Model* lets us compute an optimal allocation for the planning problem.

- from the preceding analysis, we know that it will also be an allocation associated with a competitive equilibium.

Now we're ready to bring in Python code that we require to compute additional objects that appear in a competitive equilibrium.

```
@jit
def q(pp, c_path):
    # Here we choose numeraire to be u'(c_0) -- this is q^(t_0)_t
    T = len(c_path) - 1
    q_path = np.ones(T+1)
    q_path[0] = 1
    for t in range(1, T+1):
        q_path[t] = pp.β ** t * pp.u_prime(c_path[t])
    return q_path

@jit
def w(pp, k_path):
    w_path = pp.f(k_path) - k_path * pp.f_prime(k_path)
    return w_path

@jit
def η(pp, k_path):
    η_path = pp.f_prime(k_path)
    return η_path
```

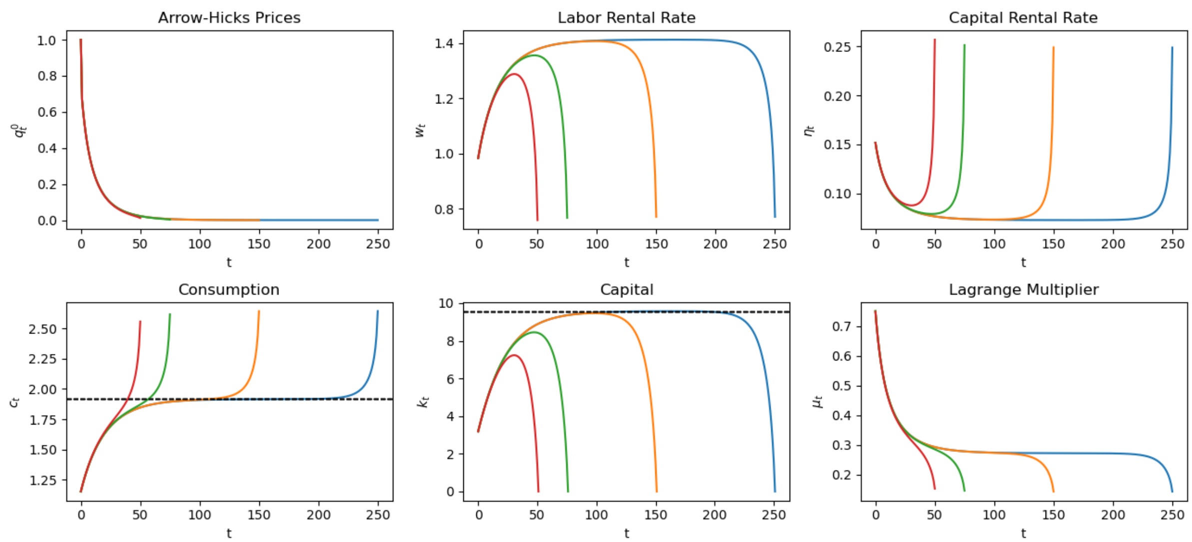
Now we calculate and plot for each T

```
T_arr = [250, 150, 75, 50]

fix, axs = plt.subplots(2, 3, figsize=(13, 6))
```

(continues on next page)

(continued from previous page)

```
titles = ['Arrow-Hicks Prices', 'Labor Rental Rate', 'Capital Rental Rate',
          'Consumption', 'Capital', 'Lagrange Multiplier']
ylabels = ['$q_t^0$', '$w_t$', r'$\eta_t$', '$c_t$', '$k_t$', r'$\mu_t$']

for T in T_arr:
    c_path, k_path = bisection(pp, 0.3, k_ss/3, T, verbose=False)
    μ_path = pp.u_prime(c_path)

    q_path = q(pp, c_path)
    w_path = w(pp, k_path)[:-1]
    η_path = η(pp, k_path)[:-1]
    paths = [q_path, w_path, η_path, c_path, k_path, μ_path]

    for i, ax in enumerate(axs.flatten()):
        ax.plot(paths[i])
        ax.set(title=titles[i], ylabel=ylabels[i], xlabel='t')
        if titles[i] == 'Capital':
            ax.axhline(k_ss, lw=1, ls='--', c='k')
        if titles[i] == 'Consumption':
            ax.axhline(c_ss, lw=1, ls='--', c='k')

plt.tight_layout()
plt.show()
```



Varying Curvature

Now we see how our results change if we keep T constant, but allow the curvature parameter, γ to vary, starting with K_0 below the steady state.

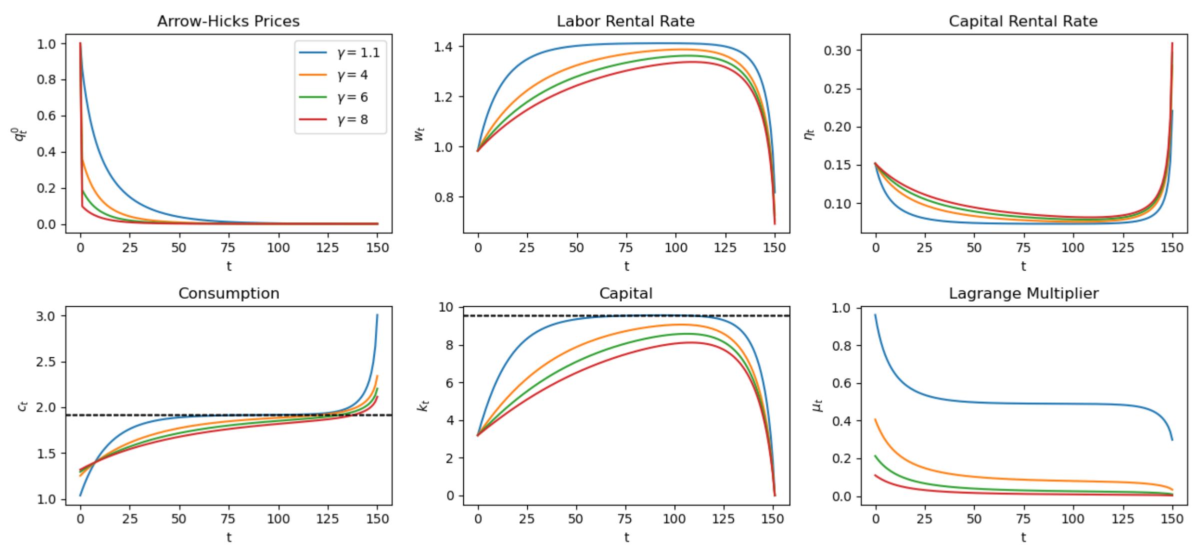
We plot the results for $T = 150$

```
T = 150
γ_arr = [1.1, 4, 6, 8]

fix, axs = plt.subplots(2, 3, figsize=(13, 6))
```

(continues on next page)

(continued from previous page)

```
for γ in γ_arr:
    pp_γ = PlanningProblem(γ=γ)
    c_path, k_path = bisection(pp_γ, 0.3, k_ss/3, T, verbose=False)
    μ_path = pp_γ.u_prime(c_path)

    q_path = q(pp_γ, c_path)
    w_path = w(pp_γ, k_path)[:-1]
    η_path = η(pp_γ, k_path)[:-1]
    paths = [q_path, w_path, η_path, c_path, k_path, μ_path]

    for i, ax in enumerate(axs.flatten()):
        ax.plot(paths[i], label=fr'$\gamma = {γ}$')
        ax.set(title=titles[i], ylabel=ylabels[i], xlabel='t')
        if titles[i] == 'Capital':
            ax.axhline(k_ss, lw=1, ls='--', c='k')
        if titles[i] == 'Consumption':
            ax.axhline(c_ss, lw=1, ls='--', c='k')

axs[0, 0].legend()
plt.tight_layout()
plt.show()
```



Adjusting γ means adjusting how much individuals prefer to smooth consumption.

Higher γ means individuals prefer to smooth more resulting in slower convergence to a steady state allocation.

Lower γ means individuals prefer to smooth less, resulting in faster convergence to a steady state allocation.

71.8 Yield Curves and Hicks-Arrow Prices

We return to Hicks-Arrow prices and calculate how they are related to **yields** on loans of alternative maturities.

This will let us plot a **yield curve** that graphs yields on bonds of maturities $j = 1, 2, \ldots$ against $j = 1, 2, \ldots$.

We use the following formulas.

A **yield to maturity** on a loan made at time t_0 that matures at time $t > t_0$

$$r_{t_0,t} = -\frac{\log q_t^{t_0}}{t - t_0}$$

A Hicks-Arrow price system for a base-year $t_0 \leq t$ satisfies

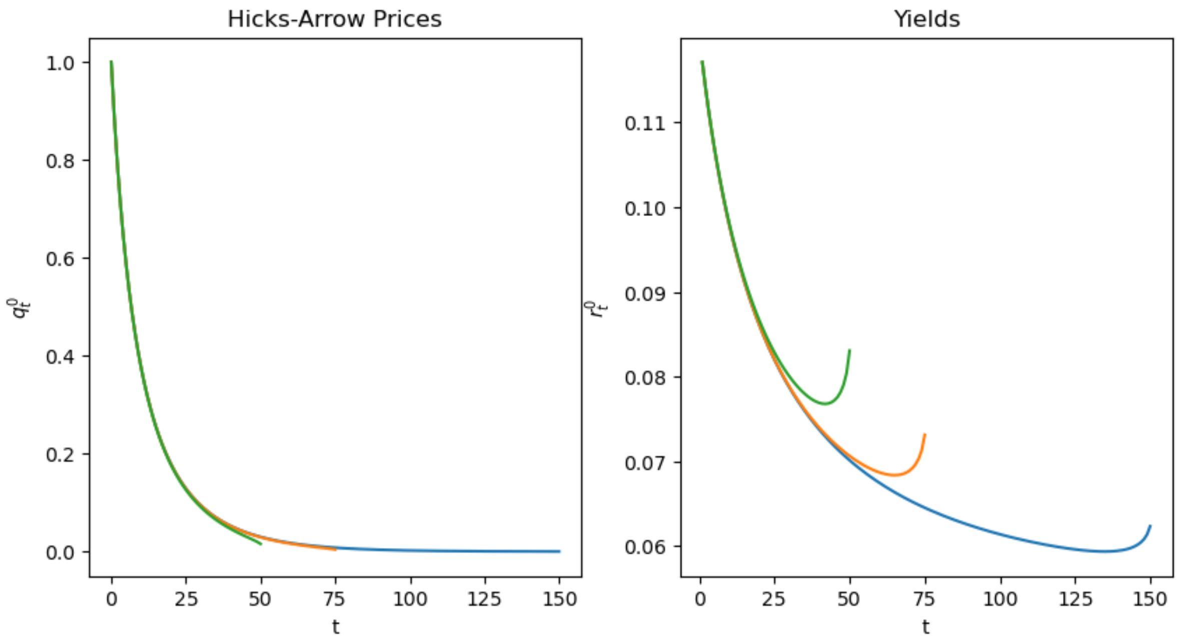
$$q_t^{t_0} = \beta^{t-t_0} \frac{u'(c_t)}{u'(c_{t_0})} = \beta^{t-t_0} \frac{c_t^{-\gamma}}{c_{t_0}^{-\gamma}}$$

We redefine our function for q to allow arbitrary base years, and define a new function for r, then plot both.

We begin by continuing to assume that $t_0 = 0$ and plot things for different maturities $t = T$, with K_0 below the steady state

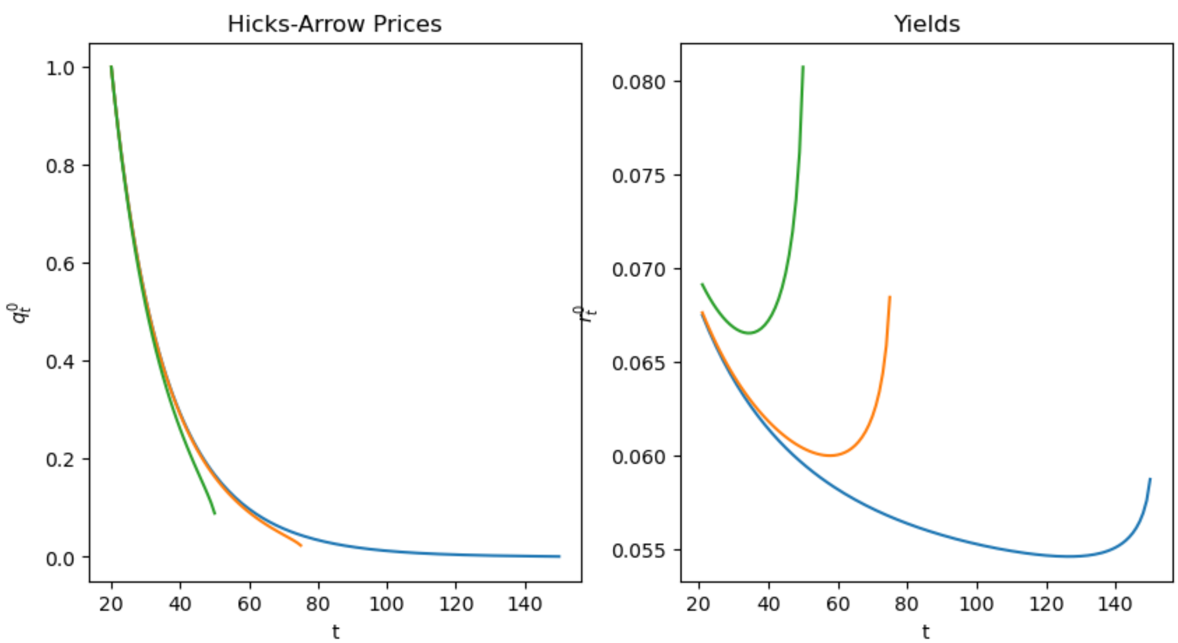
```
@jit
def q_generic(pp, t0, c_path):
    # simplify notations
    β = pp.β
    u_prime = pp.u_prime

    T = len(c_path) - 1
    q_path = np.zeros(T+1-t0)
    q_path[0] = 1
    for t in range(t0+1, T+1):
        q_path[t-t0] = β ** (t-t0) * u_prime(c_path[t]) / u_prime(c_path[t0])
    return q_path

@jit
def r(pp, t0, q_path):
    '''Yield to maturity'''
    r_path = - np.log(q_path[1:]) / np.arange(1, len(q_path))
    return r_path

def plot_yield_curves(pp, t0, c0, k0, T_arr):

    fig, axs = plt.subplots(1, 2, figsize=(10, 5))

    for T in T_arr:
        c_path, k_path = bisection(pp, c0, k0, T, verbose=False)
        q_path = q_generic(pp, t0, c_path)
        r_path = r(pp, t0, q_path)

        axs[0].plot(range(t0, T+1), q_path)
        axs[0].set(xlabel='t', ylabel='$q_t^0$', title='Hicks-Arrow Prices')

        axs[1].plot(range(t0+1, T+1), r_path)
        axs[1].set(xlabel='t', ylabel='$r_t^0$', title='Yields')
```

```
T_arr = [150, 75, 50]
plot_yield_curves(pp, 0, 0.3, k_ss/3, T_arr)
```



Now we plot when $t_0 = 20$

```
plot_yield_curves(pp, 20, 0.3, k_ss/3, T_arr)
```



CHAPTER

SEVENTYTWO

CASS-KOOPMANS MODEL WITH DISTORTING TAXES

72.1 Overview

This lecture studies effects of foreseen fiscal and technology shocks on competitive equilibrium prices and quantities in a nonstochastic version of a Cass-Koopmans growth model with features described in this QuantEcon lecture *Cass-Koopmans Competitive Equilibrium*.

This model is discussed in more detail in chapter 11 of [Ljungqvist and Sargent, 2018].

We use the model as a laboratory to experiment with numerical techniques for approximating equilibria and to display the structure of dynamic models in which decision makers have perfect foresight about future government decisions.

Following a classic paper by Robert E. Hall [Hall, 1971], we augment a nonstochastic version of the Cass-Koopmans optimal growth model with a government that purchases a stream of goods and that finances its purchases with an sequences of several distorting flat-rate taxes.

Distorting taxes prevent a competitive equilibrium allocation from solving a planning problem.

Therefore, to compute an equilibrium allocation and price system, we solve a system of nonlinear difference equations consisting of the first-order conditions for decision makers and the other equilibrium conditions.

We present two ways to approximate an equilibrium:

- The first is a shooting algorithm like the one that we deployed in *Cass-Koopmans Competitive Equilibrium*.
- The second method is a root-finding algorithm that minimizes residuals from the first-order conditions of the consumer and representative firm.

72.2 The Economy

72.2.1 Technology

Feasible allocations satisfy

$$g_t + c_t + x_t \leq F(k_t, n_t), \tag{72.1}$$

where

- g_t is government purchases of the time t good
- x_t is gross investment, and
- $F(k_t, n_t)$ is a linearly homogeneous production function with positive and decreasing marginal products of capital k_t and labor n_t.

Physical capital evolves according to

$$k_{t+1} = (1-\delta)k_t + x_t,$$

where $\delta \in (0,1)$ is a depreciation rate.

It is sometimes convenient to eliminate x_t from (72.1) and to represent it as as

$$g_t + c_t + k_{t+1} \leq F(k_t, n_t) + (1-\delta)k_t.$$

72.2.2 Components of a competitive equilibrium

All trades occurring at time 0.

The representative household owns capital, makes investment decisions, and rents capital and labor to a representative production firm.

The representative firm uses capital and labor to produce goods with the production function $F(k_t, n_t)$.

A **price system** is a triple of sequences $\{q_t, \eta_t, w_t\}_{t=0}^{\infty}$, where

- q_t is the time 0 pretax price of one unit of investment or consumption at time t (x_t or c_t),
- η_t is the pretax price at time t that the household receives from the firm for renting capital at time t, and
- w_t is the pretax price at time t that the household receives for renting labor to the firm at time t.

The prices w_t and η_t are expressed in terms of time t goods, while q_t is expressed in terms of a numeraire at time 0, as in *Cass-Koopmans Competitive Equilibrium*.

The presence of a government distinguishes this lecture from *Cass-Koopmans Competitive Equilibrium*.

Government purchases of goods at time t are $g_t \geq 0$.

A government expenditure plan is a sequence $g = \{g_t\}_{t=0}^{\infty}$.

A government tax plan is a 4-tuple of sequences $\{\tau_{ct}, \tau_{kt}, \tau_{nt}, \tau_{ht}\}_{t=0}^{\infty}$, where

- τ_{ct} is a tax rate on consumption at time t,
- τ_{kt} is a tax rate on rentals of capital at time t,
- τ_{nt} is a tax rate on wage earnings at time t, and
- τ_{ht} is a lump sum tax on a consumer at time t.

Because lump-sum taxes τ_{ht} are available, the government actually should not use any distorting taxes.

Nevertheless, we include all of these taxes because, like [Hall, 1971], they allow us to analyze how various taxes distort production and consumption decisions.

In the *experiment section*, we shall see how variations in government tax plan affect the transition path and equilibrium.

72.2.3 Representative Household

A representative household has preferences over nonnegative streams of a single consumption good c_t and leisure $1-n_t$ that are ordered by:

$$\sum_{t=0}^{\infty} \beta^t U(c_t, 1-n_t), \quad \beta \in (0,1), \tag{72.2}$$

where

- U is strictly increasing in c_t, twice continuously differentiable, and strictly concave with $c_t \geq 0$ and $n_t \in [0, 1]$.

The representative hßousehold maximizes (72.2) subject to the single budget constraint:

$$\begin{aligned}\sum_{t=0}^{\infty} q_t \left\{ (1+\tau_{ct})c_t + \underbrace{[k_{t+1} - (1-\delta)k_t]}_{\text{no tax when investing}} \right\} \\ \leq \sum_{t=0}^{\infty} q_t \left\{ \eta_t k_t - \underbrace{\tau_{kt}(\eta_t - \delta)k_t}_{\text{tax on rental return}} + (1-\tau_{nt})w_t n_t - \tau_{ht} \right\}.\end{aligned} \tag{72.3}$$

Here we have assumed that the government gives a depreciation allowance δk_t from the gross rentals on capital $\eta_t k_t$ and so collects taxes $\tau_{kt}(\eta_t - \delta)k_t$ on rentals from capital.

72.2.4 Government

Government plans $\{g_t\}_{t=0}^{\infty}$ for government purchases and taxes $\{\tau_{ct}, \tau_{kt}, \tau_{nt}, \tau_{ht}\}_{t=0}^{\infty}$ must respect the budget constraint

$$\sum_{t=0}^{\infty} q_t g_t \leq \sum_{t=0}^{\infty} q_t \left\{ \tau_{ct}c_t + \tau_{kt}(\eta_t - \delta)k_t + \tau_{nt}w_t n_t + \tau_{ht} \right\}. \tag{72.4}$$

Given a budget-feasible government policy $\{g_t\}_{t=0}^{\infty}$ and $\{\tau_{ct}, \tau_{kt}, \tau_{nt}, \tau_{ht}\}_{t=0}^{\infty}$ subject to (72.4),

- **Household** chooses $\{c_t\}_{t=0}^{\infty}$, $\{n_t\}_{t=0}^{\infty}$, and $\{k_{t+1}\}_{t=0}^{\infty}$ to maximize utility(72.2) subject to budget constraint(72.3), and
- **Firm** chooses sequences of capital $\{k_t\}_{t=0}^{\infty}$ and $\{n_t\}_{t=0}^{\infty}$ to maximize profits

$$\sum_{t=0}^{\infty} q_t [F(k_t, n_t) - \eta_t k_t - w_t n_t] \tag{72.5}$$

- A **feasible allocation** is a sequence $\{c_t, x_t, n_t, k_t\}_{t=0}^{\infty}$ that satisfies feasibility condition (72.1).

72.3 Equilibrium

Definition 72.3.1

A **competitive equilibrium with distorting taxes** is a **budget-feasible government policy**, **a feasible allocation**, and **a price system** for which, given the price system and the government policy, the allocation solves the household's problem and the firm's problem.

72.4 No-arbitrage Condition

A no-arbitrage argument implies a restriction on prices and tax rates across time.

By rearranging (72.3) and group k_t at the same t, we can get

$$\begin{aligned}\sum_{t=0}^{\infty} q_t \left[(1+\tau_{ct})c_t\right] \leq \sum_{t=0}^{\infty} q_t (1-\tau_{nt}) w_t n_t - \sum_{t=0}^{\infty} q_t \tau_{ht} \\ + \sum_{t=1}^{\infty} \left\{ \left[(1-\tau_{kt})(\eta_t - \delta) + 1\right] q_t - q_{t-1} \right\} k_t \\ + \left[(1-\tau_{k0})(\eta_0 - \delta) + 1\right] q_0 k_0 - \lim_{T\to\infty} q_T k_{T+1}\end{aligned} \tag{72.6}$$

The household inherits a given k_0 that it takes as initial condition and is free to choose $\{c_t, n_t, k_{t+1}\}_{t=0}^{\infty}$.

The household's budget constraint (72.3) must be bounded in equilibrium due to finite resources.

This imposes a restriction on price and tax sequences.

Specifically, for $t \geq 1$, the terms multiplying k_t must equal zero.

If they were strictly positive (negative), the household could arbitrarily increase (decrease) the right-hand side of (72.3) by selecting an arbitrarily large positive (negative) k_t, leading to unbounded profit or arbitrage opportunities:

- For strictly positive terms, the household could purchase large capital stocks k_t and profit from their rental services and undepreciated value.
- For strictly negative terms, the household could engage in "short selling" synthetic units of capital. Both cases would make (72.3) unbounded.

Hence, by setting the terms multiplying k_t to 0 we have the non-arbitrage condition:

$$\frac{q_t}{q_{t+1}} = \left[(1 - \tau_{kt+1})(\eta_{t+1} - \delta) + 1\right]. \tag{72.7}$$

Moreover, we have terminal condition:

$$-\lim_{T \to \infty} q_T k_{T+1} = 0. \tag{72.8}$$

Zero-profit conditions for the representative firm impose additional restrictions on equilibrium prices and quantities.

The present value of the firm's profits is

$$\sum_{t=0}^{\infty} q_t \left[F(k_t, n_t) - w_t n_t - \eta_t k_t\right].$$

Applying Euler's theorem on linearly homogeneous functions to $F(k, n)$, the firm's present value is:

$$\sum_{t=0}^{\infty} q_t \left[(F_{kt} - \eta_t) k_t + (F_{nt} - w_t) n_t\right].$$

No-arbitrage (or zero-profit) conditions are:

$$\eta_t = F_{kt}, \quad w_t = F_{nt}. \tag{72.9}$$

72.5 Household's First Order Condition

Household maximize (72.2) under (72.3).

Let $U_1 = \frac{\partial U}{\partial c}, U_2 = \frac{\partial U}{\partial (1-n)} = -\frac{\partial U}{\partial n}$., we can derive FOC from the Lagrangian

$$\mathcal{L} = \sum_{t=0}^{\infty} \beta^t U(c_t, 1 - n_t) + \mu \left(\sum_{t=0}^{\infty} q_t \left[(1 + \tau_{ct}) c_t - (1 - \tau_{nt}) w_t n_t + \ldots\right] \right),$$

First-order necessary conditions for the representative household's problem are

$$\frac{\partial \mathcal{L}}{\partial c_t} = \beta^t U_1(c_t, 1 - n_t) - \mu q_t (1 + \tau_{ct}) = 0 \tag{72.10}$$

and

$$\frac{\partial \mathcal{L}}{\partial n_t} = \beta^t \left(-U_{2t}(c_t, 1 - n_t)\right) - \mu q_t (1 - \tau_{nt}) w_t = 0 \tag{72.11}$$

Rearranging (72.10) and (72.11), we have

$$\beta^t U_1(c_t, 1-n_t) = \beta^t U_{1t} = \mu q_t (1+\tau_{ct}), \tag{72.12}$$

$$\beta^t U_2(c_t, 1-n_t) = \beta^t U_{2t} = \mu q_t (1-\tau_{nt}) w_t. \tag{72.13}$$

Plugging (72.12) into (72.8) and replacing q_t, we get terminal condition

$$-\lim_{T\to\infty} \beta^T \frac{U_{1T}}{(1+\tau_{cT})} k_{T+1} = 0. \tag{72.14}$$

72.6 Computing Equilibria

To compute an equilibrium, we seek a price system $\{q_t, \eta_t, w_t\}$, a budget feasible government policy $\{g_t, \tau_t\} \equiv \{g_t, \tau_{ct}, \tau_{nt}, \tau_{kt}, \tau_{ht}\}$, and an allocation $\{c_t, n_t, k_{t+1}\}$ that solve a system of nonlinear difference equations consisting of

- feasibility condition (72.1), no-arbitrage condition for household (72.7) and firms (72.9), household's first order conditions (72.12) and (72.13).
- an initial condition k_0 and a terminal condition (72.14).

72.7 Python Code

We require the following imports

```
import numpy as np
from scipy.optimize import root
import matplotlib.pyplot as plt
from collections import namedtuple
from mpmath import mp, mpf
from warnings import warn

# Set the precision
mp.dps = 40
mp.pretty = True
```

We use the `mpmath` library to perform high-precision arithmetic in the shooting algorithm in cases where the solution diverges due to numerical instability.

Note

In the functions below, we include routines to handle the growth component, which will be discussed further in the section *Exogenous growth*.

We include them here to avoid code duplication.

We set the following parameters

```
# Create a namedtuple to store the model parameters
Model = namedtuple("Model",
             ["β", "γ", "δ", "α", "A"])
```

(continues on next page)

(continued from previous page)

```
def create_model(β=0.95, # discount factor
                 γ=2.0,  # relative risk aversion coefficient
                 δ=0.2,  # depreciation rate
                 α=0.33, # capital share
                 A=1.0   # TFP
                 ):
    """Create a model instance."""
    return Model(β=β, γ=γ, δ=δ, α=α, A=A)

model = create_model()

# Total number of periods
S = 100
```

72.7.1 Inelastic Labor Supply

In this lecture, we consider the special case where $U(c, 1-n) = u(c)$ and $f(k) := F(k, 1)$.

We rewrite (72.1) with $f(k) := F(k, 1)$,

$$k_{t+1} = f(k_t) + (1-\delta)k_t - g_t - c_t. \tag{72.15}$$

```
def next_k(k_t, g_t, c_t, model, μ_t=1):
    """
    Capital next period: k_{t+1} = f(k_t) + (1 - δ) * k_t - c_t - g_t
    with optional growth adjustment: k_{t+1} = (f(k_t) + (1 - δ) * k_t - c_t - g_t) /
↪μ_{t+1}
    """
    return (f(k_t, model) + (1 - model.δ) * k_t - g_t - c_t) / μ_t
```

By the properties of a linearly homogeneous production function, we have $F_k(k, n) = f'(k)$ and $F_n(k, 1) = f(k, 1) - f'(k)k$.

Substituting (72.12), (72.9), and (72.15) into (72.7), we obtain the Euler equation

$$\begin{aligned} &\frac{u'(f(k_t) + (1-\delta)k_t - g_t - k_{t+1})}{(1+\tau_{ct})} \\ &- \beta \frac{u'(f(k_{t+1}) + (1-\delta)k_{t+1} - g_{t+1} - k_{t+2})}{(1+\tau_{ct+1})} \\ &\times [(1-\tau_{kt+1})(f'(k_{t+1}) - \delta) + 1] = 0. \end{aligned} \tag{72.16}$$

This can be simplified to:

$$u'(c_t) = \beta u'(c_{t+1}) \frac{(1+\tau_{ct})}{(1+\tau_{ct+1})} [(1-\tau_{kt+1})(f'(k_{t+1}) - \delta) + 1]. \tag{72.17}$$

Equation (72.17) will appear prominently in our equilibrium computation algorithms.

72.7.2 Steady state

Tax rates and government expenditures act as **forcing functions** for difference equations (72.15) and (72.17).

Define $z_t = [g_t, \tau_{kt}, \tau_{ct}]'$.

Represent the second-order difference equation as:

$$H(k_t, k_{t+1}, k_{t+2}; z_t, z_{t+1}) = 0. \tag{72.18}$$

We assume that a government policy reaches a steady state such that $\lim_{t\to\infty} z_t = \bar{z}$ and that the steady state prevails for $t > T$.

The terminal steady-state capital stock $\bar{k}$ satisfies:

$$H(\bar{k}, \bar{k}, \bar{k}, \bar{z}, \bar{z}) = 0.$$

From difference equation (72.17), we can infer a restriction on the steady-state:

$$\begin{aligned} u'(\bar{c}) &= \beta u'(\bar{c}) \frac{(1+\bar{\tau}_c)}{(1+\bar{\tau}_c)} [(1-\bar{\tau}_k)(f'(\bar{k})-\delta)+1]. \\ \implies 1 &= \beta[(1-\bar{\tau}_k)(f'(\bar{k})-\delta)+1]. \end{aligned} \tag{72.19}$$

72.7.3 Other equilibrium quantities and prices

Price:

$$q_t = \frac{\beta^t u'(c_t)}{u'(c_0)} \tag{72.20}$$

```
def compute_q_path(c_path, model, S=100, A_path=None):
    """
    Compute q path: q_t = (β^t * u'(c_t)) / u'(c_0)
    with optional A_path for growth models.
    """
    A = np.ones_like(c_path) if A_path is None else np.asarray(A_path)
    q_path = np.zeros_like(c_path)
    for t in range(S):
        q_path[t] = (model.β ** t *
                     u_prime(c_path[t], model, A[t])) / u_prime(c_path[0], model,␣
↪A[0])
    return q_path
```

Capital rental rate

$$\eta_t = f'(k_t)$$

```
def compute_η_path(k_path, model, S=100, A_path=None):
    """
    Compute η path: η_t = f'(k_t)
    with optional A_path for growth models.
    """
    A = np.ones_like(k_path) if A_path is None else np.asarray(A_path)
    η_path = np.zeros_like(k_path)
    for t in range(S):
        η_path[t] = f_prime(k_path[t], model, A[t])
    return η_path
```

Labor rental rate:

$$w_t = f(k_t) - k_t f'(k_t)$$

```
def compute_w_path(k_path, η_path, model, S=100, A_path=None):
    """
    Compute w path: w_t = f(k_t) - k_t * f'(k_t)
    with optional A_path for growth models.
    """
    A = np.ones_like(k_path) if A_path is None else np.asarray(A_path)
    w_path = np.zeros_like(k_path)
    for t in range(S):
        w_path[t] = f(k_path[t], model, A[t]) - k_path[t] * η_path[t]
    return w_path
```

Gross one-period return on capital:

$$\bar{R}_{t+1} = \frac{(1+\tau_{ct})}{(1+\tau_{ct+1})}\left[(1-\tau_{kt+1})(f'(k_{t+1})-\delta)+1\right] = \frac{(1+\tau_{ct})}{(1+\tau_{ct+1})}R_{t,t+1} \tag{72.21}$$

```
def compute_R_bar(τ_ct, τ_ctp1, τ_ktp1, k_tp1, model):
    """
    Gross one-period return on capital:
    R_bar = [(1 + τ_c_t) / (1 + τ_c_{t+1})]
        * { [1 - τ_k_{t+1}] * [f'(k_{t+1}) - δ] + 1 }
    """
    return ((1 + τ_ct) / (1 + τ_ctp1)) * (
        (1 - τ_ktp1) * (f_prime(k_tp1, model) - model.δ) + 1)
```

```
def compute_R_bar_path(shocks, k_path, model, S=100):
    """
    Compute R_bar path over time.
    """
    R_bar_path = np.zeros(S + 1)
    for t in range(S):
        R_bar_path[t] = compute_R_bar(
            shocks['τ_c'][t], shocks['τ_c'][t + 1], shocks['τ_k'][t + 1],
            k_path[t + 1], model)
    R_bar_path[S] = R_bar_path[S - 1]
    return R_bar_path
```

One-period discount factor:

$$R^{-1}_{t,t+1} = \frac{q_{t+1}}{q_t} = m_{t,t+1} = \beta\frac{u'(c_{t+1})}{u'(c_t)}\frac{(1+\tau_{ct})}{(1+\tau_{ct+1})} \tag{72.22}$$

Net one-period rate of interest:

$$r_{t,t+1} \equiv R_{t,t+1} - 1 = (1-\tau_{k,t+1})(f'(k_{t+1})-\delta) \tag{72.23}$$

By (72.22) and $r_{t,t+1} = -\ln(\frac{q_{t+1}}{q_t})$, we have

$$R_{t,t+s} = e^{s\cdot r_{t,t+s}}.$$

Then by (72.23), we have

$$\frac{q_{t+s}}{q_t} = e^{-s\cdot r_{t,t+s}}.$$

Rearranging the above equation, we have

$$r_{t,t+s} = -\frac{1}{s}\ln\left(\frac{q_{t+s}}{q_t}\right).$$

```
def compute_rts_path(q_path, S, t):
    """
    Compute r path:
    r_t,t+s = - (1/s) * ln(q_{t+s} / q_t)
    """
    s = np.arange(1, S + 1)
    q_path = np.array([float(q) for q in q_path])

    with np.errstate(divide='ignore', invalid='ignore'):
        rts_path = - np.log(q_path[t + s] / q_path[t]) / s
    return rts_path
```

72.8 Some functional forms

We assume that the representative household' period utility has the following CRRA (constant relative risk aversion) form

$$u(c) = \frac{c^{1-\gamma}}{1-\gamma}$$

```
def u_prime(c, model, A_t=1):
    """
    Marginal utility: u'(c) = c^{-γ}
    with optional technology adjustment: u'(cA) = (cA)^{-γ}
    """
    return (c * A_t) ** (-model.γ)
```

By substituting (72.21) into (72.17), we obtain

$$c_{t+1} = c_t\left[\beta\frac{(1+\tau_{ct})}{(1+\tau_{ct+1})}\left[(1-\tau_{k,t+1})(f'(k_{t+1})-\delta)+1\right]\right]^{\frac{1}{\gamma}} = c_t\left[\beta\overline{R}_{t+1}\right]^{\frac{1}{\gamma}} \tag{72.24}$$

```
def next_c(c_t, R_bar, model, μ_t=1):
    """
    Consumption next period: c_{t+1} = c_t * (β * R̄)^{1/γ}
    with optional growth adjustment: c_{t+1} = c_t * (β * R_bar)^{1/γ} * μ_{t+1}^{-1}
    """
    return c_t * (model.β * R_bar) ** (1 / model.γ) / μ_t
```

For the production function we assume a Cobb-Douglas form:

$$F(k,1) = Ak^{\alpha}$$

```
def f(k, model, A=1):
    """
    Production function: f(k) = A * k^{α}
    """
    return A * k ** model.α

def f_prime(k, model, A=1):
```

(continues on next page)

(continued from previous page)

```
    """
    Marginal product of capital: f'(k) = α * A * k^{α - 1}
    """
    return model.α * A * k ** (model.α - 1)
```

72.9 Computation

We describe two ways to compute an equilibrium:

- a shooting algorithm
- a residual-minimization method that focuses on imposing Euler equation (72.17) and the feasibility condition (72.15).

72.9.1 Shooting Algorithm

This algorithm deploys the following steps.

1. Solve the equation (72.19) for the terminal steady-state capital $\bar{k}$ that corresponds to the permanent policy vector $\bar{z}$.
2. Select a large time index $S \gg T$, guess an initial consumption rate c_0, and use the equation (72.15) to solve for k_1.
3. Use the equation (72.24) to determine c_{t+1}. Then, apply the equation (72.15) to compute k_{t+2}.
4. Iterate step 3 to compute candidate values $\hat{k}_t$ for $t = 1, \dots, S$.
5. Compute the difference $\hat{k}_S - \bar{k}$. If $\left|\hat{k}_S - \bar{k}\right| > \epsilon$ for some small ϵ, adjust c_0 and repeat steps 2-5.
6. Adjust c_0 iteratively using the bisection method to find a value that ensures $\left|\hat{k}_S - \bar{k}\right| < \epsilon$.

The following code implements these steps.

```
# Steady-state calculation
def steady_states(model, g_ss, τ_k_ss=0.0, μ_ss=None):
    """
    Calculate steady state values for capital and
    consumption with optional A_path for growth models.
    """

    β, δ, α, γ = model.β, model.δ, model.α, model.γ

    A = model.A or 1.0

    # growth-adjustment in the numerator: μ^γ or 1
    μ_eff = μ_ss**γ if μ_ss is not None else 1.0

    num = δ + (μ_eff/β - 1) / (1 - τ_k_ss)
    k_ss = (num / (α * A)) ** (1 / (α - 1))

    c_ss = (
        A * k_ss**α - δ * k_ss - g_ss
        if μ_ss is None
        else k_ss**α + (1 - δ - μ_ss) * k_ss - g_ss
```

(continues on next page)

(continued from previous page)

```
    )

    return k_ss, c_ss

def shooting_algorithm(
    c0, k0, shocks, S, model, A_path=None):
    """
    Shooting algorithm for given initial c0 and k0
    with optional A_path for growth models.
    """
    # unpack & mpf-ify shocks, fill μ with ones if missing
    g = np.array(list(map(mpf, shocks['g'])), dtype=object)
    τ_c = np.array(list(map(mpf, shocks['τ_c'])), dtype=object)
    τ_k = np.array(list(map(mpf, shocks['τ_k'])), dtype=object)
    μ = (np.array(list(map(mpf, shocks['μ'])), dtype=object)
           if 'μ' in shocks else np.ones_like(g))
    A = np.ones_like(g) if A_path is None else A_path

    k_path = np.empty(S+1, dtype=object)
    c_path = np.empty(S+1, dtype=object)
    k_path[0], c_path[0] = mpf(k0), mpf(c0)

    for t in range(S):
        k_t, c_t = k_path[t], c_path[t]
        k_tp1 = next_k(k_t, g[t], c_t, model, μ[t+1])
        if k_tp1 < 0:
            return None, None
        k_path[t+1] = k_tp1

        R_bar = compute_R_bar(
            τ_c[t], τ_c[t+1], τ_k[t+1], k_tp1, model
        )
        c_tp1 = next_c(c_t, R_bar, model, μ[t+1])
        if c_tp1 < 0:
            return None, None
        c_path[t+1] = c_tp1

    return k_path, c_path


def bisection_c0(
    c0_guess, k0, shocks, S, model, tol=mpf('1e-6'),
    max_iter=1000, verbose=False, A_path=None):
    """
    Bisection method to find initial c0
    """
    # steady-state uses last shocks (μ=1 if missing)
    g_last    = mpf(shocks['g'][-1])
    τ_k_last  = mpf(shocks['τ_k'][-1])
    μ_last    = mpf(shocks['μ'][-1]) if 'μ' in shocks else mpf('1')
    k_ss_fin, _ = steady_states(model, g_last, τ_k_last, μ_last)

    c0_lo, c0_hi = mpf('0'), f(k_ss_fin, model)
    c0 = mpf(c0_guess)

    for i in range(1, max_iter+1):
        k_path, _ = shooting_algorithm(c0, k0, shocks, S, model, A_path)
```

(continues on next page)

(continued from previous page)

```
        if k_path is None:
            if verbose:
                print(f"[{i}] shoot failed at c0={c0}")
            c0_hi = c0
        else:
            err = k_path[-1] - k_ss_fin
            if verbose and i % 100 == 0:
                print(f"[{i}] c0={c0}, err={err}")
            if abs(err) < tol:
                if verbose:
                    print(f"Converged after {i} iter")
                return c0
            # update bounds in one line
            c0_lo, c0_hi = (c0, c0_hi) if err > 0 else (c0_lo, c0)
        c0 = (c0_lo + c0_hi) / mpf('2')

    warn(f"bisection did not converge after {max_iter} iters; returning c0={c0}")
    return c0


def run_shooting(
    shocks, S, model, A_path=None,
    c0_finder=bisection_c0, shooter=shooting_algorithm):
    """
    Compute initial SS, find c0, and return [k,c] paths
    with optional A_path for growth models.
    """
    # initial SS at t=0 (μ=1 if missing)
    g0    = mpf(shocks['g'][0])
    τ_k0  = mpf(shocks['τ_k'][0])
    μ0    = mpf(shocks['μ'][0]) if 'μ' in shocks else mpf('1')
    k0, c0 = steady_states(model, g0, τ_k0, μ0)

    optimal_c0 = c0_finder(c0, k0, shocks, S, model, A_path=A_path)
    print(f"Model: {model}\nOptimal initial consumption c0 = {mpf(optimal_c0)}")

    k_path, c_path = shooter(optimal_c0, k0, shocks, S, model, A_path)
    return np.column_stack([k_path, c_path])
```

72.9.2 Experiments

Let's run some experiments.

1. A foreseen once-and-for-all increase in g from 0.2 to 0.4 occurring in period 10,
2. A foreseen once-and-for-all increase in τ_c from 0.0 to 0.2 occurring in period 10,
3. A foreseen once-and-for-all increase in τ_k from 0.0 to 0.2 occurring in period 10, and
4. A foreseen one-time increase in g from 0.2 to 0.4 in period 10, after which g reverts to 0.2 permanently.

To start, we prepare sequences that we'll used to initialize our iterative algorithm.

We will start from an initial steady state and apply shocks at an the indicated time.

```
def plot_results(
    solution, k_ss, c_ss, shocks, shock_param, axes, model,
```

(continues on next page)

(continued from previous page)

```
        A_path=None, label='', linestyle='-', T=40):
    """
    Plot simulation results (k, c, R, η, and a policy shock)
    with optional A_path for growth models.
    """
    k_path = solution[:, 0]
    c_path = solution[:, 1]
    T = min(T, k_path.size)

    # handle growth parameters
    μ0 = shocks['μ'][0] if 'μ' in shocks else 1.0
    A0 = A_path[0] if A_path is not None else (model.A or 1.0)

    # steady-state lines
    R_bar_ss = (1 / model.β) * (μ0**model.γ)
    η_ss     = model.α * A0 * k_ss**(model.α - 1)

    # plot k
    axes[0].plot(k_path[:T], linestyle=linestyle, label=label)
    axes[0].axhline(k_ss, linestyle='--', color='black')
    axes[0].set_title('k')

    # plot c
    axes[1].plot(c_path[:T], linestyle=linestyle, label=label)
    axes[1].axhline(c_ss, linestyle='--', color='black')
    axes[1].set_title('c')

    # plot R bar
    S_full    = k_path.size - 1
    R_bar_path = compute_R_bar_path(shocks, k_path, model, S_full)
    axes[2].plot(R_bar_path[:T], linestyle=linestyle, label=label)
    axes[2].axhline(R_bar_ss, linestyle='--', color='black')
    axes[2].set_title(r'$\bar{R}$')

    # plot η
    η_path = compute_η_path(k_path, model, S_full)
    axes[3].plot(η_path[:T], linestyle=linestyle, label=label)
    axes[3].axhline(η_ss, linestyle='--', color='black')
    axes[3].set_title(r'$\eta$')

    # plot shock
    shock_series = np.array(shocks[shock_param], dtype=object)
    axes[4].plot(shock_series[:T], linestyle=linestyle, label=label)
    axes[4].axhline(shock_series[0], linestyle='--', color='black')
    axes[4].set_title(rf'${shock_param}$')

    if label:
        for ax in axes[:5]:
            ax.legend()
```

Experiment 1: Foreseen once-and-for-all increase in g from 0.2 to 0.4 in period 10

The figure below shows consequences of a foreseen permanent increase in g at $t = T = 10$ that is financed by an increase in lump-sum taxes

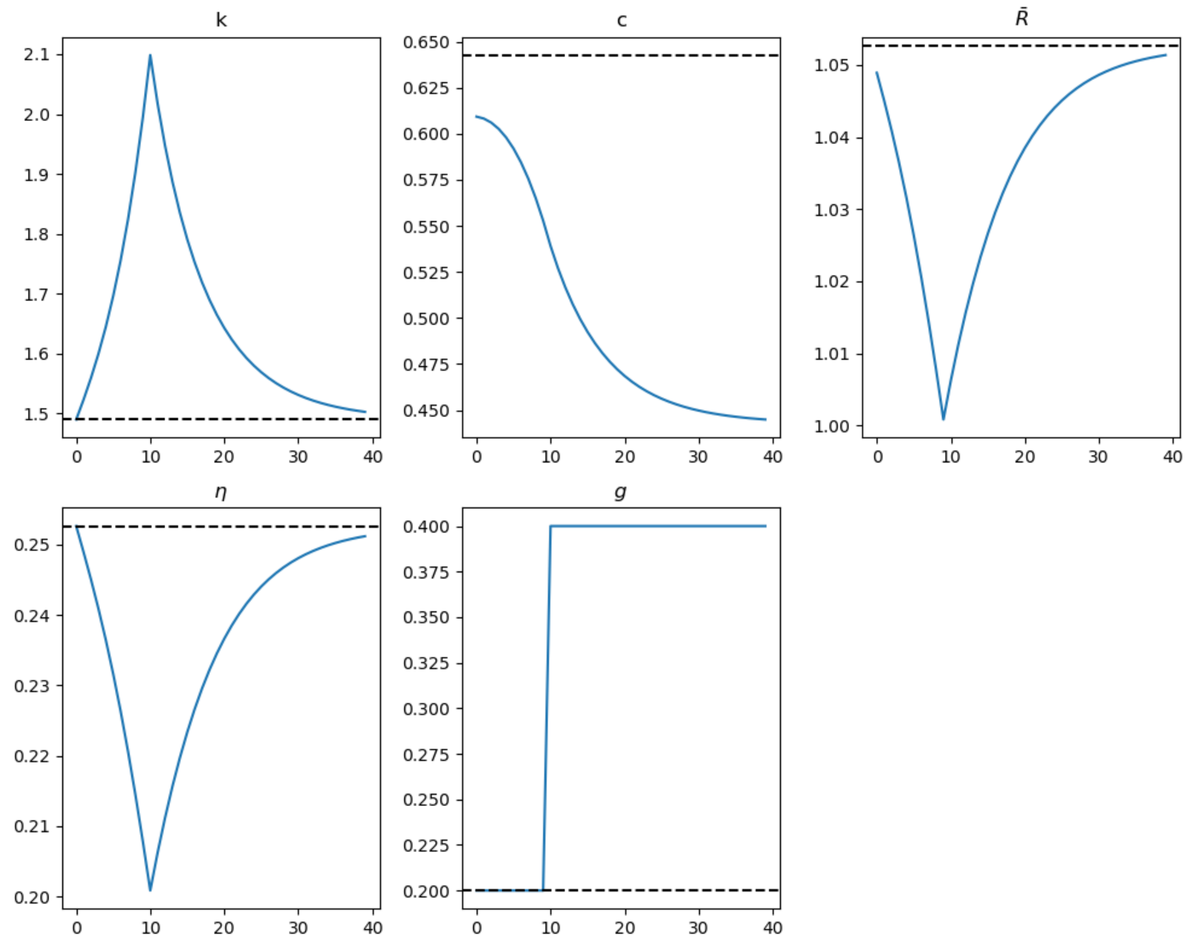
```
# Define shocks as a dictionary
shocks = {
```

(continues on next page)

(continued from previous page)

```
    'g': np.concatenate(
        (np.repeat(0.2, 10), np.repeat(0.4, S - 9))
    ),
    'τ_c': np.repeat(0.0, S + 1),
    'τ_k': np.repeat(0.0, S + 1)
}

k_ss_initial, c_ss_initial = steady_states(model,
                                           shocks['g'][0],
                                           shocks['τ_k'][0])

print(f"Steady-state capital: {k_ss_initial:.4f}")
print(f"Steady-state consumption: {c_ss_initial:.4f}")

solution = run_shooting(shocks, S, model)

fig, axes = plt.subplots(2, 3, figsize=(10, 8))
axes = axes.flatten()

plot_results(solution, k_ss_initial,
             c_ss_initial, shocks, 'g', axes, model, T=40)

for ax in axes[5:]:
    fig.delaxes(ax)

plt.tight_layout()
plt.show()
```

```
Steady-state capital: 1.4900
Steady-state consumption: 0.6426
Model: Model(β=0.95, γ=2.0, δ=0.2, α=0.33, A=1.0)
Optimal initial consumption c0 = 0.6092419528879239645312185699727132533517
```



The above figures indicate that an equilibrium **consumption smoothing** mechanism is at work, driven by the representative consumer's preference for smooth consumption paths coming from the curvature of its one-period utility function.

- The steady-state value of the capital stock remains unaffected:
  - This follows from the fact that g disappears from the steady state version of the Euler equation ((72.19)).
- Consumption begins to decline gradually before time T due to increased government consumption:
  - Households reduce consumption to offset government spending, which is financed through increased lump-sum taxes.
  - The competitive economy signals households to consume less through an increase in the stream of lump-sum taxes.
  - Households, caring about the present value rather than the timing of taxes, experience an adverse wealth effect on consumption, leading to an immediate response.
- Capital gradually accumulates between time 0 and T due to increased savings and reduces gradually after time T:
  - This temporal variation in capital stock smooths consumption over time, driven by the representative consumer's consumption-smoothing motive.

Let's collect the procedures used above into a function that runs the solver and draws plots for a given experiment

The following figure compares responses to a foreseen increase in g at $t = 10$ for two economies:

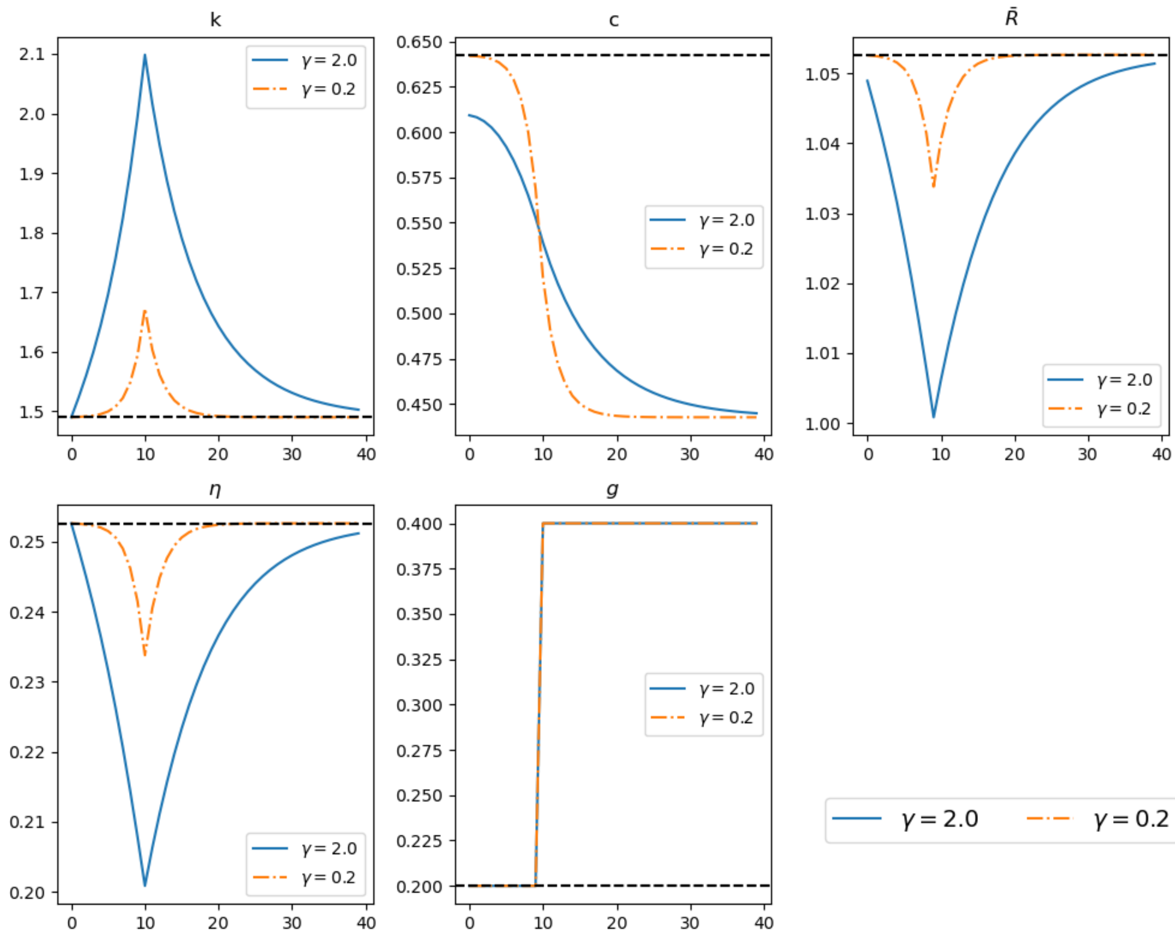
- our original economy with $\gamma = 2$, shown in the solid line, and

- an otherwise identical economy with $\gamma = 0.2$.

This comparison interest us because the utility curvature parameter γ governs the household's willingness to substitute consumption over time, and thus it preferences about smoothness of consumption paths over time.

```
# Solve the model using shooting
solution = run_shooting(shocks, S, model)

# Compute the initial steady states
k_ss_initial, c_ss_initial = steady_states(model,
                                           shocks['g'][0],
                                           shocks['τ_k'][0])

# Plot the solution for γ=2
fig, axes = plt.subplots(2, 3, figsize=(10, 8))
axes = axes.flatten()

label = fr"$\gamma = {model.γ}$"
plot_results(solution, k_ss_initial, c_ss_initial,
             shocks, 'g', axes, model, label=label,
             T=40)

# Solve and plot the result for γ=0.2
model_γ2 = create_model(γ=0.2)
solution = run_shooting(shocks, S, model_γ2)

plot_results(solution, k_ss_initial, c_ss_initial,
             shocks, 'g', axes, model_γ2,
             label=fr"$\gamma = {model_γ2.γ}$",
             linestyle='-.', T=40)

handles, labels = axes[0].get_legend_handles_labels()
fig.legend(handles, labels, loc='lower right',
           ncol=3, fontsize=14, bbox_to_anchor=(1, 0.1))

for ax in axes[5:]:
    fig.delaxes(ax)

plt.tight_layout()
plt.show()
```

```
Model: Model(β=0.95, γ=2.0, δ=0.2, α=0.33, A=1.0)
Optimal initial consumption c0 = 0.6092419528879239645312185699727132533517
Model: Model(β=0.95, γ=0.2, δ=0.2, α=0.33, A=1.0)
Optimal initial consumption c0 = 0.6420330412987902926414768724607623681745
```



The outcomes indicate that lowering γ affects both the consumption and capital stock paths because - it increases the representative consumer's willingness to substitute consumption across time:

- Consumption path:
  - When $\gamma = 0.2$, consumption becomes less smooth compared to $\gamma = 2$.
  - For $\gamma = 0.2$, consumption mirrors the government expenditure path more closely, staying higher until $t = 10$.
- Capital stock path:
  - With $\gamma = 0.2$, there are smaller build-ups and drawdowns of capital stock.
  - There are also smaller fluctuations in $\bar{R}$ and η.

Let's write another function that runs the solver and draws plots for these two experiments

Now we plot other equilibrium quantities:

```python
def plot_prices(solution, c_ss, shock_param, axes,
                model, label='', linestyle='-', T=40):
    """
    Compares and plots prices
    """
    α, β, δ, γ, A = model.α, model.β, model.δ, model.γ, model.A

    k_path = solution[:, 0]
    c_path = solution[:, 1]
```

(continues on next page)

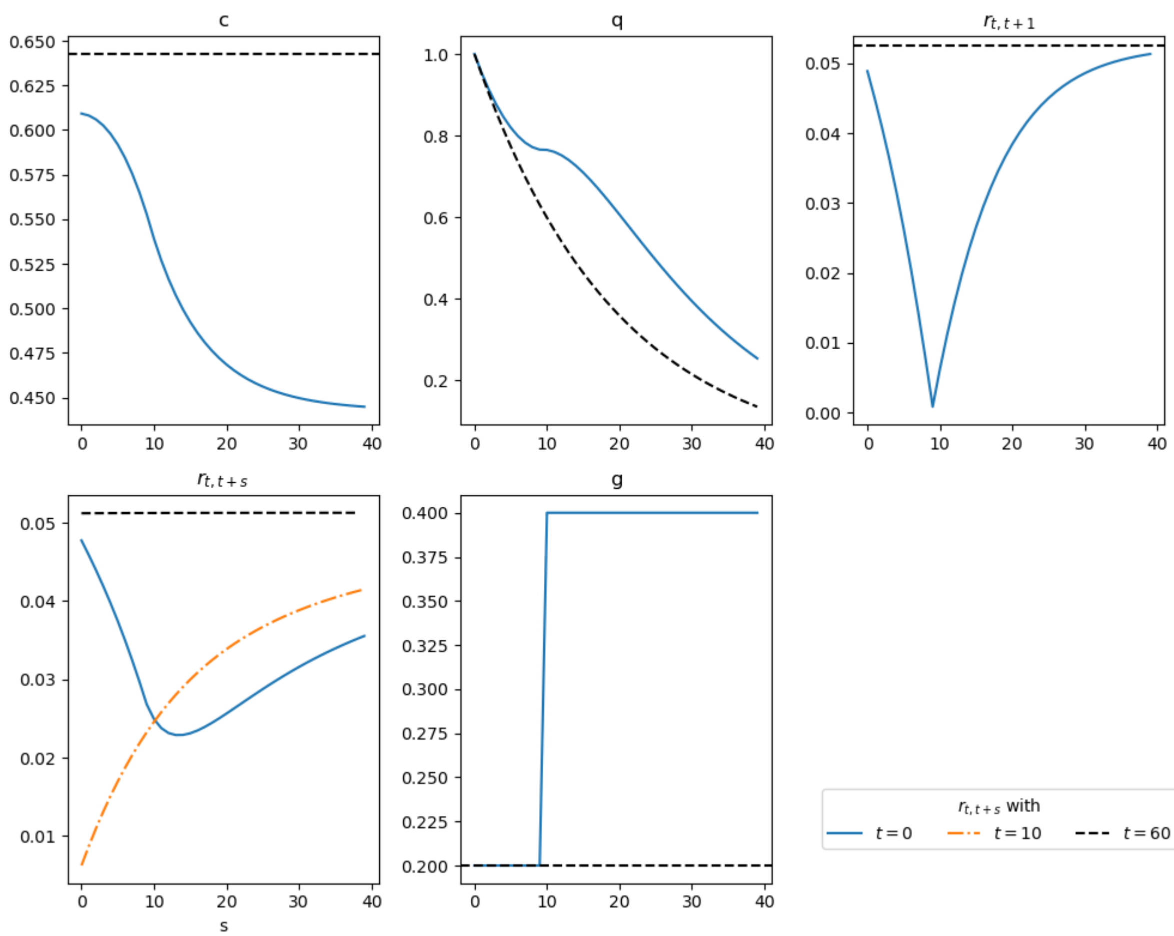
(continued from previous page)

```
    # Plot for c
    axes[0].plot(c_path[:T], linestyle=linestyle, label=label)
    axes[0].axhline(c_ss, linestyle='--', color='black')
    axes[0].set_title('c')

    # Plot for q
    q_path = compute_q_path(c_path, model, S=S)
    axes[1].plot(q_path[:T], linestyle=linestyle, label=label)
    axes[1].plot(β**np.arange(T), linestyle='--', color='black')
    axes[1].set_title('q')

    # Plot for r_{t,t+1}
    R_bar_path = compute_R_bar_path(shocks, k_path, model, S)
    axes[2].plot(R_bar_path[:T] - 1, linestyle=linestyle, label=label)
    axes[2].axhline(1 / β - 1, linestyle='--', color='black')
    axes[2].set_title('$r_{t,t+1}$')

    # Plot for r_{t,t+s}
    for style, s in zip(['-', '-.', '--'], [0, 10, 60]):
        rts_path = compute_rts_path(q_path, T, s)
        axes[3].plot(rts_path, linestyle=style,
                     color='black' if style == '--' else None,
                     label=f'$t={s}$')
        axes[3].set_xlabel('s')
        axes[3].set_title('$r_{t,t+s}$')

    # Plot for g
    axes[4].plot(shocks[shock_param][:T], linestyle=linestyle, label=label)
    axes[4].axhline(shocks[shock_param][0], linestyle='--', color='black')
    axes[4].set_title(shock_param)
```

For $\gamma = 2$ again, the next figure describes the response of q_t and the term structure of interest rates to a foreseen increase in g_t at $t = 10$

```
solution = run_shooting(shocks, S, model)

fig, axes = plt.subplots(2, 3, figsize=(10, 8))
axes = axes.flatten()

plot_prices(solution, c_ss_initial, 'g', axes, model, T=40)

for ax in axes[5:]:
    fig.delaxes(ax)

handles, labels = axes[3].get_legend_handles_labels()
fig.legend(handles, labels, title=r"$r_{t,t+s}$ with ", loc='lower right',
           ncol=3, fontsize=10, bbox_to_anchor=(1, 0.1))
plt.tight_layout()
plt.show()
```

```
Model: Model(β=0.95, γ=2.0, δ=0.2, α=0.33, A=1.0)
Optimal initial consumption c0 = 0.6092419528879239645312185699727132533517
```



The second panel on the top compares q_t for the initial steady state with q_t after the increase in g is foreseen at $t = 0$, while the third panel compares the implied short rate r_t.

The fourth panel shows the term structure of interest rates at $t = 0$, $t = 10$, and $t = 60$.

Notice that, by $t = 60$, the system has converged to the new steady state, and the term structure of interest rates becomes flat.

At $t = 10$, the term structure of interest rates is upward sloping.

This upward slope reflects the expected increase in the rate of growth of consumption over time, as shown in the consumption panel.

At $t = 0$, the term structure of interest rates exhibits a "U-shaped" pattern:

- It declines until maturity at $s = 10$.
- After $s = 10$, it increases for longer maturities.

This pattern aligns with the pattern of consumption growth in the first two figures, which declines at an increasing rate until $t = 10$ and then declines at a decreasing rate afterward.

Experiment 2: Foreseen once-and-for-all increase in τ_c from 0.0 to 0.2 in period 10

With an inelastic labor supply, the Euler equation (72.16) and the other equilibrium conditions show that

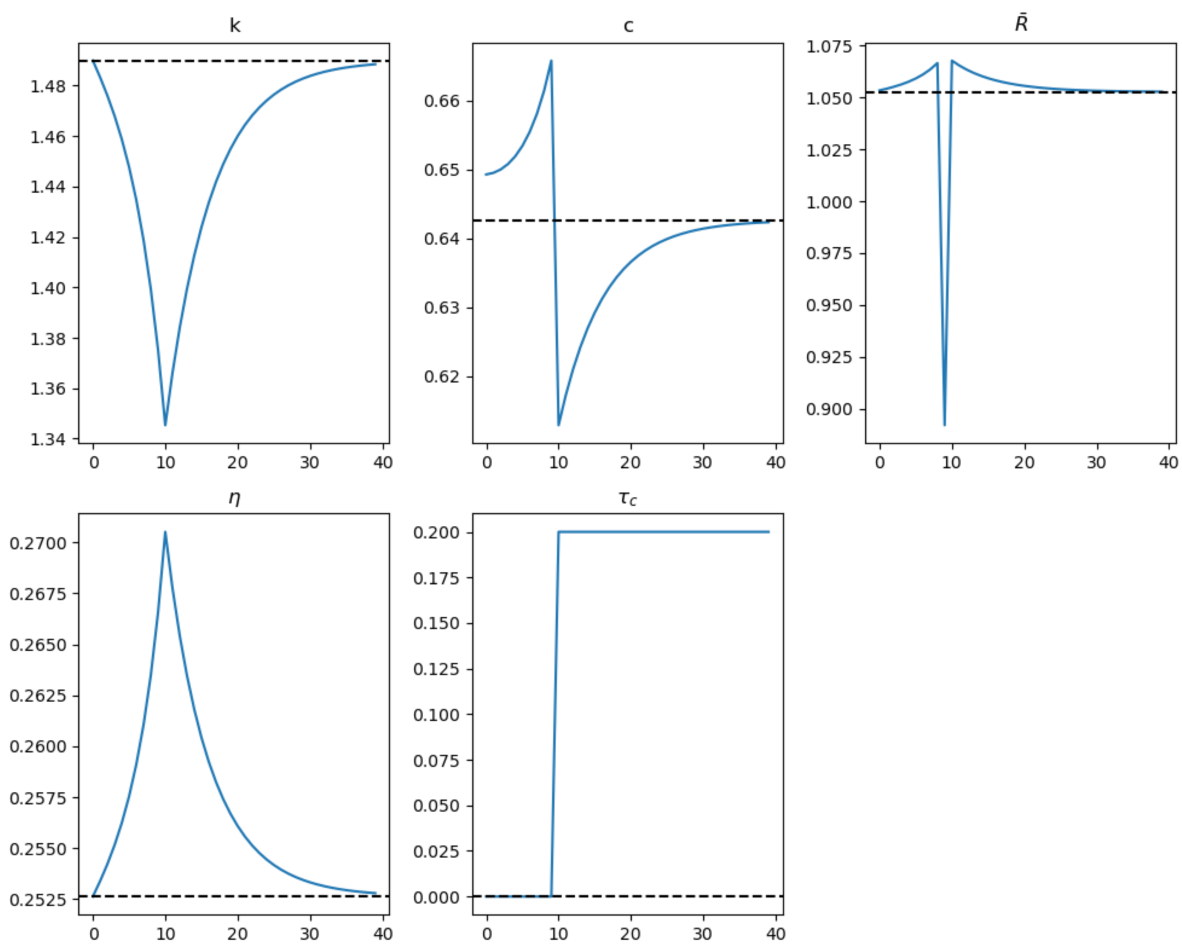
- constant consumption taxes do not distort decisions, but
- anticipated changes in them do.

Indeed, (72.16) or (72.17) indicates that a foreseen in- crease in τ_{ct} (i.e., a decrease in $(1+\tau_{ct})\ (1+\tau_{ct+1})$) operates like an increase in τ_{kt}.

The following figure portrays the response to a foreseen increase in the consumption tax τ_c.

```
shocks = {
    'g': np.repeat(0.2, S + 1),
    'τ_c': np.concatenate((np.repeat(0.0, 10), np.repeat(0.2, S - 9))),
    'τ_k': np.repeat(0.0, S + 1)
}

experiment_model(shocks, S, model,
                 solver=run_shooting,
                 plot_func=plot_results,
                 policy_shock='τ_c')
```

```
Steady-state capital: 1.4900
Steady-state consumption: 0.6426
----------------------------------------------------------------
Model: Model(β=0.95, γ=2.0, δ=0.2, α=0.33, A=1.0)
Optimal initial consumption c0 = 0.6492795614681543372301864705195788398396
```



Evidently all variables in the figures above eventually return to their initial steady-state values.

The anticipated increase in τ_{ct} leads to variations in consumption and capital stock across time:

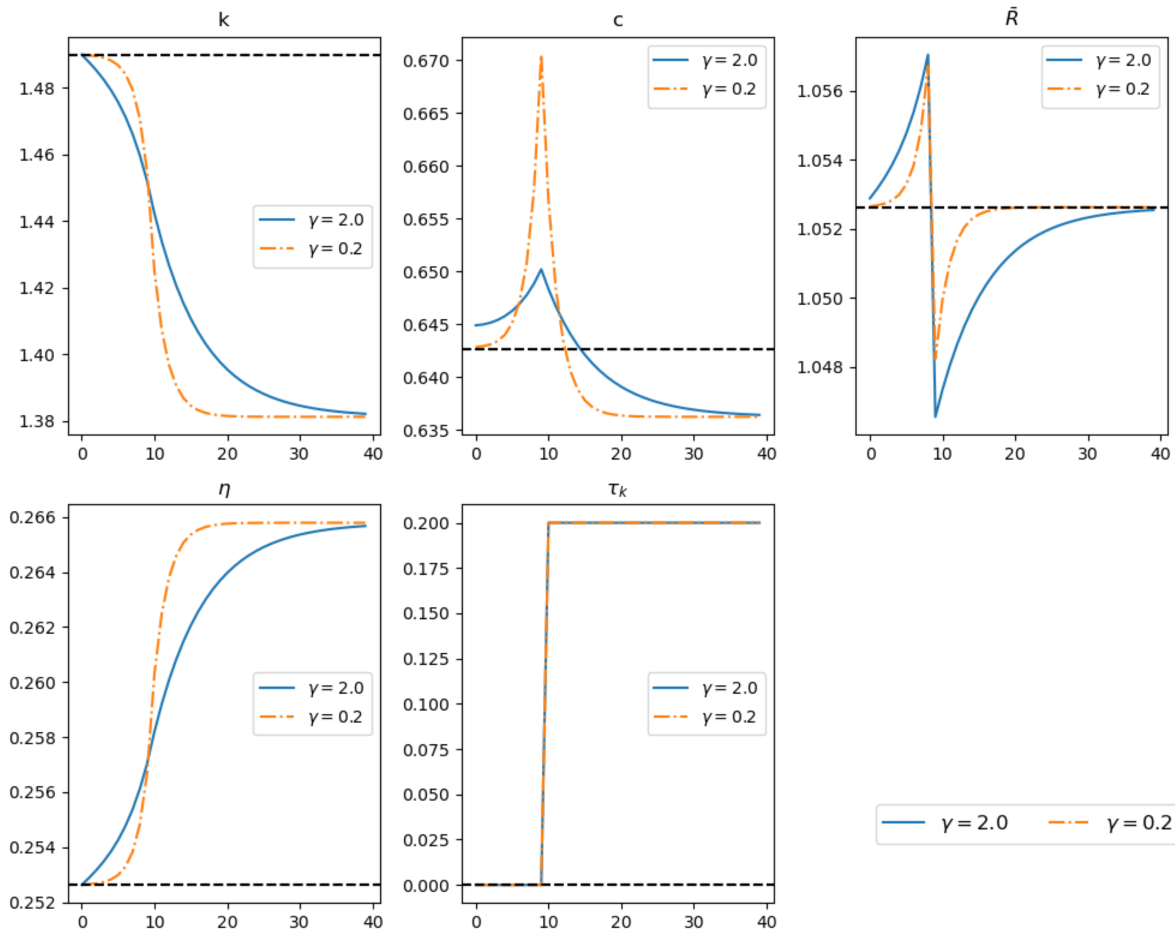
- At $t = 0$:
  - Anticipation of the increase in τ_c causes an *immediate jump in consumption*.
  - This is followed by a *consumption binge* that sends the capital stock downward until $t = T = 10$.
- Between $t = 0$ and $t = T = 10$:
  - The decline in the capital stock raises $\bar{R}$ over time.
  - The equilibrium conditions require the growth rate of consumption to rise until $t = T$.
- At $t = T = 10$:
  - The jump in τ_c depresses $\bar{R}$ below 1, causing a *sharp drop in consumption*.
- After $T = 10$:
  - The effects of anticipated distortion are over, and the economy gradually adjusts to the lower capital stock.
  - Capital must now rise, requiring *austerity* —consumption plummets after $t = T$, indicated by lower levels of consumption.
  - The interest rate gradually declines, and consumption grows at a diminishing rate along the path to the terminal steady-state.

Experiment 3: Foreseen once-and-for-all increase in τ_k from 0.0 to 0.2 in period 10

For the two γ values 2 and 0.2, the next figure shows the response to a foreseen permanent jump in τ_{kt} at $t = T = 10$.

```
shocks = {
    'g': np.repeat(0.2, S + 1),
    'τ_c': np.repeat(0.0, S + 1),
    'τ_k': np.concatenate((np.repeat(0.0, 10), np.repeat(0.2, S - 9)))
}

experiment_two_models(shocks, S, model, model_γ2,
                solver=run_shooting,
                 plot_func=plot_results,
                 policy_shock='τ_k')
```

```
Model 1 (γ=2.0): steady state k=1.4900, c=0.6426
Model 2 (γ=0.2): steady state k=1.4900, c=0.6426
----------------------------------------------------------------
Model: Model(β=0.95, γ=2.0, δ=0.2, α=0.33, A=1.0)
Optimal initial consumption c0 = 0.6448856400318608460996300822707014890199
Model: Model(β=0.95, γ=0.2, δ=0.2, α=0.33, A=1.0)
Optimal initial consumption c0 = 0.6428407772240506727464152695921513767415
```



The path of government expenditures remains fixed

- the increase in τ_{kt} is offset by a reduction in the present value of lump-sum taxes to keep the budget balanced.

The figure shows that:

- Anticipation of the increase in τ_{kt} leads to immediate decline in capital stock due to increased current consumption and a growing consumption flow.
- $\bar{R}$ starts rising at $t = 0$ and peaks at $t = 9$, and at $t = 10$, $\bar{R}$ drops sharply due to the tax change.
  - Variations in $\bar{R}$ align with the impact of the tax increase at $t = 10$ on consumption across time.
- Transition dynamics push k_t (capital stock) toward a new, lower steady-state level. In the new steady state:
  - Consumption is lower due to reduced output from the lower capital stock.
  - Smoother consumption paths occur when $\gamma = 2$ than when $\gamma = 0.2$.

So far we have explored consequences of foreseen once-and-for-all changes in government policy. Next we describe some experiments in which there is a foreseen one-time change in a policy variable (a "pulse").

Experiment 4: Foreseen one-time increase in g from 0.2 to 0.4 in period 10, after which g returns to 0.2 forever

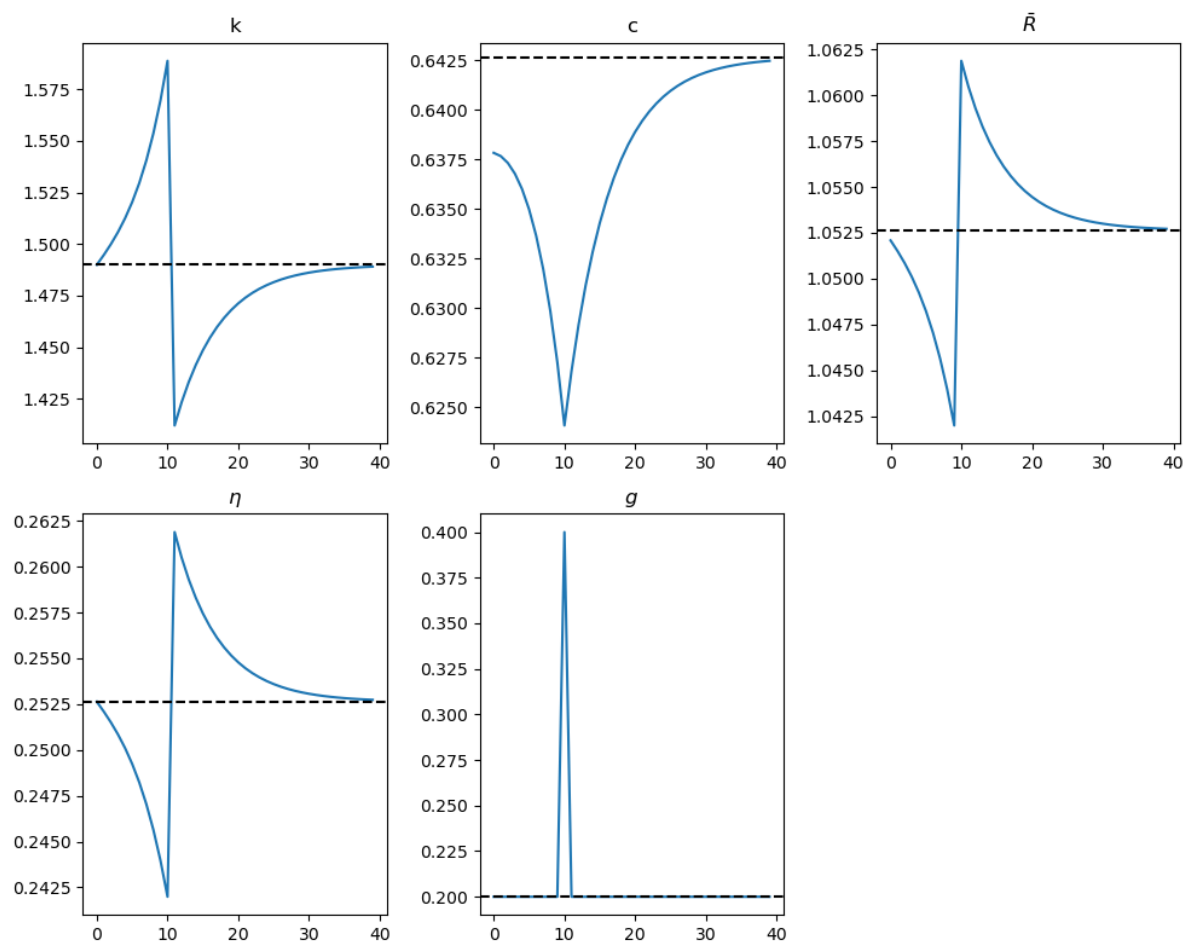
```
g_path = np.repeat(0.2, S + 1)
g_path[10] = 0.4

shocks = {
```

(continues on next page)

(continued from previous page)

```
    'g': g_path,
    'τ_c': np.repeat(0.0, S + 1),
    'τ_k': np.repeat(0.0, S + 1)
}

experiment_model(shocks, S, model,
                 solver=run_shooting,
                 plot_func=plot_results,
                 policy_shock='g')
```

```
Steady-state capital: 1.4900
Steady-state consumption: 0.6426
--------------------------------------------------------------
Model: Model(β=0.95, γ=2.0, δ=0.2, α=0.33, A=1.0)
Optimal initial consumption c0 = 0.6378298012463969247674771825320030214755
```



The figure indicates how:

- Consumption:
  - Drops immediately upon announcement of the policy and continues to decline over time in anticipation of the one-time surge in g.
  - After the shock at $t = 10$, consumption begins to recover, rising at a diminishing rate toward its steady-state value.

- Capital and $\bar{R}$:
  - Before $t = 10$, capital accumulates as interest rate changes induce households to prepare for the anticipated increase in government spending.
  - At $t = 10$, the capital stock sharply decreases as the government consumes part of it.
  - $\bar{R}$ jumps above its steady-state value due to the capital reduction and then gradually declines toward its steady-state level.

72.9.3 Method 2: Residual Minimization

The second method involves minimizing residuals (i.e., deviations from equalities) of the following equations:

- The Euler equation (72.17):

$$1 = \beta \left(\frac{c_{t+1}}{c_t}\right)^{-\gamma} \frac{(1+\tau_{ct})}{(1+\tau_{ct+1})} \left[(1-\tau_{kt+1})(\alpha A k_{t+1}^{\alpha-1} - \delta) + 1\right]$$

- The feasibility condition (72.15):

$$k_{t+1} = A k_t^{\alpha} + (1-\delta)k_t - g_t - c_t.$$

```python
# Euler's equation and feasibility condition
def euler_residual(c_t, c_tp1, τ_c_t, τ_c_tp1, τ_k_tp1, k_tp1, model, μ_tp1=1):
    """
    Computes the residuals for Euler's equation
    with optional growth model parameters μ_tp1.
    """
    R_bar = compute_R_bar(τ_c_t, τ_c_tp1, τ_k_tp1, k_tp1, model)

    c_expected = next_c(c_t, R_bar, model, μ_tp1)

    return c_expected / c_tp1 - 1.0

def feasi_residual(k_t, k_tm1, c_tm1, g_t, model, μ_t=1):
    """
    Computes the residuals for feasibility condition
    with optional growth model parameter μ_t.
    """
    k_t_expected = next_k(k_tm1, g_t, c_tm1, model, μ_t)
    return k_t_expected - k_t
```

The algorithm proceeds follows:

1. Find initial steady state k_0 based on the government plan at $t = 0$.
2. Initialize a sequence of initial guesses $\{\hat{c}_t, \hat{k}_t\}_{t=0}^{S}$.
3. Compute residuals l_a and l_k for $t = 0, \dots, S$, as well as l_{k_0} for $t = 0$ and l_{k_S} for $t = S$:
   - Compute the Euler equation residual for $t = 0, \dots, S$ using (72.17):

$$l_{ta} = \beta u'(c_{t+1}) \frac{(1+\tau_{ct})}{(1+\tau_{ct+1})} \left[(1-\tau_{kt+1})(f'(k_{t+1}) - \delta) + 1\right] - 1$$

   - Compute the feasibility condition residual for $t = 1, \dots, S-1$ using (72.15):

$$l_{tk} = k_{t+1} - f(k_t) - (1-\delta)k_t + g_t + c_t$$

- Compute the residual for the initial condition for k_0 using (72.19) and the initial capital k_0:

$$l_{k_0} = 1 - \beta \left[(1 - \tau_{k0}) \left(f'(k_0) - \delta \right) + 1 \right]$$

- Compute the residual for the terminal condition for $t = S$ using (72.17) under the assumptions $c_t = c_{t+1} = c_S$, $k_t = k_{t+1} = k_S$, $\tau_{ct} = \tau_{ct+1} = \tau_{c_s}$, and $\tau_{kt} = \tau_{kt+1} = \tau_{k_s}$:

$$l_{k_S} = \beta u'(c_S) \frac{(1 + \tau_{c_s})}{(1 + \tau_{c_s})} \left[(1 - \tau_{k_s})(f'(k_S) - \delta) + 1 \right] - 1$$

4. Iteratively adjust guesses for $\{\hat{c}_t, \hat{k}_t\}_{t=0}^{S}$ to minimize residuals l_{k_0}, l_{ta}, l_{tk}, and l_{k_S} for $t = 0, \dots, S$.

```
def compute_residuals(vars_flat, k_init, S, shock_paths, model):
    """
    Compute the residuals for the Euler equation and feasibility condition.
    """
    g, τ_c, τ_k, μ = (shock_paths[key] for key in ('g','τ_c','τ_k','μ'))
    k, c = vars_flat.reshape((S+1, 2)).T
    res = np.empty(2*S+2, dtype=float)

    # boundary condition on initial capital
    res[0] = k[0] - k_init

    # interior Euler and feasibility
    for t in range(S):
        res[2*t + 1] = euler_residual(
            c[t],    c[t+1],
            τ_c[t],  τ_c[t+1],
            τ_k[t+1],k[t+1],
            model, μ[t+1])
        res[2*t + 2] = feasi_residual(
            k[t+1], k[t], c[t],
            g[t],  model,
            μ[t+1])

    # terminal Euler condition at t=S
    res[-1] = euler_residual(
        c[S],   c[S],
        τ_c[S], τ_c[S],
        τ_k[S], k[S],
        model,
        μ[S])

    return res


def run_min(shocks, S, model, A_path=None):
    """
    Solve for the full (k,c) path by root-finding the residuals.
    """
    shocks['μ'] = shocks['μ'] if 'μ' in shocks else np.ones_like(shocks['g'])

    # compute the steady-state to serve as both initial capital and guess
    k_ss, c_ss = steady_states(
        model,
        shocks['g'][0],
```

(continues on next page)

(continued from previous page)

```
        shocks['τ_k'][0],
        shocks['μ'][0]  # =1 if no growth
    )

    # initial guess: flat at the steady-state
    guess = np.column_stack([
        np.full(S+1, k_ss),
        np.full(S+1, c_ss)
    ]).flatten()

    sol = root(
        compute_residuals,
        guess,
        args=(k_ss, S, shocks, model),
        tol=1e-8
    )

    return sol.x.reshape((S+1, 2))
```

We found that method 2 did not encounter numerical stability issues, so using `mp.mpf` is not necessary.

We leave as exercises replicating some of our experiments by using the second method.

Exercise 72.9.1

Replicate the plots of our four experiments using the second method of residual minimization:

1. A foreseen once-and-for-all increase in g from 0.2 to 0.4 occurring in period 10,
2. A foreseen once-and-for-all increase in τ_c from 0.0 to 0.2 occurring in period 10,
3. A foreseen once-and-for-all increase in τ_k from 0.0 to 0.2 occurring in period 10, and
4. A foreseen one-time increase in g from 0.2 to 0.4 in period 10, after which g reverts to 0.2 permanently,

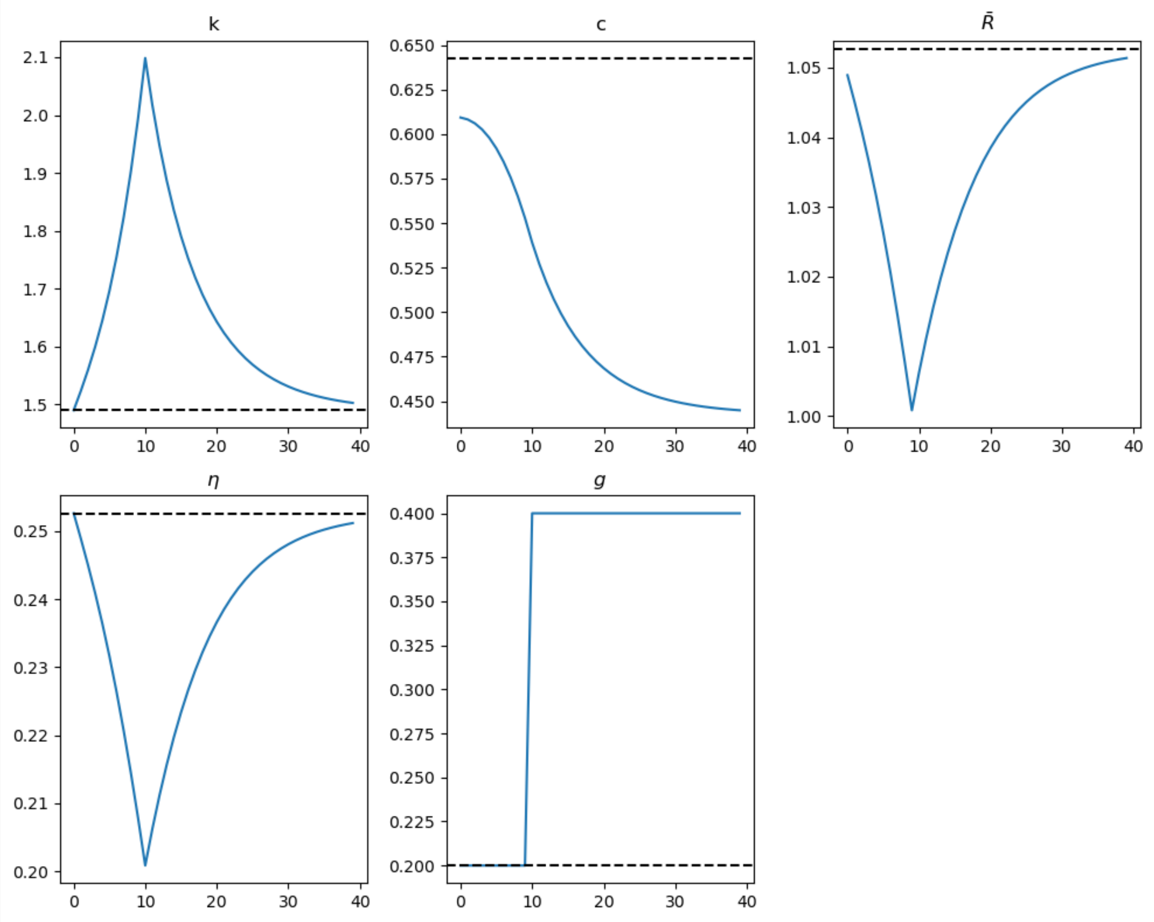
Solution

Here is one solution:

Experiment 1: Foreseen once-and-for-all increase in g from 0.2 to 0.4 in period 10

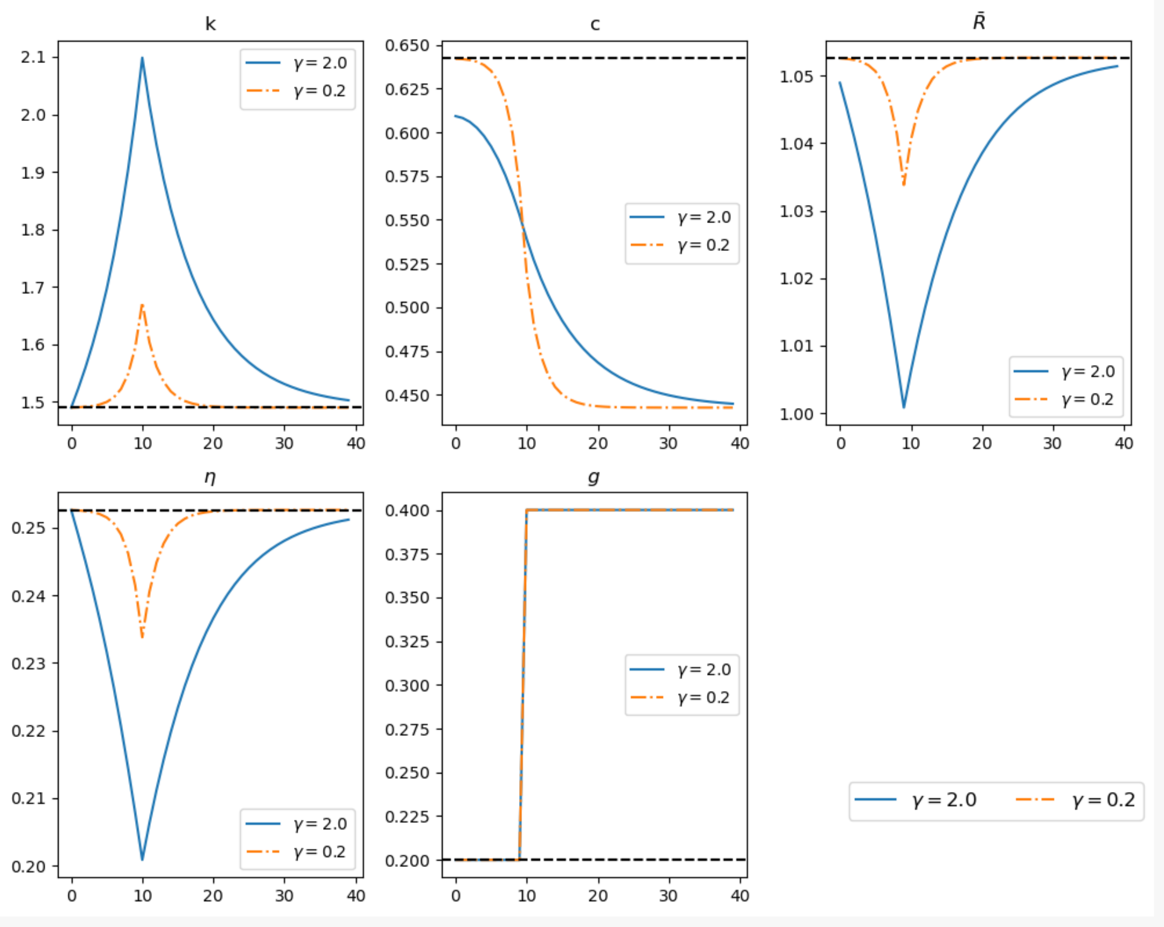
```
shocks = {
    'g': np.concatenate((np.repeat(0.2, 10), np.repeat(0.4, S - 9))),
    'τ_c': np.repeat(0.0, S + 1),
    'τ_k': np.repeat(0.0, S + 1)
}

experiment_model(shocks, S, model, solver=run_min,
                 plot_func=plot_results,
                 policy_shock='g')
```

```
Steady-state capital: 1.4900
Steady-state consumption: 0.6426
----------------------------------------------------------------
```

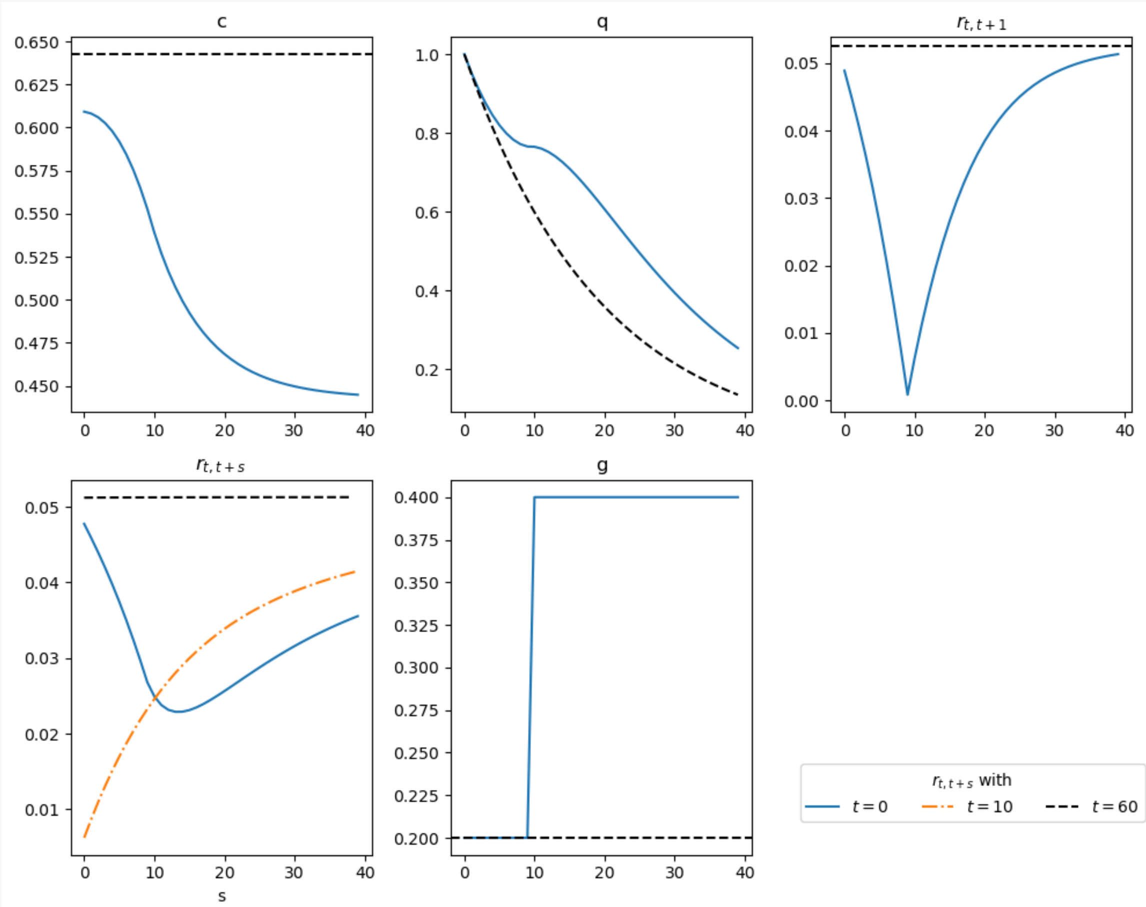



```
experiment_two_models(shocks, S, model, model_γ2,
                      run_min, plot_results, 'g')
```

```
Model 1 (γ=2.0): steady state k=1.4900, c=0.6426
Model 2 (γ=0.2): steady state k=1.4900, c=0.6426
----------------------------------------------------------------
```



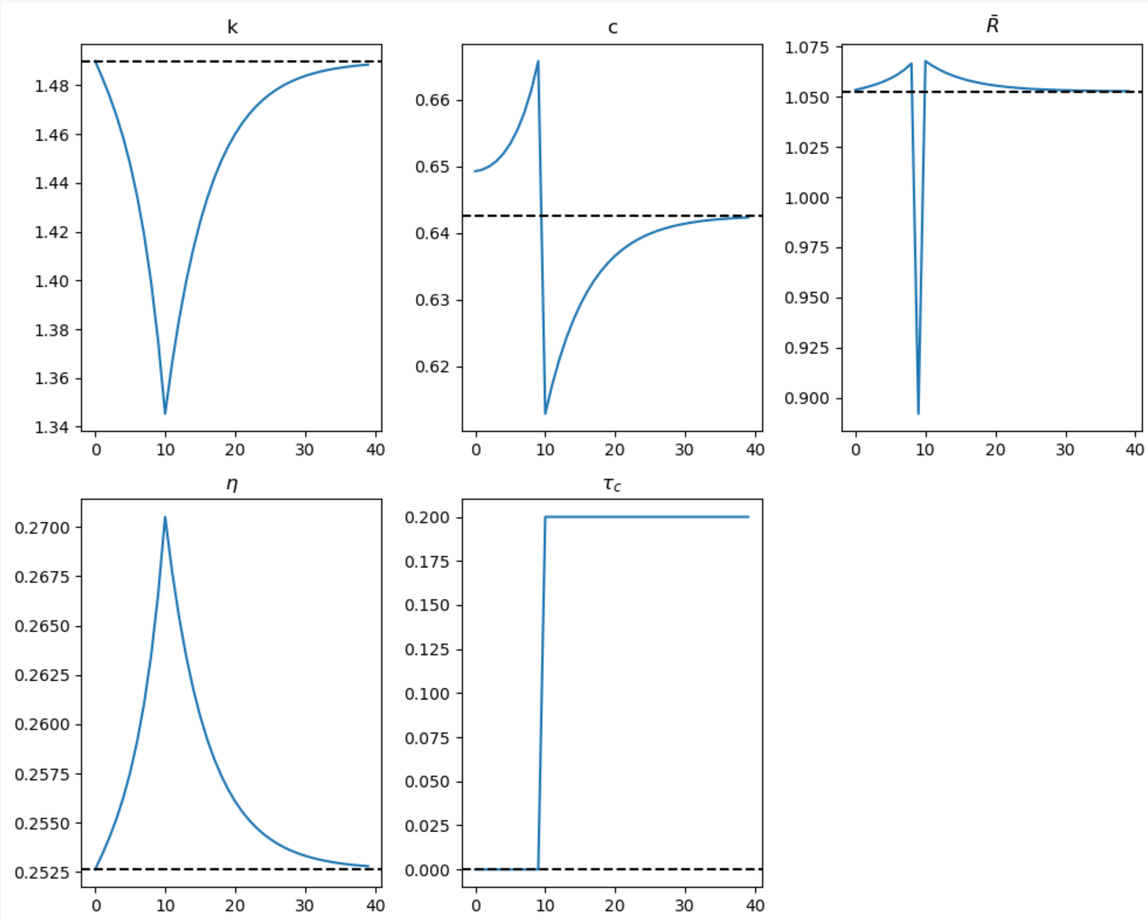

```
solution = run_min(shocks, S, model)

fig, axes = plt.subplots(2, 3, figsize=(10, 8))
axes = axes.flatten()

plot_prices(solution, c_ss_initial, 'g', axes, model, T=40)

for ax in axes[5:]:
    fig.delaxes(ax)

handles, labels = axes[3].get_legend_handles_labels()
fig.legend(handles, labels, title=r"$r_{t,t+s}$ with ", loc='lower right', ncol=3,␣
↪fontsize=10, bbox_to_anchor=(1, 0.1))
plt.tight_layout()
plt.show()
```



Experiment 2: Foreseen once-and-for-all increase in τ_c from 0.0 to 0.2 in period 10.

```
shocks = {
    'g': np.repeat(0.2, S + 1),
    'τ_c': np.concatenate((np.repeat(0.0, 10), np.repeat(0.2, S - 9))),
    'τ_k': np.repeat(0.0, S + 1)
}

experiment_model(shocks, S, model, solver=run_min,
                 plot_func=plot_results,
                 policy_shock='τ_c')
```

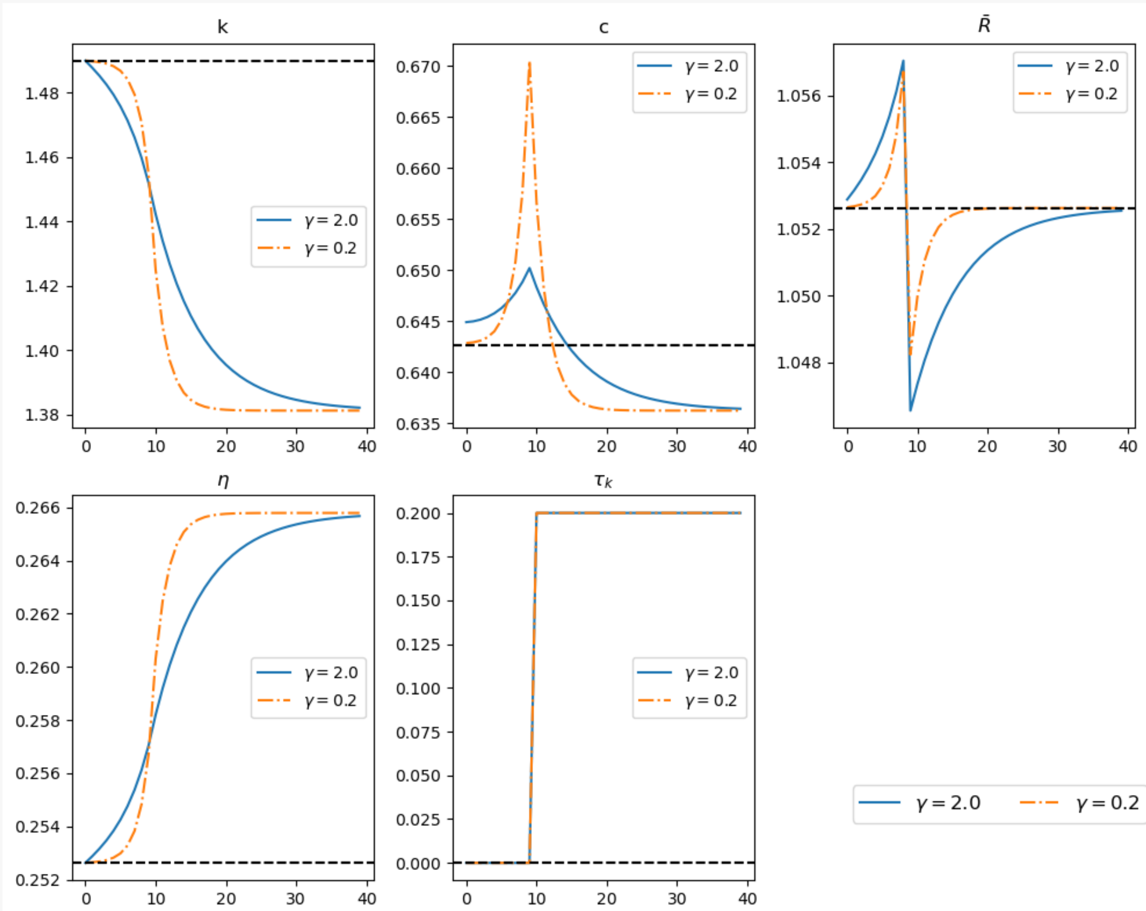
```
Steady-state capital: 1.4900
Steady-state consumption: 0.6426
----------------------------------------------------------------
```



Experiment 3: Foreseen once-and-for-all increase in τ_k from 0.0 to 0.2 in period 10.

```
shocks = {
    'g': np.repeat(0.2, S + 1),
    'τ_c': np.repeat(0.0, S + 1),
    'τ_k': np.concatenate((np.repeat(0.0, 10), np.repeat(0.2, S - 9)))
}

experiment_two_models(shocks, S, model, model_γ2,
                    solver=run_min,
                     plot_func=plot_results,
                     policy_shock='τ_k')
```

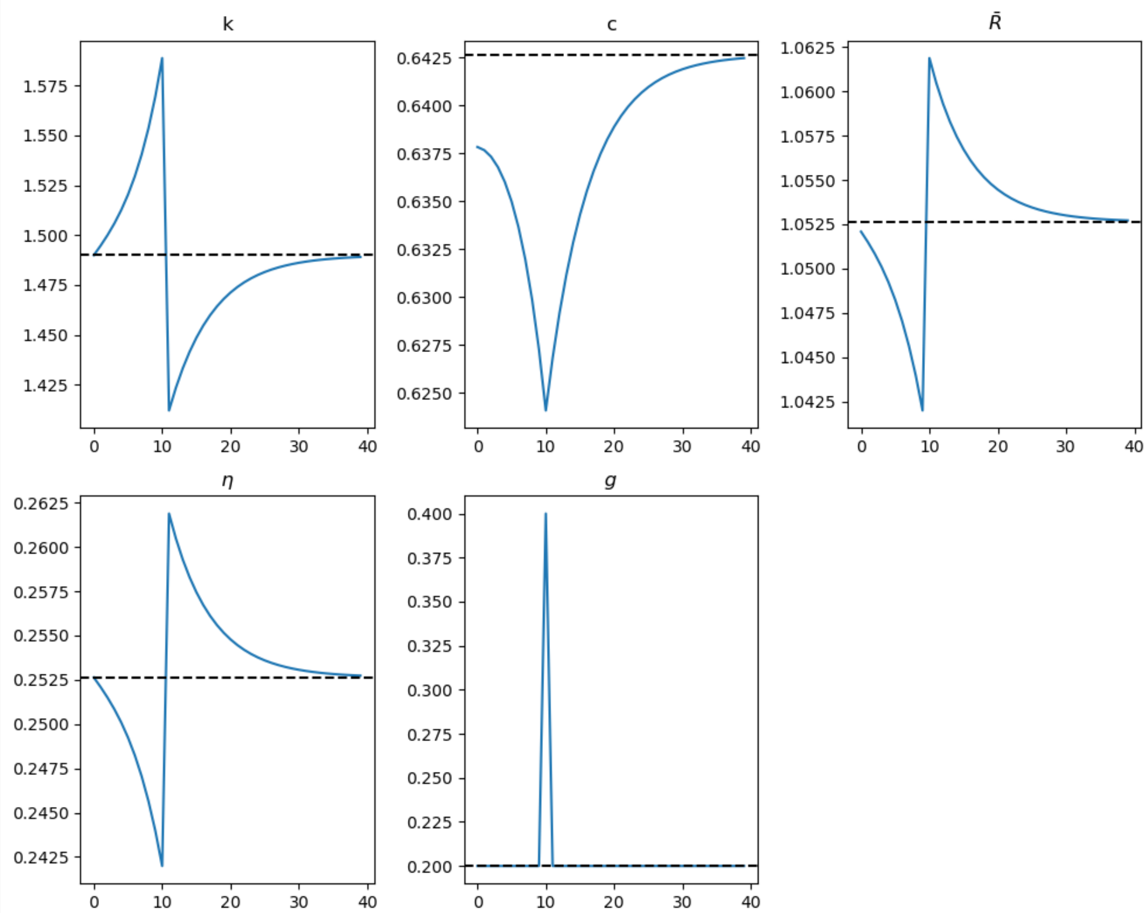
```
Model 1 (γ=2.0): steady state k=1.4900, c=0.6426
Model 2 (γ=0.2): steady state k=1.4900, c=0.6426
----------------------------------------------------------------
```



Experiment 4: Foreseen one-time increase in g from 0.2 to 0.4 in period 10, after which g returns to 0.2 forever

```
g_path = np.repeat(0.2, S + 1)
g_path[10] = 0.4

shocks = {
    'g': g_path,
    'τ_c': np.repeat(0.0, S + 1),
    'τ_k': np.repeat(0.0, S + 1)
}

experiment_model(shocks, S, model, solver=run_min,
                 plot_func=plot_results,
                 policy_shock='g')
```

```
Steady-state capital: 1.4900
Steady-state consumption: 0.6426
---------------------------------------------------------------
```



Exercise 72.9.2

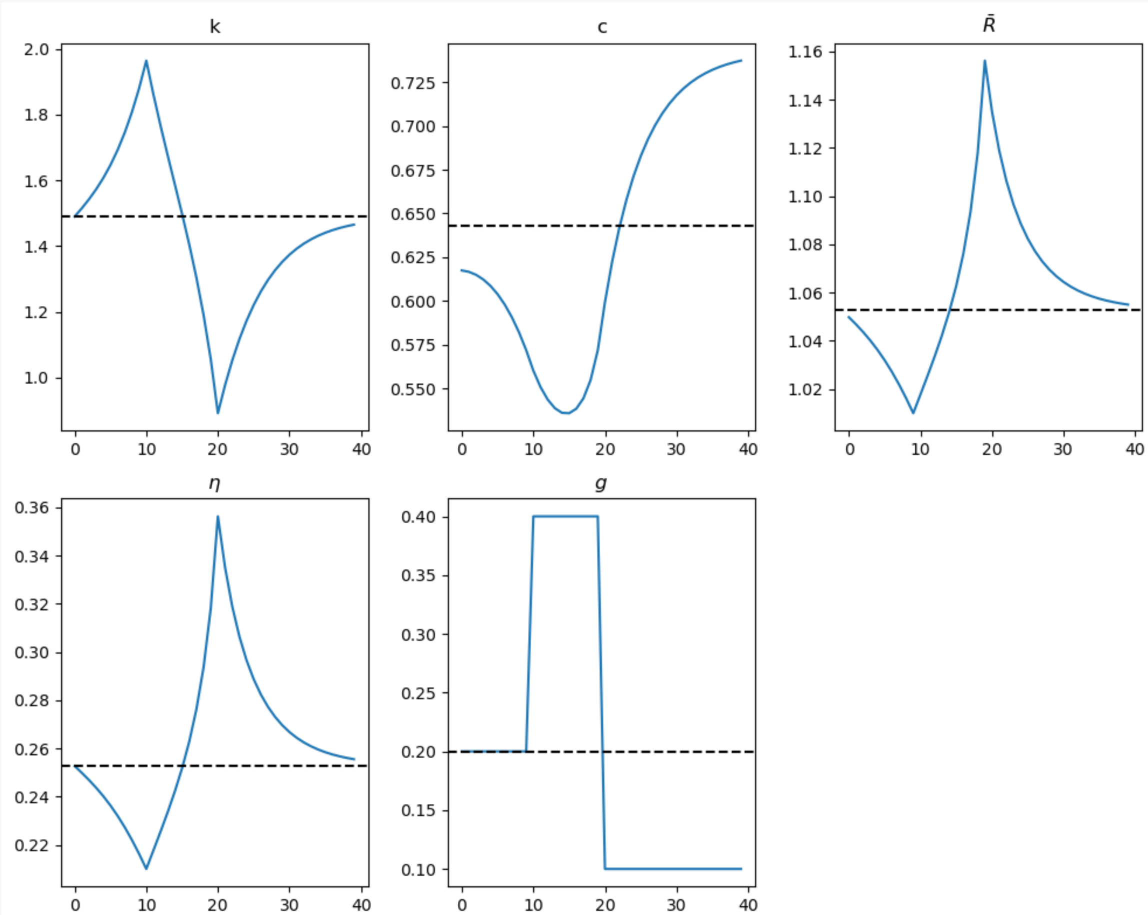
Design a new experiment where the government expenditure g increases from 0.2 to 0.4 in period 10, and then decreases to 0.1 in period 20 permanently.

Solution

Here is one solution:

```
g_path = np.repeat(0.2, S + 1)
g_path[10:20] = 0.4
g_path[20:] = 0.1

shocks = {
    'g': g_path,
    'τ_c': np.repeat(0.0, S + 1),
    'τ_k': np.repeat(0.0, S + 1)
}

experiment_model(shocks, S, model, solver=run_min,
                 plot_func=plot_results,
                 policy_shock='g')
```

```
Steady-state capital: 1.4900
Steady-state consumption: 0.6426
----------------------------------------------------------------
```



72.10 Exogenous growth

In the previous section, we considered a model without exogenous growth.

We set the term A_t in the production function to a constant by setting $A_t = 1$ for all t.

Now we are ready to consider growth.

To incorporate growth, we modify the production function to be

$$Y_t = F(K_t, A_t n_t)$$

where Y_t is aggregate output, N_t is total employment, A_t is labor-augmenting technical change, and $F(K, AN)$ is the same linearly homogeneous production function as before.

We assume that A_t follows the process

$$A_{t+1} = \mu_{t+1} A_t \tag{72.25}$$

and that $\mu_{t+1} = \bar{\mu} > 1$.

```python
# Set the constant A parameter to None
model = create_model(A=None)
```

```python
def compute_A_path(A0, shocks, S=100):
    """
    Compute A path over time.
    """
    A_path = np.full(S + 1, A0)
    for t in range(1, S + 1):
        A_path[t] = A_path[t-1] * shocks['μ'][t-1]
    return A_path
```

72.10.1 Inelastic Labor Supply

By linear homogeneity, the production function can be expressed as

$$y_t = f(k_t)$$

where $f(k) = F(k, 1) = k^\alpha$ and $k_t = \frac{K_t}{n_t A_t}$, $y_t = \frac{Y_t}{n_t A_t}$.

k_t and y_t are measured per unit of "effective labor" $A_t n_t$.

We also let $c_t = \frac{C_t}{A_t n_t}$ and $g_t = \frac{G_t}{A_t n_t}$, where C_t and G_t are total consumption and total government expenditures.

We continue to consider the case of inelastic labor supply.

Based on this, feasibility can be summarized by the following modified version of equation (72.15):

$$k_{t+1} = \mu_{t+1}^{-1}[f(k_t) + (1-\delta)k_t - g_t - c_t] \tag{72.26}$$

Again, by the properties of a linearly homogeneous production function, we have

$$\eta_t = F_k(k_t, 1) = f'(k_t), w_t = F_n(k_t, 1) = f(k_t) - f'(k_t)k_t$$

Since per capita consumption is now $c_t A_t$, the counterpart to the Euler equation (72.17) is

$$u'(c_t A_t) = \beta u'(c_{t+1} A_{t+1}) \frac{(1+\tau_{ct})}{(1+\tau_{ct+1})}[(1-\tau_{kt+1})(f'(k_{t+1}) - \delta) + 1]. \tag{72.27}$$

$\bar{R}_{t+1}$ continues to be defined by (72.21), except that now k_t is capital per effective unit of labor.

Thus, substituting (72.21), (72.27) becomes

$$u'(c_t A_t) = \beta u'(c_{t+1} A_{t+1}) \bar{R}_{t+1}$$

Assuming that the household's utility function is the same as before, we have

$$(c_t A_t)^{-\gamma} = \beta (c_{t+1} A_{t+1})^{-\gamma} \bar{R}_{t+1}$$

Thus, the counterpart to (72.24) is

$$c_{t+1} = c_t \left[\beta \bar{R}_{t+1}\right]^{\frac{1}{\gamma}} \mu_{t+1}^{-1} \tag{72.28}$$

72.10.2 Steady State

In a steady state, $c_{t+1} = c_t$. Then (72.27) becomes

$$1 = \mu^{-\gamma}\beta[(1-\tau_k)(f'(k)-\delta)+1] \tag{72.29}$$

from which we can compute that the steady-state level of capital per unit of effective labor satisfies

$$f'(k) = \delta + (\frac{\frac{1}{\beta}\mu^{\gamma}-1}{1-\tau_k}) \tag{72.30}$$

and that

$$\bar{R} = \frac{\mu^{\gamma}}{\beta} \tag{72.31}$$

The steady-state level of consumption per unit of effective labor can be found using (72.26):

$$c = f(k) + (1-\delta-\mu)k - g$$

Since the algorithm and plotting routines are the same as before, we include the steady-state calculations and shooting routine in the section *Python Code*.

72.10.3 Shooting Algorithm

Now we can apply the shooting algorithm to compute equilibrium. We augment the vector of shock variables by including μ_t, then proceed as before.

72.10.4 Experiments

Let's run some experiments:

1. A foreseen once-and-for-all increase in μ from 1.02 to 1.025 in period 10
2. An unforeseen once-and-for-all increase in μ to 1.025 in period 0

Experiment 1: A foreseen increase in μ from 1.02 to 1.025 at t=10

The figures below show the effects of a permanent increase in productivity growth μ from 1.02 to 1.025 at t=10.

They now measure c and k in effective units of labor.

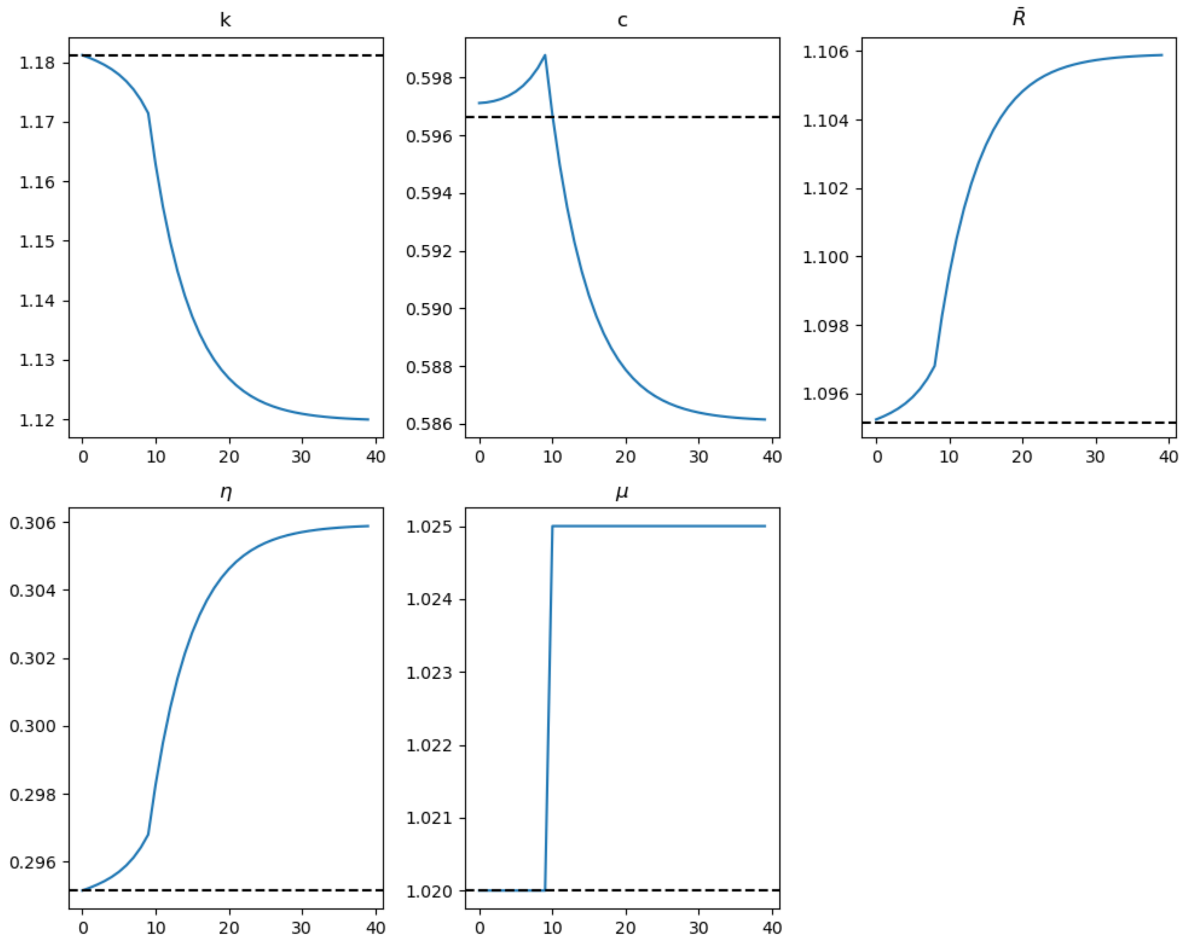
```
shocks = {
    'g': np.repeat(0.2, S + 1),
    'τ_c': np.repeat(0.0, S + 1),
    'τ_k': np.repeat(0.0, S + 1),
    'μ': np.concatenate((np.repeat(1.02, 10), np.repeat(1.025, S - 9)))
}

A_path = compute_A_path(1.0, shocks, S)

k_ss_initial, c_ss_initial = steady_states(model,
                                            shocks['g'][0],
                                            shocks['τ_k'][0],
                                            shocks['μ'][0]
```

(continues on next page)

(continued from previous page)

```
                                        )

print(f"Steady-state capital: {k_ss_initial:.4f}")
print(f"Steady-state consumption: {c_ss_initial:.4f}")

# Run the shooting algorithm with the A_path parameter
solution = run_shooting(shocks, S, model, A_path)

fig, axes = plt.subplots(2, 3, figsize=(10, 8))
axes = axes.flatten()

plot_results(solution, k_ss_initial,
             c_ss_initial, shocks, 'μ', axes, model,
             A_path, T=40)

for ax in axes[5:]:
    fig.delaxes(ax)

plt.tight_layout()
plt.show()
```

```
Steady-state capital: 1.1812
Steady-state consumption: 0.5966
Model: Model(β=0.95, γ=2.0, δ=0.2, α=0.33, A=None)
Optimal initial consumption c0 = 0.5971184749344462396270918337183607339919
```



The results in the figures are mainly driven by (72.29) and imply that a permanent increase in μ will lead to a decrease in the steady-state value of capital per unit of effective labor.

The figures indicate the following:

- As capital becomes more efficient, even with less of it, consumption per capita can be raised.
- Consumption smoothing drives an *immediate jump in consumption* in anticipation of the increase in μ.
- The increased productivity of capital leads to an increase in the gross return $\bar{R}$.
- Perfect foresight makes the effects of the increase in the growth of capital precede it, with the effect visible at $t = 0$.

Experiment 2: An unforeseen increase in μ from 1.02 to 1.025 at t=0

The figures below show the effects of an immediate jump in μ to 1.025 at t=0.

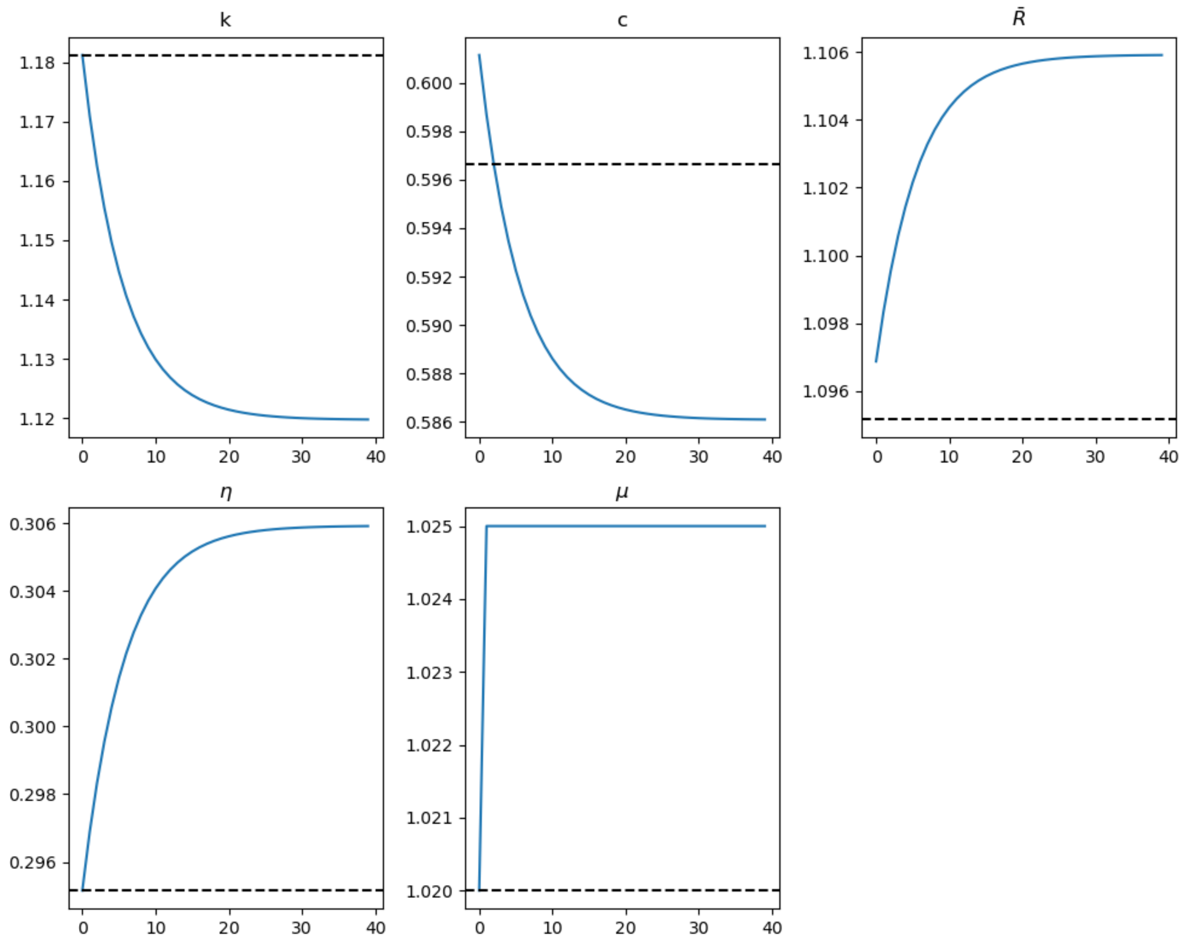
```
shocks = {
    'g': np.repeat(0.2, S + 1),
    'τ_c': np.repeat(0.0, S + 1),
    'τ_k': np.repeat(0.0, S + 1),
    'μ': np.concatenate((np.repeat(1.02, 1), np.repeat(1.025, S)))
}

A_path = compute_A_path(1.0, shocks, S)
```

(continues on next page)

(continued from previous page)

```
k_ss_initial, c_ss_initial = steady_states(model,
                                           shocks['g'][0],
                                           shocks['τ_k'][0],
                                           shocks['μ'][0]
                                          )

print(f"Steady-state capital: {k_ss_initial:.4f}")
print(f"Steady-state consumption: {c_ss_initial:.4f}")

# Run the shooting algorithm with the A_path parameter
solution = run_shooting(shocks, S, model, A_path)

fig, axes = plt.subplots(2, 3, figsize=(10, 8))
axes = axes.flatten()

plot_results(solution, k_ss_initial,
             c_ss_initial, shocks, 'μ', axes, model, A_path, T=40)

for ax in axes[5:]:
    fig.delaxes(ax)

plt.tight_layout()
plt.show()
```

```
Steady-state capital: 1.1812
Steady-state consumption: 0.5966
Model: Model(β=0.95, γ=2.0, δ=0.2, α=0.33, A=None)
Optimal initial consumption c0 = 0.6011494930430641150395883753109588316638
```

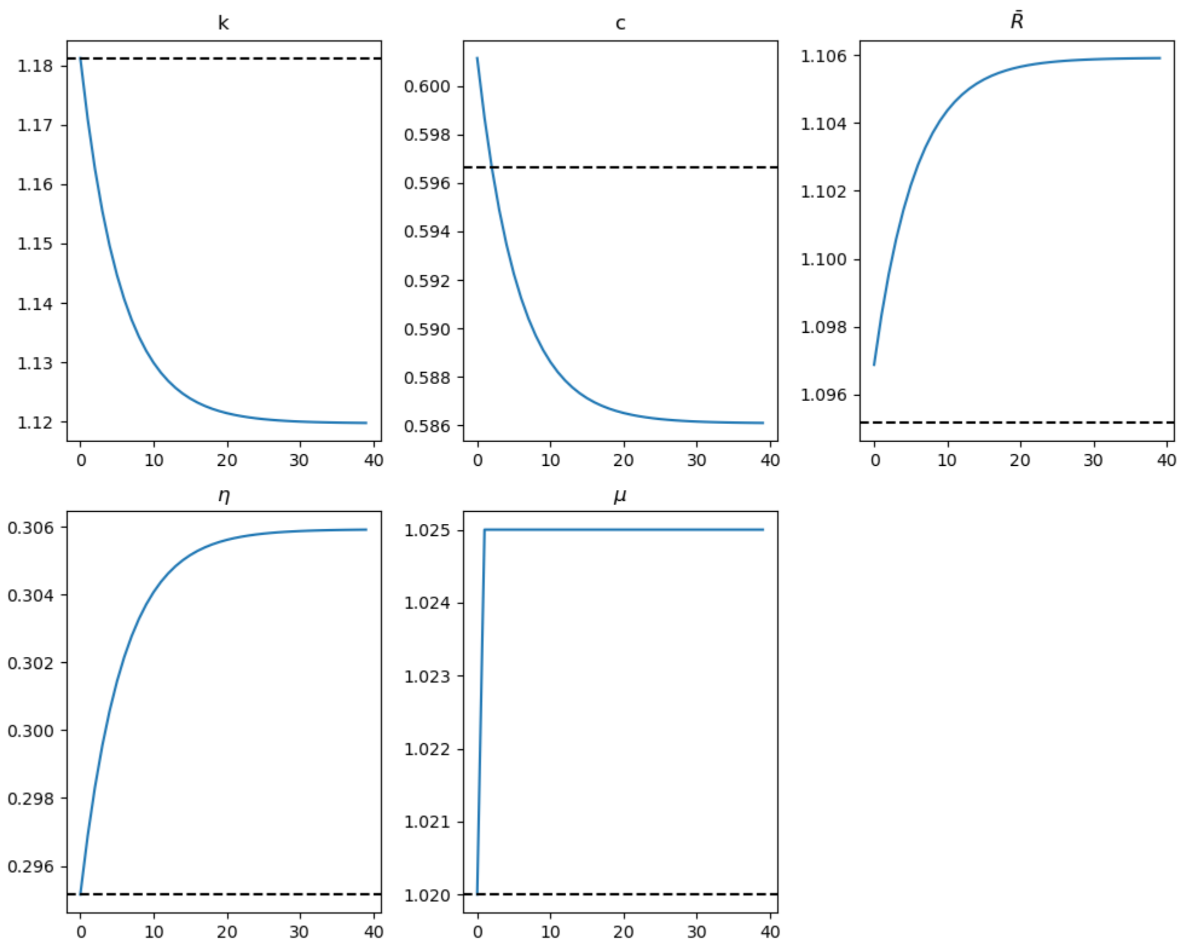


Again, we can collect the procedures used above into a function that runs the solver and draws plots for a given experiment.

```
def experiment_model(shocks, S, model, A_path, solver, plot_func, policy_shock, T=40):
    """
    Run the shooting algorithm given a model and plot the results.
    """
    k0, c0 = steady_states(model, shocks['g'][0], shocks['τ_k'][0], shocks['μ'][0])

    print(f"Steady-state capital: {k0:.4f}")
    print(f"Steady-state consumption: {c0:.4f}")
    print('-'*64)

    fig, axes = plt.subplots(2, 3, figsize=(10, 8))
    axes = axes.flatten()

    solution = solver(shocks, S, model, A_path)
    plot_func(solution, k0, c0,
              shocks, policy_shock, axes, model, A_path, T=T)

    for ax in axes[5:]:
        fig.delaxes(ax)

    plt.tight_layout()
    plt.show()
```

```
shocks = {
    'g': np.repeat(0.2, S + 1),
    'τ_c': np.repeat(0.0, S + 1),
    'τ_k': np.repeat(0.0, S + 1),
    'μ': np.concatenate((np.repeat(1.02, 1), np.repeat(1.025, S)))
}

experiment_model(shocks, S, model, A_path, run_shooting, plot_results, 'μ')
```

```
Steady-state capital: 1.1812
Steady-state consumption: 0.5966
--------------------------------------------------------------
Model: Model(β=0.95, γ=2.0, δ=0.2, α=0.33, A=None)
Optimal initial consumption c0 = 0.6011494930430641150395883753109588316638
```



The figures show that:

- The paths of all variables are now smooth due to the absence of feedforward effects.
- Capital per effective unit of labor gradually declines to a lower steady-state level.
- Consumption per effective unit of labor jumps immediately and then declines smoothly toward its lower steady-state value.
- The after-tax gross return $\bar{R}$ once again co-moves with the consumption growth rate, verifying the Euler equation (72.29).

Exercise 72.10.1

Replicate the plots of our two experiments using the second method of residual minimization:

1. A foreseen increase in μ from 1.02 to 1.025$ at t=10
2. An unforeseen increase in μ from 1.02 to 1.025$ at t=0

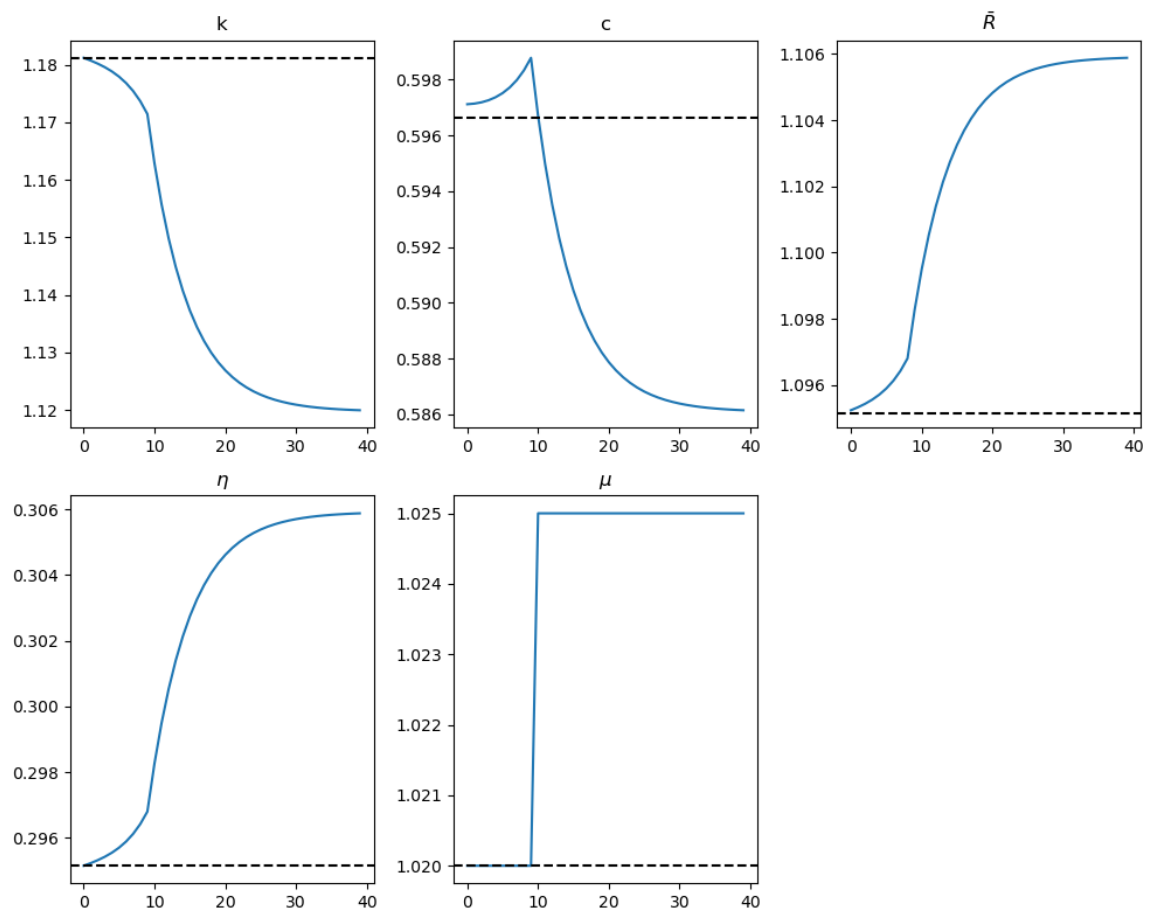
Solution

Here is one solution:

Experiment 1: A foreseen increase in μ from 1.02 to 1.025 at $t = 10$

```
shocks = {
    'g': np.repeat(0.2, S + 1),
    'τ_c': np.repeat(0.0, S + 1),
    'τ_k': np.repeat(0.0, S + 1),
    'μ': np.concatenate((np.repeat(1.02, 10), np.repeat(1.025, S - 9)))
}

A_path = compute_A_path(1.0, shocks, S)

experiment_model(shocks, S, model, A_path, run_min, plot_results, 'μ')
```

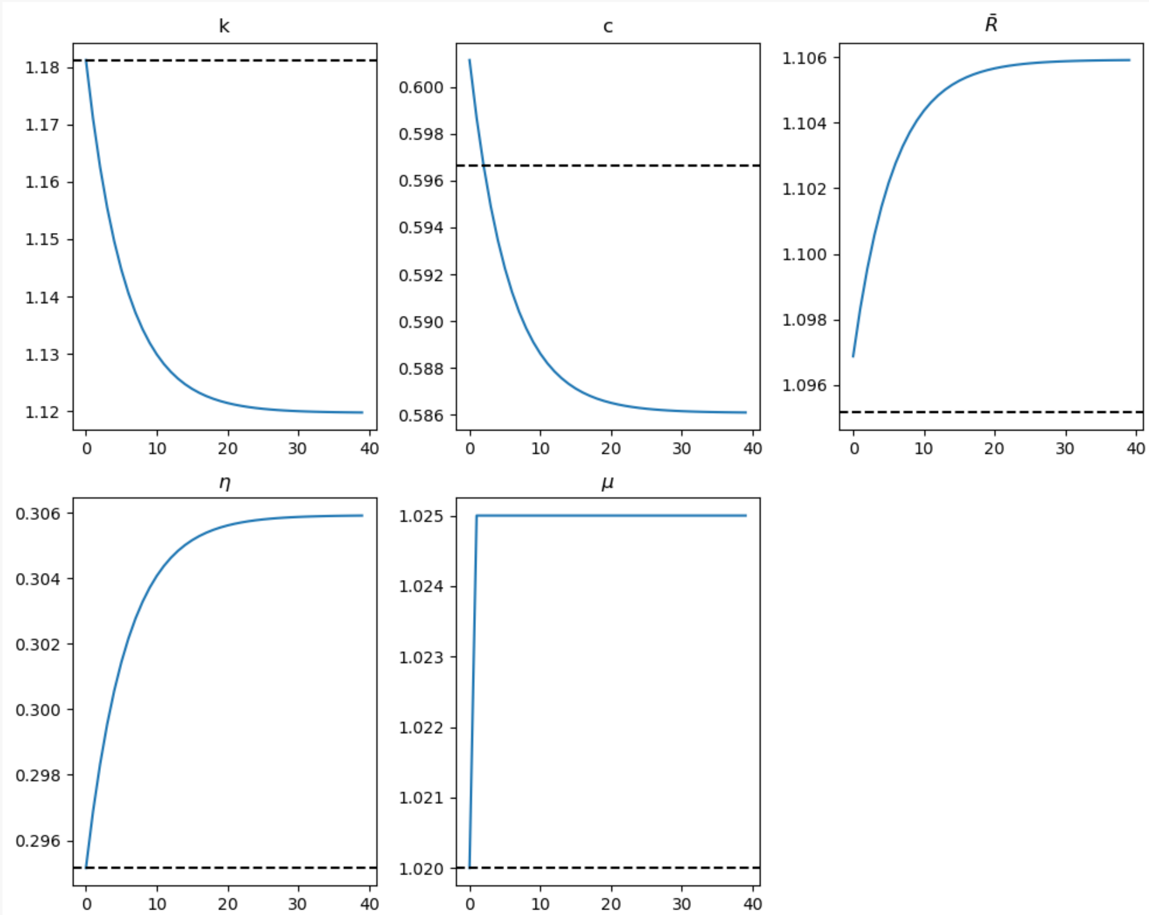
```
Steady-state capital: 1.1812
Steady-state consumption: 0.5966
---------------------------------------------------------------
```



Experiment 2: An unforeseen increase in μ from 1.02 to 1.025 at $t = 0$

```
shocks = {
    'g': np.repeat(0.2, S + 1),
    'τ_c': np.repeat(0.0, S + 1),
    'τ_k': np.repeat(0.0, S + 1),
    'μ': np.concatenate((np.repeat(1.02, 1), np.repeat(1.025, S)))
}

experiment_model(shocks, S, model, A_path, run_min, plot_results, 'μ')
```

```
Steady-state capital: 1.1812
Steady-state consumption: 0.5966
---------------------------------------------------------------
```



In this this sequel *Two-Country Model with Distorting Taxes*, we study a two-country version of our one-country model that is closely related to Mendoza and Tesar [1998].

CHAPTER

SEVENTYTHREE

TWO-COUNTRY MODEL WITH DISTORTING TAXES

73.1 Overview

This lecture is a sequel to this QuantEcon lecture *Cass-Koopmans Model with Distorting Taxes* in which we studied consequences of foreseen fiscal and technology shocks on competitive equilibrium prices and quantities in a nonstochastic version of a Cass-Koopmans growth model like the one described in this QuantEcon lecture *Cass-Koopmans Competitive Equilibrium*.

Here we study a two-country version of that model.

We construct it by putting instances of two *Cass-Koopmans Competitive Equilibrium* economies together back to back, and then opening international trade in some commodities, but not in others.

This lets us focus on some of the issues studied by Mendoza and Tesar [1998].

Let's start with some imports:

```
import numpy as np
from scipy.optimize import root
import matplotlib.pyplot as plt
from collections import namedtuple
from mpmath import mp, mpf
from warnings import warn

# Set the precision
mp.dps = 40
mp.pretty = True
```

73.2 A Two-Country Cass-Koopmans Model

This section describes a two-country version of the basic model of *The Economy*.

The model has a structure similar to ones used in the international real business cycle literature and is in the spirit of an analysis of distorting taxes by Mendoza and Tesar [1998].

We allow two countries to trade goods and claims on future goods, but not labor.

Both countries have production technologies, and consumers in each country can hold capital in either country, subject to different tax treatments.

We denote variables in the second country with asterisks (*).

Households in both countries maximize lifetime utility:

$$\sum_{t=0}^{\infty} \beta^t u(c_t) \quad \text{and} \quad \sum_{t=0}^{\infty} \beta^t u(c_t^*),$$

where $u(c) = \frac{c^{1-\gamma}}{1-\gamma}$ with $\gamma > 0$.

There are Cobb-Douglas functions with identical technology parameters in the two countries.

The world resource constraint in this two-country economy is:

$$(c_t + c_t^*) + (g_t + g_t^*) + (k_{t+1} - (1-\delta)k_t) + (k_{t+1}^* - (1-\delta)k_t^*) = f(k_t) + f(k_t^*)$$

which combines the feasibility constraints for the two countries.

Later, we will use this constraint as a global feasibility constraint in our computation.

To connect the two countries, we need to specify how capital flows across borders and how taxes are levied in different jurisdictions.

73.2.1 Capital Mobility and Taxation

A consumer in country one can hold capital in either country but pays taxes on rentals from foreign holdings of capital at the rate set by the foreign country.

Residents in both countries can purchase consumption at date t at a common Arrow-Debreu price q_t. We assume capital markets are complete.

Let B_t^f be the amount of time t goods that the representative domestic consumer raises by issuing a one-period IOU to the representative foreign consumer.

So $B_t^f > 0$ indicates the domestic consumer is borrowing from abroad at t, and $B_t^f < 0$ indicates the domestic consumer is lending abroad at t.

Hence, the budget constraint of a representative consumer in country one is:

$$\sum_{t=0}^{\infty} q_t \left(c_t + (k_{t+1} - (1-\delta)k_t) + (\tilde{k}_{t+1} - (1-\delta)\tilde{k}_t) + R_{t-1,t} B_{t-1}^f \right) \leq$$
$$\sum_{t=0}^{\infty} q_t \left((\eta_t - \tau_{kt}(\eta_t - \delta))k_t + (\eta_t^* - \tau_{kt}^*(\eta_t^* - \delta))\tilde{k}_t + (1-\tau_{nt})w_t n_t - \tau_{ht} + B_t^f \right).$$

No-arbitrage conditions for k_t and $\tilde{k}_t$ for $t \geq 1$ imply

$$\begin{aligned} q_{t-1} &= [(1-\tau_{kt})(\eta_t - \delta) + 1] q_t, \\ q_{t-1} &= [(1-\tau_{kt}^*)(\eta_t^* - \delta) + 1] q_t, \end{aligned}$$

which together imply that after-tax rental rates on capital are equalized across the two countries:

$$(1-\tau_{kt}^*)(\eta_t^* - \delta) = (1-\tau_{kt})(\eta_t - \delta).$$

The no-arbitrage conditions for B_t^f for $t \geq 0$ are $q_t = q_{t+1} R_{t+1,t}$, which implies that

$$q_{t-1} = q_t R_{t-1,t}$$

for $t \geq 1$.

Since domestic capital, foreign capital, and consumption loans bear the same rates of return, portfolios are indeterminate.

We can set holdings of foreign capital equal to zero in each country if we allow B_t^f to be nonzero.

This way of resolving portfolio indeterminacy is convenient because it reduces the number of initial conditions we need to specify.

Therefore, we set holdings of foreign capital equal to zero in both countries while allowing international lending.

Given an initial level B_{-1}^f of debt from the domestic country to the foreign country, and where $R_{t-1,t} = \frac{q_{t-1}}{q_t}$, international debt dynamics satisfy

$$B_t^f = R_{t-1,t} B_{t-1}^f + c_t + (k_{t+1} - (1-\delta)k_t) + g_t - f(k_t)$$

```
def Bf_path(k, c, g, model):
    """
    Compute B^{f}_t:
      Bf_t = R_{t-1} Bf_{t-1} + c_t + (k_{t+1}-(1-δ)k_t) + g_t - f(k_t)
    with Bf_0 = 0.
    """
    S = len(c) - 1
    R = c[:-1]**(-model.γ) / (model.β * c[1:]**(-model.γ))

    Bf = np.zeros(S + 1)
    for t in range(1, S + 1):
        inv = k[t] - (1 - model.δ) * k[t-1]
        Bf[t] = (
            R[t-1] * Bf[t-1] + c[t] + inv + g[t-1]
            - f(k[t-1], model))
    return Bf

def Bf_ss(c_ss, k_ss, g_ss, model):
    """
    Compute the steady-state B^f
    """
    R_ss   = 1.0 / model.β
    inv_ss = model.δ * k_ss
    num    = c_ss + inv_ss + g_ss - f(k_ss, model)
    den    = 1.0 - R_ss
    return num / den
```

and

$$c_t^* + (k_{t+1}^* - (1-\delta)k_t^*) + g_t^* - R_{t-1,t} B_{t-1}^f = f(k_t^*) - B_t^f.$$

The firms' first-order conditions in the two countries are:

$$\begin{aligned} \eta_t &= f'(k_t), \quad w_t = f(k_t) - k_t f'(k_t) \\ \eta_t^* &= f'(k_t^*), \quad w_t^* = f(k_t^*) - k_t^* f'(k_t^*). \end{aligned}$$

International trade in goods establishes:

$$\frac{q_t}{\beta^t} = \frac{u'(c_t)}{1+\tau_{ct}} = \mu^* \frac{u'(c_t^*)}{1+\tau_{ct}^*},$$

where μ^* is a nonnegative number that is a function of the Lagrange multiplier on the budget constraint for a consumer in country $*$.

We have normalized the Lagrange multiplier on the budget constraint of the domestic country to set the corresponding μ for the domestic country to unity.

```
def compute_rs(c_t, c_tp1, c_s_t, c_s_tp1, τc_t,
               τc_tp1, τc_s_t, τc_s_tp1, model):
    """
    Compute international risk sharing after trade starts.
    """

    return (c_t**(-model.γ)/(1+τc_t)) * ((1+τc_s_t)/c_s_t**(-model.γ)) - (
        c_tp1**(-model.γ)/(1+τc_tp1)) * ((1+τc_s_tp1)/c_s_tp1**(-model.γ))
```

Equilibrium requires that the following two national Euler equations be satisfied for $t \geq 0$:

$$
\begin{aligned}
u'(c_t) &= \beta u'(c_{t+1}) \left[(1-\tau_{kt+1})(f'(k_{t+1})-\delta)+1\right] \left[\frac{1+\tau_{ct+1}}{1+\tau_{ct}}\right], \\
u'(c_t^*) &= \beta u'(c_{t+1}^*) \left[(1-\tau_{kt+1}^*)(f'(k_{t+1}^*)-\delta)+1\right] \left[\frac{1+\tau_{ct+1}^*}{1+\tau_{ct}^*}\right].
\end{aligned}
$$

The following code computes both the domestic and foreign Euler equations.

Since they have the same form but use different variables, we can write a single function that handles both cases.

```
def compute_euler(c_t, c_tp1, τc_t,
                  τc_tp1, τk_tp1, k_tp1, model):
    """
    Compute the Euler equation.
    """
    Rbar = (1 - τk_tp1)*(f_prime(k_tp1, model) - model.δ) + 1
    return model.β * (c_tp1/c_t)**(-model.γ) * (1+τc_t)/(1+τc_tp1) * Rbar - 1
```

73.2.2 Initial condition and steady state

For the initial conditions, we choose the pre-trade allocation of capital (k_0, k_0^*) and the initial level B_{-1}^f of international debt owed by the unstarred (domestic) country to the starred (foreign) country.

73.2.3 Equilibrium steady state values

The steady state of the two-country model is characterized by two sets of equations.

First, the following equations determine the steady-state capital-labor ratios $\bar{k}$ and $\bar{k}^*$ in each country:

$$
f'(\bar{k}) = \delta + \frac{\rho}{1-\tau_k} \tag{73.1}
$$

$$
f'(\bar{k}^*) = \delta + \frac{\rho}{1-\tau_k^*} \tag{73.2}
$$

Given these steady-state capital-labor ratios, the domestic and foreign consumption values $\bar{c}$ and $\bar{c}^*$ are determined by:

$$
(\bar{c}+\bar{c}^*) = f(\bar{k}) + f(\bar{k}^*) - \delta(\bar{k}+\bar{k}^*) - (\bar{g}+\bar{g}^*) \tag{73.3}
$$

$$
\bar{c} = f(\bar{k}) - \delta\bar{k} - \bar{g} - \rho\bar{B}^f \tag{73.4}
$$

Equation (73.3) expresses feasibility at the steady state, while equation (73.4) represents the trade balance, including interest payments, at the steady state.

The steady-state level of debt $\bar{B}^f$ from the domestic country to the foreign country influences the consumption allocation between countries but not the total world capital stock.

We assume $\bar{B}^f = 0$ in the steady state, which gives us the following function to compute the steady-state values of capital and consumption

```
def compute_steady_state_global(model, g_ss=0.2):
    """
    Calculate steady state values for capital, consumption, and investment.
    """
    k_ss = ((1/model.β - (1-model.δ)) / (model.A * model.α)) ** (1/(model.α-1))
    c_ss = f(k_ss, model) - model.δ * k_ss - g_ss
    return k_ss, c_ss
```

Now, we can apply the residual minimization method to compute the steady-state values of capital and consumption.

Again, we minimize the residuals of the Euler equation, the global resource constraint, and the no-arbitrage condition.

```
def compute_residuals_global(z, model, shocks, T, k0_ss, k_star, Bf_star):
    """
    Compute residuals for the two-country model.
    """
    k, c, k_s, c_s = z.reshape(T+1, 4).T
    g, gs = shocks['g'], shocks['g_s']
    τc, τk = shocks['τ_c'], shocks['τ_k']
    τc_s, τk_s = shocks['τ_c_s'], shocks['τ_k_s']

    res = [k[0] - k0_ss, k_s[0] - k0_ss]

    for t in range(T):
        e_d = compute_euler(
            c[t], c[t+1],
            τc[t], τc[t+1], τk[t+1],
            k[t+1], model)

        e_f = compute_euler(
            c_s[t], c_s[t+1],
            τc_s[t], τc_s[t+1], τk_s[t+1],
            k_s[t+1], model)

        rs = compute_rs(
            c[t], c[t+1], c_s[t], c_s[t+1],
            τc[t], τc[t+1], τc_s[t], τc_s[t+1],
            model)

        # Global resource constraint
        grc = k[t+1] + k_s[t+1] - (
            f(k[t], model) + f(k_s[t], model) +
            (1-model.δ)*(k[t] + k_s[t]) -
            c[t] - c_s[t] - g[t] - gs[t]
        )

        res.extend([e_d, e_f, rs, grc])

    Bf_term = Bf_path(k, c, shocks['g'], model)[-1]
    res.append(k[T] - k_star)
    res.append(Bf_term - Bf_star)
    return np.array(res)
```

Now we plot the results

```python
# Function to plot global two-country model results
def plot_global_results(k, k_s, c, c_s, shocks, model,
                        k0_ss, c0_ss, g_ss, S, T=40, shock='g',
                        # a dictionary storing sequence for lower left panel
                        ll_series='None'):
    """
    Plot results for the two-country model.
    """
    fig, axes = plt.subplots(2, 3, figsize=(10, 8))
    x = np.arange(T)
    τc, τk = shocks['τ_c'], shocks['τ_k']
    Bf = Bf_path(k, c, shocks['g'], model)

    # Compute derived series
    R_ratio = c[:-1]**(-model.γ) / (model.β * c[1:]**(-model.γ)) \
    *(1+τc[:-1])/(1+τc[1:])
    inv = k[1:] - (1-model.δ)*k[:-1]
    inv_s = k_s[1:] - (1-model.δ)*k_s[:-1]

    # Add initial conditions into the series
    R_ratio = np.append(1/model.β, R_ratio)
    c = np.append(c0_ss, c)
    c_s = np.append(c0_ss, c_s)
    k = np.append(k0_ss, k)
    k_s = np.append(k0_ss, k_s)

    # Capital
    axes[0,0].plot(x, k[:T], '-', lw=1.5)
    axes[0,0].plot(x, np.full(T, k0_ss), 'k-.', lw=1.5)
    axes[0,0].plot(x, k_s[:T], '--', lw=1.5)
    axes[0,0].set_title('k')
    axes[0,0].set_xlim(0, T-1)

    # Consumption
    axes[0,1].plot(x, c[:T], '-', lw=1.5)
    axes[0,1].plot(x, np.full(T, c0_ss), 'k-.', lw=1.5)
    axes[0,1].plot(x, c_s[:T], '--', lw=1.5)
    axes[0,1].set_title('c')
    axes[0,1].set_xlim(0, T-1)

    # Interest rate
    axes[0,2].plot(x, R_ratio[:T], '-', lw=1.5)
    axes[0,2].plot(x, np.full(T, 1/model.β), 'k-.', lw=1.5)
    axes[0,2].set_title(r'$\bar{R}$')
    axes[0,2].set_xlim(0, T-1)

    # Investment
    axes[1,0].plot(x, np.full(T, model.δ * k0_ss),
    'k-.', lw=1.5)
    axes[1,0].plot(x, np.append(model.δ*k0_ss, inv[:T-1]),
    '-', lw=1.5)
    axes[1,0].plot(x, np.append(model.δ*k0_ss, inv_s[:T-1]),
    '--', lw=1.5)
    axes[1,0].set_title('x')
    axes[1,0].set_xlim(0, T-1)

    # Shock
```

(continues on next page)

(continued from previous page)

```
    axes[1,1].plot(x, shocks[shock][:T], '-', lw=1.5)
    axes[1,1].plot(x, np.full(T, shocks[shock][0]), 'k-.', lw=1.5)
    axes[1,1].set_title(f'${shock}$')
    axes[1,1].set_ylim(-0.1, 0.5)
    axes[1,1].set_xlim(0, T-1)

    # Capital flow
    axes[1,2].plot(x, np.append(0, Bf[1:T]), lw=1.5)
    axes[1,2].plot(x, np.zeros(T), 'k-.', lw=1.5)
    axes[1,2].set_title(r'$B^{f}$')
    axes[1,2].set_xlim(0, T-1)

    plt.tight_layout()
    return fig, axes
```

As in our in the one-country model in *Cass-Koopmans Model with Distorting Taxes*, we assume a Cobb-Douglas production function:

$$F(k,1) = Ak^{\alpha}$$

```
def f(k, model, A=1):
    """
    Production function: f(k) = A * k^{α}
    """
    return A * k ** model.α

def f_prime(k, model, A=1):
    """
    Marginal product of capital: f'(k) = α * A * k^{α - 1}
    """
    return model.α * A * k ** (model.α - 1)
```

Similarly, we define the capital rental rate

$$\eta_t = f'(k_t)$$

```
def compute_η_path(k_path, model, S=100, A_path=None):
    """
    Compute η path: η_t = f'(k_t)
    with optional A_path for growth models.
    """
    A = np.ones_like(k_path) if A_path is None else np.asarray(A_path)
    η_path = np.zeros_like(k_path)
    for t in range(S):
        η_path[t] = f_prime(k_path[t], model, A[t])
    return η_path
```

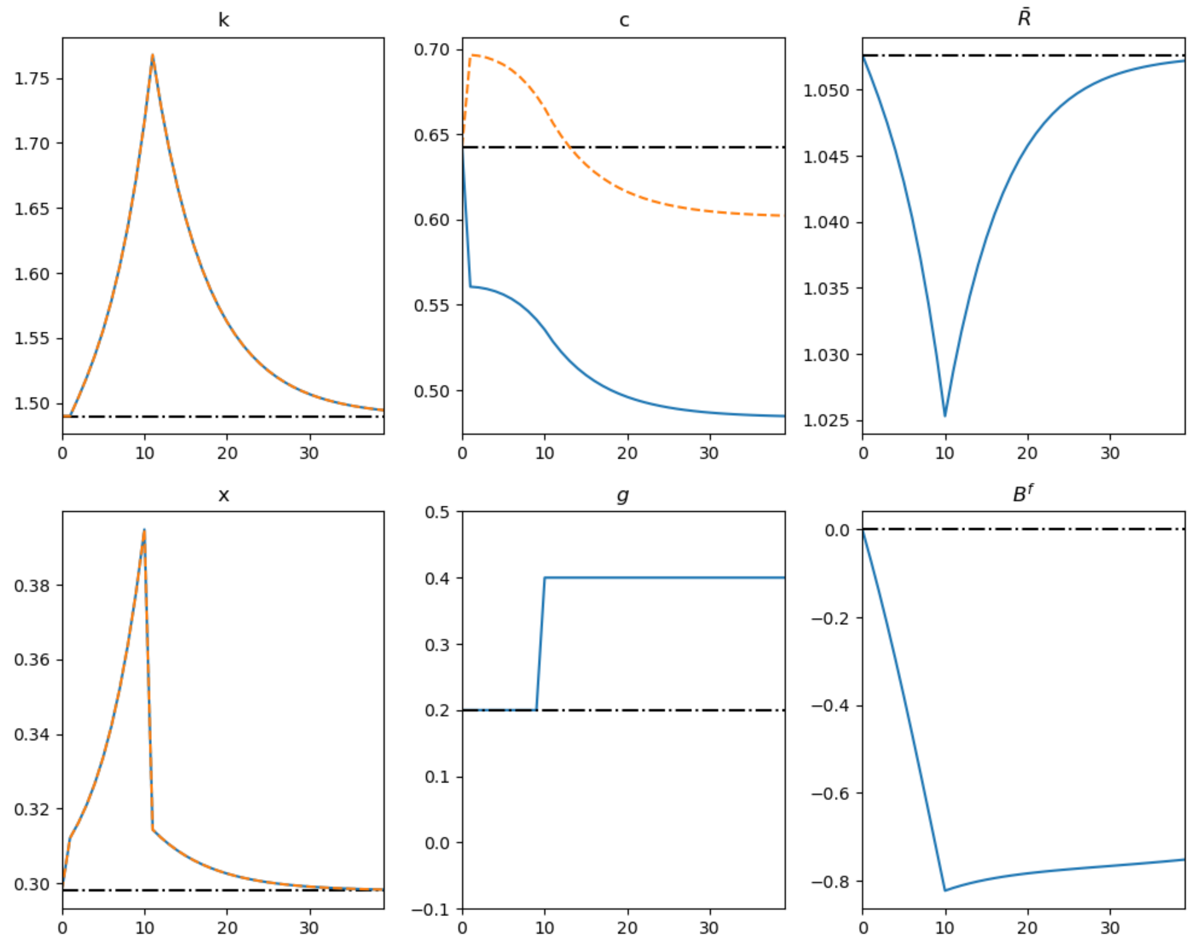
Experiment 1: A foreseen increase in g from 0.2 to 0.4 at t=10

The figure below presents transition dynamics after an increase in g in the domestic economy from 0.2 to 0.4 that is announced ten periods in advance.

We start both economies from a steady state with $B_0^f = 0$.

In the figure below, the blue lines represent the domestic economy and orange dotted lines represent the foreign economy.

```
Model = namedtuple("Model", ["β", "γ", "δ", "α", "A"])
model = Model(β=0.95, γ=2.0, δ=0.2, α=0.33, A=1.0)
S = 100

shocks_global = {
    'g': np.concatenate((np.full(10, 0.2), np.full(S-9, 0.4))),
    'g_s': np.full(S+1, 0.2),
    'τ_c': np.zeros(S+1),
    'τ_k': np.zeros(S+1),
    'τ_c_s': np.zeros(S+1),
    'τ_k_s': np.zeros(S+1)
}
g_ss = 0.2
k0_ss, c0_ss = compute_steady_state_global(model, g_ss)

k_star = k0_ss
Bf_star = Bf_ss(c0_ss, k_star, g_ss, model)

init_glob = np.tile([k0_ss, c0_ss, k0_ss, c0_ss], S+1)
sol_glob = root(
    lambda z: compute_residuals_global(z, model, shocks_global,
                                        S, k0_ss, k_star, Bf_star),
    init_glob, tol=1e-12
)
k, c, k_s, c_s = sol_glob.x.reshape(S+1, 4).T

# Plot global results via function
plot_global_results(k, k_s, c, c_s,
                        shocks_global, model,
                        k0_ss, c0_ss, g_ss,
                        S)
plt.show()
```



At time 1, the government announces that domestic government purchases g will rise ten periods later, cutting into future private resources.

To smooth consumption, domestic households immediately increase saving, offsetting the anticipated hit to their future wealth.

In a closed economy, they would save solely by accumulating extra domestic capital; with open capital markets, they can also lend to foreigners.

Once the capital flow opens up at time 1, the no-arbitrage conditions connect adjustments of both types of saving: the increase in savings by domestic households will reduce the equilibrium return on bonds and capital in the foreign economy to prevent arbitrage opportunities.

Because no-arbitrage equalizes the ratio of marginal utilities, the resulting paths of consumption and capital are synchronized across the two economies.

Up to the date the higher g takes effect, both countries continue to build their capital stocks.

When government spending finally rises 10 periods later, domestic households begin to draw down part of that capital to cushion consumption.

Again by no-arbitrage conditions, when g actually increases, both countries reduce their investment rates.

The domestic economy, in turn, starts running current-account deficits partially to fund the increase in g.

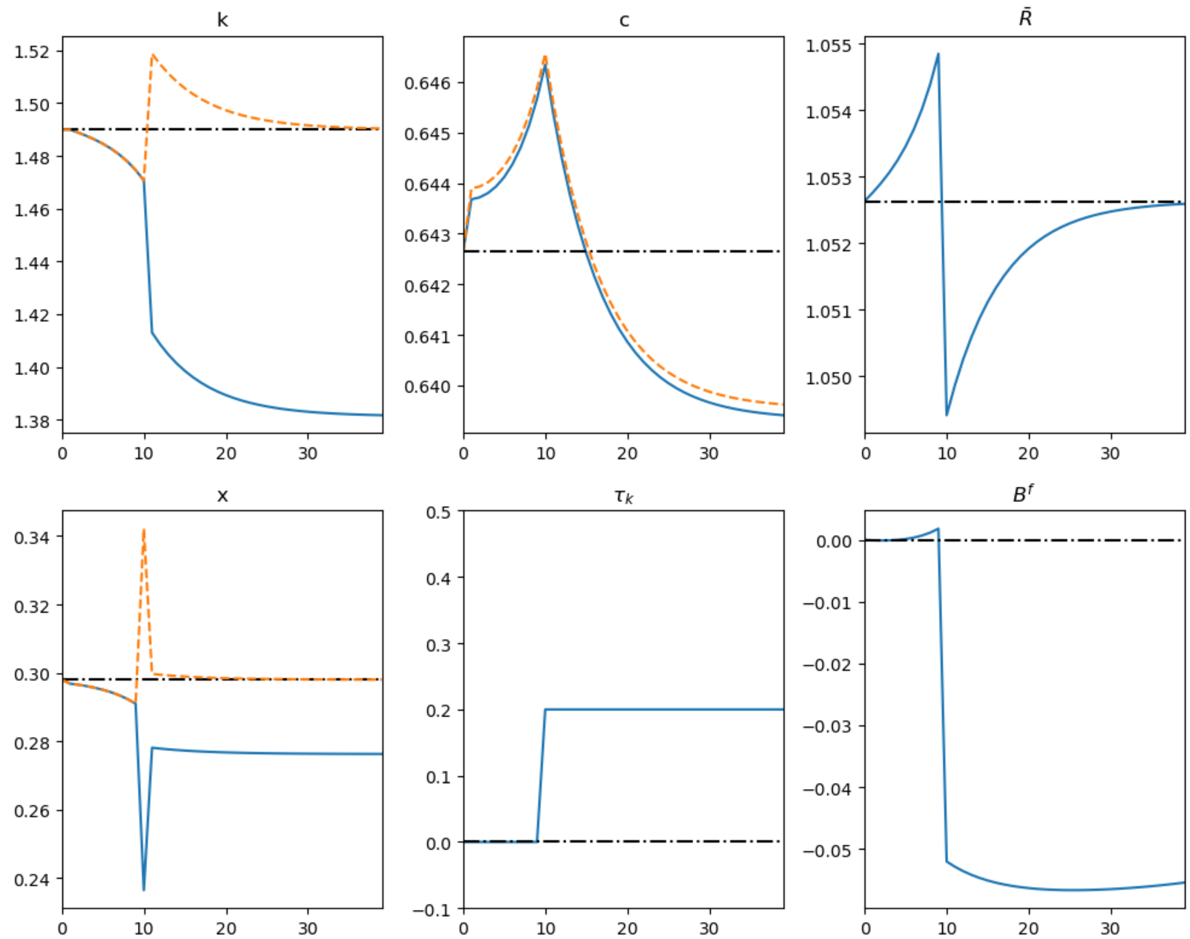
This means that foreign households begin repaying part of their external debt by reducing their capital stock.

Experiment 2: A foreseen increase in g from 0.2 to 0.4 at t=10

We now explore the impact of an increase in capital taxation in the domestic economy 10 periods after its announcement at $t = 1$.

Because the change is anticipated, households in both countries adjust immediately—even though the tax does not take effect until period $t = 11$.

```
shocks_global = {
    'g': np.full(S+1, g_ss),
    'g_s': np.full(S+1, g_ss),
    'τ_c': np.zeros(S+1),
    'τ_k': np.concatenate((np.zeros(10), np.full(S-9, 0.2))),
    'τ_c_s': np.zeros(S+1),
    'τ_k_s': np.zeros(S+1),
}

k0_ss, c0_ss = compute_steady_state_global(model, g_ss)
k_star = k0_ss
Bf_star = Bf_ss(c0_ss, k_star, g_ss, model)

init_glob = np.tile([k0_ss, c0_ss, k0_ss, c0_ss], S+1)

sol_glob = root(
    lambda z: compute_residuals_global(z, model,
            shocks_global, S, k0_ss, k_star, Bf_star),
            init_glob, tol=1e-12)

k, c, k_s, c_s = sol_glob.x.reshape(S+1, 4).T

# plot
fig, axes = plot_global_results(k, k_s, c, c_s, shocks_global, model,
                                k0_ss, c0_ss, g_ss, S, shock='τ_k')
plt.tight_layout()
plt.show()
```



After the tax increase is announced, domestic households foresee lower after-tax returns on capital, so they shift toward higher present consumption and allow the domestic capital stock to decline.

This shrinkage of the world capital supply drives the global real interest rate upward, prompting foreign households to raise current consumption as well.

Prior to the actual tax hike, the domestic economy finances part of its consumption by importing capital, generating a current-account deficit.

When τ_k finally rises, international arbitrage leads investors to reallocate capital quickly toward the untaxed foreign market, compressing the yield on bonds everywhere.

The bond-rate drop reflects the lower after-tax return on domestic capital and the higher foreign capital stock, which depresses its marginal product.

Foreign households fund their capital purchases by borrowing abroad, creating a pronounced current-account deficit and a buildup of external debt.

After the policy change, both countries move smoothly toward a new steady state in which:

- Consumption levels in each economy settle below their pre-announcement paths.
- Capital stocks differ just enough to equalize after-tax returns across borders.

Despite carrying positive net liabilities, the foreign country enjoys higher steady-state consumption because its larger capital stock yields greater output.

The episode demonstrates how open capital markets transmit a domestic tax shock internationally: capital flows and interest-rate movements share the burden, smoothing consumption adjustments in both the taxed and untaxed economies over time.

Exercise 73.2.1

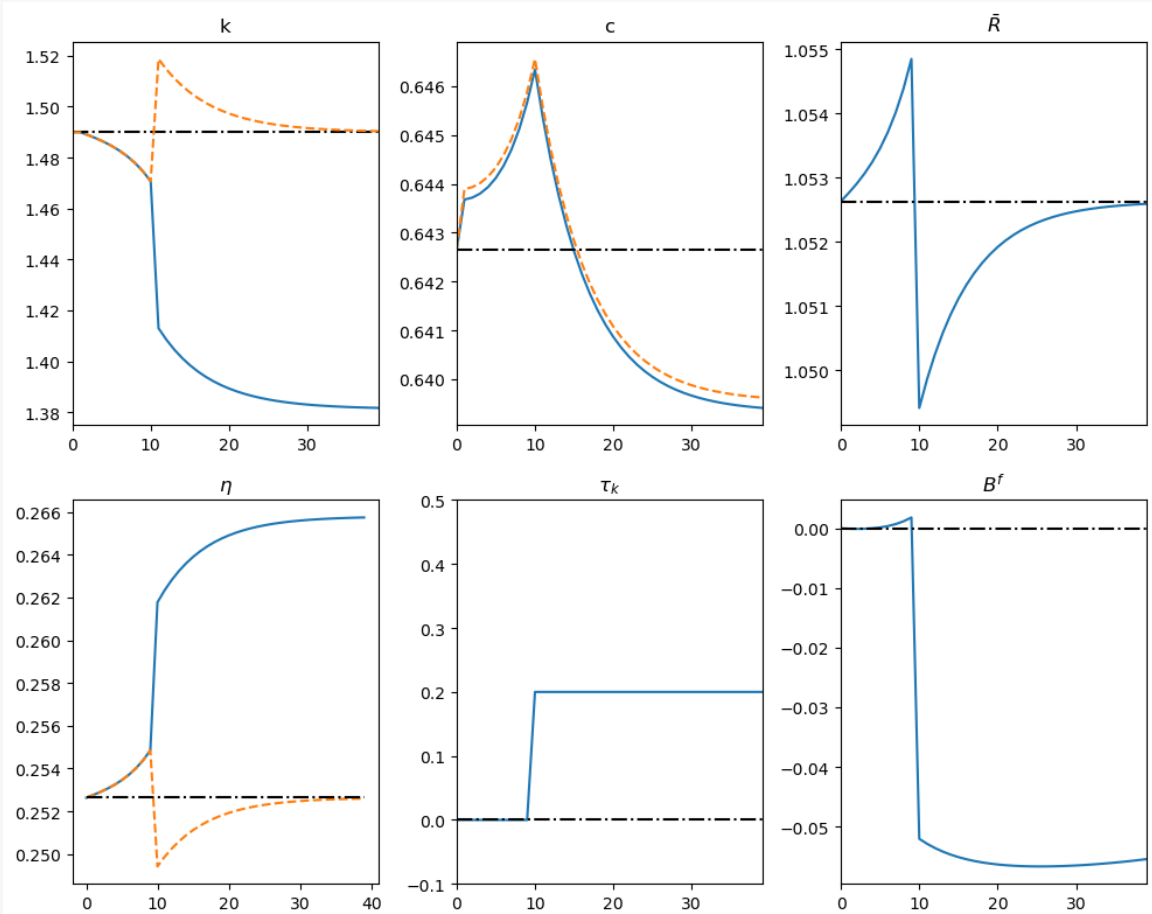
In this exercise, replace the plot for x_t with η_t to replicate the figure in [Ljungqvist and Sargent, 2018].

Compare the figures for k_t and η_t and discuss the economic intuition.

Solution

Here is one solution.

```
fig, axes = plot_global_results(k, k_s, c, c_s, shocks_global, model,
                                k0_ss, c0_ss, g_ss, S, shock='τ_k')

# Clear the plot for x_t
axes[1,0].cla()

# Plot η_t
axes[1,0].plot(compute_η_path(k, model)[:40])
axes[1,0].plot(compute_η_path(k_s, model)[:40], '--')
axes[1,0].plot(np.full(40, f_prime(k_s, model)[0]), 'k-.', lw=1.5)
axes[1,0].set_title(r'$\eta$')

plt.tight_layout()
plt.show()
```



When capital k_t decreases in the domestic country after the tax shock, the rental rate η_t increases in that country.

This happens because when capital becomes scarcer, its marginal product rises.

CHAPTER

SEVENTYFOUR

TRANSITIONS IN AN OVERLAPPING GENERATIONS MODEL

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install --upgrade quantecon
```

74.1 Introduction

This lecture presents a life-cycle model consisting of overlapping generations of two-period lived people proposed by Peter Diamond [Diamond, 1965].

We'll present the version that was analyzed in chapter 2 of Auerbach and Kotlikoff (1987) [Auerbach and Kotlikoff, 1987].

Auerbach and Kotlikoff (1987) used their two period model as a warm-up for their analysis of overlapping generation models of long-lived people that is the main topic of their book.

Their model of two-period lived overlapping generations is a useful starting point because

- it sets forth the structure of interactions between generations of different agents who are alive at a given date
- it activates forces and tradeoffs confronting the government and successive generations of people
- it is good laboratory for studying connections between government tax and subsidy programs and for policies for issuing and servicing government debt
- some interesting experiments involving transitions from one steady state to another can be computed by hand
- it is a good setting for illustrating a **shooting method** for solving a system of non-linear difference equations with initial and terminal condition

> **Note**
>
> Auerbach and Kotlikoff use computer code to calculate transition paths for their models with long-lived people.

We take the liberty of extending Auerbach and Kotlikoff's chapter 2 model to study some arrangements for redistributing resources across generations

- these take the form of a sequence of age-specific lump sum taxes and transfers

We study how these arrangements affect capital accumulation and government debt

74.2 Setting

Time is discrete and is indexed by $t = 0, 1, 2, ...$.

The economy lives forever, but the people inside it do not.

At each time $t \geq 0$ a representative old person and a representative young person are alive.

At time t a representative old person coexists with a representative young person who will become an old person at time $t + 1$.

We assume that the population size is constant over time.

A young person works, saves, and consumes.

An old person dissaves and consumes, but does not work,

A government lives forever, i.e., at $t = 0, 1, 2, ...$.

Each period $t \geq 0$, the government taxes, spends, transfers, and borrows.

Initial conditions set outside the model at time $t = 0$ are

- K_0 – initial capital stock brought into time $t = 0$ by a representative initial old person
- D_0 – government debt falling due at $t = 0$ and owned by a representative old person at time $t = 0$

K_0 and D_0 are both measured in units of time 0 goods.

A government **policy** consists of five sequences $\{G_t, D_t, \tau_t, \delta_{ot}, \delta_{yt}\}_{t=0}^{\infty}$ whose components are

- τ_t – flat rate tax at time t on wages and earnings from capital and government bonds
- D_t – one-period government bond principal due at time t, per capita
- G_t – government purchases of goods at time t, per capita
- δ_{yt} – a lump sum tax on each young person at time t
- δ_{ot} – a lump sum tax on each old person at time t

An **allocation** is a collection of sequences $\{C_{yt}, C_{ot}, K_{t+1}, L_t, Y_t, G_t\}_{t=0}^{\infty}$; constituents of the sequences include

- K_t – physical capital per capita
- L_t – labor per capita
- Y_t – output per capita

and also

- C_{yt} – consumption of young person at time $t \geq 0$
- C_{ot} – consumption of old person at time $t \geq 0$
- $K_{t+1} - K_t \equiv I_t$ – investment in physical capital at time $t \geq 0$
- G_t – government purchases

National income and product accounts consist of a sequence of equalities

- $Y_t = C_{yt} + C_{ot} + (K_{t+1} - K_t) + G_t, \quad t \geq 0$

A **price system** is a pair of sequences $\{W_t, r_t\}_{t=0}^{\infty}$; constituents of a price sequence include rental rates for the factors of production

- W_t – rental rate for labor at time $t \geq 0$
- r_t – rental rate for capital at time $t \geq 0$

74.3 Production

There are two factors of production, physical capital K_t and labor L_t.

Capital does not depreciate.

The initial capital stock K_0 is owned by the representative initial old person, who rents it to the firm at time 0.

Net investment rate I_t at time t is

$$I_t = K_{t+1} - K_t$$

The capital stock at time t emerges from cumulating past rates of investment:

$$K_t = K_0 + \sum_{s=0}^{t-1} I_s$$

A Cobb-Douglas technology converts physical capital K_t and labor services L_t into output Y_t

$$Y_t = K_t^\alpha L_t^{1-\alpha}, \quad \alpha \in (0,1) \tag{74.1}$$

74.4 Government

At time $t-1$, the government issues one-period risk-free debt that promises to pay D_t time t goods per capita at time t.

Young people at time t purchase government debt D_{t+1} that matures at time $t+1$.

Government debt issued at t bears a before-tax net rate of interest rate of r_t at time $t+1$.

The government budget constraint at time $t \geq 0$ is

$$D_{t+1} - D_t = r_t D_t + G_t - T_t$$

or

$$D_{t+1} = (1 + r_t) D_t + G_t - T_t. \tag{74.2}$$

Total tax collections net of transfers equal T_t and satisfy

$$T_t = \tau_t W_t L_t + \tau_t r_t (D_t + K_t) + \delta_{yt} + \delta_{ot}$$

74.5 Activities in Factor Markets

Old people: At each $t \geq 0$, a representative old person

- brings K_t and D_t into the period,
- rents capital to a representative firm for $r_t K_t$,
- pays taxes $\tau_t r_t (K_t + D_t)$ on its rental and interest earnings,
- pays a lump sum tax δ_{ot} to the government,
- sells K_t to a young person.

Young people: At each $t \geq 0$, a representative young person

- sells one unit of labor services to a representative firm for W_t in wages,
- pays taxes $\tau_t W_t$ on its labor earnings
- pays a lump sum tax δ_{yt} to the goverment,
- spends C_{yt} on consumption,
- acquires non-negative assets A_{t+1} consisting of a sum of physical capital K_{t+1} and one-period government bonds D_{t+1} that mature at $t+1$.

Note

If a lump-sum tax is negative, it means that the government pays the person a subsidy.

74.6 Representative firm's problem

The representative firm hires labor services from young people at competitive wage rate W_t and hires capital from old people at competitive rental rate r_t.

The rental rate on capital r_t equals the interest rate on government one-period bonds.

Units of the rental rates are:

- for W_t, output at time t per unit of labor at time t
- for r_t, output at time t per unit of capital at time t

We take output at time t as **numeraire**, so the price of output at time t is one.

The firm's profits at time t are

$$K_t^\alpha L_t^{1-\alpha} - r_t K_t - W_t L_t.$$

To maximize profits a firm equates marginal products to rental rates:

$$\begin{aligned} W_t &= (1-\alpha) K_t^\alpha L_t^{-\alpha} \\ r_t &= \alpha K_t^\alpha L_t^{1-\alpha} \end{aligned} \tag{74.3}$$

Output can be consumed either by old people or young people; or sold to young people who use it to augment the capital stock; or sold to the government for uses that do not generate utility for the people in the model (i.e., "it is thrown into the ocean").

The firm thus sells output to old people, young people, and the government.

74.7 Individuals' problems

74.7.1 Initial old person

At time $t = 0$, a representative initial old person is endowed with $(1 + r_0(1-\tau_0))A_0$ in initial assets.

It must pay a lump sum tax to (if positive) or receive a subsidy from (if negative) δ_{ot} the government.

An old person's budget constraint is

$$C_{o0} = (1 + r_0(1-\tau_0))A_0 - \delta_{ot}. \tag{74.4}$$

An initial old person's utility function is C_{o0}, so the person's optimal consumption plan is provided by equation (74.4).

74.7.2 Young person

At each $t \geq 0$, a young person inelastically supplies one unit of labor and in return receives pre-tax labor earnings of W_t units of output.

A young person's post-tax-and-transfer earnings are $W_t(1-\tau_t) - \delta_{yt}$.

At each $t \geq 0$, a young person chooses a consumption plan C_{yt}, C_{ot+1} to maximize the Cobb-Douglas utility function

$$U_t = C_{yt}^{\beta} C_{o,t+1}^{1-\beta}, \quad \beta \in (0,1) \tag{74.5}$$

subject to the following budget constraints at times t and $t+1$:

$$\begin{aligned} C_{yt} + A_{t+1} &= W_t(1-\tau_t) - \delta_{yt} \\ C_{ot+1} &= (1 + r_{t+1}(1-\tau_{t+1}))A_{t+1} - \delta_{ot} \end{aligned} \tag{74.6}$$

Solving the second equation of (74.6) for savings A_{t+1} and substituting it into the first equation implies the present value budget constraint

$$C_{yt} + \frac{C_{ot+1}}{1 + r_{t+1}(1-\tau_{t+1})} = W_t(1-\tau_t) - \delta_{yt} - \frac{\delta_{ot}}{1 + r_{t+1}(1-\tau_{t+1})} \tag{74.7}$$

To solve the young person's choice problem, form a Lagrangian

$$\begin{aligned} \mathcal{L} &= C_{yt}^{\beta} C_{o,t+1}^{1-\beta} \\ &+ \lambda \Big[C_{yt} + \frac{C_{ot+1}}{1 + r_{t+1}(1-\tau_{t+1})} - W_t(1-\tau_t) + \delta_{yt} + \frac{\delta_{ot}}{1 + r_{t+1}(1-\tau_{t+1})} \Big], \end{aligned} \tag{74.8}$$

where λ is a Lagrange multiplier on the intertemporal budget constraint (74.7).

After several lines of algebra, the intertemporal budget constraint (74.7) and the first-order conditions for maximizing $\mathcal{L}$ with respect to C_{yt}, C_{ot+1} imply that an optimal consumption plan satisfies

$$\begin{aligned} C_{yt} &= \beta \Big[W_t(1-\tau_t) - \delta_{yt} - \frac{\delta_{ot}}{1 + r_{t+1}(1-\tau_{t+1})} \Big] \\ \frac{C_{0t+1}}{1 + r_{t+1}(1-\tau_{t+1})} &= (1-\beta) \Big[W_t(1-\tau_t) - \delta_{yt} - \frac{\delta_{ot}}{1 + r_{t+1}(1-\tau_{t+1})} \Big] \end{aligned} \tag{74.9}$$

The first-order condition for minimizing Lagrangian (74.8) with respect to the Lagrange multipler λ recovers the budget constraint (74.7), which, using (74.9) gives the optimal savings plan

$$A_{t+1} = (1-\beta)[(1-\tau_t)W_t - \delta_{yt}] + \beta \frac{\delta_{ot}}{1 + r_{t+1}(1-\tau_{t+1})} \tag{74.10}$$

74.8 Equilbrium

Definition: An equilibrium is an allocation, a government policy, and a price system with the properties that

- given the price system and the government policy, the allocation solves
  - representative firms' problems for $t \geq 0$
  - individual persons' problems for $t \geq 0$
- given the price system and the allocation, the government budget constraint is satisfied for all $t \geq 0$.

74.9 Next steps

To begin our analysis of equilibrium outcomes, we'll study the special case of the model with which Auerbach and Kotlikoff (1987) [Auerbach and Kotlikoff, 1987] began their analysis in chapter 2.

It can be solved by hand.

We shall do that next.

After we derive a closed form solution, we'll pretend that we don't know and will compute equilibrium outcome paths.

We'll do that by first formulating an equilibrium as a fixed point of a mapping from sequences of factor prices and tax rates to sequences of factor prices and tax rates.

We'll compute an equilibrium by iterating to convergence on that mapping.

74.10 Closed form solution

To get the special chapter 2 case of Auerbach and Kotlikoff (1987) [Auerbach and Kotlikoff, 1987], we set both δ_{ot} and δ_{yt} to zero.

As our special case of (74.9), we compute the following consumption-savings plan for a representative young person:

$$\begin{aligned} C_{yt} &= \beta(1-\tau_t)W_t \\ A_{t+1} &= (1-\beta)(1-\tau_t)W_t \end{aligned}$$

Using (74.3) and $A_t = K_t + D_t$, we obtain the following closed form transition law for capital:

$$K_{t+1} = K_t^{\alpha}\left(1-\tau_t\right)\left(1-\alpha\right)\left(1-\beta\right) - D_t \tag{74.11}$$

74.10.1 Steady states

From (74.11) and the government budget constraint (74.2), we compute **time-invariant** or **steady state values** $\hat{K}, \hat{D}, \hat{T}$:

$$\begin{aligned} \hat{K} &= \hat{K}\left(1-\hat{\tau}\right)\left(1-\alpha\right)\left(1-\beta\right) - \hat{D} \\ \hat{D} &= (1+\hat{r})\hat{D} + \hat{G} - \hat{T} \\ \hat{T} &= \hat{\tau}\hat{Y} + \hat{\tau}\hat{r}\hat{D}. \end{aligned} \tag{74.12}$$

These imply

$$\begin{aligned} \hat{K} &= \left[\left(1-\hat{\tau}\right)\left(1-\alpha\right)\left(1-\beta\right)\right]^{\frac{1}{1-\alpha}} \\ \hat{\tau} &= \frac{\hat{G}+\hat{r}\hat{D}}{\hat{Y}+\hat{r}\hat{D}} \end{aligned}$$

Let's take an example in which

1. there is no initial government debt, $D_t = 0$,
2. government consumption G_t equals 15% of output Y_t

Our formulas for steady-state values tell us that

$$\begin{aligned} \hat{D} &= 0 \\ \hat{G} &= 0.15\hat{Y} \\ \hat{\tau} &= 0.15 \end{aligned}$$

74.10.2 Implementation

```
import numpy as np
import matplotlib.pyplot as plt
from numba import jit
from quantecon.optimize import brent_max
```

For parameters $\alpha = 0.3$ and $\beta = 0.5$, let's compute $\hat{K}$:

```
# parameters
α = 0.3
β = 0.5

# steady states of τ and D
τ_hat = 0.15
D_hat = 0.

# solve for steady state of K
K_hat = ((1 - τ_hat) * (1 - α) * (1 - β)) ** (1 / (1 - α))
K_hat
```

```
0.17694509514972878
```

Knowing $\hat{K}$, we can calculate other equilibrium objects.

Let's first define some Python helper functions.

```
@jit
def K_to_Y(K, α):

    return K ** α

@jit
def K_to_r(K, α):

    return α * K ** (α - 1)

@jit
def K_to_W(K, α):

    return (1 - α) * K ** α

@jit
def K_to_C(K, D, τ, r, α, β):

    # optimal consumption for the old when δ=0
    A = K + D
    Co = A * (1 + r * (1 - τ))

    # optimal consumption for the young when δ=0
    W = K_to_W(K, α)
    Cy = β * W * (1 - τ)

    return Cy, Co
```

We can use these helper functions to obtain steady state values $\hat{Y}$, $\hat{r}$, and $\hat{W}$ associated with steady state values $\hat{K}$ and $\hat{r}$.

```
Y_hat, r_hat, W_hat = K_to_Y(K_hat, α), K_to_r(K_hat, α), K_to_W(K_hat, α)
Y_hat, r_hat, W_hat
```

```
(0.5947734290747186, 1.0084033613445376, 0.41634140035230305)
```

Since steady state government debt $\hat{D}$ is 0, all taxes are used to pay for government expenditures

```
G_hat = τ_hat * Y_hat
G_hat
```

```
0.0892160143612078
```

We use the optimal consumption plans to find steady state consumptions for young and old

```
Cy_hat, Co_hat = K_to_C(K_hat, D_hat, τ_hat, r_hat, α, β)
Cy_hat, Co_hat
```

```
(0.17694509514972878, 0.32861231956378195)
```

Let's store the steady state quantities and prices using an array called `init_ss`

```
init_ss = np.array([K_hat, Y_hat, Cy_hat, Co_hat,     # quantities
                    W_hat, r_hat,                     # prices
                    τ_hat, D_hat, G_hat               # policies
                    ])
```

74.10.3 Transitions

We have computed a steady state in which the government policy sequences are each constant over time.

We'll use this steady state as an initial condition at time $t = 0$ for another economy in which government policy sequences are with time-varying sequences.

To make sense of our calculation, we'll treat $t = 0$ as time when a huge unanticipated shock occurs in the form of

- a time-varying government policy sequences that disrupts an original steady state
- new government policy sequences are eventually time-invariant in the sense that after some date $T > 0$, each sequence is constant over time.
- sudden revelation of a new government policy in the form of sequences starting at time $t = 0$

We assume that everyone, including old people at time $t = 0$, knows the new government policy sequence and chooses accordingly.

As the capital stock and other aggregates adjust to the fiscal policy change over time, the economy will approach a new steady state.

We can find a transition path from an old steady state to a new steady state by employing a fixed-point algorithm in a space of sequences.

But in our special case with its closed form solution, we have available a simpler and faster approach.

Here we define a Python class `ClosedFormTrans` that computes length T transition path in response to a particular fiscal policy change.

We choose T large enough so that we have gotten very close to a new steady state after T periods.

The class takes three keyword arguments, `τ_pol`, `D_pol`, and `G_pol`.

These are sequences of tax rate, government debt level, and government purchases, respectively.

In each policy experiment below, we will pass two out of three as inputs required to depict a fiscal policy.

We'll then compute the single remaining undetermined policy variable from the government budget constraint.

When we simulate transition paths, it is useful to distinguish **state variables** at time t such as K_t, Y_t, D_t, W_t, r_t from **control variables** that include $C_{yt}, C_{ot}, \tau_t, G_t$.

```python
class ClosedFormTrans:
    """
    This class simulates length T transitional path of a economy
    in response to a fiscal policy change given its initial steady
    state. The simulation is based on the closed form solution when
    the lump sum taxations are absent.

    """

    def __init__(self, α, β):

        self.α, self.β = α, β

    def simulate(self,
                T,           # length of transitional path to simulate
                init_ss,     # initial steady state
                τ_pol=None,  # sequence of tax rates
                D_pol=None,  # sequence of government debt levels
                G_pol=None): # sequence of government purchases

        α, β = self.α, self.β

        # unpack the steady state variables
        K_hat, Y_hat, Cy_hat, Co_hat = init_ss[:4]
        W_hat, r_hat = init_ss[4:6]
        τ_hat, D_hat, G_hat = init_ss[6:9]

        # initialize array containers
        # K, Y, Cy, Co
        quant_seq = np.empty((T+1, 4))

        # W, r
        price_seq = np.empty((T+1, 2))

        # τ, D, G
        policy_seq = np.empty((T+2, 3))

        # t=0, starting from steady state
        K0, Y0 = K_hat, Y_hat
        W0, r0 = W_hat, r_hat
        D0 = D_hat

        # fiscal policy
        if τ_pol is None:
            D1 = D_pol[1]
            G0 = G_pol[0]
            τ0 = (G0 + (1 + r0) * D0 - D1) / (Y0 + r0 * D0)
        elif D_pol is None:
            τ0 = τ_pol[0]
            G0 = G_pol[0]
```

(continues on next page)

(continued from previous page)

```
            D1 = (1 + r0) * D0 + G0 - τ0 * (Y0 + r0 * D0)
        elif G_pol is None:
            D1 = D_pol[1]
            τ0 = τ_pol[0]
            G0 = τ0 * (Y0 + r0 * D0) + D1 - (1 + r0) * D0

        # optimal consumption plans
        Cy0, Co0 = K_to_C(K0, D0, τ0, r0, α, β)

        # t=0 economy
        quant_seq[0, :] = K0, Y0, Cy0, Co0
        price_seq[0, :] = W0, r0
        policy_seq[0, :] = τ0, D0, G0
        policy_seq[1, 1] = D1

        # starting from t=1 to T
        for t in range(1, T+1):

            # transition of K
            K_old, τ_old = quant_seq[t-1, 0], policy_seq[t-1, 0]
            D = policy_seq[t, 1]
            K = K_old ** α * (1 - τ_old) * (1 - α) * (1 - β) - D

            # output, capital return, wage
            Y, r, W = K_to_Y(K, α), K_to_r(K, α), K_to_W(K, α)

            # to satisfy the government budget constraint
            if τ_pol is None:
                D = D_pol[t]
                D_next = D_pol[t+1]
                G = G_pol[t]
                τ = (G + (1 + r) * D - D_next) / (Y + r * D)
            elif D_pol is None:
                τ = τ_pol[t]
                G = G_pol[t]
                D = policy_seq[t, 1]
                D_next = (1 + r) * D + G - τ * (Y + r * D)
            elif G_pol is None:
                D = D_pol[t]
                D_next = D_pol[t+1]
                τ = τ_pol[t]
                G = τ * (Y + r * D) + D_next - (1 + r) * D

            # optimal consumption plans
            Cy, Co = K_to_C(K, D, τ, r, α, β)

            # store time t economy aggregates
            quant_seq[t, :] = K, Y, Cy, Co
            price_seq[t, :] = W, r
            policy_seq[t, 0] = τ
            policy_seq[t+1, 1] = D_next
            policy_seq[t, 2] = G

        self.quant_seq = quant_seq
        self.price_seq = price_seq
        self.policy_seq = policy_seq
```

(continues on next page)

(continued from previous page)

```
        return quant_seq, price_seq, policy_seq

    def plot(self):

        quant_seq = self.quant_seq
        price_seq = self.price_seq
        policy_seq = self.policy_seq

        fig, axs = plt.subplots(3, 3, figsize=(14, 10))

        # quantities
        for i, name in enumerate(['K', 'Y', 'Cy', 'Co']):
            ax = axs[i//3, i%3]
            ax.plot(range(T+1), quant_seq[:T+1, i], label=name)
            ax.hlines(init_ss[i], 0, T+1, color='r', linestyle='--')
            ax.legend()
            ax.set_xlabel('t')

        # prices
        for i, name in enumerate(['W', 'r']):
            ax = axs[(i+4)//3, (i+4)%3]
            ax.plot(range(T+1), price_seq[:T+1, i], label=name)
            ax.hlines(init_ss[i+4], 0, T+1, color='r', linestyle='--')
            ax.legend()
            ax.set_xlabel('t')

        # policies
        for i, name in enumerate(['τ', 'D', 'G']):
            ax = axs[(i+6)//3, (i+6)%3]
            ax.plot(range(T+1), policy_seq[:T+1, i], label=name)
            ax.hlines(init_ss[i+6], 0, T+1, color='r', linestyle='--')
            ax.legend()
            ax.set_xlabel('t')
```

We can create an instance `closed` for model parameters $\{\alpha, \beta\}$ and use it for various fiscal policy experiments.

```
closed = ClosedFormTrans(α, β)
```

74.10.4 Experiment 1: Tax cut

To illustrate the power of `ClosedFormTrans`, let's first experiment with the following fiscal policy change:

1. at $t = 0$, the government unexpectedly announces a one-period tax cut, $\tau_0 = (1 - \frac{1}{3})\hat{\tau}$, by issuing government debt $\bar{D}$
2. from $t = 1$, the government will keep $D_t = \bar{D}$ and adjust τ_t to collect taxation to pay for the government consumption and interest payments on the debt
3. government consumption G_t will be fixed at $0.15\hat{Y}$

The following equations completely characterize the equilibrium transition path originating from the initial steady state

$$
\begin{aligned}
K_{t+1} &= K_t^{\alpha}\left(1-\tau_t\right)\left(1-\alpha\right)\left(1-\beta\right) - \bar{D} \\
\tau_0 &= (1-\frac{1}{3})\hat{\tau} \\
\bar{D} &= \hat{G} - \tau_0 \hat{Y} \\
\tau_t &= \frac{\hat{G} + r_t \bar{D}}{\hat{Y} + r_t \bar{D}}
\end{aligned}
$$

We can simulate the transition for 20 periods, after which the economy will be close to a new steady state.

The first step is to prepare sequences of policy variables that describe fiscal policy.

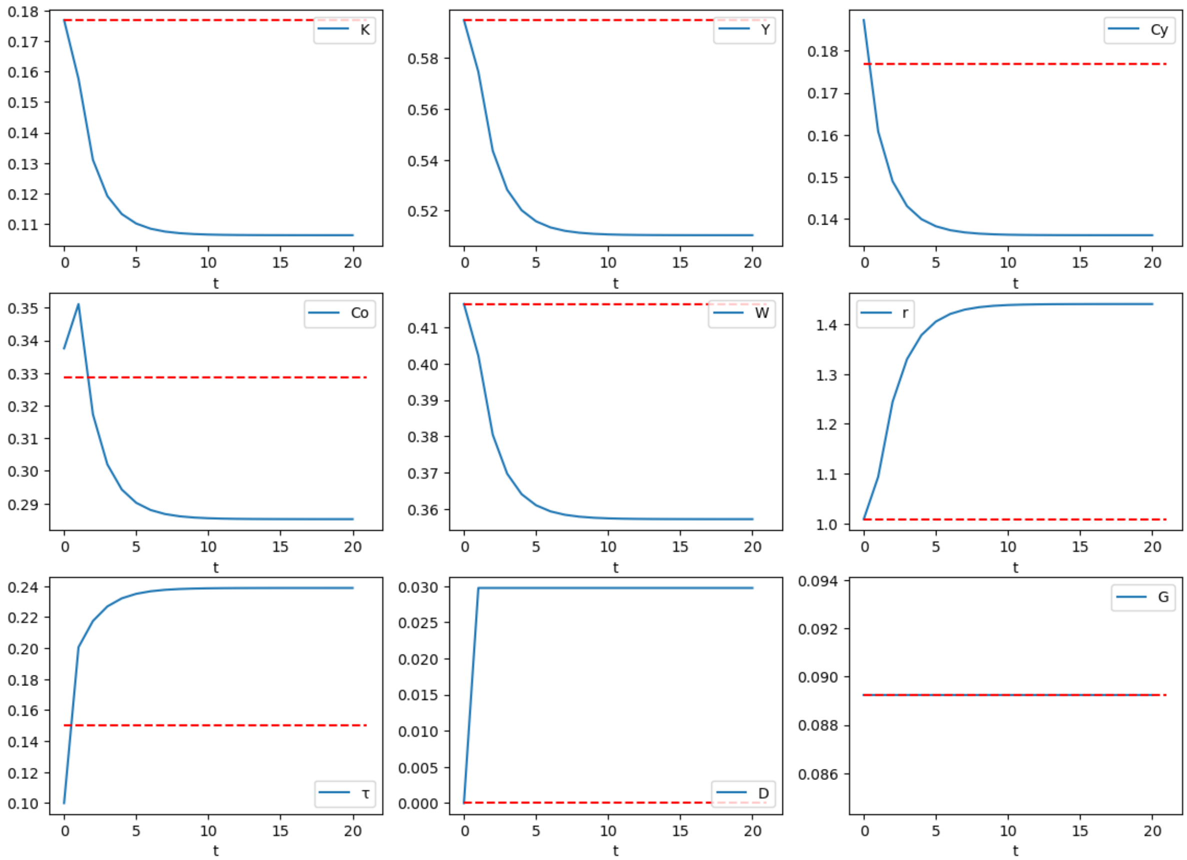
We must define sequences of government expenditure $\{G_t\}_{t=0}^{T}$ and debt level $\{D_t\}_{t=0}^{T+1}$ in advance, then pass them to the solver.

```
T = 20

# tax cut
τ0 = τ_hat * (1 - 1/3)

# sequence of government purchase
G_seq = τ_hat * Y_hat * np.ones(T+1)

# sequence of government debt
D_bar = G_hat - τ0 * Y_hat
D_seq = np.ones(T+2) * D_bar
D_seq[0] = D_hat
```

Let's use the `simulate` method of `closed` to compute dynamic transitions.

Note that we leave `τ_pol` as `None`, since the tax rates need to be determined to satisfy the government budget constraint.

```
quant_seq1, price_seq1, policy_seq1 = closed.simulate(T, init_ss,
                                                      D_pol=D_seq,
                                                      G_pol=G_seq)
closed.plot()
```



We can also experiment with a lower tax cut rate, such as 0.2.

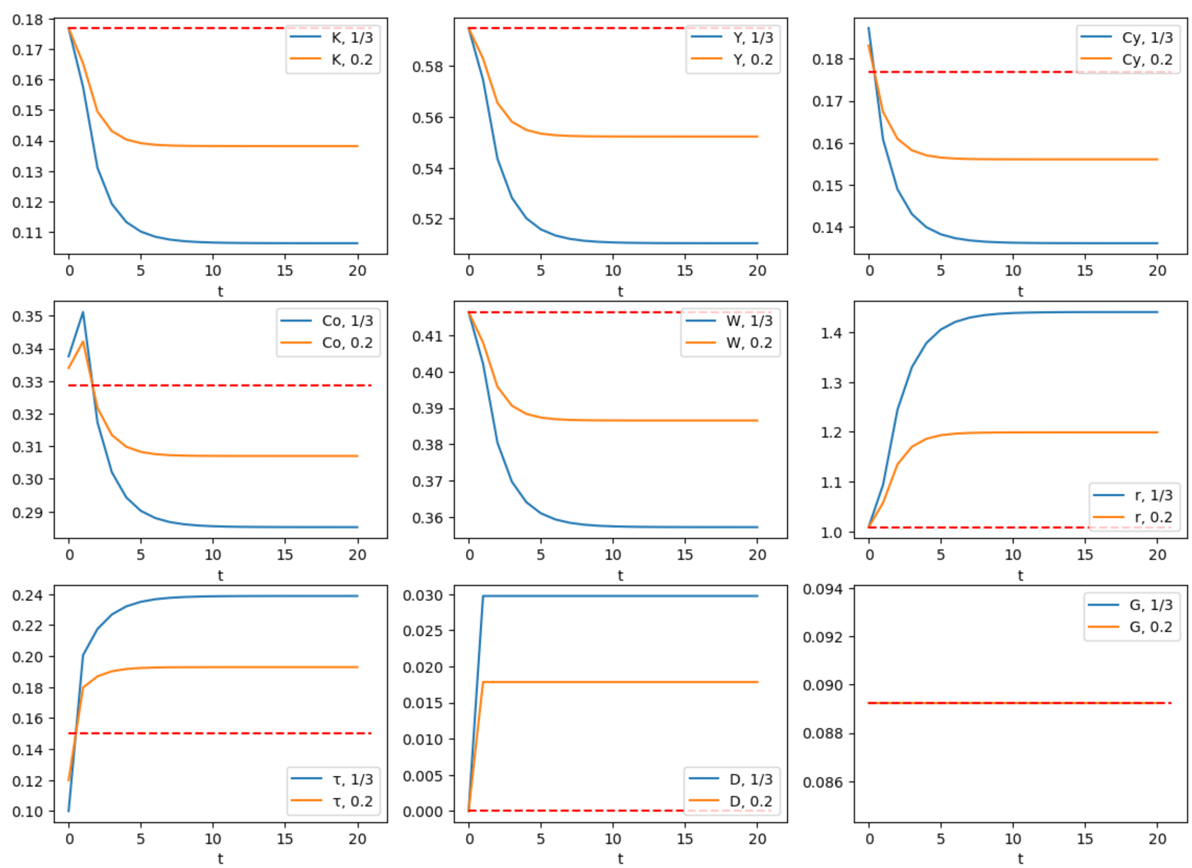
```
# lower tax cut rate
τ0 = 0.15 * (1 - 0.2)

# the corresponding debt sequence
D_bar = G_hat - τ0 * Y_hat
D_seq = np.ones(T+2) * D_bar
D_seq[0] = D_hat

quant_seq2, price_seq2, policy_seq2 = closed.simulate(T, init_ss,
                                                       D_pol=D_seq,
                                                       G_pol=G_seq)
```

```
fig, axs = plt.subplots(3, 3, figsize=(14, 10))

# quantities
for i, name in enumerate(['K', 'Y', 'Cy', 'Co']):
    ax = axs[i//3, i%3]
    ax.plot(range(T+1), quant_seq1[:T+1, i], label=f'{name}, 1/3')
    ax.plot(range(T+1), quant_seq2[:T+1, i], label=f'{name}, 0.2')
    ax.hlines(init_ss[i], 0, T+1, color='r', linestyle='--')
    ax.legend()
    ax.set_xlabel('t')

# prices
for i, name in enumerate(['W', 'r']):
```

(continues on next page)

(continued from previous page)

```
    ax = axs[(i+4)//3, (i+4)%3]
    ax.plot(range(T+1), price_seq1[:T+1, i], label=f'{name}, 1/3')
    ax.plot(range(T+1), price_seq2[:T+1, i], label=f'{name}, 0.2')
    ax.hlines(init_ss[i+4], 0, T+1, color='r', linestyle='--')
    ax.legend()
    ax.set_xlabel('t')

# policies
for i, name in enumerate(['τ', 'D', 'G']):
    ax = axs[(i+6)//3, (i+6)%3]
    ax.plot(range(T+1), policy_seq1[:T+1, i], label=f'{name}, 1/3')
    ax.plot(range(T+1), policy_seq2[:T+1, i], label=f'{name}, 0.2')
    ax.hlines(init_ss[i+6], 0, T+1, color='r', linestyle='--')
    ax.legend()
    ax.set_xlabel('t')
```



The economy with lower tax cut rate at $t = 0$ has the same transitional pattern, but is less distorted, and it converges to a new steady state with higher physical capital stock.

74.10.5 Experiment 2: Government asset accumulation

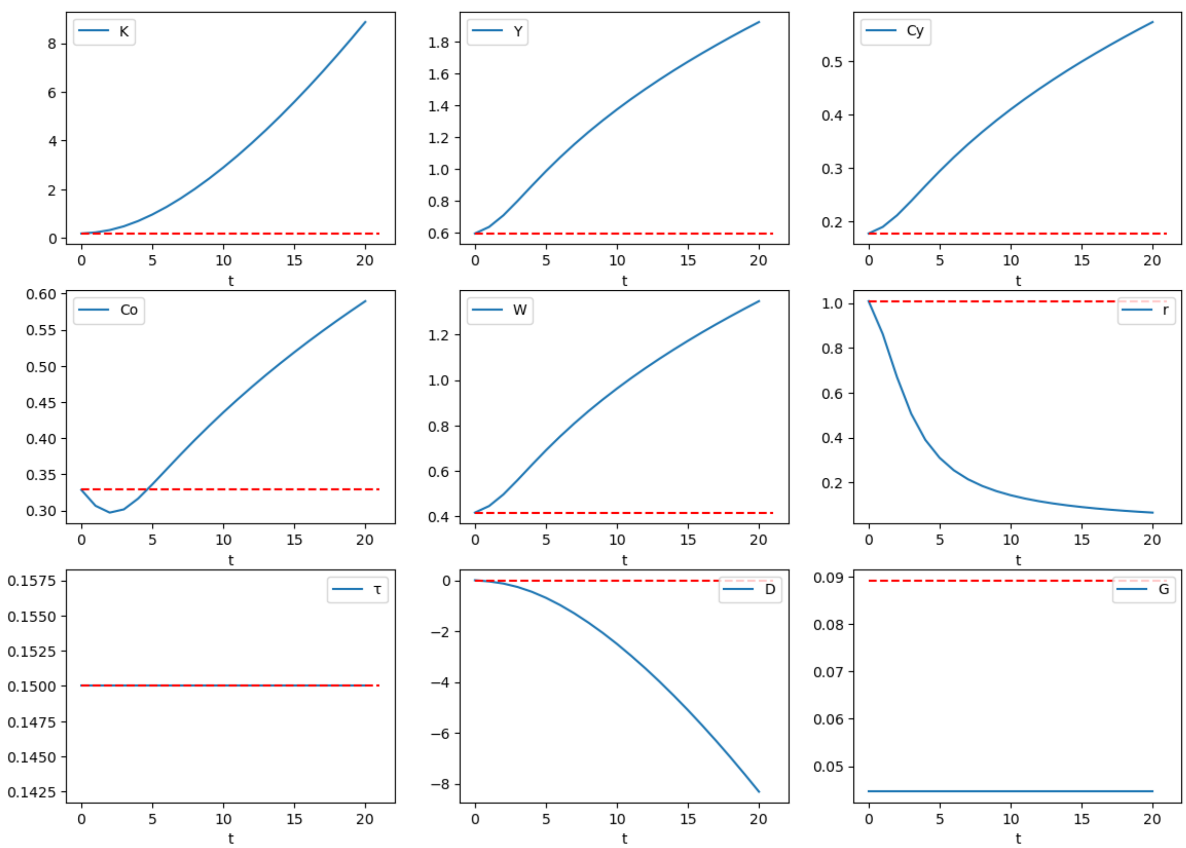
Assume that the economy is initially in the same steady state.

Now the government promises to cut its spending on services and goods by half $\forall t \geq 0$.

The government targets the same tax rate $\tau_t = \hat{\tau}$ and to accumulate assets $-D_t$ over time.

To conduct this experiment, we pass `τ_seq` and `G_seq` as inputs and let `D_pol` be determined along the path by satisfying the government budget constraint.

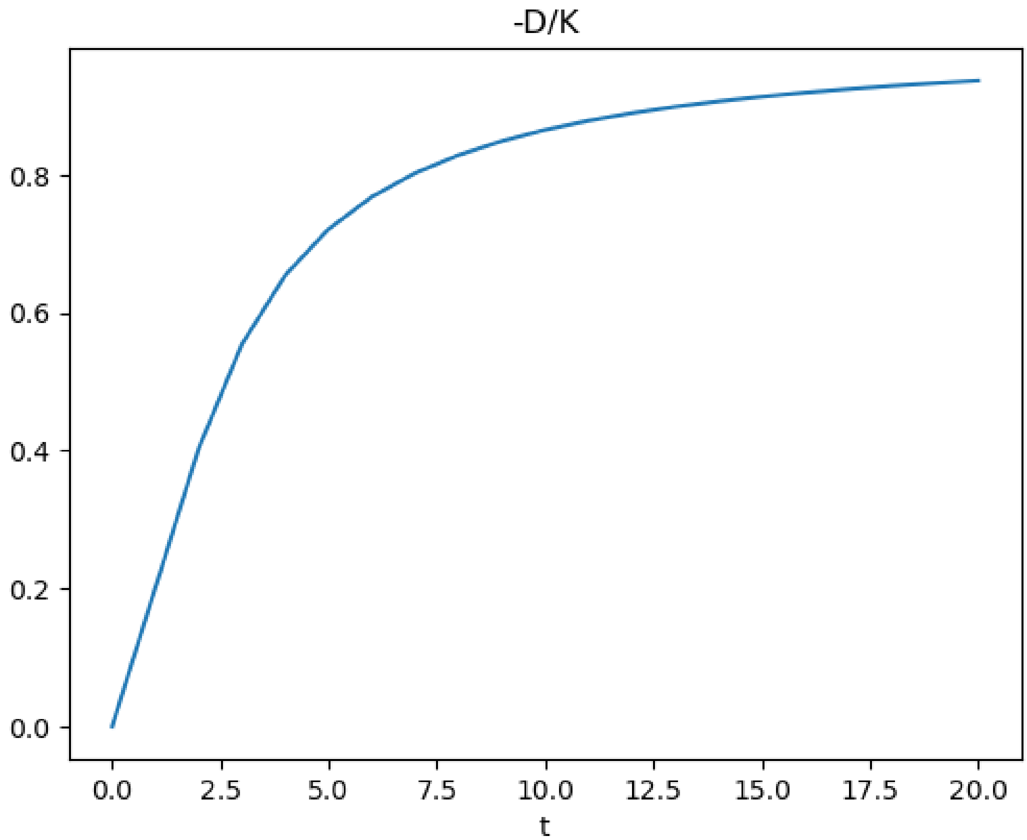
```
# government expenditure cut by a half
G_seq = τ_hat * 0.5 * Y_hat * np.ones(T+1)

# targeted tax rate
τ_seq = τ_hat * np.ones(T+1)

closed.simulate(T, init_ss, τ_pol=τ_seq, G_pol=G_seq);
closed.plot()
```



As the government accumulates the asset and uses it in production, the rental rate on capital falls and private investment falls.

As a result, the ratio $-\frac{D_t}{K_t}$ of the government asset to physical capital used in production will increase over time

```
plt.plot(range(T+1), -closed.policy_seq[:-1, 1] / closed.quant_seq[:, 0])
plt.xlabel('t')
plt.title('-D/K');
```



We want to know how this policy experiment affects individuals.

In the long run, future cohorts will enjoy higher consumption throughout their lives because they will earn higher labor income when they work.

However, in the short run, old people suffer because increases in their labor income are not big enough to offset their losses of capital income.

Such distinct long run and short run effects motivate us to study transition paths.

Note

Although the consumptions in the new steady state are strictly higher, it is at a cost of fewer public services and goods.

74.10.6 Experiment 3: Temporary expenditure cut

Let's now investigate a scenario in which the government also cuts its spending by half and accumulates the asset.

But now let the government cut its expenditures only at $t = 0$.

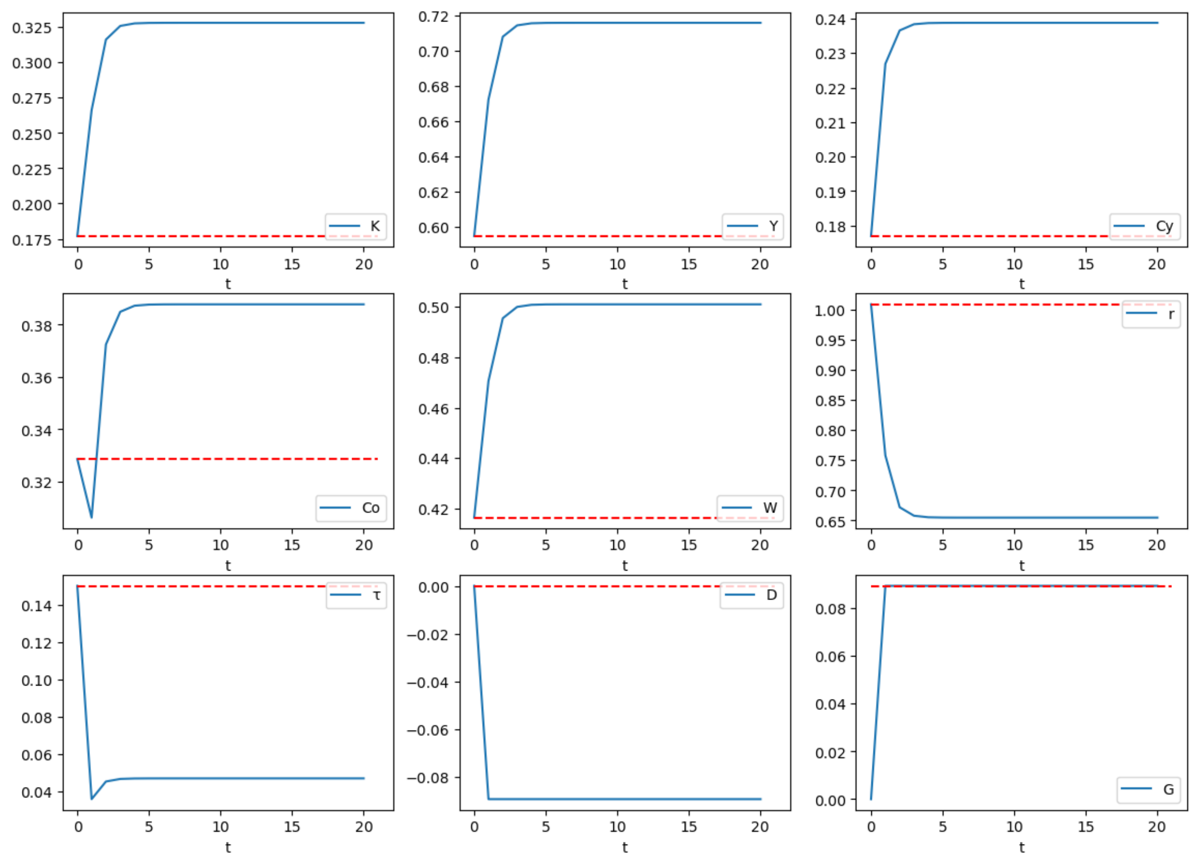
From $t \geq 1$, the government expeditures return to $\hat{G}$ and τ_t adjusts to maintain the asset level $-D_t = -D_1$.

```
# sequence of government purchase
G_seq = τ_hat * Y_hat * np.ones(T+1)
G_seq[0] = 0
```

(continues on next page)

(continued from previous page)

```
# sequence of government debt
D_bar = G_seq[0] - τ_hat * Y_hat
D_seq = D_bar * np.ones(T+2)
D_seq[0] = D_hat

closed.simulate(T, init_ss, D_pol=D_seq, G_pol=G_seq);
closed.plot()
```



The economy quickly converges to a new steady state with higher physical capital stock, lower interest rate, higher wage rate, and higher consumptions for both the young and the old.

Even though government expenditure G_t returns to its high initial level from $t \geq 1$, the government can balance the budget at a lower tax rate because it gathers additional revenue $-r_t D_t$ from the asset accumulated during the temporary cut in the spendings.

As in *Experiment 2: Government asset accumulation*, old perople early in the transition periods suffer from this policy shock.

74.11 A computational strategy

With the preceding caluations, we studied dynamic transitions instigated by alternative fiscal policies.

In all these experiments, we maintained the assumption that lump sum taxes were absent so that $\delta_{yt} = 0, \delta_{ot} = 0$.

In this section, we investigate the transition dynamics when the lump sum taxes are present.

The government will use lump sum taxes and transfers to redistribute resources across successive generations.

Including lump sum taxes disrupts closed form solution because of how they make optimal consumption and saving plans depend on future prices and tax rates.

Therefore, we compute equilibrium transitional paths by finding a fixed point of a mapping from sequences to sequences.

- that fixed point pins down an equilibrium

To set the stage for the entry of the mapping whose fixed point we seek, we return to concepts introduced in section *Equilbrium*.

Definition: Given parameters $\{\alpha, \beta\}$, a competitive equilibrium consists of

- sequences of optimal consumptions $\{C_{yt}, C_{ot}\}$
- sequences of prices $\{W_t, r_t\}$
- sequences of capital stock and output $\{K_t, Y_t\}$
- sequences of tax rates, government assets (debt), government purchases $\{\tau_t, D_t, G_t\, \delta_{yt}, \delta_{ot}\}$

with the properties that

- given the price system and government fiscal policy, consumption plans are optimal
- the government budget constraints are satisfied for all t

An equilibrium transition path can be computed by "guessing and verifying" some endogenous sequences.

In our *Experiment 1: Tax cut* example, sequences $\{D_t\}_{t=0}^{T}$ and $\{G_t\}_{t=0}^{T}$ are exogenous.

In addition, we assume that the lump sum taxes $\{\delta_{yt}, \delta_{ot}\}_{t=0}^{T}$ are given and known to everybody inside the model.

We can solve for sequences of other equilibrium sequences following the steps below

1. guess prices $\{W_t, r_t\}_{t=0}^{T}$ and tax rates $\{\tau_t\}_{t=0}^{T}$
2. solve for optimal consumption and saving plans $\{C_{yt}, C_{ot}\}_{t=0}^{T}$, treating the guesses of future prices and taxes as true
3. solve for transition of the capital stock $\{K_t\}_{t=0}^{T}$
4. update the guesses for prices and tax rates with the values implied by the equilibrium conditions
5. iterate until convergence

Let's implement this "guess and verify" approach

We start by defining the Cobb-Douglas utility function

```
@jit
def U(Cy, Co, β):

    return (Cy ** β) * (Co ** (1-β))
```

We use `Cy_val` to compute the lifetime value of an arbitrary consumption plan, C_y, given the intertemporal budget constraint.

Note that it requires knowing future prices r_{t+1} and tax rate τ_{t+1}.

```
@jit
def Cy_val(Cy, W, r_next, τ, τ_next, δy, δo_next, β):

    # Co given by the budget constraint
    Co = (W * (1 - τ) - δy - Cy) * (1 + r_next * (1 - τ_next)) - δo_next

    return U(Cy, Co, β)
```

An optimal consumption plan C_y^* can be found by maximizing `Cy_val`.

Here is an example that computes optimal consumption $C_y^* = \hat{C}_y$ in the steady state with $\delta_{yt} = \delta_{ot} = 0$, like one that we studied earlier

```
W, r_next, τ, τ_next = W_hat, r_hat, τ_hat, τ_hat
δy, δo_next = 0, 0

Cy_opt, U_opt, _ = brent_max(Cy_val,            # maximand
                             1e-6,              # lower bound
                             W*(1-τ)-δy-1e-6,   # upper bound
                             args=(W, r_next, τ, τ_next, δy, δo_next, β))

Cy_opt, U_opt
```

```
(0.17694509514972878, 0.241135518231111)
```

Let's define a Python class `AK2` that computes the transition paths with the fixed-point algorithm.

It can handle nonzero lump sum taxes

```
class AK2():
    """
    This class simulates length T transitional path of a economy
    in response to a fiscal policy change given its initial steady
    state. The transitional path is found by employing a fixed point
    algorithm to satisfy the equilibrium conditions.

    """

    def __init__(self, α, β):

        self.α, self.β = α, β

    def simulate(self,
                T,            # length of transitional path to simulate
                init_ss,      # initial steady state
                δy_seq,       # sequence of lump sum tax for the young
                δo_seq,       # sequence of lump sum tax for the old
                τ_pol=None,   # sequence of tax rates
                D_pol=None,   # sequence of government debt levels
                G_pol=None,   # sequence of government purchases
                verbose=False,
                max_iter=500,
                tol=1e-5):
```

(continues on next page)

(continued from previous page)

```
        α, β = self.α, self.β

        # unpack the steady state variables
        K_hat, Y_hat, Cy_hat, Co_hat = init_ss[:4]
        W_hat, r_hat = init_ss[4:6]
        τ_hat, D_hat, G_hat = init_ss[6:9]

        # K, Y, Cy, Co
        quant_seq = np.empty((T+2, 4))

        # W, r
        price_seq = np.empty((T+2, 2))

        # τ, D, G
        policy_seq = np.empty((T+2, 3))
        policy_seq[:, 1] = D_pol
        policy_seq[:, 2] = G_pol

        # initial guesses of prices
        price_seq[:, 0] = np.ones(T+2) * W_hat
        price_seq[:, 1] = np.ones(T+2) * r_hat

        # initial guesses of policies
        policy_seq[:, 0] = np.ones(T+2) * τ_hat

        # t=0, starting from steady state
        quant_seq[0, :2] = K_hat, Y_hat

        if verbose:
            # prepare to plot iterations until convergence
            fig, axs = plt.subplots(1, 3, figsize=(14, 4))

        # containers for checking convergence
        price_seq_old = np.empty_like(price_seq)
        policy_seq_old = np.empty_like(policy_seq)

        # start iteration
        i_iter = 0
        while True:

            if verbose:
                # plot current prices at ith iteration
                for i, name in enumerate(['W', 'r']):
                    axs[i].plot(range(T+1), price_seq[:T+1, i])
                    axs[i].set_title(name)
                    axs[i].set_xlabel('t')
                axs[2].plot(range(T+1), policy_seq[:T+1, 0],
                            label=f'{i_iter}th iteration')
                axs[2].legend(bbox_to_anchor=(1.05, 1), loc='upper left')
                axs[2].set_title('τ')
                axs[2].set_xlabel('t')

            # store old prices from last iteration
            price_seq_old[:] = price_seq
            policy_seq_old[:] = policy_seq
```

(continues on next page)

(continued from previous page)

```
            # start updating quantities and prices
            for t in range(T+1):
                K, Y = quant_seq[t, :2]
                W, r = price_seq[t, :]
                r_next = price_seq[t+1, 1]
                τ, D, G = policy_seq[t, :]
                τ_next, D_next, G_next = policy_seq[t+1, :]
                δy, δo = δy_seq[t], δo_seq[t]
                δy_next, δo_next = δy_seq[t+1], δo_seq[t+1]

                # consumption for the old
                Co = (1 + r * (1 - τ)) * (K + D) - δo

                # optimal consumption for the young
                out = brent_max(Cy_val, 1e-6, W*(1-τ)-δy-1e-6,
                                args=(W, r_next, τ, τ_next,
                                      δy, δo_next, β))
                Cy = out[0]

                quant_seq[t, 2:] = Cy, Co
                τ_num = ((1 + r) * D + G - D_next - δy - δo)
                τ_denom = (Y + r * D)
                policy_seq[t, 0] = τ_num / τ_denom

                # saving of the young
                A_next = W * (1 - τ) - δy - Cy

                # transition of K
                K_next = A_next - D_next
                Y_next = K_to_Y(K_next, α)
                W_next, r_next = K_to_W(K_next, α), K_to_r(K_next, α)

                quant_seq[t+1, :2] = K_next, Y_next
                price_seq[t+1, :] = W_next, r_next

            i_iter += 1

            if (np.max(np.abs(price_seq_old - price_seq)) < tol) & \
               (np.max(np.abs(policy_seq_old - policy_seq)) < tol):
                if verbose:
                    print(f"Converge using {i_iter} iterations")
                break

            if i_iter > max_iter:
                if verbose:
                    print(f"Fail to converge using {i_iter} iterations")
                break

        self.quant_seq = quant_seq
        self.price_seq = price_seq
        self.policy_seq = policy_seq

        return quant_seq, price_seq, policy_seq

    def plot(self):

        quant_seq = self.quant_seq
```

(continues on next page)

(continued from previous page)

```
        price_seq = self.price_seq
        policy_seq = self.policy_seq

        fig, axs = plt.subplots(3, 3, figsize=(14, 10))

        # quantities
        for i, name in enumerate(['K', 'Y', 'Cy', 'Co']):
            ax = axs[i//3, i%3]
            ax.plot(range(T+1), quant_seq[:T+1, i], label=name)
            ax.hlines(init_ss[i], 0, T+1, color='r', linestyle='--')
            ax.legend()
            ax.set_xlabel('t')

        # prices
        for i, name in enumerate(['W', 'r']):
            ax = axs[(i+4)//3, (i+4)%3]
            ax.plot(range(T+1), price_seq[:T+1, i], label=name)
            ax.hlines(init_ss[i+4], 0, T+1, color='r', linestyle='--')
            ax.legend()
            ax.set_xlabel('t')

        # policies
        for i, name in enumerate(['τ', 'D', 'G']):
            ax = axs[(i+6)//3, (i+6)%3]
            ax.plot(range(T+1), policy_seq[:T+1, i], label=name)
            ax.hlines(init_ss[i+6], 0, T+1, color='r', linestyle='--')
            ax.legend()
            ax.set_xlabel('t')
```

We can initialize an instance of class AK2 with model parameters $\{\alpha, \beta\}$ and then use it to conduct fiscal policy experiments.

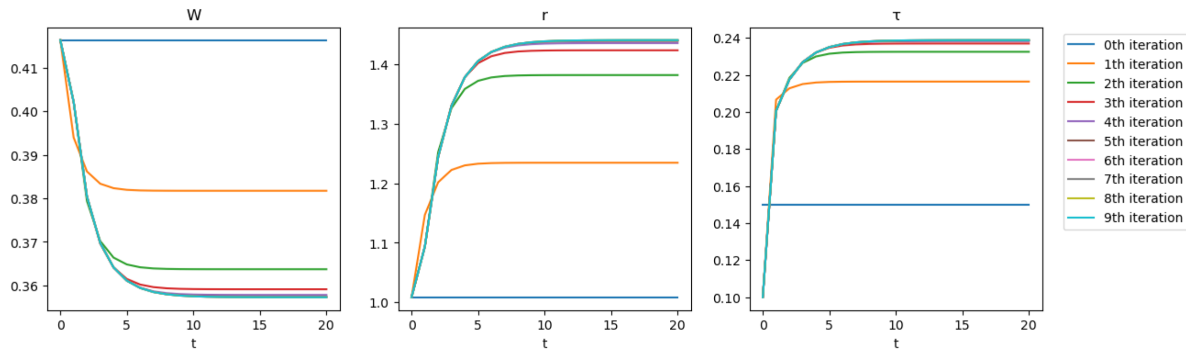
```
ak2 = AK2(α, β)
```

We first examine that the "guess and verify" method leads to the same numerical results as we obtain with the closed form solution when lump sum taxes are muted

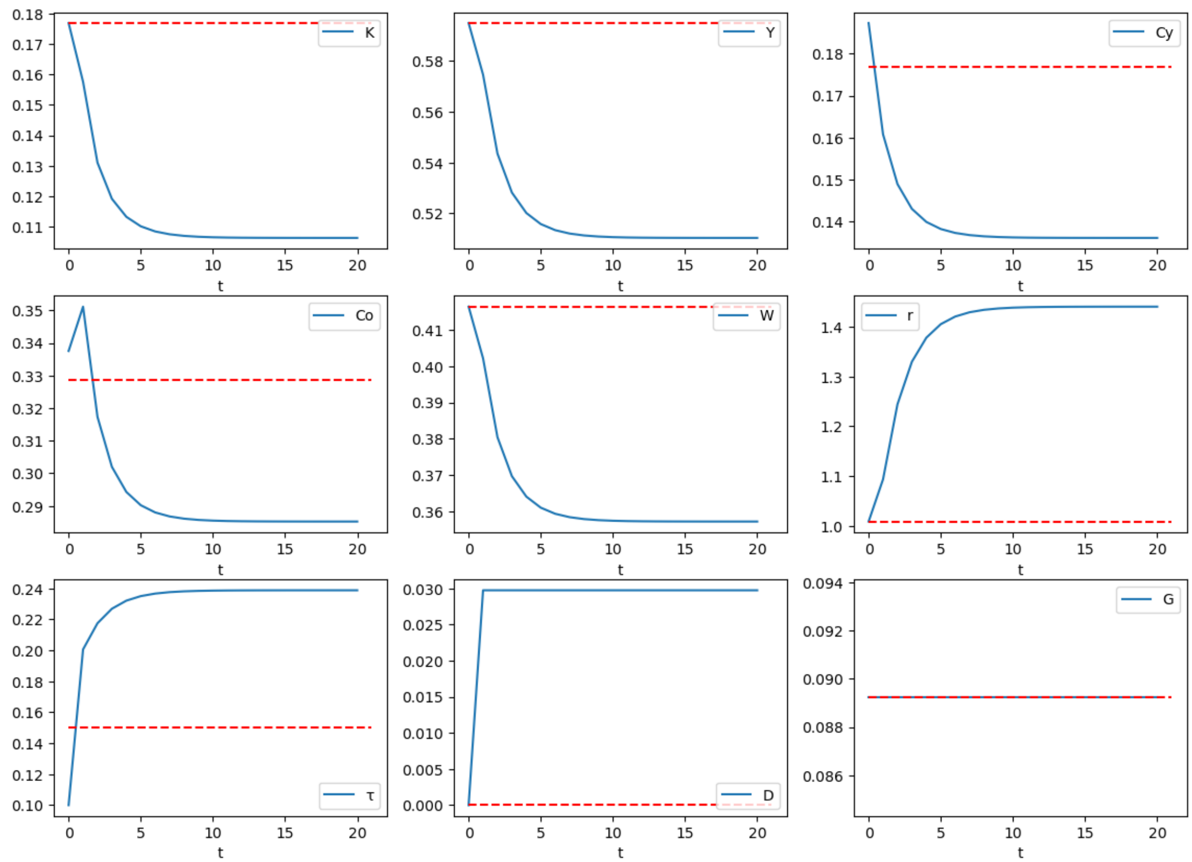
```
δy_seq = np.ones(T+2) * 0.
δo_seq = np.ones(T+2) * 0.

D_pol = np.zeros(T+2)
G_pol = np.ones(T+2) * G_hat

# tax cut
τ0 = τ_hat * (1 - 1/3)
D1 = D_hat * (1 + r_hat * (1 - τ0)) + G_hat - τ0 * Y_hat - δy_seq[0] - δo_seq[0]
D_pol[0] = D_hat
D_pol[1:] = D1
```

```
quant_seq3, price_seq3, policy_seq3 = ak2.simulate(T, init_ss,
                                                   δy_seq, δo_seq,
                                                   D_pol=D_pol, G_pol=G_pol,
                                                   verbose=True)
```

```
Converge using 10 iterations
```




```
ak2.plot()
```



Next, we activate lump sum taxes.

Let's alter our *Experiment 1: Tax cut* fiscal policy experiment by assuming that the government also increases lump sum taxes for both young and old people $\delta_{yt} = \delta_{ot} = 0.005, t \geq 0$.

```
δy_seq = np.ones(T+2) * 0.005
δo_seq = np.ones(T+2) * 0.005

D1 = D_hat * (1 + r_hat * (1 - τ0)) + G_hat - τ0 * Y_hat - δy_seq[0] - δo_seq[0]
D_pol[1:] = D1

quant_seq4, price_seq4, policy_seq4 = ak2.simulate(T, init_ss,
```

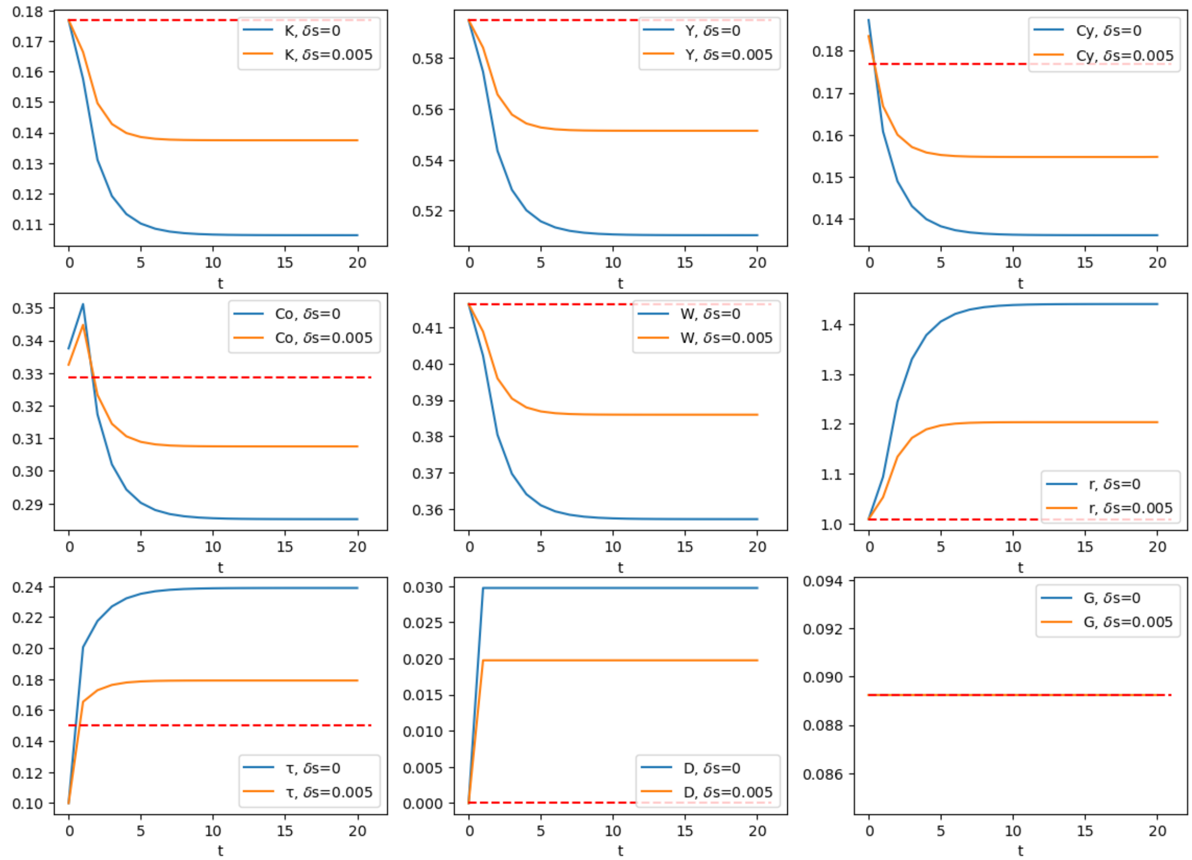
(continues on next page)

(continued from previous page)

```
                                                δy_seq, δo_seq,
                                                D_pol=D_pol, G_pol=G_pol)
```

Note how “crowding out” has been mitigated.

```
fig, axs = plt.subplots(3, 3, figsize=(14, 10))

# quantities
for i, name in enumerate(['K', 'Y', 'Cy', 'Co']):
    ax = axs[i//3, i%3]
    ax.plot(range(T+1), quant_seq3[:T+1, i], label=rf'{name}, $\delta$s=0')
    ax.plot(range(T+1), quant_seq4[:T+1, i], label=rf'{name}, $\delta$s=0.005')
    ax.hlines(init_ss[i], 0, T+1, color='r', linestyle='--')
    ax.legend()
    ax.set_xlabel('t')

# prices
for i, name in enumerate(['W', 'r']):
    ax = axs[(i+4)//3, (i+4)%3]
    ax.plot(range(T+1), price_seq3[:T+1, i], label=rf'{name}, $\delta$s=0')
    ax.plot(range(T+1), price_seq4[:T+1, i], label=rf'{name}, $\delta$s=0.005')
    ax.hlines(init_ss[i+4], 0, T+1, color='r', linestyle='--')
    ax.legend()
    ax.set_xlabel('t')

# policies
for i, name in enumerate(['τ', 'D', 'G']):
    ax = axs[(i+6)//3, (i+6)%3]
    ax.plot(range(T+1), policy_seq3[:T+1, i], label=rf'{name}, $\delta$s=0')
    ax.plot(range(T+1), policy_seq4[:T+1, i], label=rf'{name}, $\delta$s=0.005')
    ax.hlines(init_ss[i+6], 0, T+1, color='r', linestyle='--')
    ax.legend()
    ax.set_xlabel('t')
```



Comparing to *Experiment 1: Tax cut*, the government raises lump-sum taxes to finance the increasing debt interest payment, which is less distortionary comparing to raising the capital income tax rate.

74.11.1 Experiment 4: Unfunded Social Security System

In this experiment, lump-sum taxes are of equal magnitudes for old and the young, but of opposite signs.

A negative lump-sum tax is a subsidy.

Thus, in this experiment we tax the young and subsidize the old.

We start the economy at the same initial steady state that we assumed in several earlier experiments.

The government sets the lump sum taxes $\delta_{y,t} = -\delta_{o,t} = 10\% \hat{C}_y$ starting from $t = 0$.

It keeps debt levels and expenditures at their steady state levels $\hat{D}$ and $\hat{G}$.

In effect, this experiment amounts to launching an unfunded social security system.

We can use our code to compute the transition ignited by launching this system.

Let's compare the results to the *Experiment 1: Tax cut*.

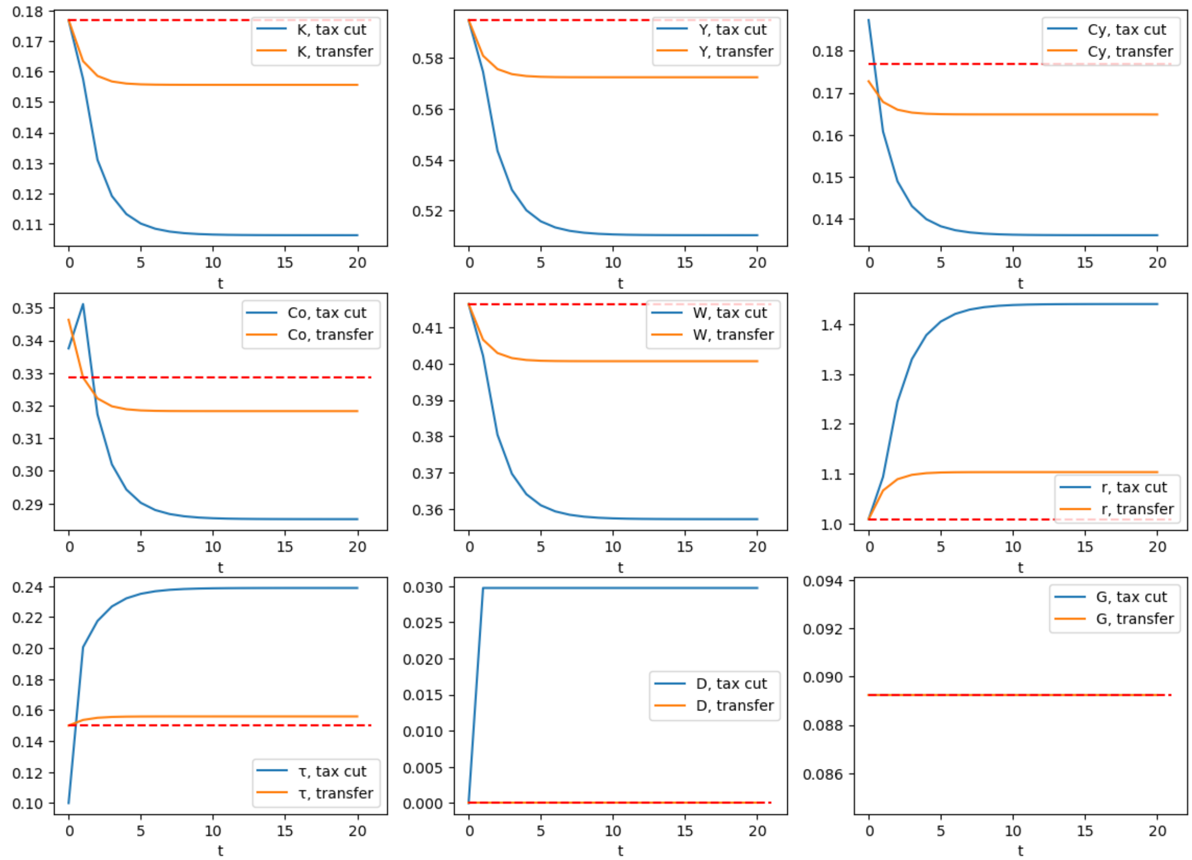
```
δy_seq = np.ones(T+2) * Cy_hat * 0.1
δo_seq = np.ones(T+2) * -Cy_hat * 0.1

D_pol[:] = D_hat
```

(continues on next page)

(continued from previous page)

```
quant_seq5, price_seq5, policy_seq5 = ak2.simulate(T, init_ss,
                                                     δy_seq, δo_seq,
                                                     D_pol=D_pol, G_pol=G_pol)
```

```
fig, axs = plt.subplots(3, 3, figsize=(14, 10))

# quantities
for i, name in enumerate(['K', 'Y', 'Cy', 'Co']):
    ax = axs[i//3, i%3]
    ax.plot(range(T+1), quant_seq3[:T+1, i], label=f'{name}, tax cut')
    ax.plot(range(T+1), quant_seq5[:T+1, i], label=f'{name}, transfer')
    ax.hlines(init_ss[i], 0, T+1, color='r', linestyle='--')
    ax.legend()
    ax.set_xlabel('t')

# prices
for i, name in enumerate(['W', 'r']):
    ax = axs[(i+4)//3, (i+4)%3]
    ax.plot(range(T+1), price_seq3[:T+1, i], label=f'{name}, tax cut')
    ax.plot(range(T+1), price_seq5[:T+1, i], label=f'{name}, transfer')
    ax.hlines(init_ss[i+4], 0, T+1, color='r', linestyle='--')
    ax.legend()
    ax.set_xlabel('t')

# policies
for i, name in enumerate(['τ', 'D', 'G']):
    ax = axs[(i+6)//3, (i+6)%3]
    ax.plot(range(T+1), policy_seq3[:T+1, i], label=f'{name}, tax cut')
    ax.plot(range(T+1), policy_seq5[:T+1, i], label=f'{name}, transfer')
    ax.hlines(init_ss[i+6], 0, T+1, color='r', linestyle='--')
    ax.legend()
    ax.set_xlabel('t')
```



An initial old person benefits especially when the social security system is launched because he receives a transfer but pays nothing for it.

But in the long run, consumption rates of both young and old people decrease because the the social security system decreases incentives to save.

That lowers the stock of physical capital and consequently lowers output.

The government must then raise tax rate in order to pay for its expenditures.

The higher rate on capital income further distorts incentives to save.

Part XII

Multiple Agent Models

CHAPTER

SEVENTYFIVE

A LAKE MODEL OF EMPLOYMENT AND UNEMPLOYMENT

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

Contents

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon jax
```

75.1 Overview

This lecture describes what has come to be called a *lake model*.

The lake model is a basic tool for modeling unemployment.

It allows us to analyze

- flows between unemployment and employment
- how these flows influence steady state employment and unemployment rates

It is a good model for interpreting monthly labor department reports on gross and net jobs created and destroyed.

The "lakes" in the model are the pools of employed and unemployed.

The "flows" between the lakes are caused by

- firing and hiring
- entry and exit from the labor force

For the first part of this lecture, the parameters governing transitions into and out of unemployment and employment are exogenous.

Later, we'll determine some of these transition rates endogenously using the *McCall search model*.

We'll also use some nifty concepts like ergodicity, which provides a fundamental link between *cross-sectional* and *long run time series* distributions.

These concepts will help us build an equilibrium model of ex-ante homogeneous workers whose different luck generates variations in their ex post experiences.

Let's start with some imports:

```
import matplotlib.pyplot as plt
import jax
import jax.numpy as jnp
from typing import NamedTuple
from quantecon.distributions import BetaBinomial
from functools import partial
import jax.scipy.stats as stats
```

75.1.1 Prerequisites

Before working through what follows, we recommend you read the *lecture on finite Markov chains*.

You will also need some basic *linear algebra* and probability.

75.2 The model

The economy is inhabited by a very large number of ex-ante identical workers.

The workers live forever, spending their lives moving between unemployment and employment.

Their rates of transition between employment and unemployment are governed by the following parameters:

- λ, the job finding rate for currently unemployed workers
- α, the dismissal rate for currently employed workers
- b, the entry rate into the labor force
- d, the exit rate from the labor force

The growth rate of the labor force evidently equals $g = b - d$.

75.2.1 Aggregate variables

We want to derive the dynamics of the following aggregates:

- E_t, the total number of employed workers at date t
- U_t, the total number of unemployed workers at t
- N_t, the number of workers in the labor force at t

75.2.2 Laws of motion for stock variables

We begin by constructing laws of motion for the aggregate variables E_t, U_t, N_t.

Of the mass of workers E_t who are employed at date t,

- $(1-d)E_t$ will remain in the labor force
- of these, $(1-\alpha)(1-d)E_t$ will remain employed

Of the mass of workers U_t workers who are currently unemployed,

- $(1-d)U_t$ will remain in the labor force
- of these, $(1-d)\lambda U_t$ will become employed

Therefore, the number of workers who will be employed at date $t+1$ will be

$$E_{t+1} = (1-d)(1-\alpha)E_t + (1-d)\lambda U_t$$

A similar analysis implies

$$U_{t+1} = (1-d)\alpha E_t + (1-d)(1-\lambda)U_t + b(E_t + U_t)$$

The value $b(E_t + U_t)$ is the mass of new workers entering the labor force unemployed.

The total stock of workers $N_t = E_t + U_t$ evolves as

$$N_{t+1} = (1+b-d)N_t = (1+g)N_t$$

Letting $X_t := \begin{pmatrix} U_t \\ E_t \end{pmatrix}$, the law of motion for X is

$$X_{t+1} = AX_t \quad \text{where} \quad A := \begin{bmatrix} (1-d)(1-\lambda)+b & (1-d)\alpha + b \\ (1-d)\lambda & (1-d)(1-\alpha) \end{bmatrix}$$

This law tells us how total employment and unemployment evolve over time.

75.2.3 Laws of motion for rates

Now let's derive the law of motion for rates.

We want to track the values of the following objects:

- The employment rate $e_t := E_t/N_t$.
- The unemployment rate $u_t := U_t/N_t$.

(Here and below, capital letters represent aggregates and lowercase letters represent rates)

To get these we can divide both sides of $X_{t+1} = AX_t$ by N_{t+1} to get

$$\begin{bmatrix} U_{t+1}/N_{t+1} \\ E_{t+1}/N_{t+1} \end{bmatrix} = \frac{1}{1+g} A \begin{bmatrix} U_t/N_t \\ E_t/N_t \end{bmatrix}$$

Letting

$$x_t := \begin{pmatrix} u_t \\ e_t \end{pmatrix} = \begin{pmatrix} U_t/N_t \\ E_t/N_t \end{pmatrix}$$

we can also write this as

$$x_{t+1} = R x_t \quad \text{where} \quad R := \frac{1}{1+g} A$$

You can check that $e_t + u_t = 1$ implies that $e_{t+1} + u_{t+1} = 1$.

This follows from the fact that the columns of R sum to 1.

75.3 Implementation

Let's code up these equations.

75.3.1 Model

To begin, we set up a class called `LakeModel` that stores the primitives α, λ, b, d.

```
class LakeModel(NamedTuple):
    """
    Parameters for the lake model
    """
    λ: float
    α: float
    b: float
    d: float
    A: jnp.ndarray
    R: jnp.ndarray
    g: float


def create_lake_model(
        λ: float = 0.283,      # job finding rate
        α: float = 0.013,      # separation rate
        b: float = 0.0124,     # birth rate
        d: float = 0.00822     # death rate
    ) -> LakeModel:
    """
    Create a LakeModel instance with default parameters.

    Computes and stores the transition matrices A and R,
    and the labor force growth rate g.

    """
    # Compute growth rate
```

(continues on next page)

(continued from previous page)

```
    g = b - d

    # Compute transition matrix A
    A = jnp.array([
        [(1-d) * (1-λ) + b, (1-d) * α + b],
        [(1-d) * λ,          (1-d) * (1-α)]
    ])

    # Compute normalized transition matrix R
    R = A / (1 + g)

    return LakeModel(λ=λ, α=α, b=b, d=d, A=A, R=R, g=g)
```

The default parameter values are:

- $\alpha = 0.013$ and $\lambda = 0.283$ are based on [Davis *et al.*, 2006]
- $b = 0.0124$ and $d = 0.00822$ are set to match monthly birth and death rates, respectively, in the U.S. population

As an experiment, let's create two instances, one with $\alpha = 0.013$ and another with $\alpha = 0.03$

```
model = create_lake_model()
print(f"Default α: {model.α}")
print(f"A matrix:\n{model.A}")
print(f"R matrix:\n{model.R}")
```

```
Default α: 0.013
A matrix:
[[0.7235063  0.02529314]
 [0.28067374 0.97888684]]
R matrix:
[[0.7204946  0.02518786]
 [0.27950543 0.97481215]]
```

```
model_new = create_lake_model(α=0.03)
print(f"New α: {model_new.α}")
print(f"New A matrix:\n{model_new.A}")
print(f"New R matrix:\n{model_new.R}")
```

```
New α: 0.03
New A matrix:
[[0.7235063  0.0421534 ]
 [0.28067374 0.9620266 ]]
New R matrix:
[[0.7204946  0.04197793]
 [0.27950543 0.9580221 ]]
```

75.3.2 Code for dynamics

We will also use a specialized function to generate time series in an efficient JAX-compatible manner.

Iteratively generating time series is somewhat nontrivial in JAX because arrays are immutable.

Here we use `lax.scan`, which allows the function to be jit-compiled.

Readers who prefer to skip the details can safely continue reading after the function definition.

```
@partial(jax.jit, static_argnames=['f', 'num_steps'])
def generate_path(f, initial_state, num_steps, **kwargs):
    """
    Generate a time series by repeatedly applying an update rule.

    Given a map f, initial state x_0, and model parameters, this
    function computes and returns the sequence {x_t}_{t=0}^{T-1} when

        x_{t+1} = f(x_t, **kwargs)

    Args:
        f: Update function mapping (x_t, **kwargs) -> x_{t+1}
        initial_state: Initial state x_0
        num_steps: Number of time steps T to simulate
        **kwargs: Optional extra arguments passed to f

    Returns:
        Array of shape (dim(x), T) containing the time series path
        [x_0, x_1, x_2, ..., x_{T-1}]
    """

    def update_wrapper(state, t):
        """
        Wrapper function that adapts f for use with JAX scan.
        """
        next_state = f(state, **kwargs)
        return next_state, state

    _, path = jax.lax.scan(update_wrapper,
                    initial_state, jnp.arange(num_steps))
    return path.T
```

Here are functions to update X_t and x_t.

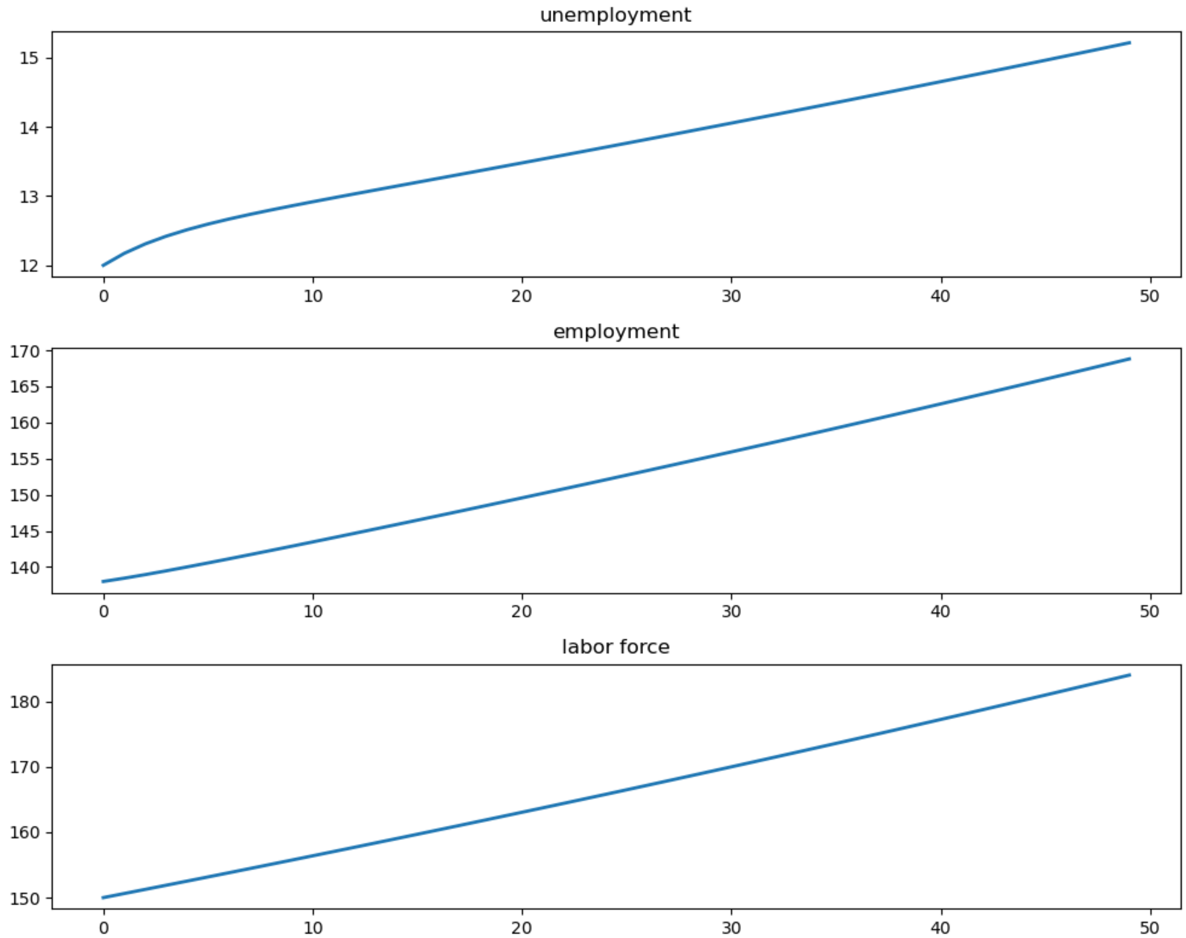
```
def stock_update(X: jnp.ndarray, model: LakeModel) -> jnp.ndarray:
    """Apply transition matrix to get next period's stocks."""
    λ, α, b, d, A, R, g = model
    return A @ X

def rate_update(x: jnp.ndarray, model: LakeModel) -> jnp.ndarray:
    """Apply normalized transition matrix for next period's rates."""
    λ, α, b, d, A, R, g = model
    return R @ x
```

75.3.3 Aggregate dynamics

Let's run a simulation under the default parameters starting from $X_0 = (12, 138)$.

We will plot the sequences $\{E_t\}$, $\{U_t\}$ and $\{N_t\}$.

```
N_0 = 150       # Population
e_0 = 0.92      # Initial employment rate
u_0 = 1 - e_0   # Initial unemployment rate
T = 50          # Simulation length

U_0 = u_0 * N_0
E_0 = e_0 * N_0

# Generate X path
X_0 = jnp.array([U_0, E_0])
X_path = generate_path(stock_update, X_0, T, model=model)

# Plot
fig, axes = plt.subplots(3, 1, figsize=(10, 8))
titles = ['unemployment', 'employment', 'labor force']
data = [X_path[0, :], X_path[1, :], X_path.sum(0)]
for ax, title, series in zip(axes, titles, data):
    ax.plot(series, lw=2)
    ax.set_title(title)
plt.tight_layout()
plt.show()
```



The aggregates E_t and U_t don't converge because their sum $E_t + U_t$ grows at rate g.

75.3.4 Rate dynamics

On the other hand, the vector of employment and unemployment rates x_t can be in a steady state $\bar{x}$ if there exists an $\bar{x}$ such that

- $\bar{x} = R\bar{x}$
- the components satisfy $\bar{e} + \bar{u} = 1$

This equation tells us that a steady state level $\bar{x}$ is an eigenvector of R associated with a unit eigenvalue.

The following function can be used to compute the steady state.

```
@jax.jit
def rate_steady_state(model: LakeModel) -> jnp.ndarray:
    r"""
    Finds the steady state of the system :math:`x_{t+1} = R x_{t}`
    by computing the eigenvector corresponding to the largest eigenvalue.

    By the Perron-Frobenius theorem, since :math:`R` is a non-negative
    matrix with columns summing to 1 (a stochastic matrix), the largest
    eigenvalue equals 1 and the corresponding eigenvector gives the steady state.
    """
```

(continues on next page)

(continued from previous page)

```
    λ, α, b, d, A, R, g = model
    eigenvals, eigenvec = jnp.linalg.eig(R)

    # Find the eigenvector corresponding to the largest eigenvalue
    # (which is 1 for a stochastic matrix by Perron-Frobenius theorem)
    max_idx = jnp.argmax(jnp.abs(eigenvals))

    # Get the corresponding eigenvector
    steady_state = jnp.real(eigenvec[:, max_idx])

    # Normalize to ensure positive values and sum to 1
    steady_state = jnp.abs(steady_state)
    steady_state = steady_state / jnp.sum(steady_state)

    return steady_state
```

We also have $x_t \to \bar{x}$ as $t \to \infty$ provided that the remaining eigenvalue of R has modulus less than 1.

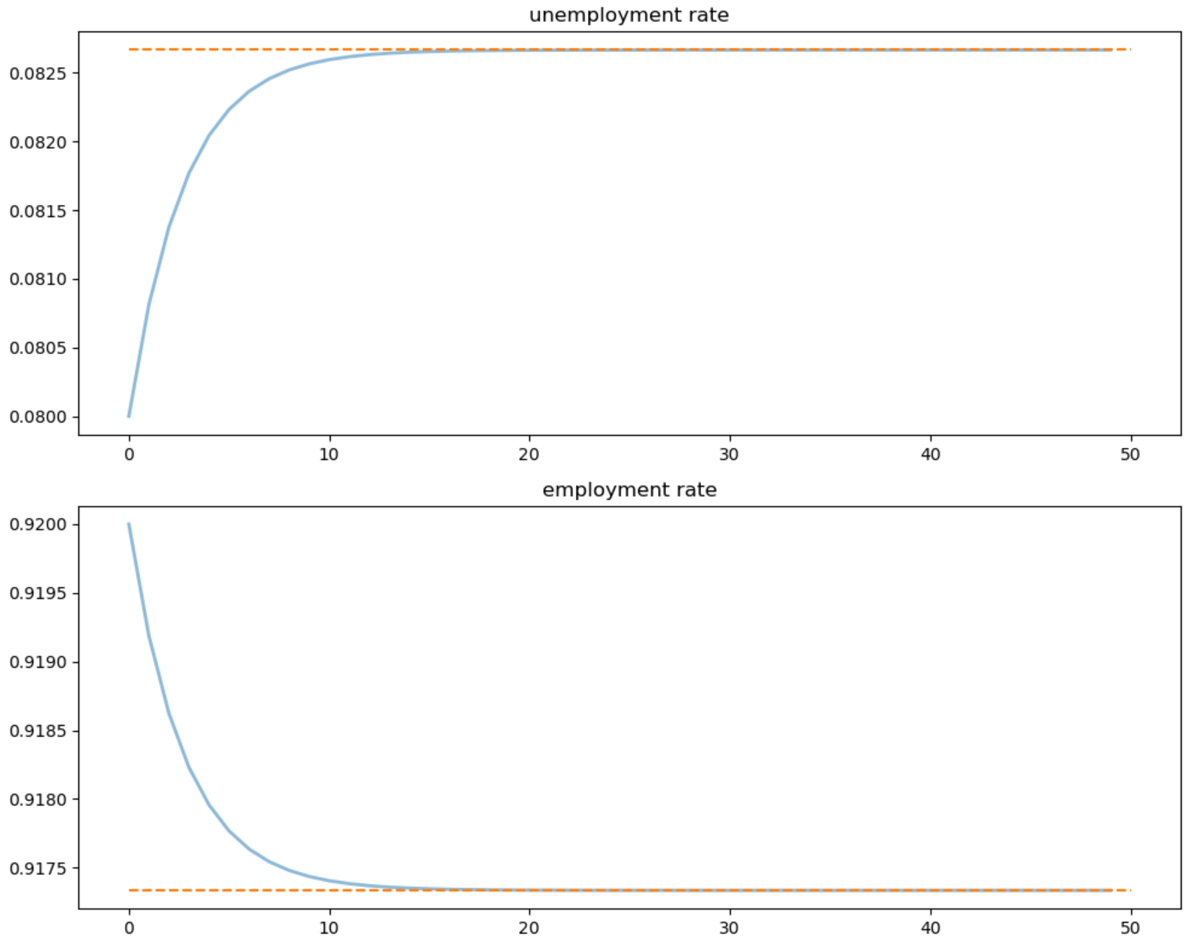
This is the case for our default parameters:

```
model = create_lake_model()
e, f = jnp.linalg.eigvals(model.R)
print(f"Eigenvalue magnitudes: {abs(e):.2f}, {abs(f):.2f}")
```

```
Eigenvalue magnitudes: 0.70, 1.00
```

Let's look at the convergence of the unemployment and employment rates to steady state levels (dashed line)

```
xbar = rate_steady_state(model)

fig, axes = plt.subplots(2, 1, figsize=(10, 8))
x_0 = jnp.array([u_0, e_0])
x_path = generate_path(rate_update, x_0, T, model=model)

titles = ['unemployment rate', 'employment rate']

for i, title in enumerate(titles):
    axes[i].plot(x_path[i, :], lw=2, alpha=0.5)
    axes[i].hlines(xbar[i], 0, T, color='C1', linestyle='--')
    axes[i].set_title(title)

plt.tight_layout()
plt.show()
```



Exercise 75.3.1

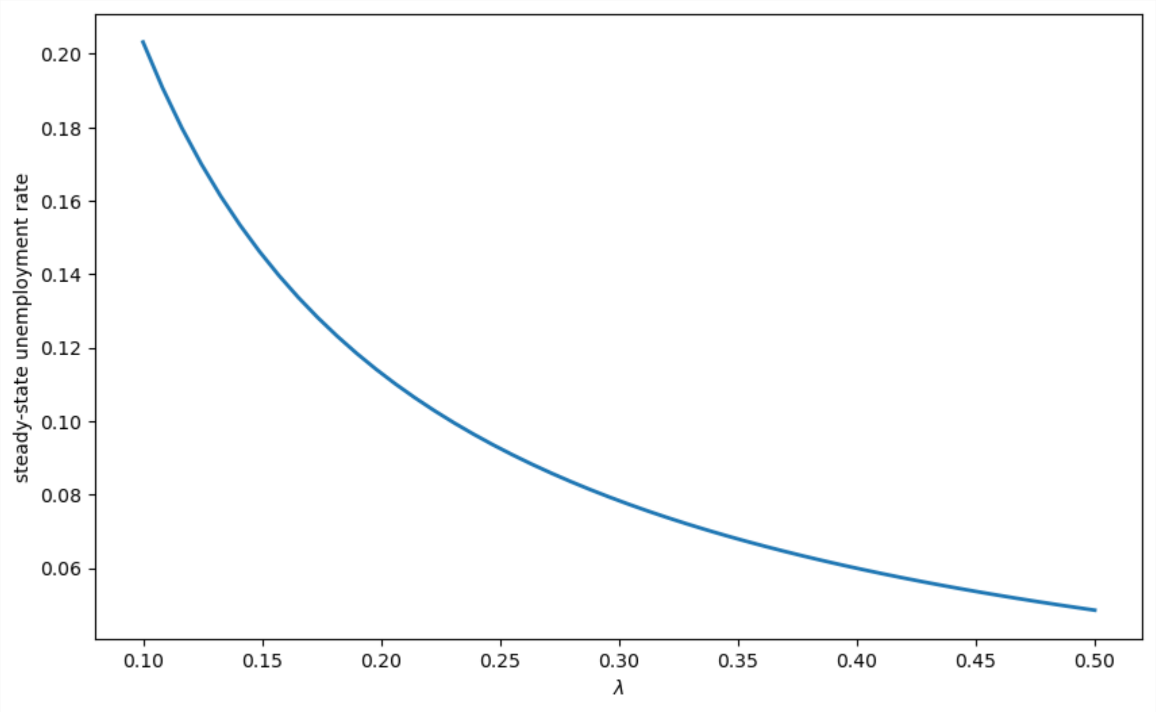
Use JAX's `vmap` to compute steady-state unemployment rates for a range of job finding rates λ (from 0.1 to 0.5), and plot the relationship.

Solution

Here is one solution

```
@jax.jit
def compute_unemployment_rate(λ_val):
    """Computes steady-state unemployment for a given λ"""
    model = create_lake_model(λ=λ_val)
    steady_state = rate_steady_state(model)
    return steady_state[0]

# Use vmap to compute for multiple λ values
λ_values = jnp.linspace(0.1, 0.5, 50)
unemployment_rates = jax.vmap(compute_unemployment_rate)(λ_values)

# Plot the results
fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(λ_values, unemployment_rates, lw=2)
```

```
ax.set_xlabel(r'$\lambda$')
ax.set_ylabel('steady-state unemployment rate')
plt.show()
```



75.4 Dynamics of an individual worker

An individual worker's employment dynamics are governed by a *finite state Markov process*.

The worker can be in one of two states:

- $s_t = 0$ means unemployed
- $s_t = 1$ means employed

Let's start off under the assumption that $b = d = 0$.

The associated transition matrix is then

$$P = \begin{pmatrix} 1-\lambda & \lambda \\ \alpha & 1-\alpha \end{pmatrix}$$

Let ψ_t denote the *marginal distribution* over employment/unemployment states for the worker at time t.

As usual, we regard it as a row vector.

We know *from an earlier discussion* that ψ_t follows the law of motion

$$\psi_{t+1} = \psi_t P$$

We also know from the *lecture on finite Markov chains* that if $\alpha \in (0,1)$ and $\lambda \in (0,1)$, then P has a unique stationary distribution, denoted here by ψ^*.

The unique stationary distribution satisfies

$$\psi^*[0] = \frac{\alpha}{\alpha + \lambda}$$

Not surprisingly, probability mass on the unemployment state increases with the dismissal rate and falls with the job finding rate.

75.4.1 Ergodicity

Let's look at a typical lifetime of employment-unemployment spells.

We want to compute the average amounts of time an infinitely lived worker would spend employed and unemployed.

Let

$$\bar{s}_{u,T} := \frac{1}{T} \sum_{t=1}^{T} \mathbb{1}\{s_t = 0\}$$

and

$$\bar{s}_{e,T} := \frac{1}{T} \sum_{t=1}^{T} \mathbb{1}\{s_t = 1\}$$

(As usual, $\mathbb{1}\{Q\} = 1$ if statement Q is true and 0 otherwise)

These are the fraction of time a worker spends unemployed and employed, respectively, up until period T.

If $\alpha \in (0,1)$ and $\lambda \in (0,1)$, then P is *ergodic*, and hence we have

$$\lim_{T \to \infty} \bar{s}_{u,T} = \psi^*[0] \quad \text{and} \quad \lim_{T \to \infty} \bar{s}_{e,T} = \psi^*[1]$$

with probability one.

Inspection tells us that P is exactly the transpose of R under the assumption $b = d = 0$.

Thus, the percentages of time that an infinitely lived worker spends employed and unemployed equal the fractions of workers employed and unemployed in the steady state distribution.

75.4.2 Convergence rate

How long does it take for time series sample averages to converge to cross-sectional averages?

We can investigate this by simulating the Markov chain.

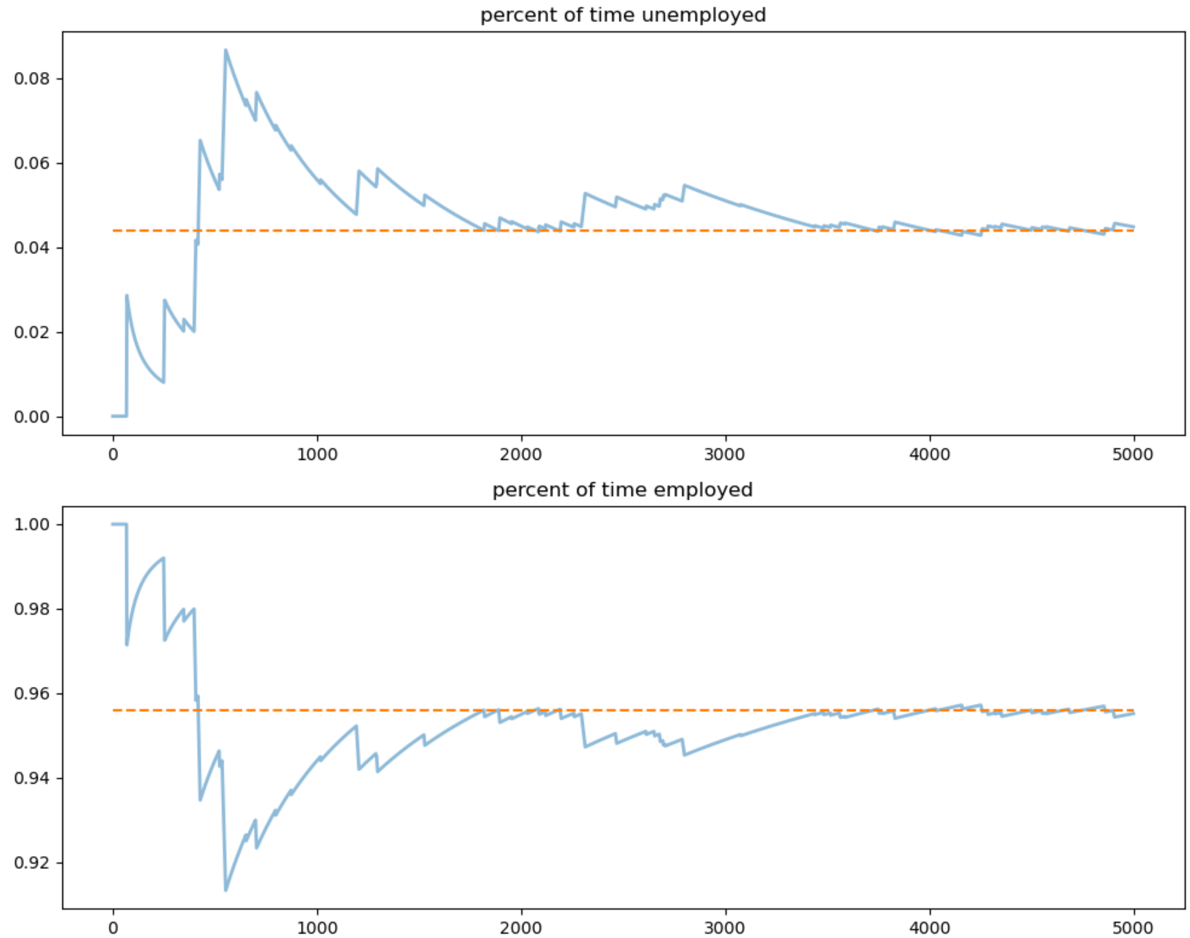
Let's plot the path of the sample averages over 5,000 periods

```
def markov_update(state, P, key):
    """
    Sample next state from transition probabilities.
    """
    probs = P[state]
    state_new = jax.random.choice(key,
                                  a=jnp.arange(len(probs)),
                                  p=probs)
    return state_new

model_markov = create_lake_model(d=0, b=0)
```

(continues on next page)

(continued from previous page)

```
T = 5000  # Simulation length

α, λ = model_markov.α, model_markov.λ

P = jnp.array([[1 - λ,        λ],
               [    α,    1 - α]])

xbar = rate_steady_state(model_markov)

# Simulate the Markov chain - we need a different approach for random updates
key = jax.random.PRNGKey(0)

def simulate_markov(P, initial_state, T, key):
    """Simulate Markov chain for T periods"""
    keys = jax.random.split(key, T)

    def scan_fn(state, key):
        next_state = markov_update(state, P, key)
        return next_state, state

    _, path = jax.lax.scan(scan_fn, initial_state, keys)
    return path

s_path = simulate_markov(P, 1, T, key)

fig, axes = plt.subplots(2, 1, figsize=(10, 8))
s_bar_e = jnp.cumsum(s_path) / jnp.arange(1, T+1)
s_bar_u = 1 - s_bar_e

to_plot = [s_bar_u, s_bar_e]
titles = ['percent of time unemployed', 'percent of time employed']

for i, plot in enumerate(to_plot):
    axes[i].plot(plot, lw=2, alpha=0.5)
    axes[i].hlines(xbar[i], 0, T, color='C1', linestyle='--')
    axes[i].set_title(titles[i])

plt.tight_layout()
plt.show()
```



The stationary probabilities are given by the dashed line.

In this case it takes much of the sample for these two objects to converge.

This is largely due to the high persistence in the Markov chain.

75.5 Exercises

Exercise 75.5.1

Consider an economy with an initial stock of workers $N_0 = 100$ at the steady state level of employment in the baseline parameterization.

Suppose that in response to new legislation the hiring rate reduces to $\lambda = 0.2$.

Plot the transition dynamics of the unemployment and employment stocks for 50 periods.

Plot the transition dynamics for the rates.

How long does the economy take to converge to its new steady state?

What is the new steady state level of employment?

Solution

We begin by constructing the model with default parameters and finding the initial steady state

```
model_initial = create_lake_model()
x0 = rate_steady_state(model_initial)
print(f"Initial Steady State: {x0}")
```

```
Initial Steady State: [0.08266623 0.9173338 ]
```

Initialize the simulation values

```
N0 = 100
T = 50
```

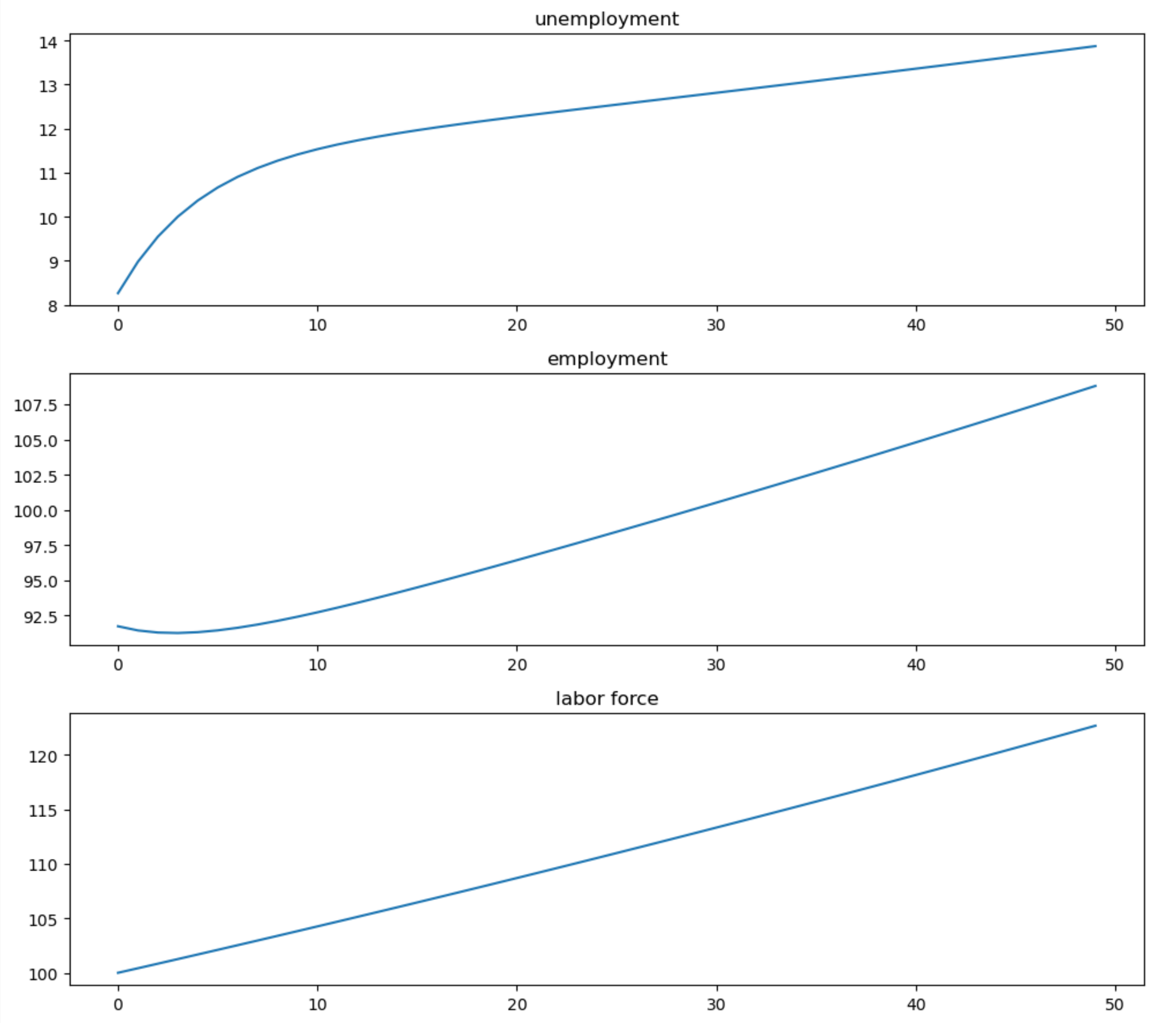
New legislation changes λ to 0.2

```
model_ex2 = create_lake_model(λ=0.2)
xbar = rate_steady_state(model_ex2)  # new steady state

# Simulate paths
X_path = generate_path(stock_update, x0 * N0, T, model=model_ex2)
x_path = generate_path(rate_update, x0, T, model=model_ex2)
print(f"New Steady State: {xbar}")
```

```
New Steady State: [0.11309288 0.8869071 ]
```

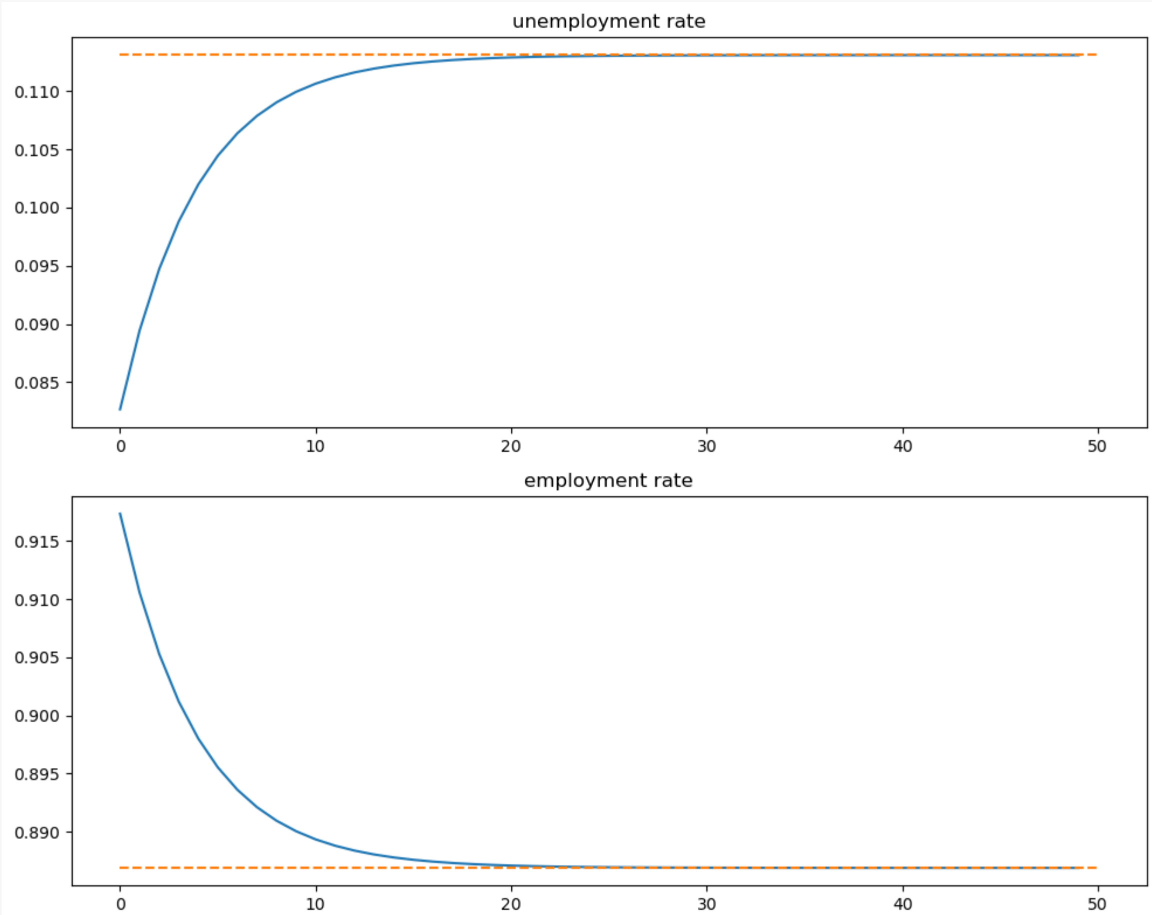
Now plot stocks

```
fig, axes = plt.subplots(3, 1, figsize=[10, 9])

axes[0].plot(X_path[0, :])
axes[0].set_title('unemployment')

axes[1].plot(X_path[1, :])
axes[1].set_title('employment')

axes[2].plot(X_path.sum(0))
axes[2].set_title('labor force')

plt.tight_layout()
plt.show()
```



And how the rates evolve

```
fig, axes = plt.subplots(2, 1, figsize=(10, 8))

titles = ['unemployment rate', 'employment rate']

for i, title in enumerate(titles):
    axes[i].plot(x_path[i, :])
    axes[i].hlines(xbar[i], 0, T, color='C1', linestyle='--')
    axes[i].set_title(title)

plt.tight_layout()
plt.show()
```



We see that it takes 20 periods for the economy to converge to its new steady state levels.

Exercise 75.5.2

Consider an economy with an initial stock of workers $N_0 = 100$ at the steady state level of employment in the baseline parameterization.

Suppose that for 20 periods the birth rate was temporarily high ($b = 0.025$) and then returned to its original level.

Plot the transition dynamics of the unemployment and employment stocks for 50 periods.

Plot the transition dynamics for the rates.

How long does the economy take to return to its original steady state?

Solution

This exercise has the economy experiencing a boom in entrances to the labor market and then later returning to the original levels.

For 20 periods the economy has a new entry rate into the labor market.

Let's start off at the baseline parameterization and record the steady state

```
model_baseline = create_lake_model()
x0 = rate_steady_state(model_baseline)
N0 = 100
T = 50
```

Here are the other parameters:

```
b_hat = 0.025
T_hat = 20
```

Let's increase b to the new value and simulate for 20 periods

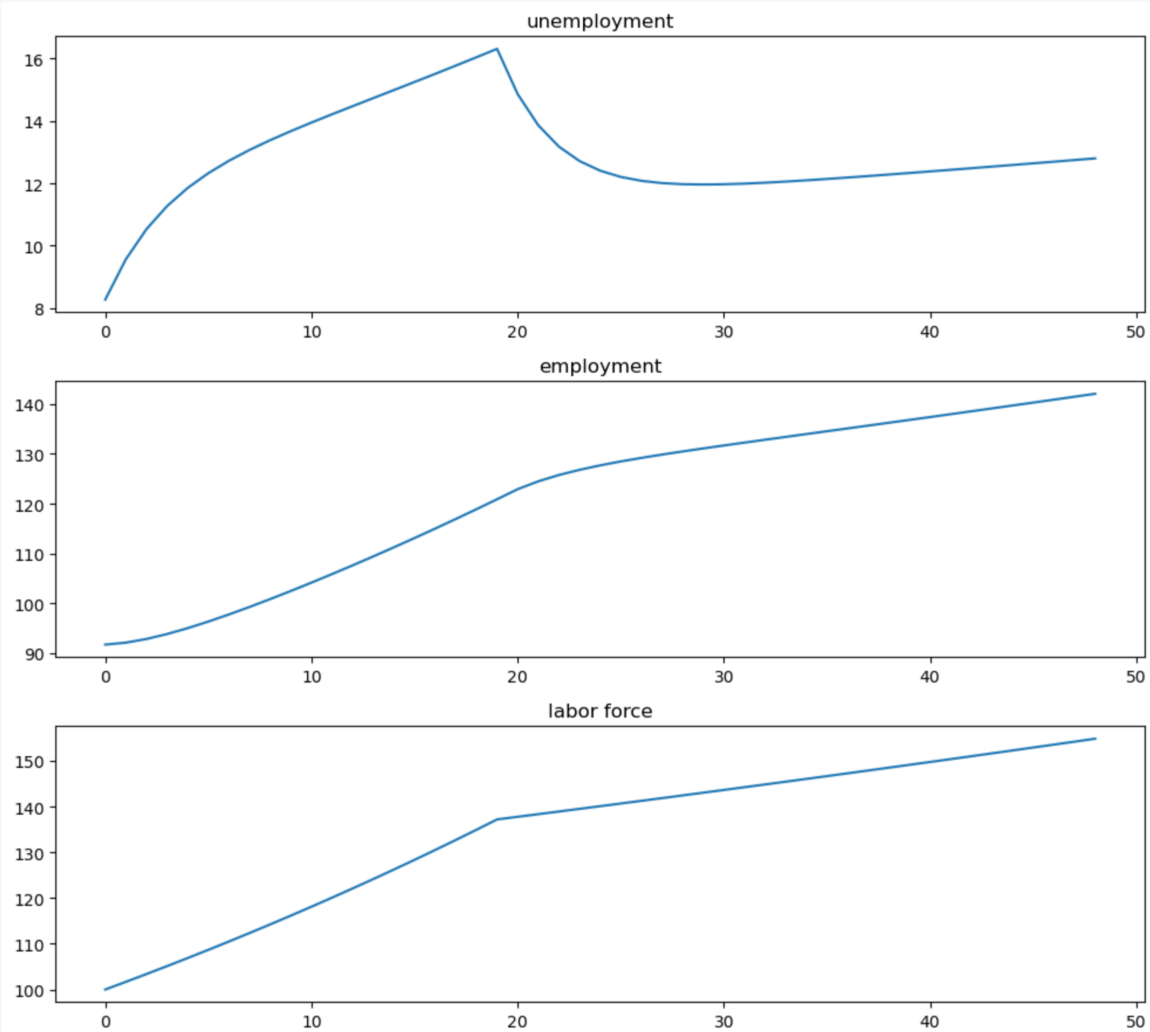
```
model_high_b = create_lake_model(b=b_hat)

# Simulate stocks and rates for first 20 periods
X_path1 = generate_path(stock_update, x0 * N0, T_hat, model=model_high_b)
x_path1 = generate_path(rate_update, x0, T_hat, model=model_high_b)
```

Now we reset b to the original value and then, using the state after 20 periods for the new initial conditions, we simulate for the additional 30 periods

```
# Use final state from period 20 as initial condition
X_path2 = generate_path(stock_update, X_path1[:, -1], T-T_hat,
                            model=model_baseline)
x_path2 = generate_path(rate_update, x_path1[:, -1], T-T_hat,
                            model=model_baseline)
```

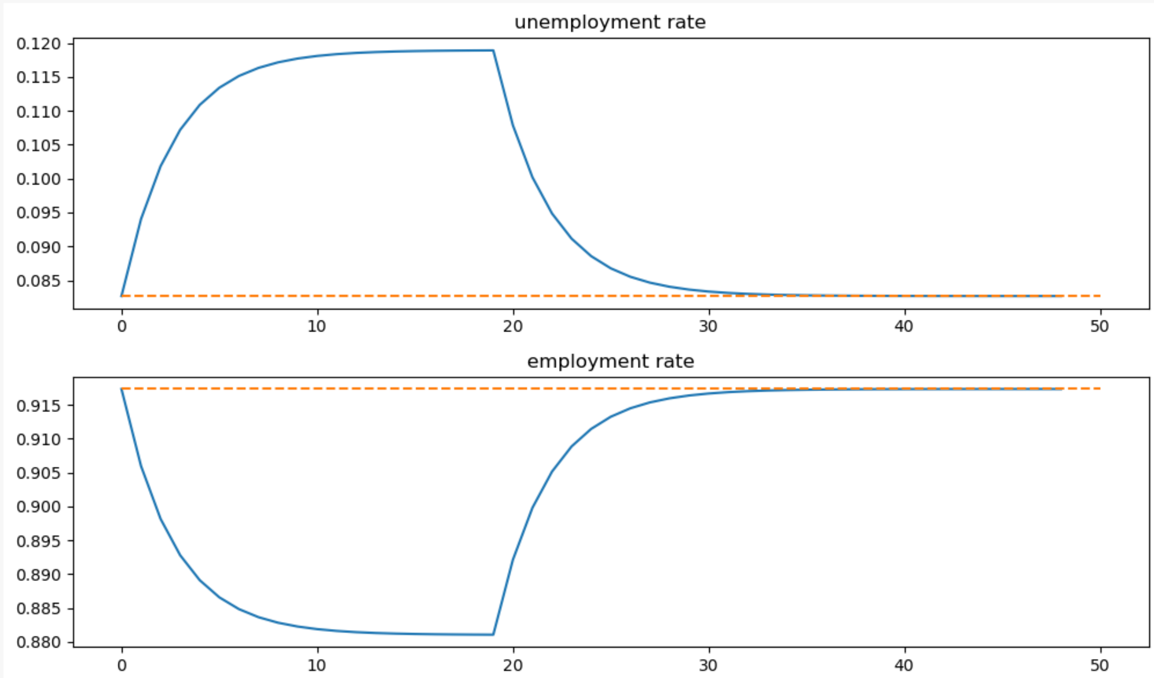
Finally, we combine these two paths and plot

```
# Combine paths
X_path = jnp.hstack([X_path1, X_path2[:, 1:]])
x_path = jnp.hstack([x_path1, x_path2[:, 1:]])

fig, axes = plt.subplots(3, 1, figsize=[10, 9])

axes[0].plot(X_path[0, :])
axes[0].set_title('unemployment')

axes[1].plot(X_path[1, :])
axes[1].set_title('employment')

axes[2].plot(X_path.sum(0))
axes[2].set_title('labor force')

plt.tight_layout()
plt.show()
```



And the rates

```
fig, axes = plt.subplots(2, 1, figsize=[10, 6])

titles = ['unemployment rate', 'employment rate']

for i, title in enumerate(titles):
    axes[i].plot(x_path[i, :])
    axes[i].hlines(x0[i], 0, T, color='C1', linestyle='--')
    axes[i].set_title(title)

plt.tight_layout()
plt.show()
```


unemployment rate
0.120
0.115
0.110
0.105
0.100
0.095
0.090
0.085
0
10
20
30
40
50
employment rate
0.915
0.910
0.905
0.900
0.895
0.890
0.885
0.880
0
10
20
30
40
50

CHAPTER

SEVENTYSIX

LAKE MODEL WITH AN ENDOGENOUS JOB FINDING RATE

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

Contents

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon jax
```

76.1 Overview

This lecture is a continuation of the *lake model lecture*.

We recommend you read that lecture first before proceeding with this one.

In the previous lecture, we studied a lake model of unemployment and employment where the transition rates between states were exogenous parameters.

In this lecture, we extend the model by making the job finding rate endogenous.

Specifically, the transition rate from unemployment to employment will be determined by the McCall search model [McCall, 1970].

Let's start with some imports:

```
import matplotlib.pyplot as plt
import jax
import jax.numpy as jnp
from typing import NamedTuple
from quantecon.distributions import BetaBinomial
from functools import partial
import jax.scipy.stats as stats
```

76.2 Set Up

The basic structure of the model will be as discussed in the *lake model lecture*.

The only difference is that the hiring rate is endogenous, determined by the decisions of optimizing agents inhabiting a McCall search model [McCall, 1970] with IID wage offers and job separation at rate α.

76.2.1 Reservation wage

In the model, the optimal policy is characterized by a reservation wage $\bar{w}$

- If the wage offer w in hand is greater than or equal to $\bar{w}$, then the worker accepts.
- Otherwise, the worker rejects.

The reservation wage depends on the wage offer distribution and the parameters

- α, the separation rate
- β, the discount factor
- γ, the offer arrival rate
- c, unemployment compensation

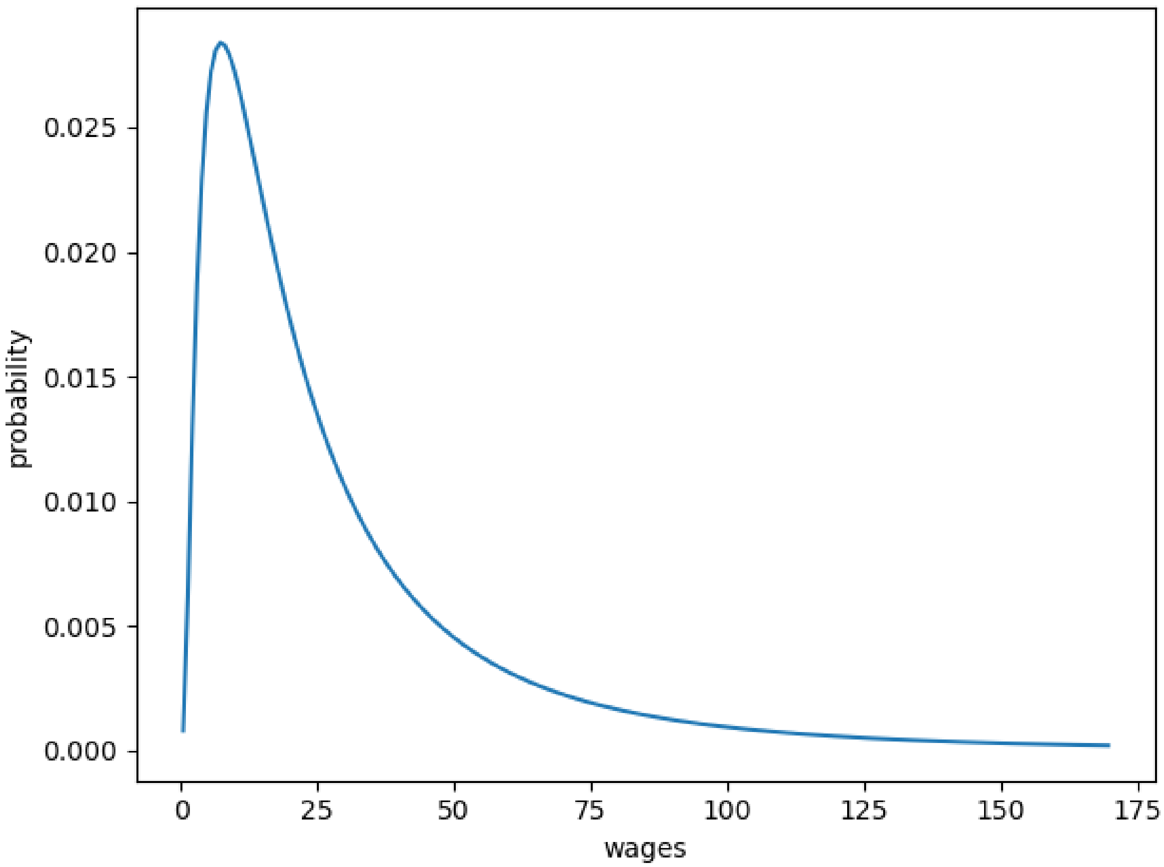
The wage offer distribution will be a discretized version of a lognormal distribution.

We first define a function to create such a discrete distribution.

```
def create_wage_distribution(
        max_wage: float,
        wage_grid_size: int,
        log_wage_mean: float
    ):
    """
    Creates a discretized version of a lognormal density LN(log(m),1), where
    m is log_wage_mean.

    """
    w_vec_temp = jnp.linspace(1e-8, max_wage, wage_grid_size + 1)
    cdf = stats.norm.cdf(
        jnp.log(w_vec_temp), loc=jnp.log(log_wage_mean), scale=1
    )
    pdf = cdf[1:] - cdf[:-1]
    p_vec = pdf / pdf.sum()
    w_vec = (w_vec_temp[1:] + w_vec_temp[:-1]) / 2
    return w_vec, p_vec
```

The cell below creates a discretized $LN(\log(20), 1)$ wage distribution and plots it.

```
w_vec, p_vec = create_wage_distribution(170, 200, 20)

fig, ax = plt.subplots()
ax.plot(w_vec, p_vec)
ax.set_xlabel('wages')
ax.set_ylabel('probability')
plt.tight_layout()
plt.show()
```



Now we organize the code for solving the McCall model, given a set of parameters.

For background on the model and our solution method, see the *lecture on the McCall model with separation*

Our first step is to define the utility function and the McCall model data structure.

```
def u(c, σ=2.0):
    return jnp.where(c > 0, (c**(1 - σ) - 1) / (1 - σ), -10e6)


class McCallModel(NamedTuple):
    """
    Stores the parameters for the McCall search model
    """
    α: float            # Job separation rate
    β: float            # Discount rate
    γ: float            # Job offer rate
```

(continues on next page)

(continued from previous page)

```
    c: float             # Unemployment compensation
    σ: float             # Utility parameter
    w_vec: jnp.ndarray  # Possible wage values
    p_vec: jnp.ndarray  # Probabilities over w_vec


def create_mccall_model(
        α=0.2, β=0.98, γ=0.7, c=6.0, σ=2.0,
        w_vec=None,
        p_vec=None
    ) -> McCallModel:
    if w_vec is None:
        n = 60  # Number of possible outcomes for wage
        # Wages between 10 and 20
        w_vec = jnp.linspace(10, 20, n)
        a, b = 600, 400  # Shape parameters
        dist = BetaBinomial(n-1, a, b)
        p_vec = jnp.array(dist.pdf())
    return McCallModel(
        α=α, β=β, γ=γ, c=c, σ=σ, w_vec=w_vec, p_vec=p_vec
    )
```

Next, we implement the Bellman operator

```
def T(mcm: McCallModel, V, U):
    """
    Update the Bellman equations.
    """
    α, β, γ, c, σ = mcm.α, mcm.β, mcm.γ, mcm.c, mcm.σ
    w_vec, p_vec = mcm.w_vec, mcm.p_vec

    V_new = u(w_vec, σ) + β * ((1 - α) * V + α * U)
    U_new = u(c, σ) + β * (1 - γ) * U + β * γ * (jnp.maximum(U, V) @ p_vec)

    return V_new, U_new
```

Now we define the value function iteration solver.

We'll use a compiled while loop for extra speed.

```
@jax.jit
def solve_mccall_model(mcm: McCallModel, tol=1e-5, max_iter=2000):
    """
    Iterates to convergence on the Bellman equations.
    """
    def cond(state):
        V, U, i, error = state
        return jnp.logical_and(error > tol, i < max_iter)

    def update(state):
        V, U, i, error = state
        V_new, U_new = T(mcm, V, U)
        error_1 = jnp.max(jnp.abs(V_new - V))
        error_2 = jnp.abs(U_new - U)
        error_new = jnp.maximum(error_1, error_2)
        return V_new, U_new, i + 1, error_new
```

(continues on next page)

(continued from previous page)

```
    # Initial state
    V_init = jnp.ones(len(mcm.w_vec))
    U_init = 1.0
    i_init = 0
    error_init = tol + 1

    init_state = (V_init, U_init, i_init, error_init)
    V_final, U_final, _, _ = jax.lax.while_loop(
        cond, update, init_state
    )
    return V_final, U_final
```

76.2.2 Lake model code

We also need the lake model functions from the previous lecture to compute steady state unemployment rates:

```
class LakeModel(NamedTuple):
    """
    Parameters for the lake model
    """
    λ: float
    α: float
    b: float
    d: float
    A: jnp.ndarray
    R: jnp.ndarray
    g: float


def create_lake_model(
        λ: float = 0.283,      # job finding rate
        α: float = 0.013,      # separation rate
        b: float = 0.0124,     # birth rate
        d: float = 0.00822     # death rate
    ) -> LakeModel:
    """
    Create a LakeModel instance with default parameters.

    Computes and stores the transition matrices A and R,
    and the labor force growth rate g.

    """
    # Compute growth rate
    g = b - d

    # Compute transition matrix A
    A = jnp.array([
        [(1-d) * (1-λ) + b, (1-d) * α + b],
        [(1-d) * λ,          (1-d) * (1-α)]
    ])

    # Compute normalized transition matrix R
    R = A / (1 + g)

    return LakeModel(λ=λ, α=α, b=b, d=d, A=A, R=R, g=g)
```

(continues on next page)

(continued from previous page)

```
@jax.jit
def rate_steady_state(model: LakeModel) -> jnp.ndarray:
    r"""
    Finds the steady state of the system :math:`x_{t+1} = R x_{t}`
    by computing the eigenvector corresponding to the largest eigenvalue.

    By the Perron-Frobenius theorem, since :math:`R` is a non-negative
    matrix with columns summing to 1 (a stochastic matrix), the largest
    eigenvalue equals 1 and the corresponding eigenvector gives the steady state.
    """
    λ, α, b, d, A, R, g = model
    eigenvals, eigenvec = jnp.linalg.eig(R)

    # Find the eigenvector corresponding to the largest eigenvalue
    # (which is 1 for a stochastic matrix by Perron-Frobenius theorem)
    max_idx = jnp.argmax(jnp.abs(eigenvals))

    # Get the corresponding eigenvector
    steady_state = jnp.real(eigenvec[:, max_idx])

    # Normalize to ensure positive values and sum to 1
    steady_state = jnp.abs(steady_state)
    steady_state = steady_state / jnp.sum(steady_state)

    return steady_state
```

76.2.3 Linking the McCall search model to the lake model

Suppose that all workers inside a lake model behave according to the McCall search model.

The exogenous probability of leaving employment remains α.

But their optimal decision rules determine the probability λ of leaving unemployment.

This is now

$$\lambda = \gamma \mathbb{P}\{w_t \geq \bar{w}\} = \gamma \sum_{w' \geq \bar{w}} p(w') \tag{76.1}$$

Here

- $\bar{w}$ is the reservation wage determined by the parameters and
- p is the wage offer distribution.

Wage offers across the population of workers are independent draws from p.

Here we calculate λ at the default parameters:

```
mcm = create_mccall_model(w_vec=w_vec, p_vec=p_vec)
V, U = solve_mccall_model(mcm)
w_idx = jnp.searchsorted(V - U, 0)
w_bar = jnp.where(w_idx == len(V), jnp.inf, mcm.w_vec[w_idx])
λ = mcm.γ * jnp.sum(p_vec * (w_vec > w_bar))
print(f"Job finding rate at default paramters ={λ}.")
```

```
Job finding rate at default paramters =0.4510621726512909.
```

76.3 Fiscal policy

In this section, we will put the lake model to work, examining outcomes associated with different levels of unemployment compensation.

Our aim is to find an optimal level of unemployment insurance.

We assume that the government sets unemployment compensation c.

The government imposes a lump-sum tax τ sufficient to finance total unemployment payments.

To attain a balanced budget at a steady state, taxes, the steady state unemployment rate u, and the unemployment compensation rate must satisfy

$$\tau = uc$$

The lump-sum tax applies to everyone, including unemployed workers.

- The post-tax income of an employed worker with wage w is $w - \tau$.
- The post-tax income of an unemployed worker is $c - \tau$.

For each specification (c, τ) of government policy, we can solve for the worker's optimal reservation wage.

This determines λ via (76.1) evaluated at post tax wages, which in turn determines a steady state unemployment rate $u(c, \tau)$.

For a given level of unemployment benefit c, we can solve for a tax that balances the budget in the steady state

$$\tau = u(c, \tau)c$$

To evaluate alternative government tax-unemployment compensation pairs, we require a welfare criterion.

We use a steady state welfare criterion

$$W := e\,\mathbb{E}[V \,|\, \text{employed}] + u\,U$$

where the notation V and U is as defined above and the expectation is at the steady state.

76.3.1 Computing optimal unemployment insurance

Now we set up the infrastructure to compute optimal unemployment insurance levels.

First, we define a container for the economy's parameters:

```
class Economy(NamedTuple):
    """Parameters for the economy"""
    α: float
    b: float
    d: float
    β: float
    γ: float
    σ: float
    log_wage_mean: float
    wage_grid_size: int
```

(continues on next page)

(continued from previous page)

```
    max_wage: float

def create_economy(
        α=0.013,
        b=0.0124,
        d=0.00822,
        β=0.98,
        γ=1.0,
        σ=2.0,
        log_wage_mean=20,
        wage_grid_size=200,
        max_wage=170
    ) -> Economy:
    """
    Create an economy with a set of default values"""
    return Economy(α=α, b=b, d=d, β=β, γ=γ, σ=σ,
                        log_wage_mean=log_wage_mean,
                        wage_grid_size=wage_grid_size,
                        max_wage=max_wage)
```

Next, we define a function that computes optimal worker behavior given policy parameters:

```
@jax.jit
def compute_optimal_quantities(
        c: float,
        τ: float,
        economy: Economy,
        w_vec: jnp.array,
        p_vec: jnp.array
    ):
    """
    Compute the reservation wage, job finding rate and value functions
    of the workers given c and τ.

    """
    mcm = create_mccall_model(
        α=economy.α,
        β=economy.β,
        γ=economy.γ,
        c=c-τ,          # Post tax compensation
        σ=economy.σ,
        w_vec=w_vec-τ,  # Post tax wages
        p_vec=p_vec
    )

    # Compute reservation wage under given parameters
    V, U = solve_mccall_model(mcm)
    w_idx = jnp.searchsorted(V - U, 0)
    w_bar = jnp.where(w_idx == len(V), jnp.inf, mcm.w_vec[w_idx])

    # Compute job finding rate
    λ = economy.γ * jnp.sum(p_vec * (w_vec - τ > w_bar))

    return w_bar, λ, V, U
```

This function computes the steady state outcomes given unemployment insurance and tax levels:

```python
@jax.jit
def compute_steady_state_quantities(
        c, τ, economy: Economy, w_vec, p_vec
    ):
    """
    Compute the steady state unemployment rate given c and τ using optimal
    quantities from the McCall model and computing corresponding steady
    state quantities

    """

    # Find optimal values and policies by solving the McCall model, as well
    # as the corresponding job finding rate.
    w_bar, λ, V, U = compute_optimal_quantities(c, τ, economy, w_vec, p_vec)

    # Set up a lake model using the given parameters and the job finding rate.
    model = create_lake_model(λ=λ, α=economy.α, b=economy.b, d=economy.d)

    # Compute steady state employment and unemployment rates from this lake
    # model.
    u, e = rate_steady_state(model)

    # Compute expected lifetime value conditional on being employed.
    mask = (w_vec - τ > w_bar)
    w = jnp.sum(V * p_vec * mask) / jnp.sum(p_vec * mask)
    # Compute steady state welfare.
    welfare = e * w + u * U

    return e, u, welfare
```

We need a function to find the tax rate that balances the government budget:

```python
def find_balanced_budget_tax(c, economy: Economy, w_vec, p_vec):
    """
    Find the tax rate that will induce a balanced budget given unemployment
    compensation c.

    """

    def steady_state_budget(t):
        """
        For given tax rate t, compute the budget surplus.

        """
        e, u, w = compute_steady_state_quantities(c, t, economy, w_vec, p_vec)
        return t - u * c

    # Use a simple bisection method to find the tax rate that balances the
    # budget (but setting the surplus to zero

    t_low, t_high = 0.0, 0.9 * c
    tol = 1e-6
    max_iter = 100
    for i in range(max_iter):
        t_mid = (t_low + t_high) / 2
        budget = steady_state_budget(t_mid)
        if abs(budget) < tol:
```

(continues on next page)

(continued from previous page)

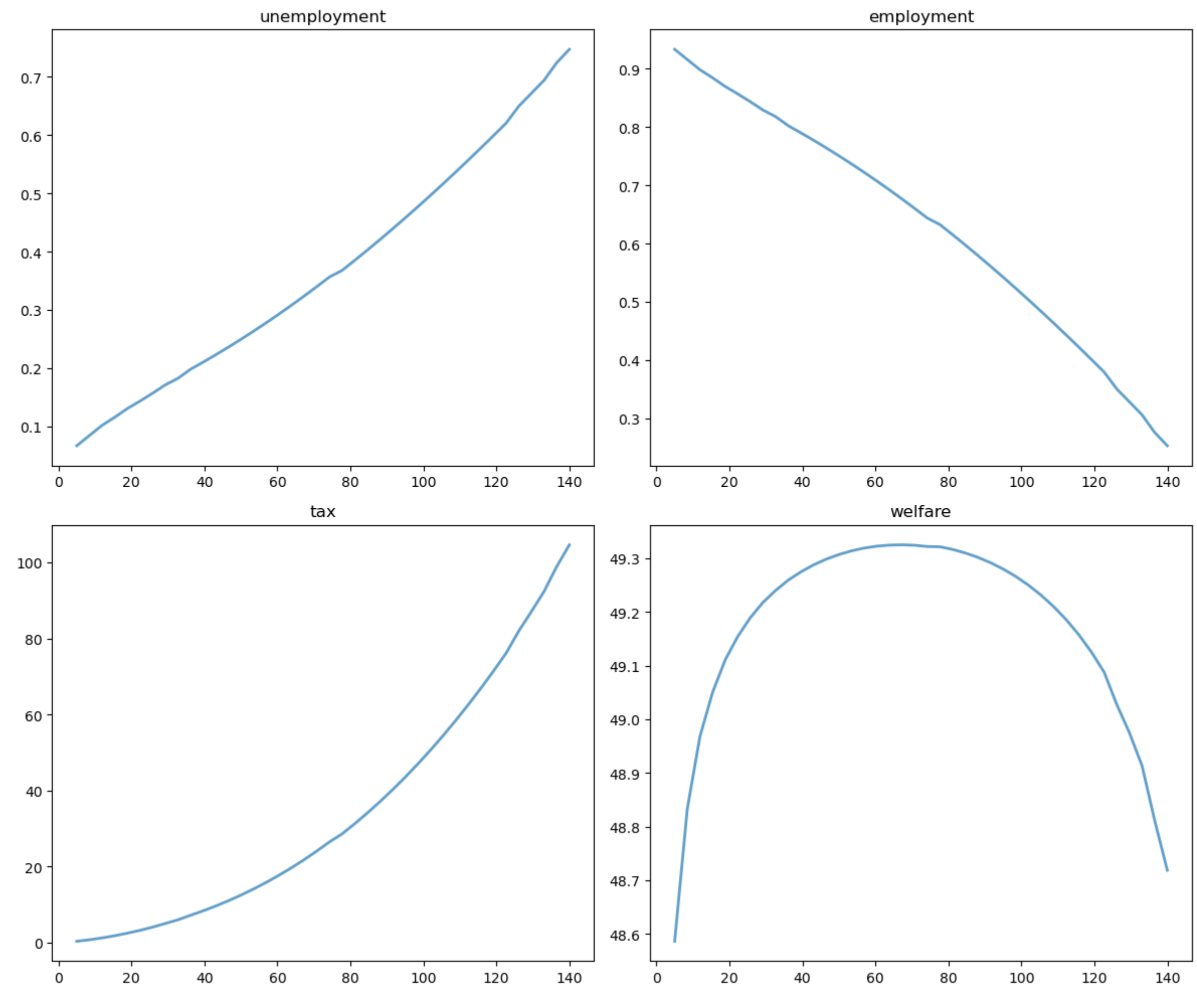
```
            return t_mid
        elif budget < 0:
            t_low = t_mid
        else:
            t_high = t_mid

    return t_mid
```

Now we compute how employment, unemployment, taxes, and welfare vary with the unemployment compensation rate:

```
# Create economy and wage distribution
economy = create_economy()
w_vec, p_vec = create_wage_distribution(
    economy.max_wage, economy.wage_grid_size, economy.log_wage_mean
)

# Levels of unemployment insurance we wish to study
c_vec = jnp.linspace(5, 140, 40)

tax_vec = []
unempl_vec = []
empl_vec = []
welfare_vec = []

for c in c_vec:
    t = find_balanced_budget_tax(c, economy, w_vec, p_vec)
    e_rate, u_rate, welfare = compute_steady_state_quantities(
        c, t, economy, w_vec, p_vec
    )
    tax_vec.append(t)
    unempl_vec.append(u_rate)
    empl_vec.append(e_rate)
    welfare_vec.append(welfare)
```

Let's visualize the results:

```
fig, axes = plt.subplots(2, 2, figsize=(12, 10))

plots = [unempl_vec, empl_vec, tax_vec, welfare_vec]
titles = ['unemployment', 'employment', 'tax', 'welfare']

for ax, plot, title in zip(axes.flatten(), plots, titles):
    ax.plot(c_vec, plot, lw=2, alpha=0.7)
    ax.set_title(title)

plt.tight_layout()
plt.show()
```



Welfare first increases and then decreases as unemployment benefits rise.

The level that maximizes steady state welfare is approximately 62.

76.4 Exercises

Exercise 76.4.1

How does the welfare-maximizing level of unemployment compensation c change with the job separation rate α?

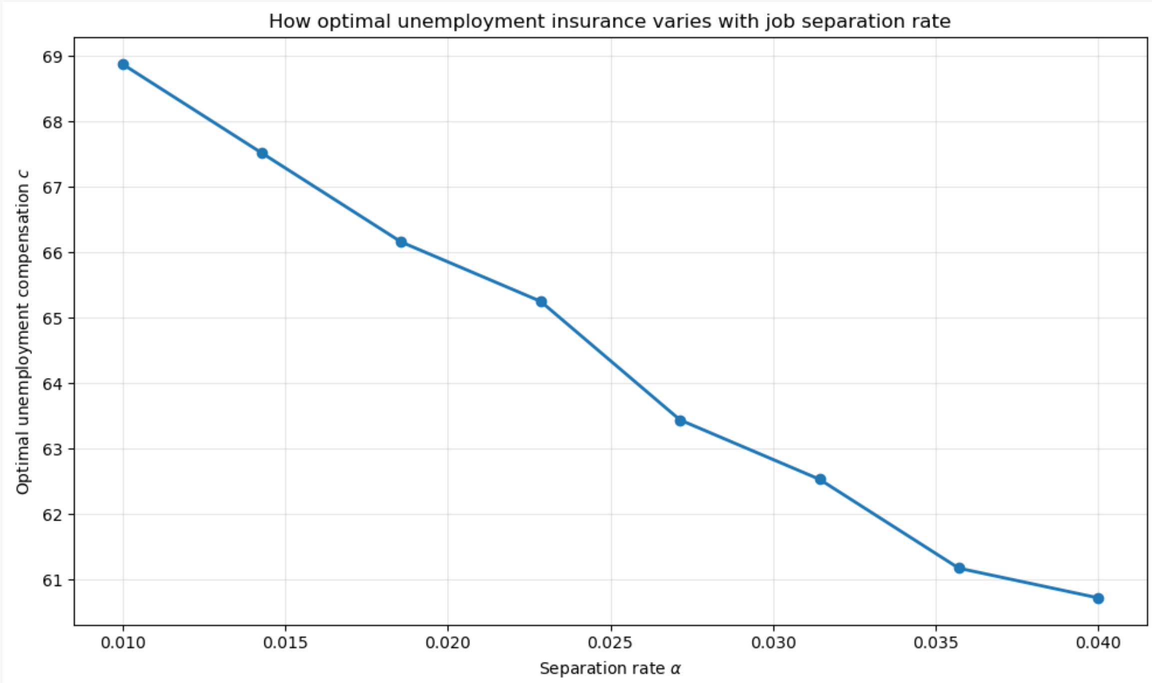
Compute and plot the optimal c (the value that maximizes welfare) for a range of separation rates α from 0.01 to 0.04.

For each α value, find the optimal c by computing welfare across the range of c values and selecting the maximum.

Solution

Here is one solution:

```python
# Range of separation rates to explore (wider range, fewer points)
α_values = jnp.linspace(0.01, 0.04, 8)

# We'll store the optimal c for each α
optimal_c_values = []

# Use a finer grid for c values to get better resolution
c_vec_fine = jnp.linspace(5, 140, 150)

for α_val in α_values:
    # Create economy parameters with this α
    params_α = create_economy(α=α_val)

    # Create wage distribution
    w_vec_α, p_vec_α = create_wage_distribution(
        params_α.max_wage, params_α.wage_grid_size, params_α.log_wage_mean
    )

    # Compute welfare for each c value
    welfare_values = []
    for c in c_vec_fine:
        t = find_balanced_budget_tax(c, params_α, w_vec_α, p_vec_α)
        e_rate, u_rate, welfare = compute_steady_state_quantities(
            c, t, params_α, w_vec_α, p_vec_α
        )
        welfare_values.append(welfare)

    # The welfare function is very flat near its maximum.
    # Using argmax on a single point can be unstable due to numerical noise.
    # Instead, we find all c values within 99.9% of maximum welfare and
    # compute their weighted average (centroid). This gives a more stable
    # estimate of the optimal unemployment compensation level.
    welfare_array = jnp.array(welfare_values)
    max_welfare = jnp.max(welfare_array)
    threshold = 0.999 * max_welfare
    near_optimal_mask = welfare_array >= threshold

    # Compute weighted average of c values in the near-optimal region
    optimal_c = jnp.sum(c_vec_fine * near_optimal_mask * welfare_array) / \
                jnp.sum(near_optimal_mask * welfare_array)
    optimal_c_values.append(optimal_c)

# Plot the relationship
fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(α_values, optimal_c_values, lw=2, marker='o')
ax.set_xlabel(r'Separation rate $\alpha$')
ax.set_ylabel('Optimal unemployment compensation $c$')
ax.set_title('How optimal unemployment insurance varies with job separation rate')
ax.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```



We see that as the separation rate increases (workers lose their jobs more frequently), the welfare-maximizing level of unemployment compensation decreases.

This occurs because higher separation rates increase steady-state unemployment, which raises the tax burden needed to finance unemployment benefits. The optimal policy balances insurance against distortionary taxation.

CHAPTER

SEVENTYSEVEN

RATIONAL EXPECTATIONS EQUILIBRIUM

Contents

> "If you're so smart, why aren't you rich?"

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
```

77.1 Overview

This lecture introduces the concept of a *rational expectations equilibrium*.

To illustrate it, we describe a linear quadratic version of a model due to Lucas and Prescott [Lucas and Prescott, 1971].

That 1971 paper is one of a small number of research articles that ignited a *rational expectations revolution*.

We follow Lucas and Prescott by employing a setting that is readily "Bellmanized" (i.e., susceptible to being formulated as a dynamic programming problems.

Because we use linear quadratic setups for demand and costs, we can deploy the LQ programming techniques described in *this lecture*.

We will learn about how a representative agent's problem differs from a planner's, and how a planning problem can be used to compute quantities and prices in a rational expectations equilibrium.

We will also learn about how a rational expectations equilibrium can be characterized as a fixed point of a mapping from a *perceived law of motion* to an *actual law of motion*.

Equality between a perceived and an actual law of motion for endogenous market-wide objects captures in a nutshell what the rational expectations equilibrium concept is all about.

Finally, we will learn about the important "Big K, little k" trick, a modeling device widely used in macroeconomics.

Except that for us

- Instead of "Big K" it will be "Big Y".
- Instead of "little k" it will be "little y".

Let's start with some standard imports:

```
import matplotlib.pyplot as plt
import numpy as np
```

We'll also use the LQ class from `QuantEcon.py`.

```
from quantecon import LQ
```

77.1.1 The Big Y, little y Trick

This widely used method applies in contexts in which a **representative firm** or agent is a "price taker" operating within a competitive equilibrium.

The following setting justifies the concept of a representative firm that stands in for a large number of other firms too.

There is a uniform unit measure of identical firms named $\omega \in \Omega = [0, 1]$.

The output of firm ω is $y(\omega)$.

The output of all firms is $Y = \int_0^1 y(\omega) d\,\omega$.

All firms end up choosing to produce the same output, so that at the end of the day $y(\omega) = y$ and $Y = y = \int_0^1 y(\omega) d\,\omega$.

This setting allows us to speak of a representative firm that chooses to produce y.

We want to impose that

- The representative firm or individual firm takes *aggregate* Y as given when it chooses individual $y(\omega)$, but ….
- At the end of the day, $Y = y(\omega) = y$, so that the representative firm is indeed representative.

The Big Y, little y trick accomplishes these two goals by

- Taking Y as beyond control when posing the choice problem of who chooses y; but ….
- Imposing $Y = y$ *after* having solved the individual's optimization problem.

Please watch for how this strategy is applied as the lecture unfolds.

We begin by applying the Big Y, little y trick in a very simple static context.

A Simple Static Example of the Big Y, little y Trick

Consider a static model in which a unit measure of firms produce a homogeneous good that is sold in a competitive market.

Each of these firms ends up producing and selling output $y(\omega) = y$.

The price p of the good lies on an inverse demand curve

$$p = a_0 - a_1 Y \tag{77.1}$$

where

- $a_i > 0$ for $i = 0, 1$
- $Y = \int_0^1 y(\omega) d\omega$ is the market-wide level of output

For convenience, we'll often just write y instead of $y(\omega)$ when we are describing the choice problem of an individual firm $\omega \in \Omega$.

Each firm has a total cost function

$$c(y) = c_1 y + 0.5 c_2 y^2, \qquad c_i > 0 \text{ for } i = 1, 2$$

The profits of a representative firm are $py - c(y)$.

Using (77.1), we can express the problem of the representative firm as

$$\max_y \Big[(a_0 - a_1 Y) y - c_1 y - 0.5 c_2 y^2 \Big] \tag{77.2}$$

In posing problem (77.2), we want the firm to be a *price taker*.

We do that by regarding p and therefore Y as exogenous to the firm.

The essence of the Big Y, little y trick is *not* to set $Y = ny$ *before* taking the first-order condition with respect to y in problem (77.2).

This assures that the firm is a price taker.

The first-order condition for problem (77.2) is

$$a_0 - a_1 Y - c_1 - c_2 y = 0 \tag{77.3}$$

At this point, *but not before*, we substitute $Y = y$ into (77.3) to obtain the following linear equation

$$a_0 - c_1 - (a_1 + c_2) Y = 0 \tag{77.4}$$

to be solved for the competitive equilibrium market-wide output Y.

After solving for Y, we can compute the competitive equilibrium price p from the inverse demand curve (77.1).

77.1.2 Related Planning Problem

Define **consumer surplus** as the area under the inverse demand curve:

$$S_c(Y) = \int_0^Y (a_0 - a_1 s) ds = a_o Y - \frac{a_1}{2} Y^2.$$

Define the social cost of production as

$$S_p(Y) = c_1 Y + \frac{c_2}{2} Y^2$$

Consider the planning problem

$$\max_Y [S_c(Y) - S_p(Y)]$$

The first-order necessary condition for the planning problem is equation (77.4).

Thus, a Y that solves (77.4) is a competitive equilibrium output as well as an output that solves the planning problem.

This type of outcome provides an intellectual justification for liking a competitive equilibrium.

77.1.3 Further Reading

References for this lecture include

- [Lucas and Prescott, 1971]
- [Sargent, 1987], chapter XIV
- [Ljungqvist and Sargent, 2018], chapter 7

77.2 Rational Expectations Equilibrium

Our first illustration of a rational expectations equilibrium involves a market with a unit measure of identical firms, each of which seeks to maximize the discounted present value of profits in the face of adjustment costs.

The adjustment costs induce the firms to make gradual adjustments, which in turn requires consideration of future prices.

Individual firms understand that, via the inverse demand curve, the price is determined by the amounts supplied by other firms.

Hence each firm wants to forecast future total industry output.

In our context, a forecast is generated by a belief about the law of motion for the aggregate state.

Rational expectations equilibrium prevails when this belief coincides with the actual law of motion generated by production choices induced by this belief.

We formulate a rational expectations equilibrium in terms of a fixed point of an operator that maps beliefs into optimal beliefs.

77.2.1 Competitive Equilibrium with Adjustment Costs

To illustrate, consider a collection of n firms producing a homogeneous good that is sold in a competitive market.

Each firm sell output $y_t(\omega) = y_t$.

The price p_t of the good lies on the inverse demand curve

$$p_t = a_0 - a_1 Y_t \tag{77.5}$$

where

- $a_i > 0$ for $i = 0, 1$
- $Y_t = \int_0^1 y_t(\omega) d\omega = y_t$ is the market-wide level of output

The Firm's Problem

Each firm is a price taker.

While it faces no uncertainty, it does face adjustment costs

In particular, it chooses a production plan to maximize

$$\sum_{t=0}^{\infty} \beta^t r_t \tag{77.6}$$

where

$$r_t := p_t y_t - \frac{\gamma(y_{t+1} - y_t)^2}{2}, \qquad y_0 \text{ given} \tag{77.7}$$

Regarding the parameters,

- $\beta \in (0, 1)$ is a discount factor
- $\gamma > 0$ measures the cost of adjusting the rate of output

Regarding timing, the firm observes p_t and y_t when it chooses y_{t+1} at time t.

To state the firm's optimization problem completely requires that we specify dynamics for all state variables.

This includes ones that the firm cares about but does not control like p_t.

We turn to this problem now.

Prices and Aggregate Output

In view of (77.5), the firm's incentive to forecast the market price translates into an incentive to forecast aggregate output Y_t.

Aggregate output depends on the choices of other firms.

The output $y_t(\omega)$ of a single firm ω has a negligible effect on aggregate output $\int_0^1 y_t(\omega) d\omega$.

That justifies firms in regarding their forecasts of aggregate output as being unaffected by their own output decisions.

Representative Firm's Beliefs

We suppose the firm believes that market-wide output Y_t follows the law of motion

$$Y_{t+1} = H(Y_t) \tag{77.8}$$

where Y_0 is a known initial condition.

The **belief function** H is an equilibrium object, and hence remains to be determined.

Optimal Behavior Given Beliefs

For now, let's fix a particular belief H in (77.8) and investigate the firm's response to it.

Let v be the optimal value function for the firm's problem given H.

The value function satisfies the Bellman equation

$$v(y, Y) = \max_{y'} \left\{ a_0 y - a_1 y Y - \frac{\gamma(y' - y)^2}{2} + \beta v(y', H(Y)) \right\} \tag{77.9}$$

Let's denote the firm's optimal policy function by h, so that

$$y_{t+1} = h(y_t, Y_t) \tag{77.10}$$

where

$$h(y, Y) := \operatorname{argmax}_{y'} \left\{ a_0 y - a_1 y Y - \frac{\gamma(y' - y)^2}{2} + \beta v(y', H(Y)) \right\} \tag{77.11}$$

Evidently v and h both depend on H.

Characterization with First-Order Necessary Conditions

In what follows it will be helpful to have a second characterization of h, based on first-order conditions.

The first-order necessary condition for choosing y' is

$$-\gamma(y' - y) + \beta v_y(y', H(Y)) = 0 \tag{77.12}$$

An important useful envelope result of Benveniste-Scheinkman [Benveniste and Scheinkman, 1979] implies that to differentiate v with respect to y we can naively differentiate the right side of (77.9), giving

$$v_y(y, Y) = a_0 - a_1 Y + \gamma(y' - y)$$

Substituting this equation into (77.12) gives the **Euler equation**

$$-\gamma(y_{t+1} - y_t) + \beta[a_0 - a_1 Y_{t+1} + \gamma(y_{t+2} - y_{t+1})] = 0 \tag{77.13}$$

The firm optimally sets an output path that satisfies (77.13), taking (77.8) as given, and subject to

- the initial conditions for (y_0, Y_0).
- the terminal condition $\lim_{t \to \infty} \beta^t y_t v_y(y_t, Y_t) = 0$.

This last condition is called the **transversality condition**, and acts as a first-order necessary condition "at infinity".

A representative firm's decision rule solves the difference equation (77.13) subject to the given initial condition y_0 and the transversality condition.

Note that solving the Bellman equation (77.9) for v and then h in (77.11) yields a decision rule that automatically imposes both the Euler equation (77.13) and the transversality condition.

The Actual Law of Motion for Output

As we've seen, a given belief translates into a particular decision rule h.

Recalling that in equilbrium $Y_t = y_t$, the **actual law of motion** for market-wide output is then

$$Y_{t+1} = h(Y_t, Y_t) \tag{77.14}$$

Thus, when firms believe that the law of motion for market-wide output is (77.8), their optimizing behavior makes the actual law of motion be (77.14).

77.2.2 Definition of Rational Expectations Equilibrium

A **rational expectations equilibrium** or **recursive competitive equilibrium** of the model with adjustment costs is a decision rule h and an aggregate law of motion H such that

1. Given belief H, the map h is the firm's optimal policy function.
2. The law of motion H satisfies $H(Y) = h(Y, Y)$ for all Y.

Thus, a rational expectations equilibrium equates the perceived and actual laws of motion (77.8) and (77.14).

Fixed Point Characterization

As we've seen, the firm's optimum problem induces a mapping Φ from a perceived law of motion H for market-wide output to an actual law of motion $\Phi(H)$.

The mapping Φ is the composition of two mappings, the first of which maps a perceived law of motion into a decision rule via (77.9)–(77.11), the second of which maps a decision rule into an actual law via (77.14).

The H component of a rational expectations equilibrium is a fixed point of Φ.

77.3 Computing an Equilibrium

Now let's compute a rational expectations equilibrium.

77.3.1 Failure of Contractivity

Readers accustomed to dynamic programming arguments might try to address this problem by choosing some guess H_0 for the aggregate law of motion and then iterating with Φ.

Unfortunately, the mapping Φ is not a contraction.

Indeed, there is no guarantee that direct iterations on Φ converge[1].

There are examples in which these iterations diverge.

Fortunately, another method works here.

The method exploits a connection between equilibrium and Pareto optimality expressed in the fundamental theorems of welfare economics (see, e.g, [Mas-Colell *et al.*, 1995]).

Lucas and Prescott [Lucas and Prescott, 1971] used this method to construct a rational expectations equilibrium.

Some details follow.

77.3.2 A Planning Problem Approach

Our plan of attack is to match the Euler equations of the market problem with those for a single-agent choice problem.

As we'll see, this planning problem can be solved by LQ control (*linear regulator*).

Optimal quantities from the planning problem are rational expectations equilibrium quantities.

The rational expectations equilibrium price can be obtained as a shadow price in the planning problem.

We first compute a sum of consumer and producer surplus at time t

$$s(Y_t, Y_{t+1}) := \int_0^{Y_t} (a_0 - a_1 x)\, dx - \frac{\gamma(Y_{t+1} - Y_t)^2}{2} \tag{77.15}$$

The first term is the area under the demand curve, while the second measures the social costs of changing output.

[1] A literature that studies whether models populated with agents who learn can converge to rational expectations equilibria features iterations on a modification of the mapping Φ that can be approximated as $\gamma\Phi + (1-\gamma)I$. Here I is the identity operator and $\gamma \in (0,1)$ is a *relaxation parameter*. See [Marcet and Sargent, 1989] and [Evans and Honkapohja, 2001] for statements and applications of this approach to establish conditions under which collections of adaptive agents who use least squares learning to converge to a rational expectations equilibrium.

The **planning problem** is to choose a production plan $\{Y_t\}$ to maximize

$$\sum_{t=0}^{\infty} \beta^t s(Y_t, Y_{t+1})$$

subject to an initial condition for Y_0.

77.3.3 Solution of Planning Problem

Evaluating the integral in (77.15) yields the quadratic form $a_0 Y_t - a_1 Y_t^2 / 2$.

As a result, the Bellman equation for the planning problem is

$$V(Y) = \max_{Y'} \left\{ a_0 Y - \frac{a_1}{2} Y^2 - \frac{\gamma (Y' - Y)^2}{2} + \beta V(Y') \right\} \tag{77.16}$$

The associated first-order condition is

$$-\gamma (Y' - Y) + \beta V'(Y') = 0 \tag{77.17}$$

Applying the same Benveniste-Scheinkman formula gives

$$V'(Y) = a_0 - a_1 Y + \gamma (Y' - Y)$$

Substituting this into equation (77.17) and rearranging leads to the Euler equation

$$\beta a_0 + \gamma Y_t - [\beta a_1 + \gamma (1 + \beta)] Y_{t+1} + \gamma \beta Y_{t+2} = 0 \tag{77.18}$$

77.3.4 Key Insight

Return to equation (77.13) and set $y_t = Y_t$ for all t.

A small amount of algebra will convince you that when $y_t = Y_t$, equations (77.18) and (77.13) are identical.

Thus, the Euler equation for the planning problem matches the second-order difference equation that we derived by

1. finding the Euler equation of the representative firm and
2. substituting into it the expression $Y_t = y_t$ that “makes the representative firm be representative”.

If it is appropriate to apply the same terminal conditions for these two difference equations, which it is, then we have verified that a solution of the planning problem is also a rational expectations equilibrium quantity sequence.

It follows that for this example we can compute equilibrium quantities by forming the optimal linear regulator problem corresponding to the Bellman equation (77.16).

The optimal policy function for the planning problem is the aggregate law of motion H that the representative firm faces within a rational expectations equilibrium.

Structure of the Law of Motion

As you are asked to show in the exercises, the fact that the planner’s problem is an LQ control problem implies an optimal policy — and hence aggregate law of motion — taking the form

$$Y_{t+1} = \kappa_0 + \kappa_1 Y_t \tag{77.19}$$

for some parameter pair κ_0, κ_1.

Now that we know the aggregate law of motion is linear, we can see from the firm's Bellman equation (77.9) that the firm's problem can also be framed as an LQ problem.

As you're asked to show in the exercises, the LQ formulation of the firm's problem implies a law of motion that looks as follows

$$y_{t+1} = h_0 + h_1 y_t + h_2 Y_t \tag{77.20}$$

Hence a rational expectations equilibrium will be defined by the parameters $(\kappa_0, \kappa_1, h_0, h_1, h_2)$ in (77.19)–(77.20).

77.4 Exercises

Exercise 77.4.1

Consider the firm problem *described above*.

Let the firm's belief function H be as given in (77.19).

Formulate the firm's problem as a discounted optimal linear regulator problem, being careful to describe all of the objects needed.

Use the class `LQ` from the QuantEcon.py package to solve the firm's problem for the following parameter values:

$$a_0 = 100, a_1 = 0.05, \beta = 0.95, \gamma = 10, \kappa_0 = 95.5, \kappa_1 = 0.95$$

Express the solution of the firm's problem in the form (77.20) and give the values for each h_j.

If there were a unit measure of identical competitive firms all behaving according to (77.20), what would (77.20) imply for the *actual* law of motion (77.8) for market supply.

Solution

To map a problem into a discounted optimal linear control problem, we need to define

- state vector x_t and control vector u_t
- matrices A, B, Q, R that define preferences and the law of motion for the state

For the state and control vectors, we choose

$$x_t = \begin{bmatrix} y_t \\ Y_t \\ 1 \end{bmatrix}, \qquad u_t = y_{t+1} - y_t$$

For B, Q, R we set

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \kappa_1 & \kappa_0 \\ 0 & 0 & 1 \end{bmatrix}, \quad B = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad R = \begin{bmatrix} 0 & a_1/2 & -a_0/2 \\ a_1/2 & 0 & 0 \\ -a_0/2 & 0 & 0 \end{bmatrix}, \quad Q = \gamma/2$$

By multiplying out you can confirm that

- $x_t' R x_t + u_t' Q u_t = -r_t$
- $x_{t+1} = A x_t + B u_t$

We'll use the module lqcontrol.py to solve the firm's problem at the stated parameter values.

This will return an LQ policy F with the interpretation $u_t = -Fx_t$, or

$$y_{t+1} - y_t = -F_0 y_t - F_1 Y_t - F_2$$

Matching parameters with $y_{t+1} = h_0 + h_1 y_t + h_2 Y_t$ leads to

$$h_0 = -F_2, \quad h_1 = 1 - F_0, \quad h_2 = -F_1$$

Here's our solution

```
# Model parameters

a0 = 100
a1 = 0.05
β = 0.95
γ = 10.0

# Beliefs

κ0 = 95.5
κ1 = 0.95

# Formulate the LQ problem

A = np.array([[1, 0, 0], [0, κ1, κ0], [0, 0, 1]])
B = np.array([1, 0, 0])
B.shape = 3, 1
R = np.array([[0, a1/2, -a0/2], [a1/2, 0, 0], [-a0/2, 0, 0]])
Q = 0.5 * γ

# Solve for the optimal policy

lq = LQ(Q, R, A, B, beta=β)
P, F, d = lq.stationary_values()
F = F.flatten()
out1 = f"F = [{F[0]:.3f}, {F[1]:.3f}, {F[2]:.3f}]"
h0, h1, h2 = -F[2], 1 - F[0], -F[1]
out2 = f"(h0, h1, h2) = ({h0:.3f}, {h1:.3f}, {h2:.3f})"

print(out1)
print(out2)
```

```
F = [-0.000, 0.046, -96.949]
(h0, h1, h2) = (96.949, 1.000, -0.046)
```

The implication is that

$$y_{t+1} = 96.949 + y_t - 0.046\, Y_t$$

For the case $n > 1$, recall that $Y_t = n y_t$, which, combined with the previous equation, yields

$$Y_{t+1} = n\,(96.949 + y_t - 0.046\, Y_t) = n 96.949 + (1 - n 0.046) Y_t$$

Exercise 77.4.2

Consider the following κ_0, κ_1 pairs as candidates for the aggregate law of motion component of a rational expectations equilibrium (see (77.19)).

Extending the program that you wrote for Exercise 77.4.1, determine which if any satisfy *the definition* of a rational expectations equilibrium

- (94.0886298678, 0.923409232937)
- (93.2119845412, 0.984323478873)
- (95.0818452486, 0.952459076301)

Describe an iterative algorithm that uses the program that you wrote for Exercise 77.4.1 to compute a rational expectations equilibrium.

(You are not being asked actually to use the algorithm you are suggesting)

Solution

To determine whether a κ_0, κ_1 pair forms the aggregate law of motion component of a rational expectations equilibrium, we can proceed as follows:

- Determine the corresponding firm law of motion $y_{t+1} = h_0 + h_1 y_t + h_2 Y_t$.
- Test whether the associated aggregate law :$Y_{t+1} = nh(Y_t/n, Y_t)$ evaluates to $Y_{t+1} = \kappa_0 + \kappa_1 Y_t$.

In the second step, we can use $Y_t = ny_t = y_t$, so that $Y_{t+1} = nh(Y_t/n, Y_t)$ becomes

$$Y_{t+1} = h(Y_t, Y_t) = h_0 + (h_1 + h_2)Y_t$$

Hence to test the second step we can test $\kappa_0 = h_0$ and $\kappa_1 = h_1 + h_2$.

The following code implements this test

```
candidates = ((94.0886298678, 0.923409232937),
              (93.2119845412, 0.984323478873),
              (95.0818452486, 0.952459076301))

for κ0, κ1 in candidates:

    # Form the associated law of motion
    A = np.array([[1, 0, 0], [0, κ1, κ0], [0, 0, 1]])

    # Solve the LQ problem for the firm
    lq = LQ(Q, R, A, B, beta=β)
    P, F, d = lq.stationary_values()
    F = F.flatten()
    h0, h1, h2 = -F[2], 1 - F[0], -F[1]

    # Test the equilibrium condition
    if np.allclose((κ0, κ1), (h0, h1 + h2)):
        print(f'Equilibrium pair = {κ0}, {κ1}')
        print('f(h0, h1, h2) = {h0}, {h1}, {h2}')
        break
```

```
Equilibrium pair = 95.0818452486, 0.952459076301
f(h0, h1, h2) = {h0}, {h1}, {h2}
```

The output tells us that the answer is pair (iii), which implies $(h_0, h_1, h_2) = (95.0819, 1.0000, -.0475)$.

(Notice we use `np.allclose` to test equality of floating-point numbers, since exact equality is too strict).

Regarding the iterative algorithm, one could loop from a given (κ_0, κ_1) pair to the associated firm law and then to a new (κ_0, κ_1) pair.

This amounts to implementing the operator Φ described in the lecture.

(There is in general no guarantee that this iterative process will converge to a rational expectations equilibrium)

Exercise 77.4.3

Recall the planner's problem *described above*

1. Formulate the planner's problem as an LQ problem.
2. Solve it using the same parameter values in exercise 1
   - $a_0 = 100, a_1 = 0.05, \beta = 0.95, \gamma = 10$
3. Represent the solution in the form $Y_{t+1} = \kappa_0 + \kappa_1 Y_t$.
4. Compare your answer with the results from exercise 2.

Solution

We are asked to write the planner problem as an LQ problem.

For the state and control vectors, we choose

$$x_t = \begin{bmatrix} Y_t \\ 1 \end{bmatrix}, \quad u_t = Y_{t+1} - Y_t$$

For the LQ matrices, we set

$$A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad B = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad R = \begin{bmatrix} a_1/2 & -a_0/2 \\ -a_0/2 & 0 \end{bmatrix}, \quad Q = \gamma/2$$

By multiplying out you can confirm that

- $x_t' R x_t + u_t' Q u_t = -s(Y_t, Y_{t+1})$
- $x_{t+1} = A x_t + B u_t$

By obtaining the optimal policy and using $u_t = -F x_t$ or

$$Y_{t+1} - Y_t = -F_0 Y_t - F_1$$

we can obtain the implied aggregate law of motion via $\kappa_0 = -F_1$ and $\kappa_1 = 1 - F_0$.

The Python code to solve this problem is below:

```
# Formulate the planner's LQ problem

A = np.array([[1, 0], [0, 1]])
B = np.array([[1], [0]])
R = np.array([[a1 / 2, -a0 / 2], [-a0 / 2, 0]])
Q = γ / 2

# Solve for the optimal policy

lq = LQ(Q, R, A, B, beta=β)
P, F, d = lq.stationary_values()

# Print the results

F = F.flatten()
κ0, κ1 = -F[1], 1 - F[0]
```

```
95.08187459214764 0.952459062703923
```

The output yields the same (κ_0, κ_1) pair obtained as an equilibrium from the previous exercise.

Exercise 77.4.4

A monopolist faces the industry demand curve (77.5) and chooses $\{Y_t\}$ to maximize $\sum_{t=0}^{\infty} \beta^t r_t$ where

$$r_t = p_t Y_t - \frac{\gamma(Y_{t+1} - Y_t)^2}{2}$$

Formulate this problem as an LQ problem.

Compute the optimal policy using the same parameters as Exercise 77.4.2.

In particular, solve for the parameters in

$$Y_{t+1} = m_0 + m_1 Y_t$$

Compare your results with Exercise 77.4.2 – comment.

Solution

The monopolist's LQ problem is almost identical to the planner's problem from the previous exercise, except that

$$R = \begin{bmatrix} a_1 & -a_0/2 \\ -a_0/2 & 0 \end{bmatrix}$$

The problem can be solved as follows

```
A = np.array([[1, 0], [0, 1]])
B = np.array([[1], [0]])
R = np.array([[a1, -a0 / 2], [-a0 / 2, 0]])
Q = γ / 2

lq = LQ(Q, R, A, B, beta=β)
P, F, d = lq.stationary_values()

F = F.flatten()
m0, m1 = -F[1], 1 - F[0]
print(m0, m1)
```

```
73.4729440350284 0.9265270559649701
```

We see that the law of motion for the monopolist is approximately $Y_{t+1} = 73.4729 + 0.9265 Y_t$.

In the rational expectations case, the law of motion was approximately $Y_{t+1} = 95.0818 + 0.9525 Y_t$.

One way to compare these two laws of motion is by their fixed points, which give long-run equilibrium output in each case.

For laws of the form $Y_{t+1} = c_0 + c_1 Y_t$, the fixed point is $c_0/(1 - c_1)$.

If you crunch the numbers, you will see that the monopolist adopts a lower long-run quantity than obtained by the competitive market, implying a higher market price.

This is analogous to the elementary static-case results

CHAPTER

SEVENTYEIGHT

STABILITY IN LINEAR RATIONAL EXPECTATIONS MODELS

Contents

In addition to what's in Anaconda, this lecture deploys the following libraries:

```
!pip install quantecon
```

```
import matplotlib.pyplot as plt
import numpy as np
import quantecon as qe
from sympy import init_printing, symbols, Matrix
init_printing()
```

78.1 Overview

This lecture studies stability in the context of an elementary rational expectations model.

We study a rational expectations version of Philip Cagan's model [Cagan, 1956] linking the price level to the money supply.

Cagan did not use a rational expectations version of his model, but Sargent [Sargent, 1977] did.

We study a rational expectations version of this model because it is intrinsically interesting and because it has a mathematical structure that appears in virtually all linear rational expectations model, namely, that a key endogenous variable equals a mathematical expectation of a geometric sum of future values of another variable.

The model determines the price level or rate of inflation as a function of the money supply or the rate of change in the money supply.

In this lecture, we'll encounter:

- a convenient formula for the expectation of geometric sum of future values of a variable
- a way of solving an expectational difference equation by mapping it into a vector first-order difference equation and appropriately manipulating an eigen decomposition of the transition matrix in order to impose stability
- a way to use a Big K, little k argument to allow apparent feedback from endogenous to exogenous variables within a rational expectations equilibrium
- a use of eigenvector decompositions of matrices that allowed Blanchard and Khan (1981) [Blanchard and Kahn, 1980] and Whiteman (1983) [Whiteman, 1983] to solve a class of linear rational expectations models
- how to use **SymPy** to get analytical formulas for some key objects comprising a rational expectations equilibrium

Matrix decompositions employed here are described in more depth in this lecture *Lagrangian formulations*.

We formulate a version of Cagan's model under rational expectations as an **expectational difference equation** whose solution is a rational expectations equilibrium.

We'll start this lecture with a quick review of deterministic (i.e., non-random) first-order and second-order linear difference equations.

78.2 Linear Difference Equations

We'll use the **backward shift** or **lag** operator L.

The lag operator L maps a sequence $\{x_t\}_{t=0}^\infty$ into the sequence $\{x_{t-1}\}_{t=0}^\infty$

We'll deploy L by using the equality $Lx_t \equiv x_{t-1}$ in algebraic expressions.

Further, the inverse L^{-1} of the lag operator is the **forward shift** operator.

We'll often use the equality $L^{-1}x_t \equiv x_{t+1}$ below.

The algebra of lag and forward shift operators can simplify representing and solving linear difference equations.

78.2.1 First Order

We want to solve a linear first-order scalar difference equation.

Let $|\lambda| < 1$ and let $\{u_t\}_{t=-\infty}^\infty$ be a bounded sequence of scalar real numbers.

Let L be the lag operator defined by $Lx_t \equiv x_{t-1}$ and let L^{-1} be the forward shift operator defined by $L^{-1}x_t \equiv x_{t+1}$.

Then

$$(1 - \lambda L)y_t = u_t, \forall t \tag{78.1}$$

has solutions

$$y_t = (1 - \lambda L)^{-1}u_t + k\lambda^t \tag{78.2}$$

or

$$y_t = \sum_{j=0}^{\infty} \lambda^j u_{t-j} + k\lambda^t$$

for any real number k.

You can verify this fact by applying $(1 - \lambda L)$ to both sides of equation (78.2) and noting that $(1 - \lambda L)\lambda^t = 0$.

To pin down k we need one condition imposed from outside (e.g., an initial or terminal condition) on the path of y.

Now let $|\lambda| > 1$.

Rewrite equation (78.1) as

$$y_{t-1} = \lambda^{-1} y_t - \lambda^{-1} u_t, \forall t \tag{78.3}$$

or

$$(1 - \lambda^{-1} L^{-1}) y_t = -\lambda^{-1} u_{t+1}. \tag{78.4}$$

A solution is

$$y_t = -\lambda^{-1} \left(\frac{1}{1 - \lambda^{-1} L^{-1}} \right) u_{t+1} + k\lambda^t \tag{78.5}$$

for any k.

To verify that this is a solution, check the consequences of operating on both sides of equation (78.5) by $(1 - \lambda L)$ and compare to equation (78.1).

For any bounded $\{u_t\}$ sequence, solution (78.2) exists for $|\lambda| < 1$ because the **distributed lag** in u converges.

Solution (78.5) exists when $|\lambda| > 1$ because the **distributed lead** in u converges.

When $|\lambda| > 1$, the distributed lag in u in (78.2) may diverge, in which case a solution of this form does not exist.

The distributed lead in u in (78.5) need not converge when $|\lambda| < 1$.

78.2.2 Second Order

Now consider the second order difference equation

$$(1 - \lambda_1 L)(1 - \lambda_2 L) y_{t+1} = u_t \tag{78.6}$$

where $\{u_t\}$ is a bounded sequence, y_0 is an initial condition, $|\lambda_1| < 1$ and $|\lambda_2| > 1$.

We seek a bounded sequence $\{y_t\}_{t=0}^{\infty}$ that satisfies (78.6). Using insights from our analysis of the first-order equation, operate on both sides of (78.6) by the forward inverse of $(1 - \lambda_2 L)$ to rewrite equation (78.6) as

$$(1 - \lambda_1 L) y_{t+1} = -\frac{\lambda_2^{-1}}{1 - \lambda_2^{-1} L^{-1}} u_{t+1}$$

or

$$y_{t+1} = \lambda_1 y_t - \lambda_2^{-1} \sum_{j=0}^{\infty} \lambda_2^{-j} u_{t+j+1}. \tag{78.7}$$

Thus, we obtained equation (78.7) by solving a stable root (in this case λ_1) **backward**, and an unstable root (in this case λ_2) **forward**.

Equation (78.7) has a form that we shall encounter often.

- $\lambda_1 y_t$ is called the **feedback part**
- $-\frac{\lambda_2^{-1}}{1-\lambda_2^{-1} L^{-1}} u_{t+1}$ is called the **feedforward part**

78.3 Illustration: Cagan's Model

Now let's use linear difference equations to represent and solve Sargent's [Sargent, 1977] rational expectations version of Cagan's model [Cagan, 1956] that connects the price level to the public's anticipations of future money supplies.

Cagan did not use a rational expectations version of his model, but Sargent [Sargent, 1977]

Let

- m_t^d be the log of the demand for money
- m_t be the log of the supply of money
- p_t be the log of the price level

It follows that $p_{t+1} - p_t$ is the rate of inflation.

The logarithm of the demand for real money balances $m_t^d - p_t$ is an inverse function of the expected rate of inflation $p_{t+1} - p_t$ for $t \geq 0$:

$$m_t^d - p_t = -\beta(p_{t+1} - p_t), \quad \beta > 0$$

Equate the demand for log money m_t^d to the supply of log money m_t in the above equation and rearrange to deduce that the logarithm of the price level p_t is related to the logarithm of the money supply m_t by

$$p_t = (1 - \lambda)m_t + \lambda p_{t+1} \tag{78.8}$$

where $\lambda \equiv \frac{\beta}{1+\beta} \in (0, 1)$.

(We note that the characteristic polynomial if $1 - \lambda^{-1}z^{-1} = 0$ so that the zero of the characteristic polynomial in this case is $\lambda \in (0, 1)$ which here is **inside** the unit circle.)

Solving the first order difference equation (78.8) forward gives

$$p_t = (1 - \lambda) \sum_{j=0}^{\infty} \lambda^j m_{t+j}, \tag{78.9}$$

which is the unique **stable** solution of difference equation (78.8) among a class of more general solutions

$$p_t = (1 - \lambda) \sum_{j=0}^{\infty} \lambda^j m_{t+j} + c\lambda^{-t} \tag{78.10}$$

that is indexed by the real number $c \in \mathbf{R}$.

Because we want to focus on stable solutions, we set $c = 0$.

Equation (78.10) attributes **perfect foresight** about the money supply sequence to the holders of real balances.

We begin by assuming that the log of the money supply is **exogenous** in the sense that it is an autonomous process that does not feed back on the log of the price level.

In particular, we assume that the log of the money supply is described by the linear state space system

$$\begin{aligned} m_t &= Gx_t \\ x_{t+1} &= Ax_t \end{aligned} \tag{78.11}$$

where x_t is an $n \times 1$ vector that does not include p_t or lags of p_t, A is an $n \times n$ matrix with eigenvalues that are less than λ^{-1} in absolute values, and G is a $1 \times n$ selector matrix.

Variables appearing in the vector x_t contain information that might help predict future values of the money supply.

We'll start with an example in which x_t includes only m_t, possibly lagged values of m, and a constant.

An example of such an $\{m_t\}$ process that fits info state space system (78.11) is one that satisfies the second order linear difference equation

$$m_{t+1} = \alpha + \rho_1 m_t + \rho_2 m_{t-1}$$

where the zeros of the characteristic polynomial $(1 - \rho_1 z - \rho_2 z^2)$ are strictly greater than 1 in modulus.

(Please see *this* QuantEcon lecture for more about characteristic polynomials and their role in solving linear difference equations.)

We seek a stable or non-explosive solution of the difference equation (78.8) that obeys the system comprised of (78.8)-(78.11).

By stable or non-explosive, we mean that neither m_t nor p_t diverges as $t \to +\infty$.

This requires that we shut down the term $c\lambda^{-t}$ in equation (78.10) above by setting $c = 0$

The solution we are after is

$$p_t = F x_t \tag{78.12}$$

where

$$F = (1 - \lambda) G (I - \lambda A)^{-1} \tag{78.13}$$

Note

As mentioned above, an **explosive solution** of difference equation (78.8) can be constructed by adding to the right hand of (78.12) a sequence $c\lambda^{-t}$ where c is an arbitrary positive constant.

78.4 Some Python Code

We'll construct examples that illustrate (78.11).

Our first example takes as the law of motion for the log money supply the second order difference equation

$$m_{t+1} = \alpha + \rho_1 m_t + \rho_2 m_{t-1} \tag{78.14}$$

that is parameterized by ρ_1, ρ_2, α

To capture this parameterization with system (78.9) we set

$$x_t = \begin{bmatrix} 1 \\ m_t \\ m_{t-1} \end{bmatrix}, \quad A = \begin{bmatrix} 1 & 0 & 0 \\ \alpha & \rho_1 & \rho_2 \\ 0 & 1 & 0 \end{bmatrix}, \quad G = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}$$

Here is Python code

```
λ = .9

α = 0
ρ1 = .9
ρ2 = .05

A = np.array([[1,  0,  0],
              [α, ρ1, ρ2],
              [0,  1,  0]])
G = np.array([[0, 1, 0]])
```

The matrix A has one eigenvalue equal to unity.

It is associated with the A_{11} component that captures a constant component of the state x_t.

We can verify that the two eigenvalues of A not associated with the constant in the state x_t are strictly less than unity in modulus.

```
eigvals = np.linalg.eigvals(A)
print(eigvals)
```

```
[-0.05249378  0.95249378  1.        ]
```

```
(abs(eigvals) <= 1).all()
```

```
np.True_
```

Now let's compute F in formulas (78.12) and (78.13).

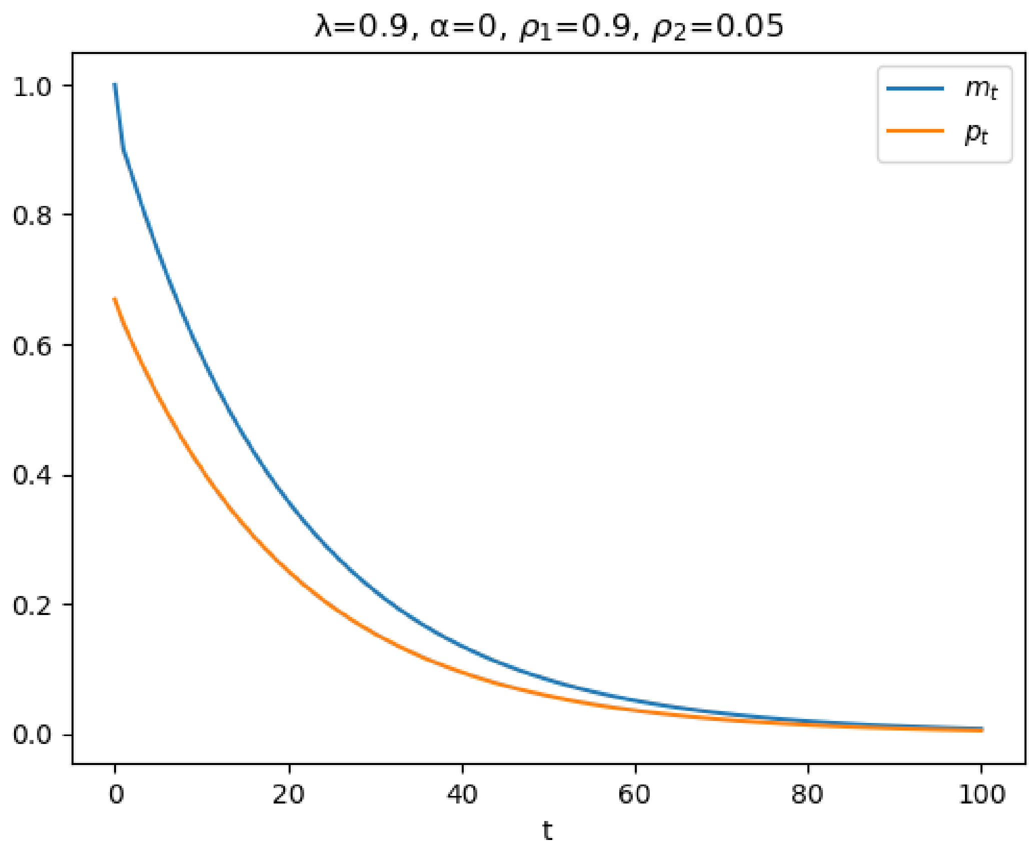
```
# compute the solution, i.e. forumula (3)
F = (1 - λ) * G @ np.linalg.inv(np.eye(A.shape[0]) - λ * A)
print("F= ",F)
```

```
F=  [[0.         0.66889632 0.03010033]]
```

Now let's simulate paths of m_t and p_t starting from an initial value x_0.

```
# set the initial state
x0 = np.array([1, 1, 0])

T = 100 # length of simulation

m_seq = np.empty(T+1)
p_seq = np.empty(T+1)

[m_seq[0]] = G @ x0
[p_seq[0]] = F @ x0

# simulate for T periods
x_old = x0
for t in range(T):

    x = A @ x_old

    [m_seq[t+1]] = G @ x
    [p_seq[t+1]] = F @ x

    x_old = x
```

```
plt.figure()
plt.plot(range(T+1), m_seq, label=r'$m_t$')
plt.plot(range(T+1), p_seq, label=r'$p_t$')
plt.xlabel('t')
plt.title(rf'λ={λ}, α={α}, $ρ_1$={ρ1}, $ρ_2$={ρ2}')
plt.legend()
plt.show()
```



In the above graph, why is the log of the price level always less than the log of the money supply?

Because

- according to equation (78.9), p_t is a geometric weighted average of current and future values of m_t, and
- it happens that in this example future m's are always less than the current m

78.5 Alternative Code

We could also have run the simulation using the quantecon **LinearStateSpace** code.

The following code block performs the calculation with that code.

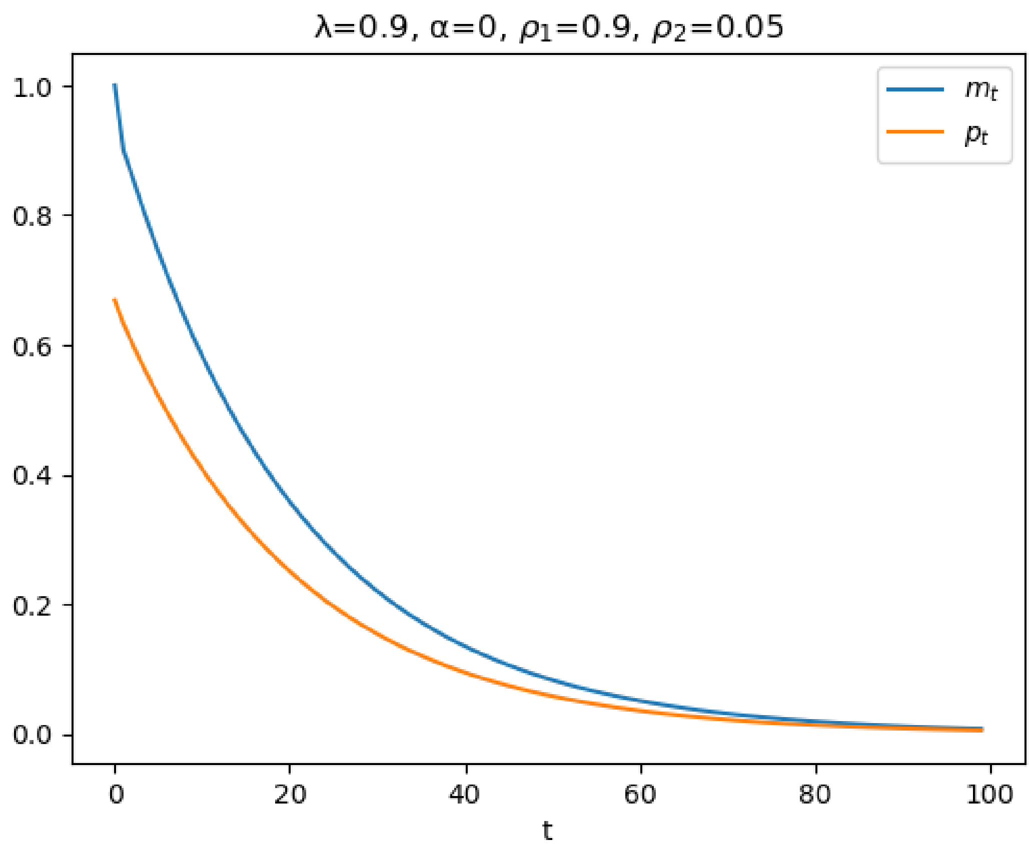
```
# construct a LinearStateSpace instance

# stack G and F
G_ext = np.vstack([G, F])

C = np.zeros((A.shape[0], 1))

ss = qe.LinearStateSpace(A, C, G_ext, mu_0=x0)
```

```
T = 100

# simulate using LinearStateSpace
```

(continues on next page)

(continued from previous page)

```
x, y = ss.simulate(ts_length=T)

# plot
plt.figure()
plt.plot(range(T), y[0,:], label='$m_t$')
plt.plot(range(T), y[1,:], label='$p_t$')
plt.xlabel('t')
plt.title(f'λ={λ}, α={α}, $ρ_1$={ρ1}, $ρ_2$={ρ2}')
plt.legend()
plt.show()
```



78.5.1 Special Case

To simplify our presentation in ways that will let focus on an important idea, in the above second-order difference equation (78.14) that governs m_t, we now set $\alpha = 0$, $\rho_1 = \rho \in (-1, 1)$, and $\rho_2 = 0$ so that the law of motion for m_t becomes

$$m_{t+1} = \rho m_t \tag{78.15}$$

and the state x_t becomes

$$x_t = m_t.$$

Consequently, we can set $G = 1$, $A = \rho$ making our formula (78.13) for F become

$$F = (1 - \lambda)(1 - \lambda\rho)^{-1}.$$

so that the log the log price level satisfies

$$p_t = F m_t.$$

Please keep these formulas in mind as we investigate an alternative route to and interpretation of our formula for F.

78.6 Another Perspective

Above, we imposed stability or non-explosiveness on the solution of the key difference equation (78.8) in Cagan's model by solving the unstable root of the characteristic polynomial forward.

To shed light on the mechanics involved in imposing stability on a solution of a potentially unstable system of linear difference equations and to prepare the way for generalizations of our model in which the money supply is allowed to feed back on the price level itself, we stack equations (78.8) and (78.15) to form the system

$$\begin{bmatrix} m_{t+1} \\ p_{t+1} \end{bmatrix} = \begin{bmatrix} \rho & 0 \\ -(1-\lambda)/\lambda & \lambda^{-1} \end{bmatrix} \begin{bmatrix} m_t \\ p_t \end{bmatrix} \tag{78.16}$$

or

$$y_{t+1} = H y_t, \quad t \geq 0 \tag{78.17}$$

where

$$H = \begin{bmatrix} \rho & 0 \\ -(1-\lambda)/\lambda & \lambda^{-1} \end{bmatrix}. \tag{78.18}$$

Transition matrix H has eigenvalues $\rho \in (0,1)$ and $\lambda^{-1} > 1$.

Because an eigenvalue of H exceeds unity, if we iterate on equation (78.17) starting from an arbitrary initial vector $y_0 = \begin{bmatrix} m_0 \\ p_0 \end{bmatrix}$ with $m_0 > 0, p_0 > 0$, we discover that in general absolute values of both components of y_t diverge toward $+\infty$ as $t \to +\infty$.

To substantiate this claim, we can use the eigenvector matrix decomposition of H that is available to us because the eigenvalues of H are distinct

$$H = Q \Lambda Q^{-1}.$$

Here Λ is a diagonal matrix of eigenvalues of H and Q is a matrix whose columns are eigenvectors associated with the corresponding eigenvalues.

Note that

$$H^t = Q \Lambda^t Q^{-1}$$

so that

$$y_t = Q \Lambda^t Q^{-1} y_0$$

For almost all initial vectors y_0, the presence of the eigenvalue $\lambda^{-1} > 1$ causes both components of y_t to diverge in absolute value to $+\infty$.

To explore this outcome in more detail, we can use the following transformation

$$y_t^* = Q^{-1} y_t$$

that allows us to represent the dynamics in a way that isolates the source of the propensity of paths to diverge:

$$y_{t+1}^* = \Lambda^t y_t^*$$

Staring at this equation indicates that unless

$$y_0^* = \begin{bmatrix} y_{1,0}^* \\ 0 \end{bmatrix} \tag{78.19}$$

the path of y_t^* and therefore the paths of both components of $y_t = Qy_t^*$ will diverge in absolute value as $t \to +\infty$. (We say that the paths *explode*)

Equation (78.19) also leads us to conclude that there is a unique setting for the initial vector y_0 for which both components of y_t do not diverge.

The required setting of y_0 must evidently have the property that

$$Qy_0 = y_0^* = \begin{bmatrix} y_{1,0}^* \\ 0 \end{bmatrix}.$$

But note that since $y_0 = \begin{bmatrix} m_0 \\ p_0 \end{bmatrix}$ and m_0 is given to us an initial condition, p_0 has to do all the adjusting to satisfy this equation.

Sometimes this situation is described by saying that while m_0 is truly a **state** variable, p_0 is a **jump** variable that must adjust at $t = 0$ in order to satisfy the equation.

Thus, in a nutshell the unique value of the vector y_0 for which the paths of y_t do not diverge must have second component p_0 that verifies equality (78.19) by setting the second component of y_0^* equal to zero.

The component p_0 of the initial vector $y_0 = \begin{bmatrix} m_0 \\ p_0 \end{bmatrix}$ must evidently satisfy

$$Q^{\{2\}} y_0 = 0$$

where $Q^{\{2\}}$ denotes the second row of Q^{-1}, a restriction that is equivalent to

$$Q^{21} m_0 + Q^{22} p_0 = 0 \tag{78.20}$$

where Q^{ij} denotes the (i, j) component of Q^{-1}.

Solving this equation for p_0, we find

$$p_0 = -(Q^{22})^{-1} Q^{21} m_0. \tag{78.21}$$

This is the unique **stabilizing value** of p_0 expressed as a function of m_0.

78.6.1 Refining the Formula

We can get an even more convenient formula for p_0 that is cast in terms of components of Q instead of components of Q^{-1}.

To get this formula, first note that because $(Q^{21}\ Q^{22})$ is the second row of the inverse of Q and because $Q^{-1}Q = I$, it follows that

$$\begin{bmatrix} Q^{21} & Q^{22} \end{bmatrix} \begin{bmatrix} Q_{11} \\ Q_{21} \end{bmatrix} = 0$$

which implies that

$$Q^{21}Q_{11} + Q^{22}Q_{21} = 0.$$

Therefore,

$$-(Q^{22})^{-1}Q^{21} = Q_{21}Q_{11}^{-1}.$$

So we can write

$$p_0 = Q_{21}Q_{11}^{-1}m_0. \tag{78.22}$$

It can be verified that this formula replicates itself over time in the sense that

$$p_t = Q_{21}Q_{11}^{-1}m_t. \tag{78.23}$$

To implement formula (78.23), we want to compute Q_1 the eigenvector of Q associated with the stable eigenvalue ρ of Q.

By hand it can be verified that the eigenvector associated with the stable eigenvalue ρ is proportional to

$$Q_1 = \begin{bmatrix} 1 - \lambda\rho \\ 1 - \lambda \end{bmatrix}.$$

Notice that if we set $A = \rho$ and $G = 1$ in our earlier formula for p_t we get

$$p_t = G(I - \lambda A)^{-1}m_t = (1 - \lambda)(1 - \lambda\rho)^{-1}m_t,$$

a formula that is equivalent with

$$p_t = Q_{21}Q_{11}^{-1}m_t,$$

where

$$Q_1 = \begin{bmatrix} Q_{11} \\ Q_{21} \end{bmatrix}.$$

78.6.2 Remarks about Feedback

We have expressed (78.16) in what superficially appears to be a form in which y_{t+1} feeds back on y_t, even though what we actually want to represent is that the component p_t feeds **forward** on p_{t+1}, and through it, on future m_{t+j}, $j = 0, 1, 2, \ldots$.

A tell-tale sign that we should look beyond its superficial "feedback" form is that $\lambda^{-1} > 1$ so that the matrix H in (78.16) is **unstable**

- it has one eigenvalue ρ that is less than one in modulus that does not imperil stability, but …
- it has a second eigenvalue λ^{-1} that exceeds one in modulus and that makes H an unstable matrix

We'll keep these observations in mind as we turn now to a case in which the log money supply actually does feed back on the log of the price level.

78.7 Log money Supply Feeds Back on Log Price Level

An arrangement of eigenvalues that split around unity, with one being below unity and another being greater than unity, sometimes prevails when there is *feedback* from the log price level to the log money supply.

Let the feedback rule be

$$m_{t+1} = \rho m_t + \delta p_t \tag{78.24}$$

where $\rho \in (0,1)$ and where we shall now allow $\delta \neq 0$.

Warning: If things are to fit together as we wish to deliver a stable system for some initial value p_0 that we want to determine uniquely, δ cannot be too large.

The forward-looking equation (78.8) continues to describe equality between the demand and supply of money.

We assume that equations (78.8) and (78.24) govern $y_t \equiv \begin{bmatrix} m_t \\ p_t \end{bmatrix}$ for $t \geq 0$.

The transition matrix H in the law of motion

$$y_{t+1} = H y_t$$

now becomes

$$H = \begin{bmatrix} \rho & \delta \\ -(1-\lambda)/\lambda & \lambda^{-1} \end{bmatrix}.$$

We take m_0 as a given initial condition and as before seek an initial value p_0 that stabilizes the system in the sense that y_t converges as $t \rightarrow +\infty$.

Our approach is identical with the one followed above and is based on an eigenvalue decomposition in which, cross our fingers, one eigenvalue exceeds unity and the other is less than unity in absolute value.

When $\delta \neq 0$ as we now assume, the eigenvalues of H will no longer be $\rho \in (0,1)$ and $\lambda^{-1} > 1$

We'll just calculate them and apply the same algorithm that we used above.

That algorithm remains valid so long as the eigenvalues split around unity as before.

Again we assume that m_0 is an initial condition, but that p_0 is not given but to be solved for.

Let's write and execute some Python code that will let us explore how outcomes depend on δ.

```
def construct_H(ρ, λ, δ):
    "contruct matrix H given parameters."

    H = np.empty((2, 2))
    H[0, :] = ρ,δ
    H[1, :] = - (1 - λ) / λ, 1 / λ

    return H

def H_eigvals(ρ=.9, λ=.5, δ=0):
    "compute the eigenvalues of matrix H given parameters."

    # construct H matrix
    H = construct_H(ρ, λ, δ)

    # compute eigenvalues
    eigvals = np.linalg.eigvals(H)

    return eigvals
```

```
H_eigvals()
```

```
array([2. , 0.9])
```

Notice that a negative δ will not imperil the stability of the matrix H, even if it has a big absolute value.

```
# small negative δ
H_eigvals(δ=-0.05)
```

```
array([0.8562829, 2.0437171])
```

```
# large negative δ
H_eigvals(δ=-1.5)
```

```
array([0.10742784, 2.79257216])
```

A sufficiently small positive δ also causes no problem.

```
# sufficiently small positive δ
H_eigvals(δ=0.05)
```

```
array([0.94750622, 1.95249378])
```

But a large enough positive δ makes both eigenvalues of H strictly greater than unity in modulus.

For example,

```
H_eigvals(δ=0.2)
```

```
array([1.12984379, 1.77015621])
```

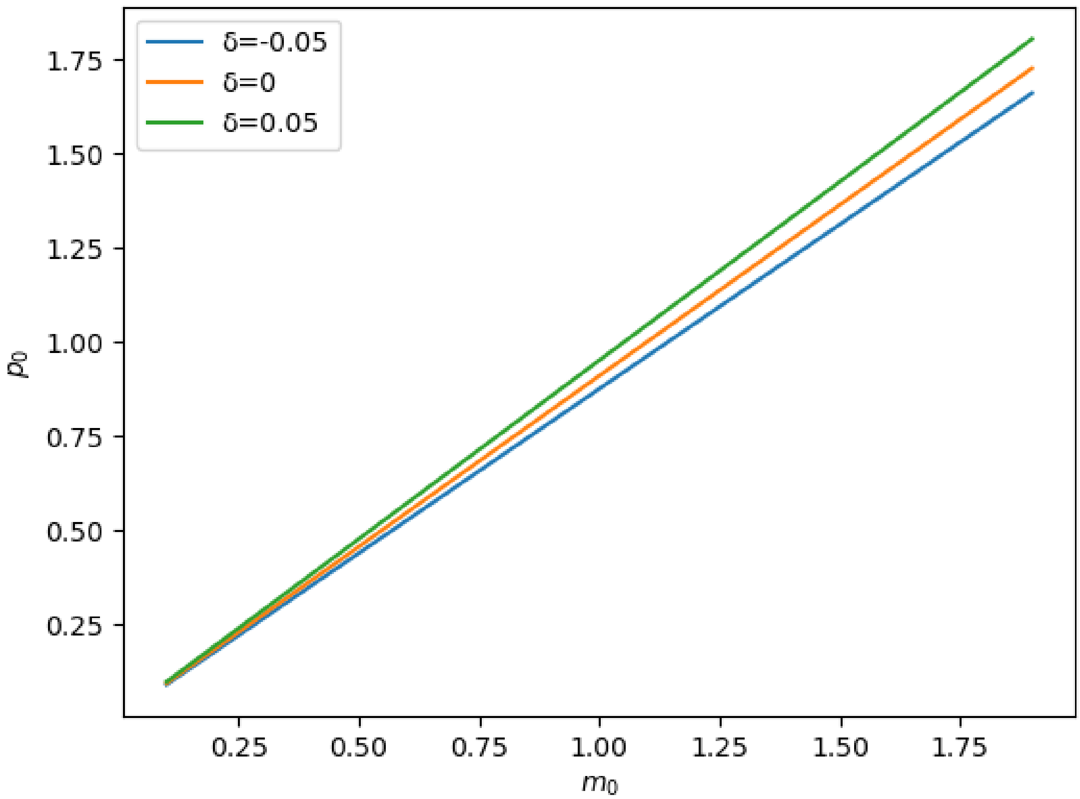
We want to study systems in which one eigenvalue exceeds unity in modulus while the other is less than unity in modulus, so we avoid values of δ that are too.

That is, we want to avoid too much positive feedback from p_t to m_{t+1}.

```
def magic_p0(m0, ρ=.9, λ=.5, δ=0):
    """
    Use the magic formula (8) to compute the level of p0
    that makes the system stable.
    """

    H = construct_H(ρ, λ, δ)
    eigvals, Q = np.linalg.eig(H)

    # find the index of the smaller eigenvalue
    ind = 0 if eigvals[0] < eigvals[1] else 1

    # verify that the eigenvalue is less than unity
    if eigvals[ind] > 1:

        print("both eigenvalues exceed unity in modulus")

        return None

    p0 = Q[1, ind] / Q[0, ind] * m0

    return p0
```

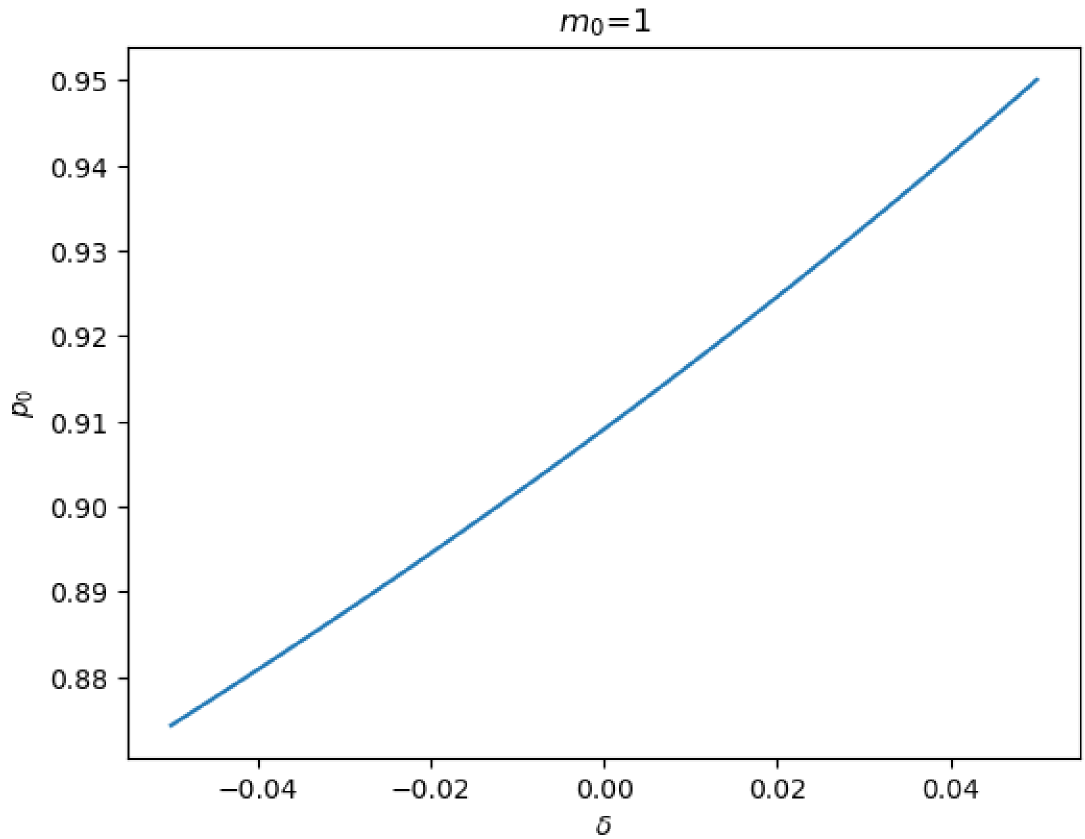
Let's plot how the solution p_0 changes as m_0 changes for different settings of δ.

```
m_range = np.arange(0.1, 2., 0.1)

for δ in [-0.05, 0, 0.05]:
    plt.plot(m_range, [magic_p0(m0, δ=δ) for m0 in m_range], label=f"δ={δ}")
plt.legend()

plt.xlabel(r"$m_0$")
plt.ylabel(r"$p_0$")
plt.show()
```



To look at things from a different angle, we can fix the initial value m_0 and see how p_0 changes as δ changes.

```
m0 = 1

δ_range = np.linspace(-0.05, 0.05, 100)
plt.plot(δ_range, [magic_p0(m0, δ=δ) for δ in δ_range])
plt.xlabel(r'$\delta$')
plt.ylabel(r'$p_0$')
plt.title(rf'$m_0$={m0}')
plt.show()
```



Notice that when δ is large enough, both eigenvalues exceed unity in modulus, causing a stabilizing value of p_0 not to exist.

```
magic_p0(1, δ=0.2)
```

```
both eigenvalues exceed unity in modulus
```

78.8 Big P, Little p Interpretation

It is helpful to view our solutions of difference equations having feedback from the price level or inflation to money or the rate of money creation in terms of the Big K, little k idea discussed in *Rational Expectations Models*.

This will help us sort out what is taken as given by the decision makers who use the difference equation (78.9) to determine p_t as a function of their forecasts of future values of m_t.

Let's write the stabilizing solution that we have computed using the eigenvector decomposition of H as $P_t = F^* m_t$, where

$$F^* = Q_{21} Q_{11}^{-1}.$$

Then from $P_{t+1} = F^* m_{t+1}$ and $m_{t+1} = \rho m_t + \delta P_t$ we can deduce the recursion $P_{t+1} = F^* \rho m_t + F^* \delta P_t$ and create the stacked system

$$\begin{bmatrix} m_{t+1} \\ P_{t+1} \end{bmatrix} = \begin{bmatrix} \rho & \delta \\ F^* \rho & F^* \delta \end{bmatrix} \begin{bmatrix} m_t \\ P_t \end{bmatrix}$$

or

$$x_{t+1} = A x_t$$

where $x_t = \begin{bmatrix} m_t \\ P_t \end{bmatrix}$.

Apply formula (78.13) for F to deduce that

$$p_t = F \begin{bmatrix} m_t \\ P_t \end{bmatrix} = F \begin{bmatrix} m_t \\ F^* m_t \end{bmatrix}$$

which implies that

$$p_t = \begin{bmatrix} F_1 & F_2 \end{bmatrix} \begin{bmatrix} m_t \\ F^* m_t \end{bmatrix} = F_1 m_t + F_2 F^* m_t$$

so that we can anticipate that

$$F^* = F_1 + F_2 F^*$$

We shall verify this equality in the next block of Python code that implements the following computations.

1. For the system with $\delta \neq 0$ so that there is feedback, we compute the stabilizing solution for p_t in the form $p_t = F^* m_t$ where $F^* = Q_{21} Q_{11}^{-1}$ as above.
2. Recalling the system (78.11), (78.12), and (78.13) above, we define $x_t = \begin{bmatrix} m_t \\ P_t \end{bmatrix}$ and notice that it is Big P_t and not little p_t here. Then we form A and G as $A = \begin{bmatrix} \rho & \delta \\ F^* \rho & F^* \delta \end{bmatrix}$ and $G = \begin{bmatrix} 1 & 0 \end{bmatrix}$ and we compute $\begin{bmatrix} F_1 & F_2 \end{bmatrix} \equiv F$ from equation (78.13) above.
3. We compute $F_1 + F_2 F^*$ and compare it with F^* and check for the anticipated equality.

```
# set parameters
ρ = .9
λ = .5
δ = .05
```

```
# solve for F_star
H = construct_H(ρ, λ, δ)
eigvals, Q = np.linalg.eig(H)

ind = 0 if eigvals[0] < eigvals[1] else 1
F_star = Q[1, ind] / Q[0, ind]
F_star
```

```
0.950124378879 11
```

```
# solve for F_check
A = np.empty((2, 2))
A[0, :] = ρ, δ
A[1, :] = F_star * A[0, :]

G = np.array([1, 0])

F_check= (1 - λ) * G @ np.linalg.inv(np.eye(2) - λ * A)
F_check
```

```
array([0.92755597, 0.02375311])
```

Compare F^* with $F_1 + F_2 F^*$

```
F_check[0] + F_check[1] * F_star, F_star
```

$$(0.95012437887911,\ 0.95012437887911)$$

78.9 Fun with SymPy

This section is a gift for readers who have made it this far.

It puts SymPy to work on our model.

Thus, we use Sympy to compute some key objects comprising the eigenvector decomposition of H.

We start by generating an H with nonzero δ.

```
λ, δ, ρ = symbols('λ, δ, ρ')
```

```
H1 = Matrix([[ρ,δ], [- (1 - λ) / λ, λ ** -1]])
```

```
H1
```

$$\begin{bmatrix} \rho & \delta \\ \frac{\lambda-1}{\lambda} & \frac{1}{\lambda} \end{bmatrix}$$

```
H1.eigenvals()
```

$$\left\{(\lambda\rho+1)/2\lambda - \sqrt{4\delta\lambda^2 - 4\delta\lambda + \lambda^2\rho^2 - 2\lambda\rho + 1}/2\lambda : 1,\ (\lambda\rho+1)/2\lambda + \sqrt{4\delta\lambda^2 - 4\delta\lambda + \lambda^2\rho^2 - 2\lambda\rho + 1}/2\lambda : 1\right\}$$

```
H1.eigenvects()
```

$$\left[\left(\frac{\lambda\rho+1}{2\lambda} - \frac{\sqrt{4\delta\lambda^2 - 4\delta\lambda + \lambda^2\rho^2 - 2\lambda\rho + 1}}{2\lambda},\ 1,\ \left[\begin{bmatrix} \frac{\lambda\left(\frac{\lambda\rho+1}{2\lambda} - \frac{\sqrt{4\delta\lambda^2-4\delta\lambda+\lambda^2\rho^2-2\lambda\rho+1}}{2\lambda}\right)}{\lambda-1} - \frac{1}{\lambda-1} \\ 1 \end{bmatrix}\right]\right),\ \left(\frac{\lambda\rho+1}{2\lambda} + \frac{\sqrt{4\delta\lambda^2 - 4\delta\lambda}}{}\right.$$

Now let's compute H when δ is zero.

```
H2 = Matrix([[ρ,0], [- (1 - λ) / λ, λ ** -1]])
```

```
H2
```

$$\begin{bmatrix} \rho & 0 \\ \frac{\lambda-1}{\lambda} & \frac{1}{\lambda} \end{bmatrix}$$

```
H2.eigenvals()
```

$\{1/\lambda: 1, \ \rho: 1\}$

```
H2.eigenvects()
```

$$\left[\left(\frac{1}{\lambda}, \ 1, \ \left[\begin{bmatrix} 0 \\ 1 \end{bmatrix} \right] \right), \ \left(\rho, \ 1, \ \left[\begin{bmatrix} \frac{\lambda\rho - 1}{\lambda - 1} \\ 1 \end{bmatrix} \right] \right) \right]$$

Below we do induce SymPy to do the following fun things for us analytically:

1. We compute the matrix Q whose first column is the eigenvector associated with ρ. and whose second column is the eigenvector associated with λ^{-1}.
2. We use SymPy to compute the inverse Q^{-1} of Q (both in symbols).
3. We use SymPy to compute $Q_{21}Q_{11}^{-1}$ (in symbols).
4. Where Q^{ij} denotes the (i, j) component of Q^{-1}, we use SymPy to compute $-(Q^{22})^{-1}Q^{21}$ (again in symbols)

```
# construct Q
vec = []
for i, (eigval, _, eigvec) in enumerate(H2.eigenvects()):

    vec.append(eigvec[0])

    if eigval == ρ:
        ind = i

Q = vec[ind].col_insert(1, vec[1-ind])
```

```
Q
```

$$\begin{bmatrix} \frac{\lambda\rho - 1}{\lambda - 1} & 0 \\ 1 & 1 \end{bmatrix}$$

Q^{-1}

```
Q_inv = Q ** (-1)
Q_inv
```

$$\begin{bmatrix} \frac{\lambda - 1}{\lambda\rho - 1} & 0 \\ \frac{1 - \lambda}{\lambda\rho - 1} & 1 \end{bmatrix}$$

$Q_{21}Q_{11}^{-1}$

```
Q[1, 0] / Q[0, 0]
```

$\frac{\lambda - 1}{\lambda\rho - 1}$

$-(Q^{22})^{-1}Q^{21}$

```
- Q_inv[1, 0] / Q_inv[1, 1]
```

$$-\frac{1-\lambda}{\lambda\rho-1}$$

CHAPTER

SEVENTYNINE

MARKOV PERFECT EQUILIBRIUM

Contents

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
```

79.1 Overview

This lecture describes the concept of Markov perfect equilibrium.

Markov perfect equilibrium is a key notion for analyzing economic problems involving dynamic strategic interaction, and a cornerstone of applied game theory.

In this lecture, we teach Markov perfect equilibrium by example.

We will focus on settings with

- two players
- quadratic payoff functions
- linear transition rules for the state

Other references include chapter 7 of [Ljungqvist and Sargent, 2018].

Let's start with some standard imports:

```
import matplotlib.pyplot as plt
import numpy as np
import quantecon as qe
```

79.2 Background

Markov perfect equilibrium is a refinement of the concept of Nash equilibrium.

It is used to study settings where multiple decision-makers interact non-cooperatively over time, each pursuing its own objective.

The agents in the model face a common state vector, the time path of which is influenced by – and influences – their decisions.

In particular, the transition law for the state that confronts each agent is affected by decision rules of other agents.

Individual payoff maximization requires that each agent solve a dynamic programming problem that includes this transition law.

Markov perfect equilibrium prevails when no agent wishes to revise its policy, taking as given the policies of all other agents.

Well known examples include

- Choice of price, output, location or capacity for firms in an industry (e.g., [Ericson and Pakes, 1995], [Ryan, 2012], [Doraszelski and Satterthwaite, 2010]).
- Rate of extraction from a shared natural resource, such as a fishery (e.g., [Levhari and Mirman, 1980], [Van Long, 2011]).

Let's examine a model of the first type.

79.2.1 Example: A Duopoly Model

Two firms are the only producers of a good, the demand for which is governed by a linear inverse demand function

$$p = a_0 - a_1(q_1 + q_2) \tag{79.1}$$

Here $p = p_t$ is the price of the good, $q_i = q_{it}$ is the output of firm $i = 1, 2$ at time t and $a_0 > 0, a_1 > 0$.

In (79.1) and what follows,

- the time subscript is suppressed when possible to simplify notation
- $\hat{x}$ denotes a next period value of variable x

Each firm recognizes that its output affects total output and therefore the market price.

The one-period payoff function of firm i is price times quantity minus adjustment costs:

$$\pi_i = p q_i - \gamma(\hat{q}_i - q_i)^2, \quad \gamma > 0, \tag{79.2}$$

Substituting the inverse demand curve (79.1) into (79.2) lets us express the one-period payoff as

$$\pi_i(q_i, q_{-i}, \hat{q}_i) = a_0 q_i - a_1 q_i^2 - a_1 q_i q_{-i} - \gamma(\hat{q}_i - q_i)^2, \tag{79.3}$$

where q_{-i} denotes the output of the firm other than i.

The objective of the firm is to maximize $\sum_{t=0}^{\infty} \beta^t \pi_{it}$.

Firm i chooses a decision rule that sets next period quantity $\hat{q}_i$ as a function f_i of the current state (q_i, q_{-i}).

An essential aspect of a Markov perfect equilibrium is that each firm takes the decision rule of the other firm as known and given.

Given f_{-i}, the Bellman equation of firm i is

$$v_i(q_i, q_{-i}) = \max_{\hat{q}_i} \{\pi_i(q_i, q_{-i}, \hat{q}_i) + \beta v_i(\hat{q}_i, f_{-i}(q_{-i}, q_i))\} \tag{79.4}$$

Definition A **Markov perfect equilibrium** of the duopoly model is a pair of value functions (v_1, v_2) and a pair of policy functions (f_1, f_2) such that, for each $i \in \{1, 2\}$ and each possible state,

- The value function v_i satisfies Bellman equation (79.4).
- The maximizer on the right side of (79.4) equals $f_i(q_i, q_{-i})$.

The adjective "Markov" denotes that the equilibrium decision rules depend only on the current values of the state variables, not other parts of their histories.

"Perfect" means complete, in the sense that the equilibrium is constructed by backward induction and hence builds in optimizing behavior for each firm at all possible future states.

- These include many states that will not be reached when we iterate forward on the pair of equilibrium strategies f_i starting from a given initial state.

79.2.2 Computation

One strategy for computing a Markov perfect equilibrium is iterating to convergence on pairs of Bellman equations and decision rules.

In particular, let v_i^j, f_i^j be the value function and policy function for firm i at the j-th iteration.

Imagine constructing the iterates

$$v_i^{j+1}(q_i, q_{-i}) = \max_{\hat{q}_i} \left\{\pi_i(q_i, q_{-i}, \hat{q}_i) + \beta v_i^j(\hat{q}_i, f_{-i}(q_{-i}, q_i))\right\} \tag{79.5}$$

These iterations can be challenging to implement computationally.

However, they simplify for the case in which one-period payoff functions are quadratic and transition laws are linear — which takes us to our next topic.

79.3 Linear Markov Perfect Equilibria

As we saw in the duopoly example, the study of Markov perfect equilibria in games with two players leads us to an interrelated pair of Bellman equations.

In linear-quadratic dynamic games, these "stacked Bellman equations" become "stacked Riccati equations" with a tractable mathematical structure.

We'll lay out that structure in a general setup and then apply it to some simple problems.

79.3.1 Coupled Linear Regulator Problems

We consider a general linear-quadratic regulator game with two players.

For convenience, we'll start with a finite horizon formulation, where t_0 is the initial date and t_1 is the common terminal date.

Player i takes $\{u_{-it}\}$ as given and minimizes

$$\sum_{t=t_0}^{t_1-1} \beta^{t-t_0} \{x_t' R_i x_t + u_{it}' Q_i u_{it} + u_{-it}' S_i u_{-it} + 2x_t' W_i u_{it} + 2u_{-it}' M_i u_{it}\} \tag{79.6}$$

while the state evolves according to

$$x_{t+1} = Ax_t + B_1 u_{1t} + B_2 u_{2t} \tag{79.7}$$

Here

- x_t is an $n \times 1$ state vector and u_{it} is a $k_i \times 1$ vector of controls for player i
- R_i is $n \times n$
- S_i is $k_{-i} \times k_{-i}$
- Q_i is $k_i \times k_i$
- W_i is $n \times k_i$
- M_i is $k_{-i} \times k_i$
- A is $n \times n$
- B_i is $n \times k_i$

79.3.2 Computing Equilibrium

We formulate a linear Markov perfect equilibrium as follows.

Player i employs linear decision rules $u_{it} = -F_{it} x_t$, where F_{it} is a $k_i \times n$ matrix.

A Markov perfect equilibrium is a pair of sequences $\{F_{1t}, F_{2t}\}$ over $t = t_0, \ldots, t_1 - 1$ such that

- $\{F_{1t}\}$ solves player 1's problem, taking $\{F_{2t}\}$ as given, and
- $\{F_{2t}\}$ solves player 2's problem, taking $\{F_{1t}\}$ as given

If we take $u_{2t} = -F_{2t} x_t$ and substitute it into (79.6) and (79.7), then player 1's problem becomes minimization of

$$\sum_{t=t_0}^{t_1-1} \beta^{t-t_0} \{x_t' \Pi_{1t} x_t + u_{1t}' Q_1 u_{1t} + 2u_{1t}' \Gamma_{1t} x_t\} \tag{79.8}$$

subject to

$$x_{t+1} = \Lambda_{1t} x_t + B_1 u_{1t}, \tag{79.9}$$

where

- $\Lambda_{it} := A - B_{-i} F_{-it}$
- $\Pi_{it} := R_i + F_{-it}' S_i F_{-it}$
- $\Gamma_{it} := W_i' - M_i' F_{-it}$

This is an LQ dynamic programming problem that can be solved by working backwards.

Decision rules that solve this problem are

$$F_{1t} = (Q_1 + \beta B_1' P_{1t+1} B_1)^{-1} (\beta B_1' P_{1t+1} \Lambda_{1t} + \Gamma_{1t}) \tag{79.10}$$

where P_{1t} solves the matrix Riccati difference equation

$$P_{1t} = \Pi_{1t} - (\beta B_1' P_{1t+1} \Lambda_{1t} + \Gamma_{1t})' (Q_1 + \beta B_1' P_{1t+1} B_1)^{-1} (\beta B_1' P_{1t+1} \Lambda_{1t} + \Gamma_{1t}) + \beta \Lambda_{1t}' P_{1t+1} \Lambda_{1t} \tag{79.11}$$

Similarly, decision rules that solve player 2's problem are

$$F_{2t} = (Q_2 + \beta B_2' P_{2t+1} B_2)^{-1} (\beta B_2' P_{2t+1} \Lambda_{2t} + \Gamma_{2t}) \tag{79.12}$$

where P_{2t} solves

$$P_{2t} = \Pi_{2t} - (\beta B_2' P_{2t+1} \Lambda_{2t} + \Gamma_{2t})' (Q_2 + \beta B_2' P_{2t+1} B_2)^{-1} (\beta B_2' P_{2t+1} \Lambda_{2t} + \Gamma_{2t}) + \beta \Lambda_{2t}' P_{2t+1} \Lambda_{2t} \tag{79.13}$$

Here, in all cases $t = t_0, \ldots, t_1 - 1$ and the terminal conditions are $P_{it_1} = 0$.

The solution procedure is to use equations (79.10), (79.11), (79.12), and (79.13), and "work backwards" from time $t_1 - 1$.

Since we're working backward, P_{1t+1} and P_{2t+1} are taken as given at each stage.

Moreover, since

- some terms on the right-hand side of (79.10) contain F_{2t}
- some terms on the right-hand side of (79.12) contain F_{1t}

we need to solve these $k_1 + k_2$ equations simultaneously.

Key Insight

A key insight is that equations (79.10) and (79.12) are linear in F_{1t} and F_{2t}.

After these equations have been solved, we can take F_{it} and solve for P_{it} in (79.11) and (79.13).

Infinite Horizon

We often want to compute the solutions of such games for infinite horizons, in the hope that the decision rules F_{it} settle down to be time-invariant as $t_1 \to +\infty$.

In practice, we usually fix t_1 and compute the equilibrium of an infinite horizon game by driving $t_0 \to -\infty$.

This is the approach we adopt in the next section.

79.3.3 Implementation

We use the function nnash from QuantEcon.py that computes a Markov perfect equilibrium of the infinite horizon linear-quadratic dynamic game in the manner described above.

79.4 Application

Let's use these procedures to treat some applications, starting with the duopoly model.

79.4.1 A Duopoly Model

To map the duopoly model into coupled linear-quadratic dynamic programming problems, define the state and controls as

$$x_t := \begin{bmatrix} 1 \\ q_{1t} \\ q_{2t} \end{bmatrix} \quad \text{and} \quad u_{it} := q_{i,t+1} - q_{it}, \quad i = 1, 2$$

If we write

$$x_t' R_i x_t + u_{it}' Q_i u_{it}$$

where $Q_1 = Q_2 = \gamma$,

$$R_1 := \begin{bmatrix} 0 & -\frac{a_0}{2} & 0 \\ -\frac{a_0}{2} & a_1 & \frac{a_1}{2} \\ 0 & \frac{a_1}{2} & 0 \end{bmatrix} \quad \text{and} \quad R_2 := \begin{bmatrix} 0 & 0 & -\frac{a_0}{2} \\ 0 & 0 & \frac{a_1}{2} \\ -\frac{a_0}{2} & \frac{a_1}{2} & a_1 \end{bmatrix}$$

then we recover the one-period payoffs in expression (79.3).

The law of motion for the state x_t is $x_{t+1} = Ax_t + B_1 u_{1t} + B_2 u_{2t}$ where

$$A := \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad B_1 := \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad B_2 := \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

The optimal decision rule of firm i will take the form $u_{it} = -F_i x_t$, inducing the following closed-loop system for the evolution of x in the Markov perfect equilibrium:

$$x_{t+1} = (A - B_1 F_1 - B_1 F_2) x_t \tag{79.14}$$

79.4.2 Parameters and Solution

Consider the previously presented duopoly model with parameter values of:

- $a_0 = 10$
- $a_1 = 2$
- $\beta = 0.96$
- $\gamma = 12$

From these, we compute the infinite horizon MPE using the preceding code

```python
import numpy as np
import quantecon as qe

# Parameters
a0 = 10.0
a1 = 2.0
β = 0.96
γ = 12.0

# In LQ form
A = np.eye(3)
B1 = np.array([[0.], [1.], [0.]])
B2 = np.array([[0.], [0.], [1.]])


R1 = [[      0.,      -a0 / 2,          0.],
      [-a0 / 2.,           a1,     a1 / 2.],
      [       0,      a1 / 2.,          0.]]

R2 = [[     0.,           0.,      -a0 / 2],
      [     0.,           0.,      a1 / 2.],
      [-a0 / 2,      a1 / 2.,           a1]]

Q1 = Q2 = γ
S1 = S2 = W1 = W2 = M1 = M2 = 0.0
```

(continues on next page)

(continued from previous page)

```
# Solve using QE's nnash function
F1, F2, P1, P2 = qe.nnash(A, B1, B2, R1, R2, Q1,
                          Q2, S1, S2, W1, W2, M1,
                          M2, beta=β)

# Display policies
print("Computed policies for firm 1 and firm 2:\n")
print(f"F1 = {F1}")
print(f"F2 = {F2}")
print("\n")
```

```
Computed policies for firm 1 and firm 2:

F1 = [[-0.66846615  0.29512482  0.07584666]]
F2 = [[-0.66846615  0.07584666  0.29512482]]
```

Running the code produces the following output.

One way to see that F_i is indeed optimal for firm i taking F_2 as given is to use QuantEcon.py's LQ class.

In particular, let's take F2 as computed above, plug it into (79.8) and (79.9) to get firm 1's problem and solve it using LQ.

We hope that the resulting policy will agree with F1 as computed above

```
Λ1 = A - B2 @ F2
lq1 = qe.LQ(Q1, R1, Λ1, B1, beta=β)
P1_ih, F1_ih, d = lq1.stationary_values()
F1_ih
```

```
array([[-0.66846613,  0.29512482,  0.07584666]])
```

This is close enough for rock and roll, as they say in the trade.

Indeed, np.allclose agrees with our assessment

```
np.allclose(F1, F1_ih)
```

```
True
```

79.4.3 Dynamics

Let's now investigate the dynamics of price and output in this simple duopoly model under the MPE policies.

Given our optimal policies $F1$ and $F2$, the state evolves according to (79.14).

The following program

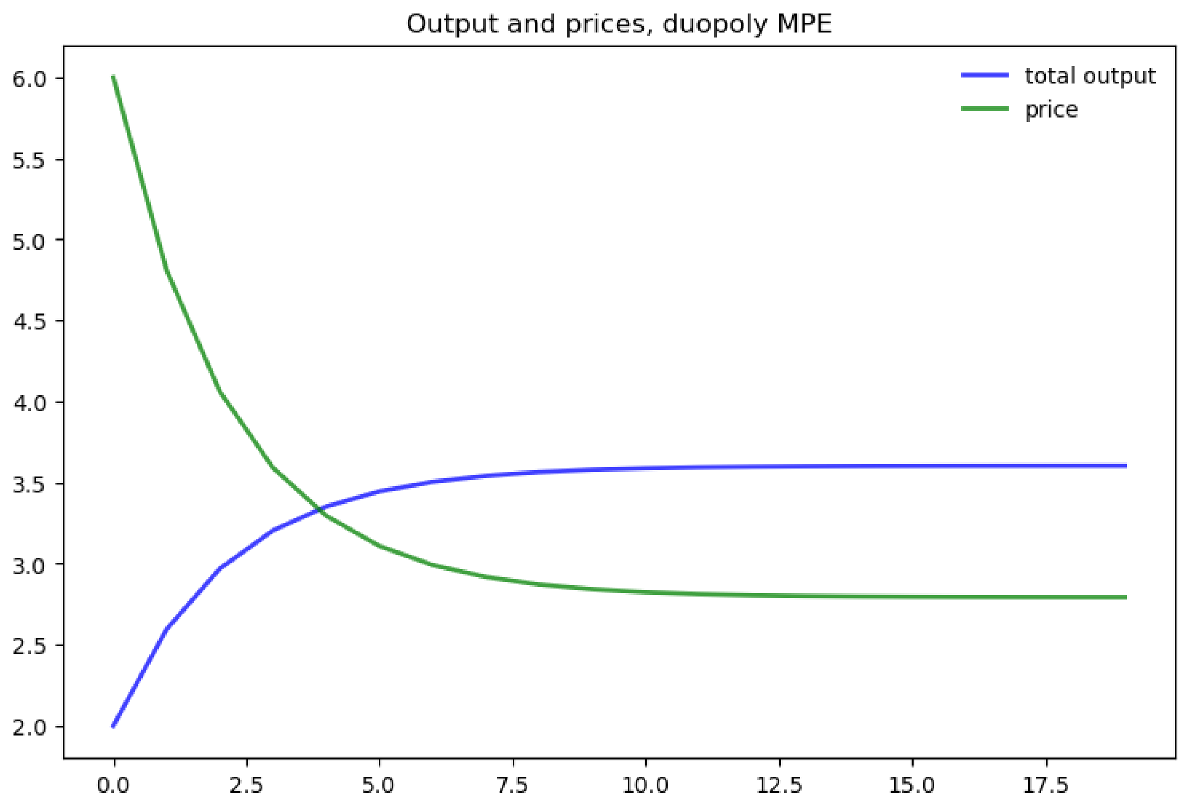
- imports $F1$ and $F2$ from the previous program along with all parameters.
- computes the evolution of x_t using (79.14).
- extracts and plots industry output $q_t = q_{1t} + q_{2t}$ and price $p_t = a_0 - a_1 q_t$.

```
AF = A - B1 @ F1 - B2 @ F2
n = 20
x = np.empty((3, n))
x[:, 0] = 1, 1, 1
```

(continues on next page)

(continued from previous page)

```
for t in range(n-1):
    x[:, t+1] = AF @ x[:, t]
q1 = x[1, :]
q2 = x[2, :]
q = q1 + q2       # Total output, MPE
p = a0 - a1 * q   # Price, MPE

fig, ax = plt.subplots(figsize=(9, 5.8))
ax.plot(q, 'b-', lw=2, alpha=0.75, label='total output')
ax.plot(p, 'g-', lw=2, alpha=0.75, label='price')
ax.set_title('Output and prices, duopoly MPE')
ax.legend(frameon=False)
plt.show()
```



Note that the initial condition has been set to $q_{10} = q_{20} = 1.0$.

To gain some perspective we can compare this to what happens in the monopoly case.

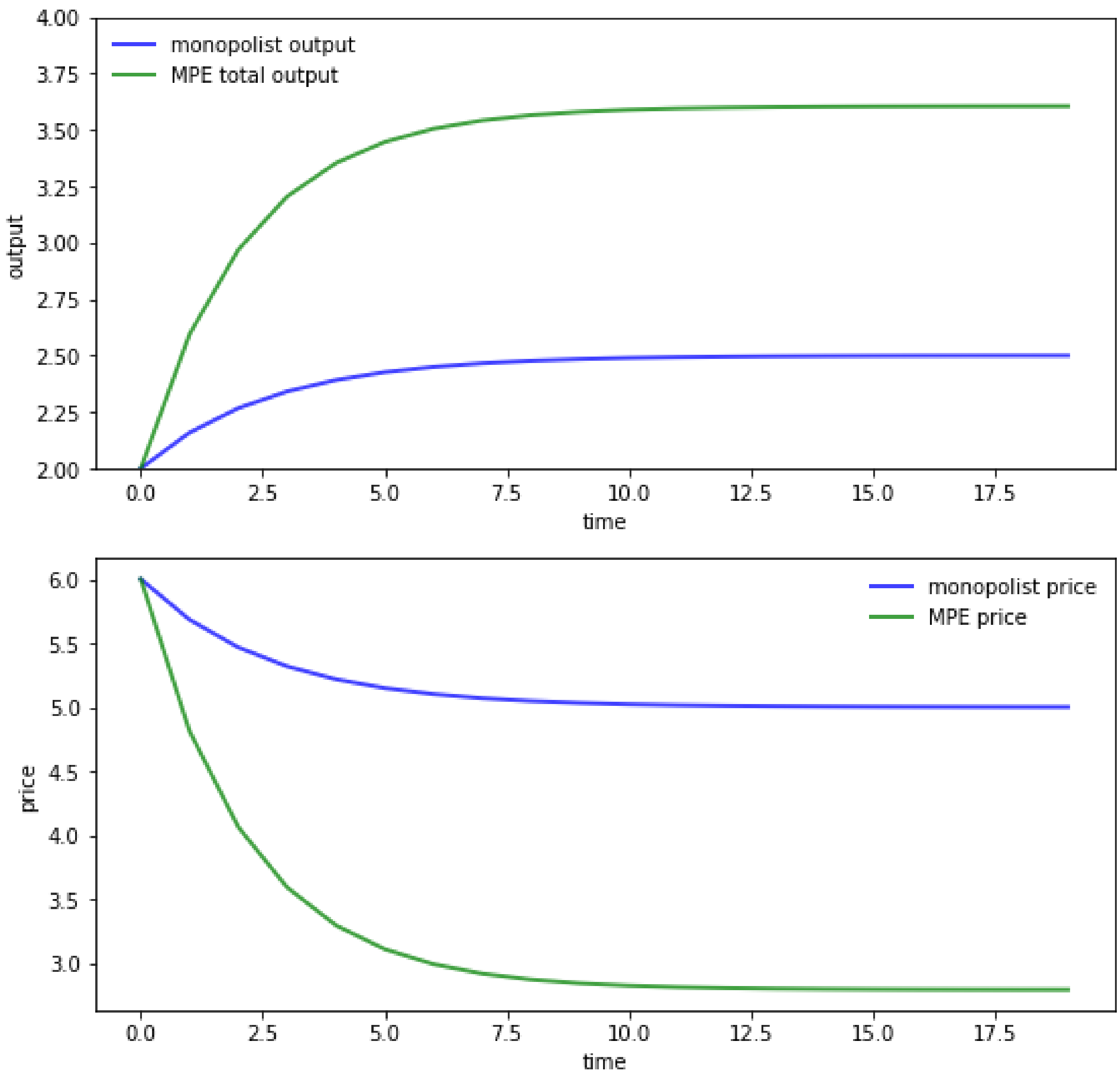
The first panel in the next figure compares output of the monopolist and industry output under the MPE, as a function of time.

The second panel shows analogous curves for price.

Here parameters are the same as above for both the MPE and monopoly solutions.

The monopolist initial condition is $q_0 = 2.0$ to mimic the industry initial condition $q_{10} = q_{20} = 1.0$ in the MPE case.

As expected, output is higher and prices are lower under duopoly than monopoly.


monopolist output
MPE total output
output
4.00
3.75
3.50
3.25
3.00
2.75
2.50
2.25
2.00
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
time
monopolist price
MPE price
price
6.0
5.5
5.0
4.5
4.0
3.5
3.0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
time

79.5 Exercises

Exercise 79.5.1

Replicate the *pair of figures* showing the comparison of output and prices for the monopolist and duopoly under MPE.

Parameters are as in duopoly_mpe.py and you can use that code to compute MPE policies under duopoly.

The optimal policy in the monopolist case can be computed using QuantEcon.py's LQ class.

Solution

First, let's compute the duopoly MPE under the stated parameters

```
# == Parameters == #
a0 = 10.0
a1 = 2.0
β = 0.96
γ = 12.0

# == In LQ form == #
A  = np.eye(3)
B1 = np.array([[0.], [1.], [0.]])
B2 = np.array([[0.], [0.], [1.]])
R1 = [[      0.,      -a0/2,          0.],
      [-a0 / 2.,          a1,     a1 / 2.],
      [       0,    a1 / 2.,          0.]]

R2 = [[     0.,           0.,     -a0 / 2],
      [     0.,           0.,     a1 / 2.],
      [-a0 / 2,     a1 / 2.,          a1]]

Q1 = Q2 = γ
S1 = S2 = W1 = W2 = M1 = M2 = 0.0

# == Solve using QE's nnash function == #
F1, F2, P1, P2 = qe.nnash(A, B1, B2, R1, R2, Q1,
                          Q2, S1, S2, W1, W2, M1,
                          M2, beta=β)
```

Now we evaluate the time path of industry output and prices given initial condition $q_{10} = q_{20} = 1$.

```
AF = A - B1 @ F1 - B2 @ F2
n = 20
x = np.empty((3, n))
x[:, 0] = 1, 1, 1
for t in range(n-1):
    x[:, t+1] = AF @ x[:, t]
q1 = x[1, :]
q2 = x[2, :]
q = q1 + q2       # Total output, MPE
p = a0 - a1 * q   # Price, MPE
```

Next, let's have a look at the monopoly solution.

For the state and control, we take

$$x_t = q_t - \bar{q} \quad \text{and} \quad u_t = q_{t+1} - q_t$$

To convert to an LQ problem we set

$$R = a_1 \quad \text{and} \quad Q = \gamma$$

in the payoff function $x_t' R x_t + u_t' Q u_t$ and

$$A = B = 1$$

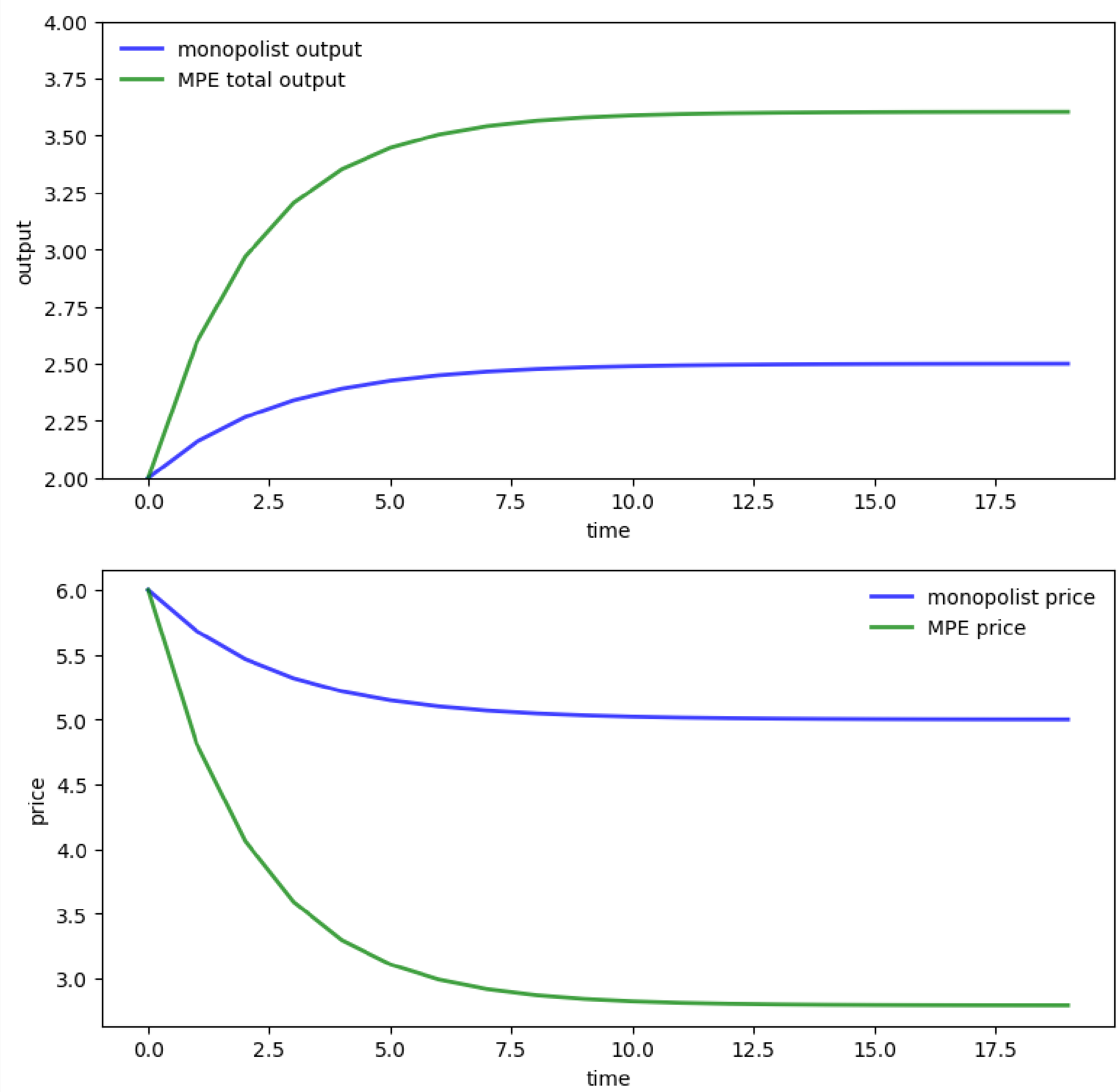
in the law of motion $x_{t+1} = A x_t + B u_t$.

We solve for the optimal policy $u_t = -F x_t$ and track the resulting dynamics of $\{q_t\}$, starting at $q_0 = 2.0$.

```
R = a1
Q = γ
A = B = 1
lq_alt = qe.LQ(Q, R, A, B, beta=β)
P, F, d = lq_alt.stationary_values()
q_bar = a0 / (2.0 * a1)
qm = np.empty(n)
qm[0] = 2
x0 = qm[0] - q_bar
x = x0
for i in range(1, n):
    x = A * x - B * F * x
    qm[i] = float(x.item()) + q_bar
pm = a0 - a1 * qm
```

Let's have a look at the different time paths

```
fig, axes = plt.subplots(2, 1, figsize=(9, 9))

ax = axes[0]
ax.plot(qm, 'b-', lw=2, alpha=0.75, label='monopolist output')
ax.plot(q, 'g-', lw=2, alpha=0.75, label='MPE total output')
ax.set(ylabel="output", xlabel="time", ylim=(2, 4))
ax.legend(loc='upper left', frameon=0)

ax = axes[1]
ax.plot(pm, 'b-', lw=2, alpha=0.75, label='monopolist price')
ax.plot(p, 'g-', lw=2, alpha=0.75, label='MPE price')
ax.set(ylabel="price", xlabel="time")
ax.legend(loc='upper right', frameon=0)
plt.show()
```



Exercise 79.5.2

In this exercise, we consider a slightly more sophisticated duopoly problem.

It takes the form of infinite horizon linear-quadratic game proposed by Judd [Judd, 1990].

Two firms set prices and quantities of two goods interrelated through their demand curves.

Relevant variables are defined as follows:

- I_{it} = inventories of firm i at beginning of t
- q_{it} = production of firm i during period t
- p_{it} = price charged by firm i during period t
- S_{it} = sales made by firm i during period t
- E_{it} = costs of production of firm i during period t

- C_{it} = costs of carrying inventories for firm i during t

The firms' cost functions are

- $C_{it} = c_{i1} + c_{i2} I_{it} + 0.5 c_{i3} I_{it}^2$
- $E_{it} = e_{i1} + e_{i2} q_{it} + 0.5 e_{i3} q_{it}^2$ where e_{ij}, c_{ij} are positive scalars

Inventories obey the laws of motion

$$I_{i,t+1} = (1 - \delta) I_{it} + q_{it} - S_{it}$$

Demand is governed by the linear schedule

$$S_t = D p_{it} + b$$

where

- $S_t = \begin{bmatrix} S_{1t} & S_{2t} \end{bmatrix}'$
- D is a 2×2 negative definite matrix and
- b is a vector of constants

Firm i maximizes the undiscounted sum

$$\lim_{T \to \infty} \frac{1}{T} \sum_{t=0}^{T} \left(p_{it} S_{it} - E_{it} - C_{it} \right)$$

We can convert this to a linear-quadratic problem by taking

$$u_{it} = \begin{bmatrix} p_{it} \\ q_{it} \end{bmatrix} \quad \text{and} \quad x_t = \begin{bmatrix} I_{1t} \\ I_{2t} \\ 1 \end{bmatrix}$$

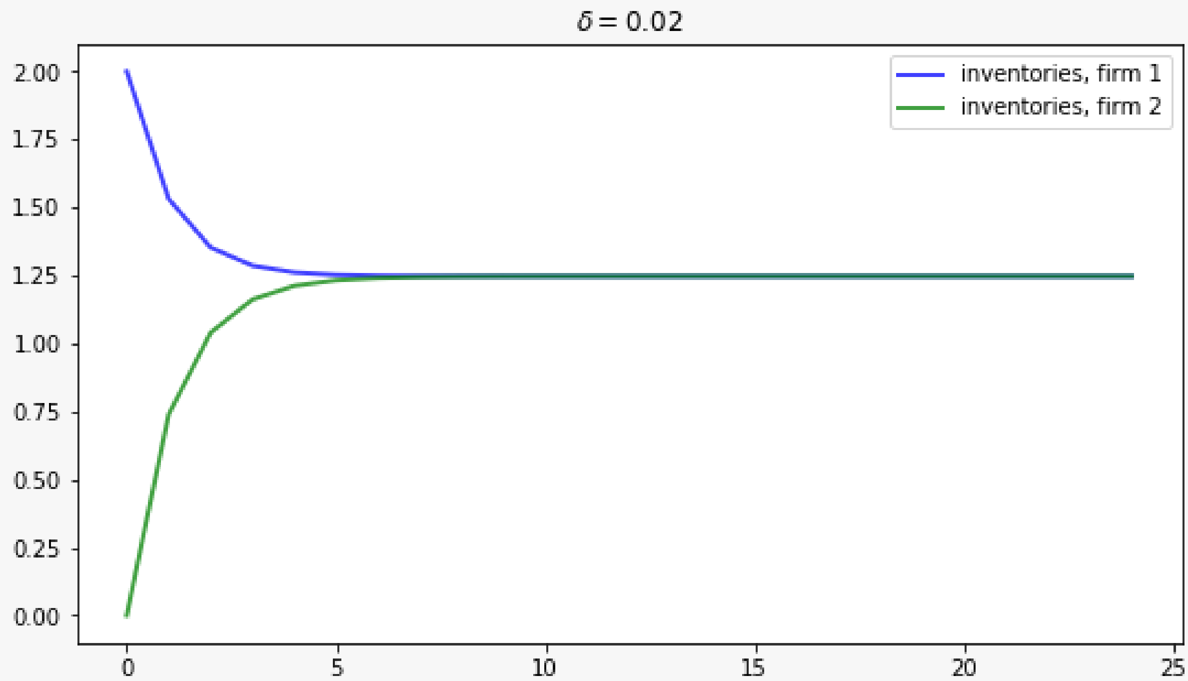
Decision rules for price and quantity take the form $u_{it} = -F_i x_t$.

The Markov perfect equilibrium of Judd's model can be computed by filling in the matrices appropriately.

The exercise is to calculate these matrices and compute the following figures.

The first figure shows the dynamics of inventories for each firm when the parameters are

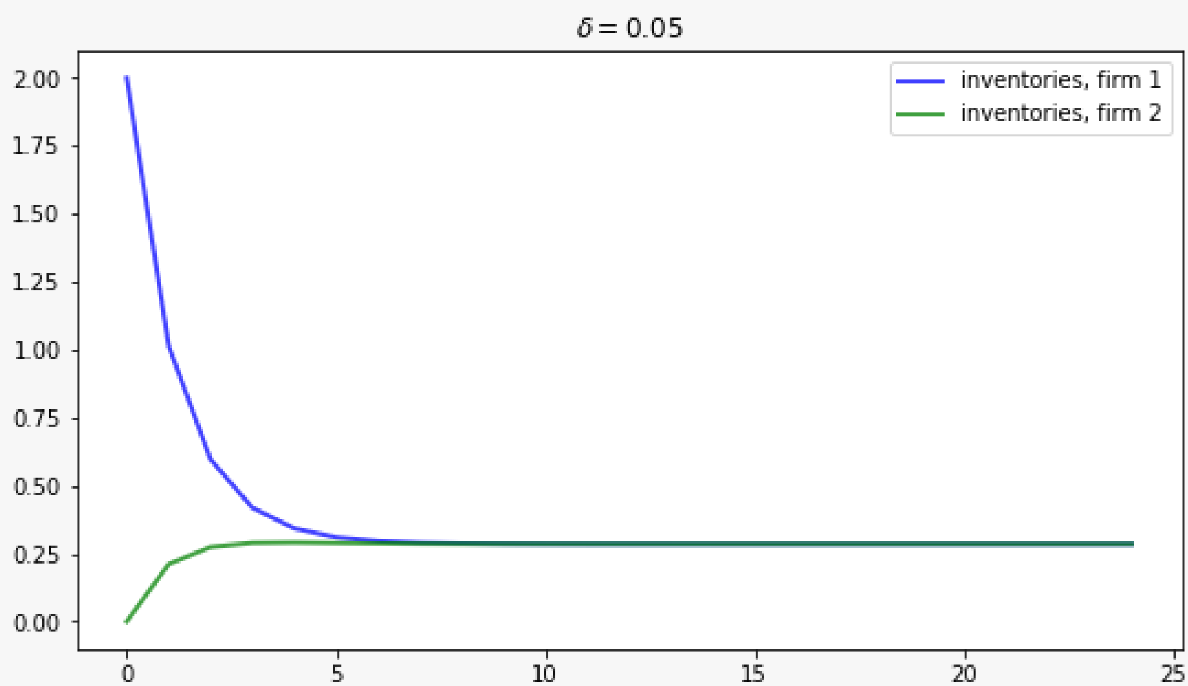
```
δ = 0.02
D = np.array([[-1, 0.5], [0.5, -1]])
b = np.array([25, 25])
c1 = c2 = np.array([1, -2, 1])
e1 = e2 = np.array([10, 10, 3])
```



Inventories trend to a common steady state.

If we increase the depreciation rate to $\delta = 0.05$, then we expect steady state inventories to fall.

This is indeed the case, as the next figure shows



In this exercise, reproduce the figure when $\delta = 0.02$.

Solution

We treat the case $\delta = 0.02$

```
δ = 0.02
D = np.array([[-1, 0.5], [0.5, -1]])
b = np.array([25, 25])
c1 = c2 = np.array([1, -2, 1])
e1 = e2 = np.array([10, 10, 3])

δ_1 = 1 - δ
```

Recalling that the control and state are

$$u_{it} = \begin{bmatrix} p_{it} \\ q_{it} \end{bmatrix} \quad \text{and} \quad x_t = \begin{bmatrix} I_{1t} \\ I_{2t} \\ 1 \end{bmatrix}$$

we set up the matrices as follows:

```
# ==  Create matrices needed to compute the Nash feedback equilibrium == #

A = np.array([[δ_1,       0,    -δ_1 * b[0]],
              [  0,     δ_1,    -δ_1 * b[1]],
              [  0,       0,              1]])

B1 = δ_1 * np.array([[1, -D[0, 0]],
                     [0, -D[1, 0]],
                     [0,       0]])
B2 = δ_1 * np.array([[0, -D[0, 1]],
                     [1, -D[1, 1]],
                     [0,       0]])

R1 = -np.array([[0.5 * c1[2],     0,    0.5 * c1[1]],
                [          0,     0,              0],
                [0.5 * c1[1],     0,          c1[0]]])
R2 = -np.array([[0,               0,              0],
                [0,     0.5 * c2[2],    0.5 * c2[1]],
                [0,     0.5 * c2[1],          c2[0]]])

Q1 = np.array([[-0.5 * e1[2], 0], [0, D[0, 0]]])
Q2 = np.array([[-0.5 * e2[2], 0], [0, D[1, 1]]])

S1 = np.zeros((2, 2))
S2 = np.copy(S1)

W1 = np.array([[           0,             0],
               [           0,             0],
               [-0.5 * e1[1],    b[0] / 2.]])
W2 = np.array([[           0,             0],
               [           0,             0],
               [-0.5 * e2[1],    b[1] / 2.]])

M1 = np.array([[0, 0], [0, D[0, 1] / 2.]])
M2 = np.copy(M1)
```

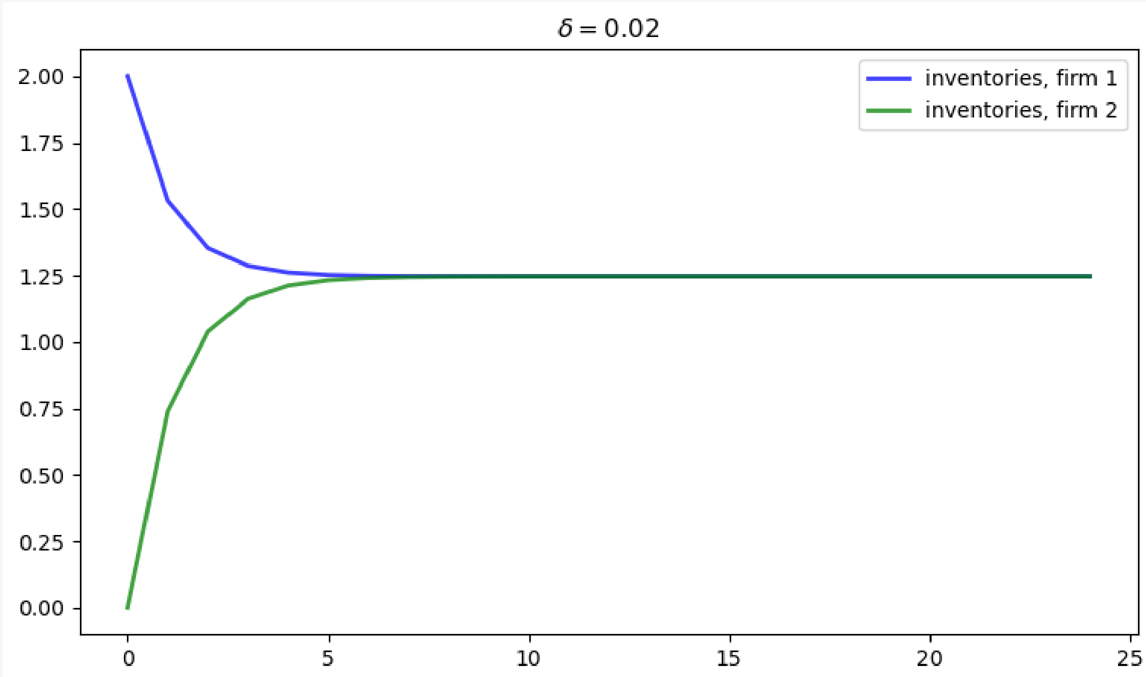
We can now compute the equilibrium using `qe.nnash`

```
F1, F2, P1, P2 = qe.nnash(A, B1, B2, R1,
                          R2, Q1, Q2, S1,
                          S2, W1, W2, M1, M2)

print("\nFirm 1's feedback rule:\n")
print(F1)

print("\nFirm 2's feedback rule:\n")
print(F2)
```

```
Firm 1's feedback rule:

[[ 2.43666582e-01  2.72360627e-02 -6.82788293e+00]
 [ 3.92370734e-01  1.39696451e-01 -3.77341073e+01]]

Firm 2's feedback rule:

[[ 2.72360627e-02  2.43666582e-01 -6.82788293e+00]
 [ 1.39696451e-01  3.92370734e-01 -3.77341073e+01]]
```

Now let's look at the dynamics of inventories, and reproduce the graph corresponding to $\delta = 0.02$

```
AF = A - B1 @ F1 - B2 @ F2
n = 25
x = np.empty((3, n))
x[:, 0] = 2, 0, 1
for t in range(n-1):
    x[:, t+1] = AF @ x[:, t]
I1 = x[0, :]
I2 = x[1, :]
fig, ax = plt.subplots(figsize=(9, 5))
ax.plot(I1, 'b-', lw=2, alpha=0.75, label='inventories, firm 1')
ax.plot(I2, 'g-', lw=2, alpha=0.75, label='inventories, firm 2')
ax.set_title(rf'$\delta = {δ}$')
ax.legend()
plt.show()
```



$\delta = 0.02$
inventories, firm 1
inventories, firm 2
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0
5
10
15
20
25

CHAPTER

EIGHTY

UNCERTAINTY TRAPS

Contents

80.1 Overview

In this lecture, we study a simplified version of an uncertainty traps model of Fajgelbaum, Schaal and Taschereau-Dumouchel [Fajgelbaum *et al.*, 2015].

The model features self-reinforcing uncertainty that has big impacts on economic activity.

In the model,

- Fundamentals vary stochastically and are not fully observable.
- At any moment there are both active and inactive entrepreneurs; only active entrepreneurs produce.
- Agents – active and inactive entrepreneurs – have beliefs about the fundamentals expressed as probability distributions.
- Greater uncertainty means greater dispersions of these distributions.
- Entrepreneurs are risk-averse and hence less inclined to be active when uncertainty is high.
- The output of active entrepreneurs is observable, supplying a noisy signal that helps everyone inside the model infer fundamentals.
- Entrepreneurs update their beliefs about fundamentals using Bayes' Law, implemented via *Kalman filtering*.

Uncertainty traps emerge because:

- High uncertainty discourages entrepreneurs from becoming active.
- A low level of participation – i.e., a smaller number of active entrepreneurs – diminishes the flow of information about fundamentals.

- Less information translates to higher uncertainty, further discouraging entrepreneurs from choosing to be active, and so on.

Uncertainty traps stem from a positive externality: high aggregate economic activity levels generates valuable information.

Let's start with some standard imports:

```
import matplotlib.pyplot as plt
import numpy as np
```

80.2 The Model

The original model described in [Fajgelbaum *et al.*, 2015] has many interesting moving parts.

Here we examine a simplified version that nonetheless captures many of the key ideas.

80.2.1 Fundamentals

The evolution of the fundamental process $\{\theta_t\}$ is given by

$$\theta_{t+1} = \rho\theta_t + \sigma_\theta w_{t+1}$$

where

- $\sigma_\theta > 0$ and $0 < \rho < 1$
- $\{w_t\}$ is IID and standard normal

The random variable θ_t is not observable at any time.

80.2.2 Output

There is a total $\bar{M}$ of risk-averse entrepreneurs.

Output of the m-th entrepreneur, conditional on being active in the market at time t, is equal to

$$x_m = \theta + \epsilon_m \quad \text{where} \quad \epsilon_m \sim N\left(0, \gamma_x^{-1}\right) \tag{80.1}$$

Here the time subscript has been dropped to simplify notation.

The inverse of the shock variance, γ_x, is called the shock's **precision**.

The higher is the precision, the more informative x_m is about the fundamental.

Output shocks are independent across time and firms.

80.2.3 Information and Beliefs

All entrepreneurs start with identical beliefs about θ_0.

Signals are publicly observable and hence all agents have identical beliefs always.

Dropping time subscripts, beliefs for current θ are represented by the normal distribution $N(\mu, \gamma^{-1})$.

Here γ is the precision of beliefs; its inverse is the degree of uncertainty.

These parameters are updated by Kalman filtering.

Let

- $\mathbb{M} \subset \{1, \dots, \bar{M}\}$ denote the set of currently active firms.
- $M := |\mathbb{M}|$ denote the number of currently active firms.
- X be the average output $\frac{1}{M} \sum_{m \in \mathbb{M}} x_m$ of the active firms.

With this notation and primes for next period values, we can write the updating of the mean and precision via

$$\mu' = \rho \frac{\gamma \mu + M \gamma_x X}{\gamma + M \gamma_x} \tag{80.2}$$

$$\gamma' = \left(\frac{\rho^2}{\gamma + M \gamma_x} + \sigma_\theta^2 \right)^{-1} \tag{80.3}$$

These are standard Kalman filtering results applied to the current setting.

Exercise 1 provides more details on how (80.2) and (80.3) are derived and then asks you to fill in remaining steps.

The next figure plots the law of motion for the precision in (80.3) as a 45 degree diagram, with one curve for each $M \in \{0, \dots, 6\}$.

The other parameter values are $\rho = 0.99, \gamma_x = 0.5, \sigma_\theta = 0.5$

Points where the curves hit the 45 degree lines are long-run steady states for precision for different values of M.

Thus, if one of these values for M remains fixed, a corresponding steady state is the equilibrium level of precision

- high values of M correspond to greater information about the fundamental, and hence more precision in steady state
- low values of M correspond to less information and more uncertainty in steady state

In practice, as we'll see, the number of active firms fluctuates stochastically.

80.2.4 Participation

Omitting time subscripts once more, entrepreneurs enter the market in the current period if

$$\mathbb{E}[u(x_m - F_m)] > c \tag{80.4}$$

Here

- the mathematical expectation of x_m is based on (80.1) and beliefs $N(\mu, \gamma^{-1})$ for θ
- F_m is a stochastic but pre-visible fixed cost, independent across time and firms
- c is a constant reflecting opportunity costs

The statement that F_m is pre-visible means that it is realized at the start of the period and treated as a constant in (80.4).

The utility function has the constant absolute risk aversion form

$$u(x) = \frac{1}{a} (1 - \exp(-ax)) \tag{80.5}$$

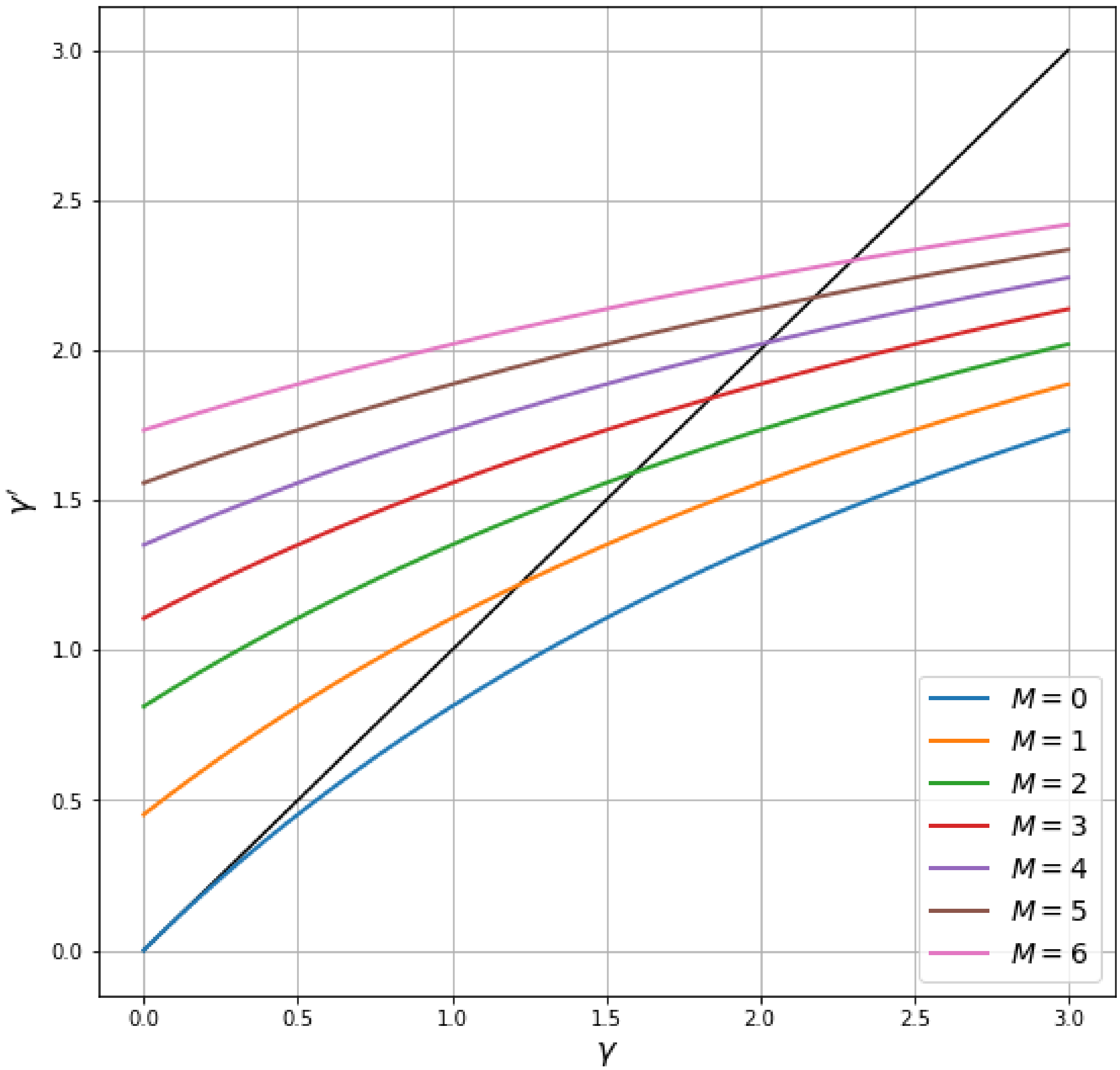
where a is a positive parameter.

Combining (80.4) and (80.5), entrepreneur m participates in the market (or is said to be active) when

$$\frac{1}{a} \{1 - \mathbb{E}[\exp(-a(\theta + \epsilon_m - F_m))]\} > c$$

Using standard formulas for expectations of lognormal random variables, this is equivalent to the condition

$$\psi(\mu, \gamma, F_m) := \frac{1}{a} \left(1 - \exp\left(-a\mu + aF_m + \frac{a^2 \left(\frac{1}{\gamma} + \frac{1}{\gamma_x} \right)}{2} \right) \right) - c > 0 \tag{80.6}$$



3.0
2.5
2.0
1.5
1.0
0.5
0.0
γ'
0.0
0.5
1.0
1.5
2.0
2.5
3.0
γ
$M = 0$
$M = 1$
$M = 2$
$M = 3$
$M = 4$
$M = 5$
$M = 6$

80.3 Implementation

We want to simulate this economy.

As a first step, let's put together a class that bundles

- the parameters, the current value of θ and the current values of the two belief parameters μ and γ
- methods to update θ, μ and γ, as well as to determine the number of active firms and their outputs

The updating methods follow the laws of motion for θ, μ and γ given above.

The method to evaluate the number of active firms generates $F_1, \ldots, F_{\bar{M}}$ and tests condition (80.6) for each firm.

The **init** method encodes as default values the parameters we'll use in the simulations below

```python
class UncertaintyTrapEcon:

    def __init__(self,
                 a=1.5,             # Risk aversion
                 γ_x=0.5,           # Production shock precision
                 ρ=0.99,            # Correlation coefficient for θ
                 σ_θ=0.5,           # Standard dev of θ shock
                 num_firms=100,     # Number of firms
                 σ_F=1.5,           # Standard dev of fixed costs
                 c=-420,            # External opportunity cost
                 μ_init=0,          # Initial value for μ
                 γ_init=4,          # Initial value for γ
                 θ_init=0):         # Initial value for θ

        # == Record values == #
        self.a, self.γ_x, self.ρ, self.σ_θ = a, γ_x, ρ, σ_θ
        self.num_firms, self.σ_F, self.c, = num_firms, σ_F, c
        self.σ_x = np.sqrt(1/γ_x)

        # == Initialize states == #
        self.γ, self.μ, self.θ = γ_init, μ_init, θ_init

    def ψ(self, F):
        temp1 = -self.a * (self.μ - F)
        temp2 = self.a**2 * (1/self.γ + 1/self.γ_x) / 2
        return (1 / self.a) * (1 - np.exp(temp1 + temp2)) - self.c

    def update_beliefs(self, X, M):
        """
        Update beliefs (μ, γ) based on aggregates X and M.
        """
        # Simplify names
        γ_x, ρ, σ_θ = self.γ_x, self.ρ, self.σ_θ
        # Update μ
        temp1 = ρ * (self.γ * self.μ + M * γ_x * X)
        temp2 = self.γ + M * γ_x
        self.μ = temp1 / temp2
        # Update γ
        self.γ = 1 / (ρ**2 / (self.γ + M * γ_x) + σ_θ**2)

    def update_θ(self, w):
        """
        Update the fundamental state θ given shock w.
```

(continues on next page)

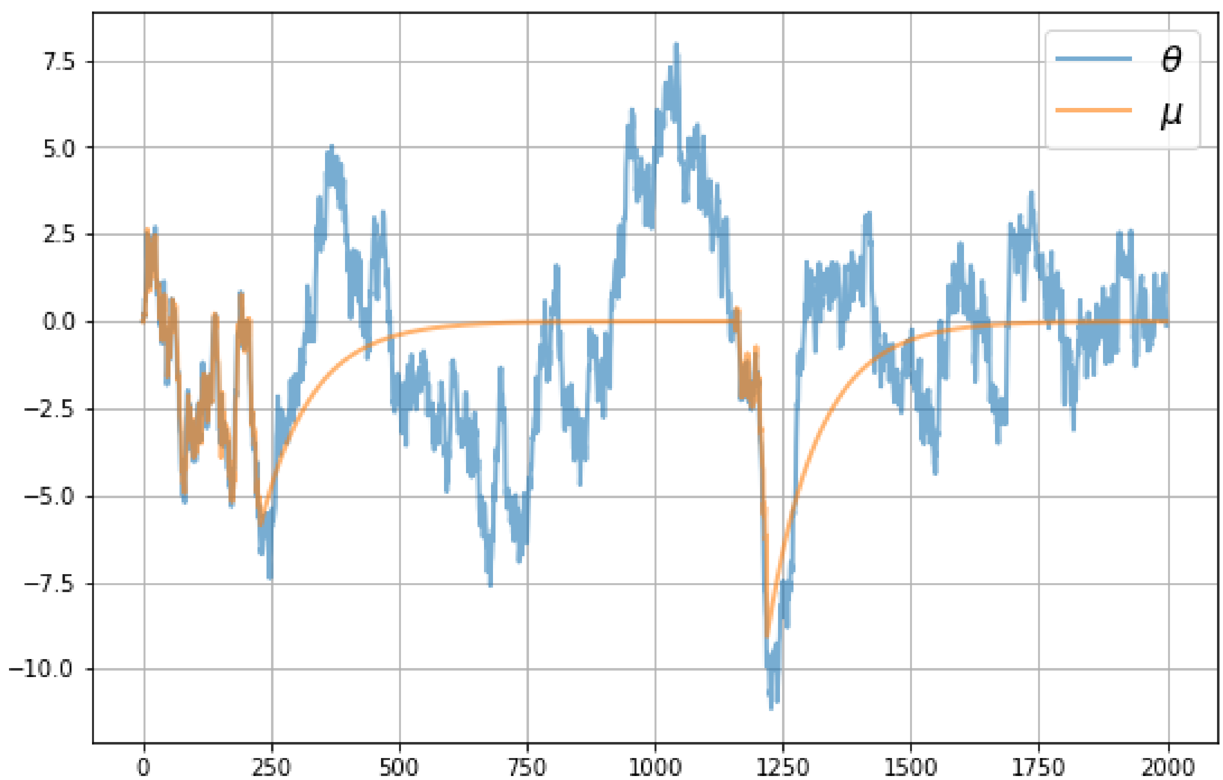
(continued from previous page)

```
        """
        self.θ = self.ρ * self.θ + self.σ_θ * w

    def gen_aggregates(self):
        """
        Generate aggregates based on current beliefs (μ, γ). This
        is a simulation step that depends on the draws for F.
        """
        F_vals = self.σ_F * np.random.randn(self.num_firms)
        M = np.sum(self.ψ(F_vals) > 0)  # Counts number of active firms
        if M > 0:
            x_vals = self.θ + self.σ_x * np.random.randn(M)
            X = x_vals.mean()
        else:
            X = 0
        return X, M
```

In the results below we use this code to simulate time series for the major variables.

80.4 Results

Let's look first at the dynamics of μ, which the agents use to track θ



We see that μ tracks θ well when there are sufficient firms in the market.

However, there are times when μ tracks θ poorly due to insufficient information.

These are episodes where the uncertainty traps take hold.

During these episodes

- precision is low and uncertainty is high
- few firms are in the market

To get a clearer idea of the dynamics, let's look at all the main time series at once, for a given set of shocks

Notice how the traps only take hold after a sequence of bad draws for the fundamental.

Thus, the model gives us a **propagation mechanism** that maps bad random draws into long downturns in economic activity.

80.5 Exercises

Exercise 80.5.1

Fill in the details behind (80.2) and (80.3) based on the following standard result (see, e.g., p. 24 of [Young and Smith, 2005]).

Fact Let $\mathbf{x} = (x_1, \ldots, x_M)$ be a vector of IID draws from common distribution $N(\theta, 1/\gamma_x)$ and let $\bar{x}$ be the sample mean. If γ_x is known and the prior for θ is $N(\mu, 1/\gamma)$, then the posterior distribution of θ given $\mathbf{x}$ is

$$\pi(\theta \,|\, \mathbf{x}) = N(\mu_0, 1/\gamma_0)$$

where

$$\mu_0 = \frac{\mu\gamma + M\bar{x}\gamma_x}{\gamma + M\gamma_x} \quad \text{and} \quad \gamma_0 = \gamma + M\gamma_x$$

Solution

This exercise asked you to validate the laws of motion for γ and μ given in the lecture, based on the stated result about Bayesian updating in a scalar Gaussian setting. The stated result tells us that after observing average output X of the M firms, our posterior beliefs will be

$$N(\mu_0, 1/\gamma_0)$$

where

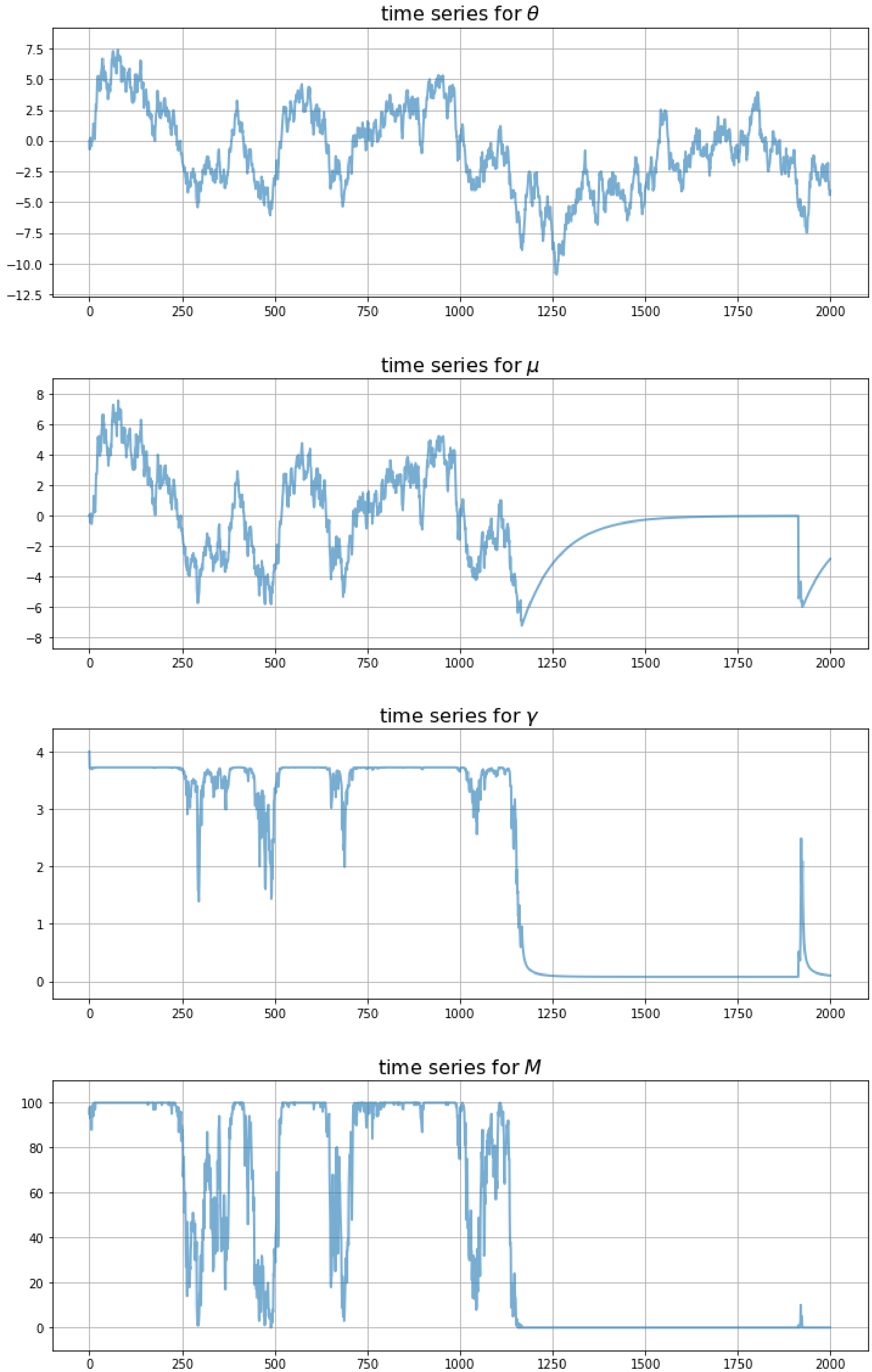
$$\mu_0 = \frac{\mu\gamma + MX\gamma_x}{\gamma + M\gamma_x} \quad \text{and} \quad \gamma_0 = \gamma + M\gamma_x$$

If we take a random variable θ with this distribution and then evaluate the distribution of $\rho\theta + \sigma_\theta w$ where w is independent and standard normal, we get the expressions for μ' and γ' given in the lecture.

Exercise 80.5.2

Modulo randomness, replicate the simulation figures shown above.

- Use the parameter values listed as defaults in the **init** method of the UncertaintyTrapEcon class.


time series for θ
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−12.5
0
250
500
750
1000
1250
1500
1750
2000
time series for μ
8
6
4
2
0
−2
−4
−6
−8
time series for γ
4
3
2
1
0
time series for M
100
80
60
40
20
0

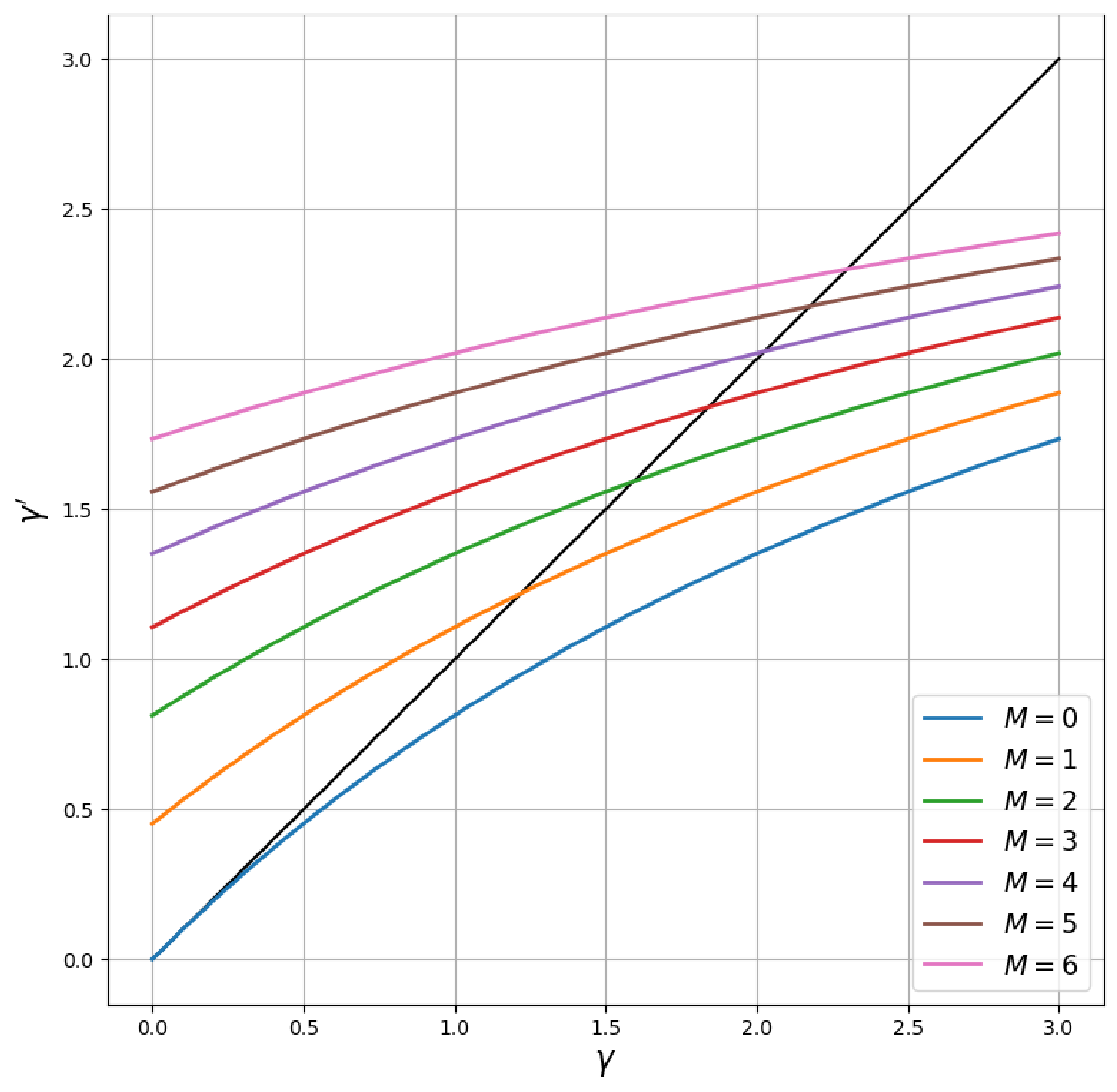
Solution

First, let's replicate the plot that illustrates the law of motion for precision, which is

$$\gamma_{t+1} = \left(\frac{\rho^2}{\gamma_t + M\gamma_x} + \sigma_\theta^2 \right)^{-1}$$

Here M is the number of active firms. The next figure plots γ_{t+1} against γ_t on a 45 degree diagram for different values of M

```
econ = UncertaintyTrapEcon()
ρ, σ_θ, γ_x = econ.ρ, econ.σ_θ, econ.γ_x     # Simplify names
γ = np.linspace(1e-10, 3, 200)              # γ grid
fig, ax = plt.subplots(figsize=(9, 9))
ax.plot(γ, γ, 'k-')                         # 45 degree line

for M in range(7):
    γ_next = 1 / (ρ**2 / (γ + M * γ_x) + σ_θ**2)
    label_string = f"$M = {M}$"
    ax.plot(γ, γ_next, lw=2, label=label_string)
ax.legend(loc='lower right', fontsize=14)
ax.set_xlabel(r'$\gamma$', fontsize=16)
ax.set_ylabel(r"$\gamma'$", fontsize=16)
ax.grid()
plt.show()
```



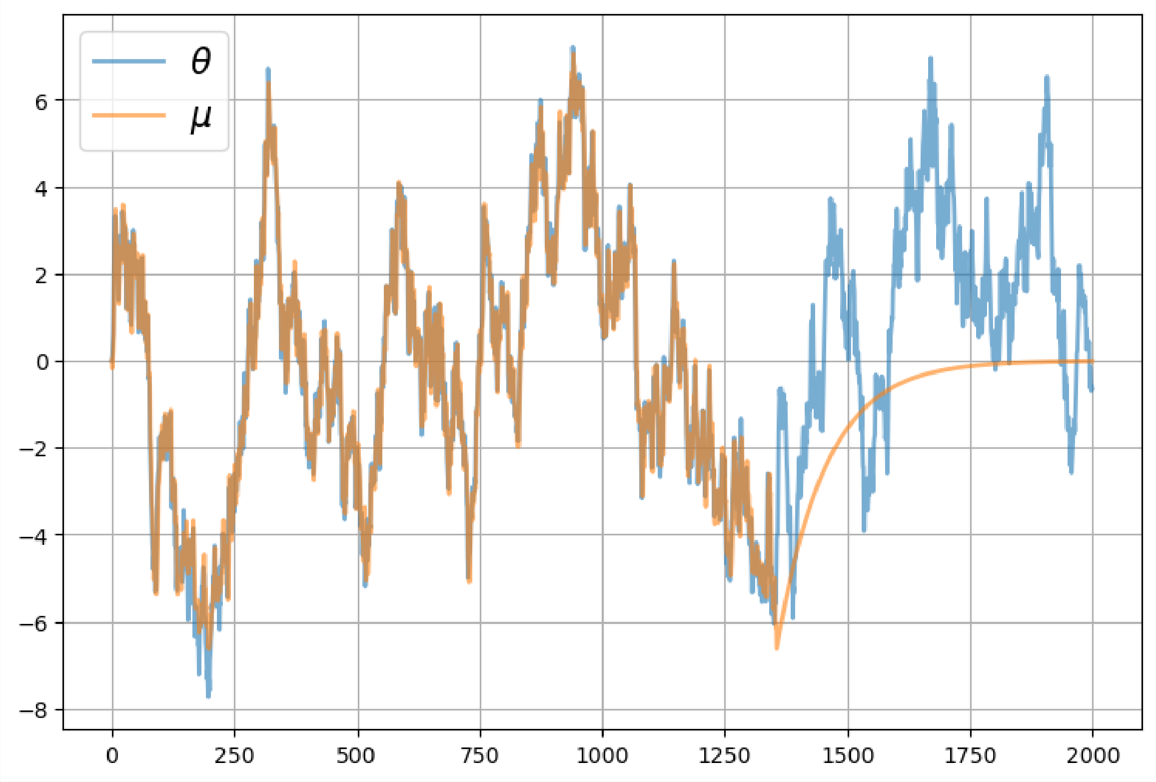
The points where the curves hit the 45 degree lines are the long-run steady states corresponding to each M, if that value of M was to remain fixed. As the number of firms falls, so does the long-run steady state of precision.

Next let's generate time series for beliefs and the aggregates – that is, the number of active firms and average output

```
sim_length=2000

μ_vec = np.empty(sim_length)
θ_vec = np.empty(sim_length)
γ_vec = np.empty(sim_length)
X_vec = np.empty(sim_length)
M_vec = np.empty(sim_length)

μ_vec[0] = econ.μ
γ_vec[0] = econ.γ
θ_vec[0] = 0

w_shocks = np.random.randn(sim_length)

for t in range(sim_length-1):
    X, M = econ.gen_aggregates()
    X_vec[t] = X
    M_vec[t] = M

    econ.update_beliefs(X, M)
    econ.update_θ(w_shocks[t])

    μ_vec[t+1] = econ.μ
    γ_vec[t+1] = econ.γ
    θ_vec[t+1] = econ.θ

# Record final values of aggregates
X, M = econ.gen_aggregates()
X_vec[-1] = X
M_vec[-1] = M
```

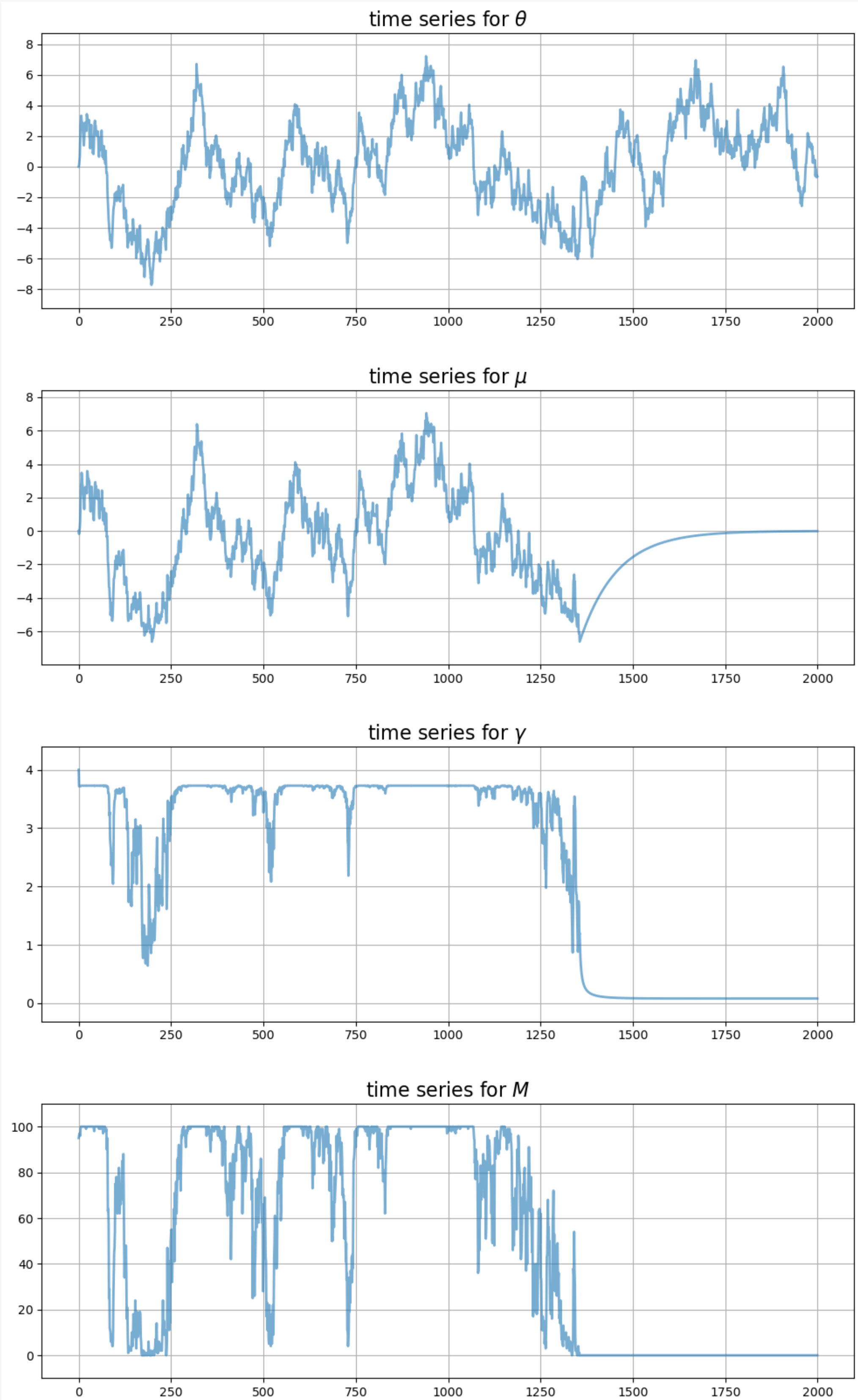
First, let's see how well μ tracks θ in these simulations

```
fig, ax = plt.subplots(figsize=(9, 6))
ax.plot(range(sim_length), θ_vec, alpha=0.6, lw=2, label=r"$\theta$")
ax.plot(range(sim_length), μ_vec, alpha=0.6, lw=2, label=r"$\mu$")
ax.legend(fontsize=16)
ax.grid()
plt.show()
```



Now let's plot the whole thing together

```
fig, axes = plt.subplots(4, 1, figsize=(12, 20))
# Add some spacing
fig.subplots_adjust(hspace=0.3)

series = (θ_vec, μ_vec, γ_vec, M_vec)
names = r'$\theta$', r'$\mu$', r'$\gamma$', r'$M$'

for ax, vals, name in zip(axes, series, names):
    # Determine suitable y limits
    s_max, s_min = max(vals), min(vals)
    s_range = s_max - s_min
    y_max = s_max + s_range * 0.1
    y_min = s_min - s_range * 0.1
    ax.set_ylim(y_min, y_max)
    # Plot series
    ax.plot(range(sim_length), vals, alpha=0.6, lw=2)
    ax.set_title(f"time series for {name}", fontsize=16)
    ax.grid()

plt.show()
```


time series for θ
time series for μ
time series for γ
time series for M

If you run the code above you'll get different plots, of course.

Try experimenting with different parameters to see the effects on the time series.

(It would also be interesting to experiment with non-Gaussian distributions for the shocks, but this is a big exercise since it takes us outside the world of the standard Kalman filter)

CHAPTER

EIGHTYONE

THE AIYAGARI MODEL

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

Contents

In addition to what's included in base Anaconda, we need to install JAX

```
!pip install quantecon jax
```

81.1 Overview

In this lecture, we describe the structure of a class of models that build on work by Truman Bewley [Bewley, 1977].

We begin by discussing an example of a Bewley model due to Rao Aiyagari [Aiyagari, 1994].

The model features

- heterogeneous agents
- a single exogenous vehicle for borrowing and lending
- limits on amounts individual agents may borrow

The Aiyagari model has been used to investigate many topics, including

- precautionary savings and the effect of liquidity constraints [Aiyagari, 1994]
- risk sharing and asset pricing [Heaton and Lucas, 1996]
- the shape of the wealth distribution [Benhabib *et al.*, 2015]
- etc., etc., etc.

81.1.1 Preliminaries

We use the following imports:

```
import quantecon as qe
import matplotlib.pyplot as plt
import jax
import jax.numpy as jnp
from typing import NamedTuple
from scipy.optimize import bisect
```

We will use 64-bit floats with JAX in order to increase precision.

```
jax.config.update("jax_enable_x64", True)
```

We will use the following function to compute stationary distributions of stochastic matrices (for a reference to the algorithm, see p. 88 of Economic Dynamics).

```
@jax.jit
def compute_stationary(P):
    n = P.shape[0]
    I = jnp.identity(n)
    O = jnp.ones((n, n))
    A = I - jnp.transpose(P) + O
    return jnp.linalg.solve(A, jnp.ones(n))
```

81.1.2 References

The primary reference for this lecture is [Aiyagari, 1994].

A textbook treatment is available in chapter 18 of [Ljungqvist and Sargent, 2018].

A continuous time version of the model by SeHyoun Ahn and Benjamin Moll can be found here.

81.2 The Economy

81.2.1 Households

Infinitely lived households / consumers face idiosyncratic income shocks.

A unit interval of *ex-ante* identical households face a common borrowing constraint.

The savings problem faced by a typical household is

$$\max \mathbb{E} \sum_{t=0}^{\infty} \beta^t u(c_t)$$

subject to

$$a_{t+1} + c_t \le wz_t + (1+r)a_t \quad c_t \ge 0, \quad \text{and} \quad a_t \ge -B$$

where

- c_t is current consumption
- a_t is assets
- z_t is an exogenous component of labor income capturing stochastic unemployment risk, etc.
- w is a wage rate
- r is a net interest rate
- B is the maximum amount that the agent is allowed to borrow

The exogenous process $\{z_t\}$ follows a finite state Markov chain with given stochastic matrix P.

The wage and interest rate are fixed over time.

In this simple version of the model, households supply labor inelastically because they do not value leisure.

81.2.2 Firms

Firms produce output by hiring capital and labor.

Firms act competitively and face constant returns to scale.

Since returns to scale are constant, the number of firms does not matter.

Hence we can consider a single (but nonetheless competitive) representative firm.

The firm's output is

$$Y = AK^{\alpha}N^{1-\alpha}$$

where

- A and α are parameters with $A > 0$ and $\alpha \in (0, 1)$
- K is aggregate capital
- N is total labor supply (which is constant in this simple version of the model)

The firm's problem is

$$\max_{K,N} \{AK^{\alpha}N^{1-\alpha} - (r+\delta)K - wN\}$$

The parameter δ is the depreciation rate.

These parameters are stored in the following namedtuple:

```
class Firm(NamedTuple):
    A: float = 1.0     # Total factor productivity
    N: float = 1.0     # Total labor supply
    α: float = 0.33    # Capital share
    δ: float = 0.05    # Depreciation rate
```

From the first-order condition with respect to capital, the firm's inverse demand for capital is

$$r = A\alpha \left(\frac{N}{K}\right)^{1-\alpha} - \delta \tag{81.1}$$

```
def r_given_k(K, firm):
    """
    Inverse demand curve for capital. The interest rate associated with a
    given demand for capital K.
    """
    A, N, α, δ = firm
    return A * α * (N / K)**(1 - α) - δ
```

Using this expression and the firm's first-order condition for labor, we can pin down the equilibrium wage rate as a function of r as

$$w(r) = A(1-\alpha)(A\alpha/(r+\delta))^{\alpha/(1-\alpha)} \tag{81.2}$$

```
def r_to_w(r, firm):
    """
    Equilibrium wages associated with a given interest rate r.
    """
    A, N, α, δ = firm
    return A * (1 - α) * (A * α / (r + δ))**(α / (1 - α))
```

81.2.3 Equilibrium

We construct a **stationary rational expectations equilibrium (SREE)**.

In such an equilibrium

- prices induce behavior that generates aggregate quantities consistent with the prices
- aggregate quantities and prices are constant over time

In more detail, an SREE lists a set of prices, savings and production policies such that

- households want to choose the specified savings policies taking the prices as given
- firms maximize profits taking the same prices as given
- the resulting aggregate quantities are consistent with the prices; in particular, the demand for capital equals the supply
- aggregate quantities (defined as cross-sectional averages) are constant

81.3 Implementation

Let's look at how we might compute such an equilibrium in practice.

Below we provide code to solve the household problem, taking r and w as fixed.

81.3.1 Primitives and operators

We will solve the household problem using value function iteration.

First we set up a `NamedTuple` to store the parameters that define a household asset accumulation problem, as well as the grids used to solve it

```
class Household(NamedTuple):
    β: float              # Discount factor
    a_grid: jnp.ndarray # Asset grid
    z_grid: jnp.ndarray # Exogenous states
    Π: jnp.ndarray      # Transition matrix

def create_household(β=0.96,                      # Discount factor
                     Π=[[0.9, 0.1], [0.1, 0.9]],  # Markov chain
                     z_grid=[0.1, 1.0],           # Exogenous states
                     a_min=1e-10, a_max=12.5,     # Asset grid
                     a_size=100):
    """
    Create a Household namedtuple with custom grids.
    """
    a_grid = jnp.linspace(a_min, a_max, a_size)
    z_grid, Π = map(jnp.array, (z_grid, Π))
    return Household(β=β, a_grid=a_grid, z_grid=z_grid, Π=Π)
```

For now we assume that $u(c) = \log(c)$

```
u = jnp.log
```

Here's a namedtuple that stores the wage rate and interest rate with default values

```
class Prices(NamedTuple):
    r: float = 0.01  # Interest rate
    w: float = 1.0   # Wages
```

Now we set up a vectorized version of the right-hand side of the Bellman equation (before maximization), which is a 3D array representing

$$B(a, z, a') = u(wz + (1 + r)a - a') + \beta \sum_{z'} v(a', z')\Pi(z, z')$$

for all (a, z, a').

```
def B(v, household, prices):
    # Unpack
    β, a_grid, z_grid, Π = household
    a_size, z_size = len(a_grid), len(z_grid)
    r, w = prices

    # Compute current consumption as array c[i, j, ip]
    a  = jnp.reshape(a_grid, (a_size, 1, 1))    # a[i]   ->  a[i, j, ip]
    z  = jnp.reshape(z_grid, (1, z_size, 1))    # z[j]   ->  z[i, j, ip]
    ap = jnp.reshape(a_grid, (1, 1, a_size))    # ap[ip] -> ap[i, j, ip]
    c = w * z + (1 + r) * a - ap

    # Calculate continuation rewards at all combinations of (a, z, ap)
    v = jnp.reshape(v, (1, 1, a_size, z_size)) # v[ip, jp] -> v[i, j, ip, jp]
    Π = jnp.reshape(Π, (1, z_size, 1, z_size)) # Π[j, jp]  -> Π[i, j, ip, jp]
```

(continues on next page)

(continued from previous page)

```
    EV = jnp.sum(v * Π, axis=-1)                # sum over last index jp

    # Compute the right-hand side of the Bellman equation
    return jnp.where(c > 0, u(c) + β * EV, -jnp.inf)
```

The next function computes greedy policies

```
def get_greedy(v, household, prices):
    """
    Computes a v-greedy policy σ, returned as a set of indices. If
    σ[i, j] equals ip, then a_grid[ip] is the maximizer at i, j.
    """
    # argmax over ap
    return jnp.argmax(B(v, household, prices), axis=-1)
```

We define the Bellman operator T, which takes a value function v and returns Tv as given in the Bellman equation

```
def T(v, household, prices):
    """
    The Bellman operator. Takes a value function v and returns Tv.
    """
    return jnp.max(B(v, household, prices), axis=-1)
```

Here's value function iteration, which repeatedly applies the Bellman operator until convergence

```
@jax.jit
def value_function_iteration(household, prices, tol=1e-4, max_iter=10_000):
    """
    Implements value function iteration using a compiled JAX loop.
    """
    β, a_grid, z_grid, Π = household
    a_size, z_size = len(a_grid), len(z_grid)

    def condition_function(loop_state):
        i, v, error = loop_state
        return jnp.logical_and(error > tol, i < max_iter)

    def update(loop_state):
        i, v, error = loop_state
        v_new = T(v, household, prices)
        error = jnp.max(jnp.abs(v_new - v))
        return i + 1, v_new, error

    # Initial loop state
    v_init = jnp.zeros((a_size, z_size))
    loop_state_init = (0, v_init, tol + 1)

    # Run the fixed point iteration
    i, v, error = jax.lax.while_loop(condition_function, update, loop_state_init)

    return get_greedy(v, household, prices)
```

As a first example of what we can do, let's compute and plot an optimal accumulation policy at fixed prices

```
# Create an instance of Household
household = create_household()
prices = Prices()
```

(continues on next page)

(continued from previous page)

```
r, w = prices
print(f"Interest rate: {r}, Wage: {w}")
```

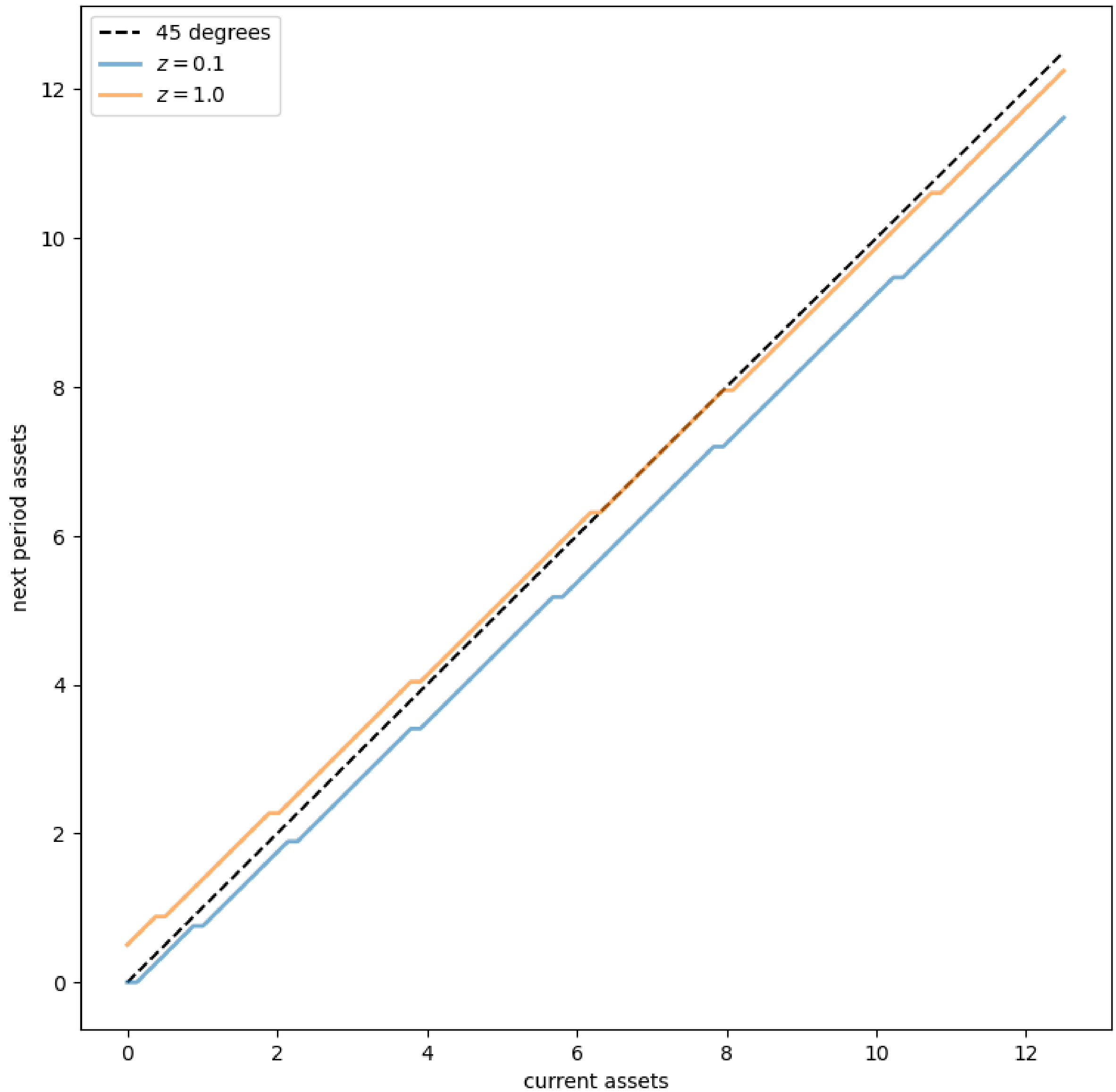
```
Interest rate: 0.01, Wage: 1.0
```

```
with qe.Timer():
    σ_star = value_function_iteration(household, prices).block_until_ready()
```

```
0.52 seconds elapsed
```

The next plot shows asset accumulation policies at different values of the exogenous state

```
β, a_grid, z_grid, Π = household

fig, ax = plt.subplots(figsize=(9, 9))
ax.plot(a_grid, a_grid, 'k--', label="45 degrees")
for j, z in enumerate(z_grid):
    lb = f'$z = {z:.2}$'
    policy_vals = a_grid[σ_star[:, j]]
    ax.plot(a_grid, policy_vals, lw=2, alpha=0.6, label=lb)
    ax.set_xlabel('current assets')
    ax.set_ylabel('next period assets')
ax.legend(loc='upper left')
plt.show()
```



The plot shows asset accumulation policies at different values of the exogenous state.

81.3.2 Capital supply

To start thinking about equilibrium, we need to know how much capital households supply at a given interest rate r.

This quantity can be calculated by taking the stationary distribution of assets under the optimal policy and computing the mean.

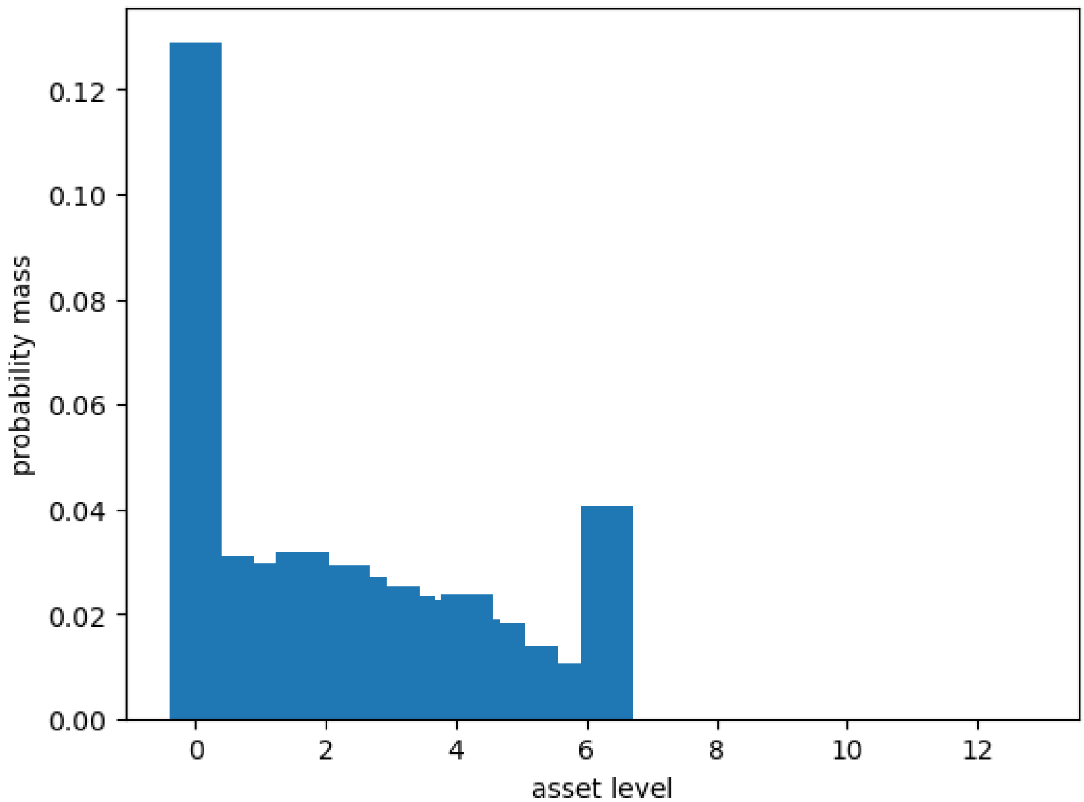
The next function computes the stationary distribution for a given policy σ via the following steps:

- Compute the stationary distribution $\psi = (\psi(a, z))$ of P_σ, which defines the Markov chain of the state (a_t, z_t) under policy σ.
- Sum out z_t to get the marginal distribution for a_t.

```
@jax.jit
def compute_asset_stationary(σ, household):
    # Unpack
    β, a_grid, z_grid, Π = household
    a_size, z_size = len(a_grid), len(z_grid)

    # Construct P_σ as an array of the form P_σ[i, j, ip, jp]
    ap_idx = jnp.arange(a_size)
    ap_idx = jnp.reshape(ap_idx, (1, 1, a_size, 1))
    σ = jnp.reshape(σ, (a_size, z_size, 1, 1))
    A = jnp.where(σ == ap_idx, 1, 0)
    Π = jnp.reshape(Π, (1, z_size, 1, z_size))
    P_σ = A * Π

    # Reshape P_σ into a matrix
    n = a_size * z_size
    P_σ = jnp.reshape(P_σ, (n, n))

    # Get stationary distribution and reshape back onto [i, j] grid
    ψ = compute_stationary(P_σ)
    ψ = jnp.reshape(ψ, (a_size, z_size))

    # Sum along the rows to get the marginal distribution of assets
    ψ_a = jnp.sum(ψ, axis=1)
    return ψ_a
```

Let’s give this a test run.

```
ψ_a = compute_asset_stationary(σ_star, household)

fig, ax = plt.subplots()
ax.bar(household.a_grid, ψ_a)
ax.set_xlabel("asset level")
ax.set_ylabel("probability mass")
plt.show()
```



The distribution should sum to one:

```
ψ_a.sum()
```

```
Array(1., dtype=float64)
```

The next function computes aggregate capital supply by households under policy σ, given wages and interest rates

```
def capital_supply(σ, household):
    """
    Induced level of capital stock under the policy, taking r and w as given.
    """
    β, a_grid, z_grid, Π = household
    ψ_a = compute_asset_stationary(σ, household)
    return float(jnp.sum(ψ_a * a_grid))
```

81.3.3 Equilibrium

We compute a SREE as follows:

1. Set $n = 0$ and start with an initial guess K_0 for aggregate capital.
2. Determine prices r, w from the firm decision problem, given K_n.
3. Compute the optimal savings policy of households given these prices.
4. Compute aggregate capital K_{n+1} as the mean of steady-state capital given this savings policy.
5. If $K_{n+1} \approx K_n$, stop; otherwise, go to step 2.

We can write the sequence of operations in steps 2-4 as

$$K_{n+1} = G(K_n)$$

If K_{n+1} agrees with K_n then we have a SREE.

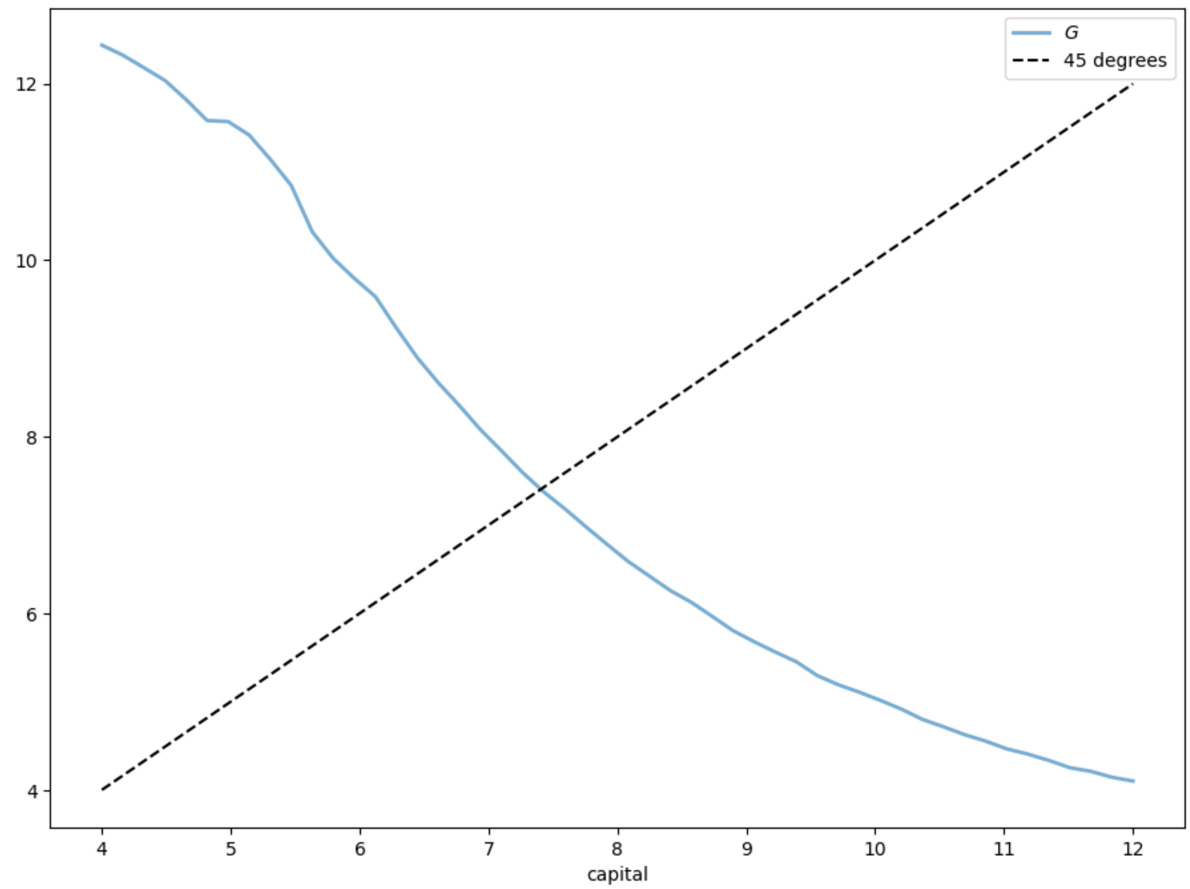
In other words, our problem is to find the fixed point of the one-dimensional map G.

Here's G expressed as a Python function

```
def G(K, firm, household):
    # Get prices r, w associated with K
    r = r_given_k(K, firm)
    w = r_to_w(r, firm)

    # Generate a household object with these prices, compute
    # aggregate capital.
    prices = Prices(r=r, w=w)
    σ_star = value_function_iteration(household, prices)
    return capital_supply(σ_star, household)
```

Let's inspect visually as a first pass

```
num_points = 50
firm = Firm()
household = create_household()
k_vals = jnp.linspace(4, 12, num_points)
out = [G(k, firm, household) for k in k_vals]

fig, ax = plt.subplots(figsize=(11, 8))
ax.plot(k_vals, out, lw=2, alpha=0.6, label='$G$')
ax.plot(k_vals, k_vals, 'k--', label="45 degrees")
ax.set_xlabel('capital')
ax.legend()
plt.show()
```



Now let's compute the equilibrium.

Looking at the figure above, we see that a simple iteration scheme $K_{n+1} = G(K_n)$ will cycle from high to low values, leading to slow convergence.

As a result, we use a damped iteration scheme of the form

$$K_{n+1} = \alpha K_n + (1 - \alpha) G(K_n)$$

```
def compute_equilibrium(firm, household,
                        K0=6, α=0.99, max_iter=1_000, tol=1e-4,
                        print_skip=10, verbose=False):
    n = 0
    K = K0
    error = tol + 1
    while error > tol and n < max_iter:
        new_K = α * K + (1 - α) * G(K, firm, household)
        error = abs(new_K - K)
        K = new_K
        n += 1
        if verbose and n % print_skip == 0:
            print(f"At iteration {n} with error {error}")
    return K, n
```

```
firm = Firm()
household = create_household()
```

(continues on next page)

(continued from previous page)

```
print("\nComputing equilibrium capital stock")
with qe.Timer():
    K_star, n = compute_equilibrium(firm, household, K0=6.0)
print(f"Computed equilibrium {K_star:.5} in {n} iterations")
```

```
Computing equilibrium capital stock
2.26 seconds elapsed
Computed equilibrium 7.4095 in 259 iterations
```

This convergence is not very fast, given how quickly we can solve the household problem.

You can try varying α, but usually this parameter is hard to set a priori.

In the exercises below you will be asked to use bisection instead, which generally performs better.

81.3.4 Supply and demand curves

We can visualize the equilibrium using supply and demand curves.

The following code draws the aggregate supply and demand curves.

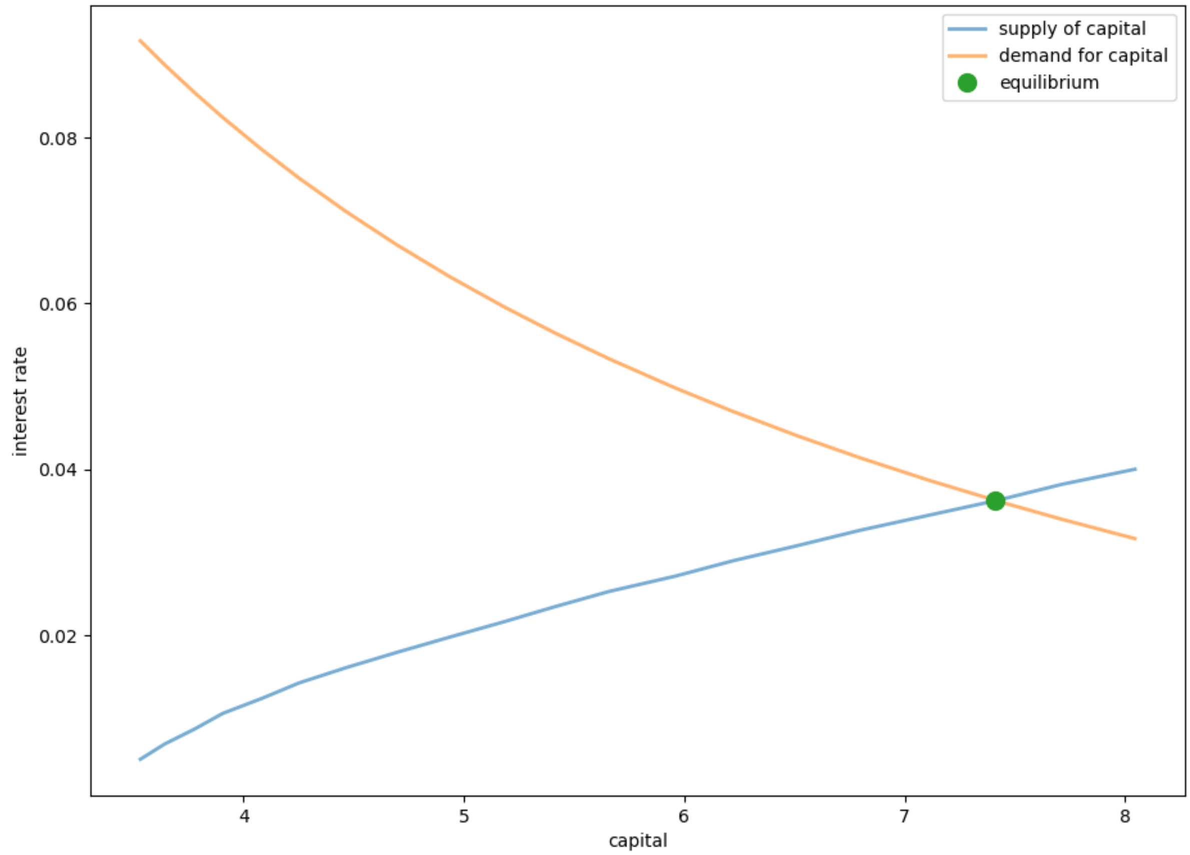
The intersection gives the equilibrium interest rate and capital

```
def prices_to_capital_stock(household, r, firm):
    """
    Map prices to the induced level of capital stock.
    """
    w = r_to_w(r, firm)
    prices = Prices(r=r, w=w)

    # Compute the optimal policy
    σ_star = value_function_iteration(household, prices)

    # Compute capital supply
    return capital_supply(σ_star, household)

# Create a grid of r values to compute demand and supply of capital
num_points = 20
r_vals = jnp.linspace(0.005, 0.04, num_points)

# Compute supply of capital
k_vals = []
for r in r_vals:
    k_vals.append(prices_to_capital_stock(household, r, firm))

# Plot against demand for capital by firms
fig, ax = plt.subplots(figsize=(11, 8))
ax.plot(k_vals, r_vals, lw=2, alpha=0.6,
        label='supply of capital')
ax.plot(k_vals, r_given_k(
        jnp.array(k_vals), firm), lw=2, alpha=0.6,
        label='demand for capital')

# Add marker at equilibrium
r_star = r_given_k(K_star, firm)
ax.plot(K_star, r_star, 'o', markersize=10, label='equilibrium')
```

(continues on next page)

(continued from previous page)

```
ax.set_xlabel('capital')
ax.set_ylabel('interest rate')
ax.legend(loc='upper right')

plt.show()
```



81.4 Exercises

Exercise 81.4.1

Write a new version of `compute_equilibrium` that uses `bisect` from `scipy.optimize` instead of damped iteration.

See if you can make it faster than the previous version.

In `bisect`,

- you should set `xtol=1e-4` to have the same error tolerance as the previous version.
- for the lower and upper bounds of the bisection routine try `a = 1.0` and `b = 20.0`.

Solution

We use bisection to find the zero of the function $h(k) = k - G(k)$

```
def compute_equilibrium_bisect(firm, household, a=1.0, b=20.0):
    K = bisect(lambda k: k - G(k, firm, household), a, b, xtol=1e-4)
    return K

firm = Firm()
household = create_household()
print("\nComputing equilibrium capital stock using bisection")
with qe.Timer():
    K_star = compute_equilibrium_bisect(firm, household)
print(f"Computed equilibrium capital stock {K_star:.5}")
```

```
Computing equilibrium capital stock using bisection
0.18 seconds elapsed
Computed equilibrium capital stock 7.4178
```

The bisection method is faster than the damped iteration scheme.

Exercise 81.4.2

Show how equilibrium capital stock changes with β.

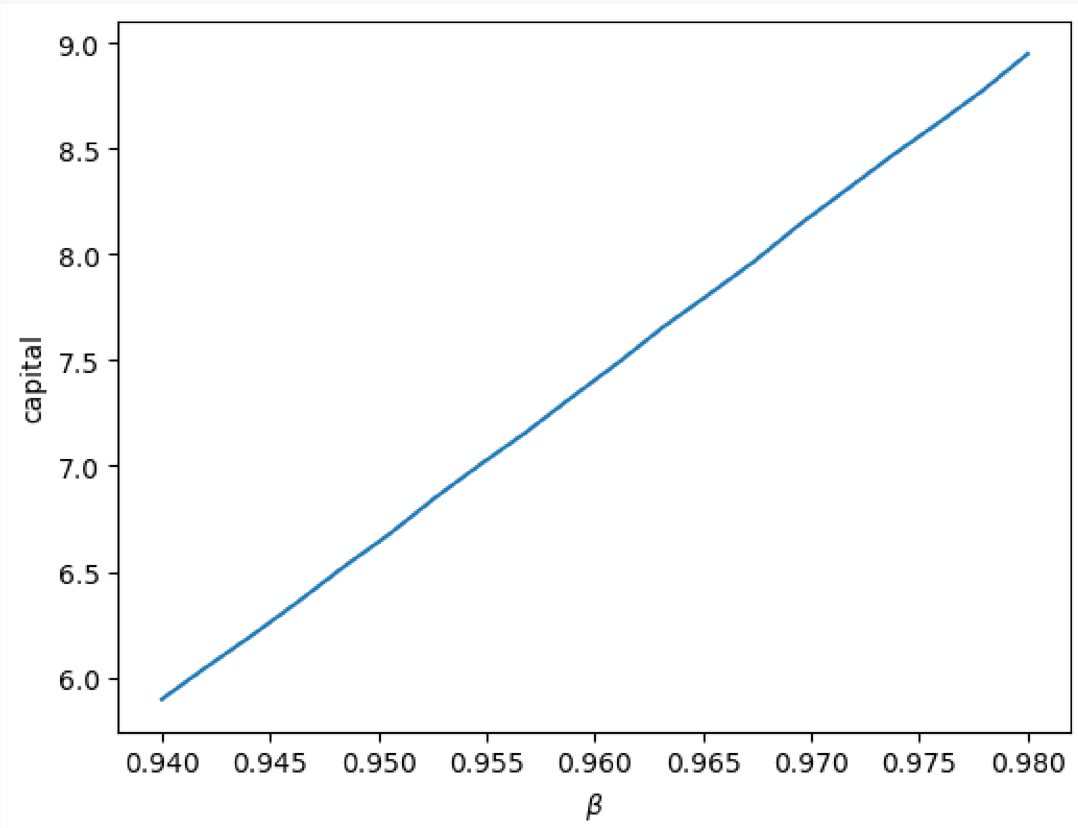
Use the following values of β and plot the relationship you find.

```
β_vals = jnp.linspace(0.94, 0.98, 20)
```

Solution

```
K_vals = []
K = 6.0  # initial guess

for β in β_vals:
    household = create_household(β=β)
    K = compute_equilibrium_bisect(firm, household, 0.5 * K, 1.5 * K)
    print(f"Computed equilibrium {K:.4} at β = {β}")
    K_vals.append(K)

fig, ax = plt.subplots()
ax.plot(β_vals, K_vals, ms=2)
ax.set_xlabel(r'$\beta$')
ax.set_ylabel('capital')
plt.show()
```

```
Computed equilibrium 5.897 at β = 0.94
Computed equilibrium 6.052 at β = 0.9421052631578948
Computed equilibrium 6.202 at β = 0.9442105263157894
Computed equilibrium 6.36 at β = 0.9463157894736841
Computed equilibrium 6.524 at β = 0.9484210526315789
Computed equilibrium 6.679 at β = 0.9505263157894737
Computed equilibrium 6.85 at β = 0.9526315789473683
Computed equilibrium 7.008 at β = 0.9547368421052631
Computed equilibrium 7.16 at β = 0.9568421052631579
Computed equilibrium 7.325 at β = 0.9589473684210527
Computed equilibrium 7.486 at β = 0.9610526315789474
Computed equilibrium 7.657 at β = 0.963157894736842
Computed equilibrium 7.81 at β = 0.9652631578947368
Computed equilibrium 7.967 at β = 0.9673684210526317
Computed equilibrium 8.142 at β = 0.9694736842105263
Computed equilibrium 8.302 at β = 0.971578947368421
Computed equilibrium 8.463 at β = 0.9736842105263158
Computed equilibrium 8.616 at β = 0.9757894736842105
Computed equilibrium 8.773 at β = 0.9778947368421053
Computed equilibrium 8.948 at β = 0.98
```



Exercise 81.4.3

In this lecture, we used value function iteration to solve the household problem.

An alternative is Howard policy iteration (HPI), which is discussed in detail in Dynamic Programming.

HPI can be faster than VFI for some problems because it uses fewer but more computationally intensive iterations.

Your task is to implement Howard policy iteration and compare the results with value function iteration.

Key concepts you'll need:

Howard policy iteration requires computing the value v_σ of a policy σ, defined as:

$$v_\sigma = (I - \beta P_\sigma)^{-1} r_\sigma$$

where r_σ is the reward vector under policy σ, and P_σ is the transition matrix induced by σ.

To solve this, you'll need to:

1. Compute current rewards $r_\sigma(a, z) = u((1 + r)a + wz - \sigma(a, z))$
2. Set up the linear operator R_σ where $(R_\sigma v)(a, z) = v(a, z) - \beta \sum_{z'} v(\sigma(a, z), z')\Pi(z, z')$
3. Solve $v_\sigma = R_\sigma^{-1} r_\sigma$ using `jax.scipy.sparse.linalg.bicgstab`

You can use the `get_greedy` function that's already defined in this lecture.

Implement the following Howard policy iteration routine:

```
def howard_policy_iteration(household, prices,
                            tol=1e-4, max_iter=10_000, verbose=False):
    """
    Howard policy iteration routine.
    """
    # Your code here
    pass
```

Once implemented, compute the equilibrium capital stock using HPI and verify that it produces approximately the same result as VFI at the default parameter values.

Solution

First, we need to implement the helper functions for Howard policy iteration.

The following function computes the array r_σ which gives current rewards given policy σ:

```
def compute_r_σ(σ, household, prices):
    """
    Compute current rewards at each i, j under policy σ. In particular,

        r_σ[i, j] = u((1 + r)a[i] + wz[j] - a'[ip])

    when ip = σ[i, j].
    """
    # Unpack
    β, a_grid, z_grid, Π = household
    a_size, z_size = len(a_grid), len(z_grid)
    r, w = prices

    # Compute r_σ[i, j]
    a = jnp.reshape(a_grid, (a_size, 1))
    z = jnp.reshape(z_grid, (1, z_size))
    ap = a_grid[σ]
    c = (1 + r) * a + w * z - ap
    r_σ = u(c)

    return r_σ
```

The linear operator R_σ is defined as:

```
def R_σ(v, σ, household):
    # Unpack
    β, a_grid, z_grid, Π = household
    a_size, z_size = len(a_grid), len(z_grid)

    # Set up the array v[σ[i, j], jp]
    zp_idx = jnp.arange(z_size)
    zp_idx = jnp.reshape(zp_idx, (1, 1, z_size))
    σ = jnp.reshape(σ, (a_size, z_size, 1))
    V = v[σ, zp_idx]

    # Expand Π[j, jp] to Π[i, j, jp]
    Π = jnp.reshape(Π, (1, z_size, z_size))

    # Compute and return v[i, j] - β Σ_jp v[σ[i, j], jp] * Π[j, jp]
    return v - β * jnp.sum(V * Π, axis=-1)
```

The next function computes the lifetime value of a given policy:

```
def get_value(σ, household, prices):
    """
    Get the lifetime value of policy σ by computing

        v_σ = R_σ^{-1} r_σ
    """
    r_σ = compute_r_σ(σ, household, prices)

    # Reduce R_σ to a function in v
    _R_σ = lambda v: R_σ(v, σ, household)

    # Compute v_σ = R_σ^{-1} r_σ using an iterative routine.
    return jax.scipy.sparse.linalg.bicgstab(_R_σ, r_σ)[0]
```

Now we can implement Howard policy iteration:

```
@jax.jit
def howard_policy_iteration(household, prices, tol=1e-4, max_iter=10_000):
    """
    Howard policy iteration routine using a compiled JAX loop.
    """
    β, a_grid, z_grid, Π = household
    a_size, z_size = len(a_grid), len(z_grid)

    def condition_function(loop_state):
        i, σ, v_σ, error = loop_state
        return jnp.logical_and(error > tol, i < max_iter)

    def update(loop_state):
        i, σ, v_σ, error = loop_state
        σ_new = get_greedy(v_σ, household, prices)
        v_σ_new = get_value(σ_new, household, prices)
        error = jnp.max(jnp.abs(v_σ_new - v_σ))
        return i + 1, σ_new, v_σ_new, error

    # Initial loop state
    σ_init = jnp.zeros((a_size, z_size), dtype=int)
    v_σ_init = get_value(σ_init, household, prices)
    loop_state_init = (0, σ_init, v_σ_init, tol + 1)
```

```
    # Run the fixed point iteration
    i, σ, v_σ, error = jax.lax.while_loop(condition_function, update, loop_state_
↪init)

    return σ
```

Now let's create a modified version of the G function that uses HPI:

```
def G_hpi(K, firm, household):
    # Get prices r, w associated with K
    r = r_given_k(K, firm)
    w = r_to_w(r, firm)

    # Generate prices and compute aggregate capital using HPI.
    prices = Prices(r=r, w=w)
    σ_star = howard_policy_iteration(household, prices)
    return capital_supply(σ_star, household)
```

And compute the equilibrium using HPI:

```
def compute_equilibrium_bisect_hpi(firm, household, a=1.0, b=20.0):
    K = bisect(lambda k: k - G_hpi(k, firm, household), a, b, xtol=1e-4)
    return K

firm = Firm()
household = create_household()
print("\nComputing equilibrium capital stock using HPI")
with qe.Timer():
    K_star_hpi = compute_equilibrium_bisect_hpi(firm, household)
print(f"Computed equilibrium capital stock with HPI: {K_star_hpi:.5}")
print(f"Previous equilibrium capital stock with VFI: {K_star:.5}")
print(f"Difference: {abs(K_star_hpi - K_star):.6}")
```

```
Computing equilibrium capital stock using HPI
1.67 seconds elapsed
Computed equilibrium capital stock with HPI: 7.4172
Previous equilibrium capital stock with VFI: 7.4178
Difference: 0.000579834
```

The results show that both methods produce approximately the same equilibrium, confirming that HPI is a valid alternative to VFI.

CHAPTER

EIGHTYTWO

A LONG-LIVED, HETEROGENEOUS AGENT, OVERLAPPING GENERATIONS MODEL

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

In addition to what's in Anaconda, this lecture will need the following library

```
!pip install jax
```

82.1 Overview

This lecture describes an overlapping generations model with these features:

- A competitive equilibrium with incomplete markets determines prices and quantities
- Agents live many periods as in [Auerbach and Kotlikoff, 1987]
- Agents receive idiosyncratic labor productivity shocks that cannot be fully insured as in [Aiyagari, 1994]
- Government fiscal policy instruments include tax rates, debt, and transfers as in chapter 2 of [Auerbach and Kotlikoff, 1987] and *Transitions in an Overlapping Generations Model*
- Among other equilibrium objects, a competitive equilibrium determines a sequence of cross-section densities of heterogeneous agents' consumptions, labor incomes, and savings

We use the model to study:

- How fiscal policies affect different generations
- How market incompleteness promotes precautionary savings
- How life-cycle savings and buffer-stock savings motives interact
- How fiscal policies redistribute resources across and within generations

As prerequisites for this lecture, we recommend two quantecon lectures:

1. Discrete State Dynamic Programming
2. *Transitions in an Overlapping Generations Model*

as well as the optional reading *The Aiyagari Model*

As usual, let's start by importing some Python modules

```
from collections import namedtuple
import numpy as np
import matplotlib.pyplot as plt
import jax.numpy as jnp
import jax.scipy as jsp
import jax
```

82.2 Environment

We start by introducing the economic environment we are operating in.

82.2.1 Demographics and time

We work in discrete time indexed by $t = 0, 1, 2, ...$.

Each agent lives for $J = 50$ periods and faces no mortality risk.

We index age by $j = 0, 1, ..., 49$, and the population size remains fixed at $1/J$.

82.2.2 Individuals' state variables

Each agent i of age j at time t is characterized by two state variables: asset holdings $a_{i,j,t}$ and idiosyncratic labor productivity $\gamma_{i,j,t}$.

The idiosyncratic labor productivity process follows a two-state Markov chain that takes values γ_l and γ_h with transition matrix Π.

Newborn agents begin with an initial distribution $\pi = [0.5, 0.5]$ over these productivity states.

82.2.3 Labor supply

An agent with productivity $\gamma_{i,j,t}$ supplies $l(j)\gamma_{i,j,t}$ efficiency units of labor.

$l(j)$ is a deterministic age-specific labor efficiency units profile.

An agent's effective labor supply depends on a life-cycle efficiency profile and an idiosyncratic stochastic process.

82.2.4 Initial conditions

Newborns start with zero assets $a_{i,0,t} = 0$.

Initial idiosyncratic productivities are drawn from distribution π.

Agents leave no bequests and have terminal value function $V_J(a) = 0$.

82.3 Production

A representative firm operates a constant returns to scale Cobb-Douglas production:

$$Y_t = Z_t K_t^{\alpha} L_t^{1-\alpha}$$

where:

- K_t is aggregate capital
- L_t is aggregate efficiency units of labor
- Z_t is total factor productivity
- α is the capital share

82.4 Government

The government follows a fiscal policy that includes debt, taxes, transfers, and government spending.

The government issues one-period debt D_t to finance its operations and collects revenues through a flat-rate tax τ_t on both labor and capital income.

The government also implements age-specific lump-sum taxes or transfers $\delta_{j,t}$ that can redistribute resources across different age groups.

Additionally, it makes government purchases G_t for public goods and services.

The government budget constraint at time t is

$$D_{t+1} - D_t = r_t D_t + G_t - T_t$$

where total tax revenues T_t satisfy

$$T_t = \tau_t w_t L_t + \tau_t r_t (D_t + K_t) + \sum_j \delta_{j,t}$$

82.5 Activities in factor markets

At each time $t \geq 0$, agents supply labor and capital.

82.5.1 Age-specific labor supplies

Agents of age $j \in \{0, 1, ..., J-1\}$ supply labor according to:

- Their deterministic age-efficiency profile $l(j)$
- Their current idiosyncratic productivity shock $\gamma_{i,j,t}$

Each agent supplies $l(j)\gamma_{i,j,t}$ effective units of labor and earns a competitive wage w_t per effective unit, subject to a flat tax rate τ_t on labor earnings.

82.5.2 Asset market participation

Summarizing activities in the asset market, all agents, regardless of age $j \in \{0, 1, ..., J-1\}$, can:

- Hold assets $a_{i,j,t}$ (subject to borrowing constraints)
- Earn a risk-free one-period return r_t on savings
- Pay capital income taxes at flat rate τ_t
- Receive or pay age-specific transfers $\delta_{j,t}$

82.5.3 Key features

Lifecycle patterns shape economic behavior across ages:

- Labor productivity varies systematically with age according to the profile $l(j)$, while asset holdings typically follow a lifecycle pattern of accumulation during working years and decumulation during retirement.
- Age-specific fiscal transfers $\delta_{j,t}$ redistribute resources across generations.

Within-cohort heterogeneity creates dispersion among agents of the same age:

- Agents of the same age differ in their asset holdings $a_{i,j,t}$ due to different histories of idiosyncratic productivity shocks, their current productivities $\gamma_{i,j,t}$, and consequently their labor incomes and financial wealth.

Cross-cohort interactions determine equilibrium outcomes through market aggregation:

- All cohorts participate together in factor markets, with asset supplies from all cohorts determining aggregate capital and effective labor supplies from all cohorts determining aggregate labor.
- Equilibrium prices reflect both lifecycle and redistributive forces.

82.6 Representative firm's problem

A representative firm chooses capital and effective labor to maximize profits

$$\max_{K,L} Z_t K_t^{\alpha} L_t^{1-\alpha} - r_t K_t - w_t L_t$$

First-order necessary conditions imply that

$$w_t = (1-\alpha) Z_t (K_t/L_t)^{\alpha}$$

and

$$r_t = \alpha Z_t (K_t/L_t)^{\alpha-1}$$

82.7 Households' problems

A household's value function satisfies a Bellman equation

$$V_{j,t}(a,\gamma) = \max_{c,a'}\{u(c) + \beta\mathbb{E}[V_{j+1,t+1}(a',\gamma')]\}$$

where maximization is subject to

$$c + a' = (1 + r_t(1-\tau_t))a + (1-\tau_t)w_t l(j)\gamma - \delta_{j,t}$$

$$c \geq 0$$

and a terminal condition $V_{J,t}(a,\gamma) = 0$

82.8 Population dynamics

The joint probability density function $\mu_{j,t}(a,\gamma)$ of asset holdings and idiosyncratic labor productivity evolves according to

- For newborns $(j = 0)$:

$$\mu_{0,t+1}(a',\gamma') = \begin{cases} \pi(\gamma') & \text{if } a' = 0, \\ 0, & \text{otherwise} \end{cases}$$

- For other cohorts:

$$\mu_{j+1,t+1}(a',\gamma') = \int \mathbf{1}_{\sigma_{j,t}(a,\gamma)=a'}\Pi(\gamma,\gamma')\mu_{j,t}(a,\gamma)d(a,\gamma)$$

where $\sigma_{j,t}(a,\gamma)$ is the optimal saving policy function.

82.9 Equilibrium

An equilibrium consists of:

- Value functions $V_{j,t}$
- Policy functions $\sigma_{j,t}$
- Joint probability distributions $\mu_{j,t}$
- Prices r_t, w_t
- Government policies $\tau_t, D_t, \delta_{j,t}, G_t$

that satisfy the following conditions

- Given prices and government policies, value and policy functions solve households' problems
- Given prices, the representative firm maximizes profits
- Government budget constraints are satisfied
- Markets clear:
  - Asset market: $K_t = \sum_j \int a\mu_{j,t}(a,\gamma)d(a,\gamma) - D_t$
  - Labor market: $L_t = \sum_j \int l(j)\gamma\mu_{j,t}(a,\gamma)d(a,\gamma)$

Relative to the model presented in *Transitions in an Overlapping Generations Model*, the present model adds

- Heterogeneity within generations due to productivity shocks
- A precautionary savings motive
- More re-distributional effects
- More complicated transition dynamics

82.10 Implementation

Using tools in Discrete State Dynamic Programming, we solve our model by combining value function iteration with equilibrium price determination.

A sensible approach is to nest a discrete DP solver inside an outer loop that searches for market-clearing prices.

For a candidate sequence of prices interest rates r_t and wages w_t, we can solve individual households' dynamic programming problems using either value function iteration or policy iteration to obtain optimal policy functions.

We then deduce associated stationary joint probability distributions of asset holdings and idiosyncratic labor efficiency units for each age cohort.

This will give us an aggregate capital supply (from household savings) and a labor supply (from the age-efficiency profile and productivity shocks).

We can then compare these with capital and labor demand from firms, compute deviations between factor market supplies and demands, then update price guesses until we find market-clearing prices.

To construct transition dynamics, we can compute sequences of time-varying prices by using *backward induction* to compute value and policy functions, and *forward iteration* for the distributions of agents across states:

1. Outer loop (market clearing)
   - Guess initial prices (r_t, w_t)
   - Iterate until asset and labor markets clear
   - Use firms' first-order necessary conditions to update prices
2. Inner loop (individual dynamic programming)
   - For each age cohort:
     - Discretize asset and productivity state space
     - Use value function iteration or policy iteration
     - Solve for optimal savings policies
     - Compute stationary distributions
3. Aggregation
   - Sum across individual states within each cohort
   - Sum across cohorts both
     - Aggregate capital supply, and
     - Aggregate effective labor supply
   - Take into account population weights $1/J$
4. Transition dynamics

- Backward induction:
  - Start from final steady state
  - Solve sequence of value functions
- Forward iteration:
  - Start from initial distribution
  - Track cohort distributions over time
- Market clearing in each period:
  - Solve for price sequences
  - Update until all markets clear in all periods

We start coding by defining helper functions that describe preferences, firms, and government budget constraints.

```
ϕ, k_bar = 0., 0.

@jax.jit
def V_bar(a):
    "Terminal value function depending on the asset holding."

    return - ϕ * (a - k_bar) ** 2
```

```
ν = 0.5

@jax.jit
def u(c):
    "Utility from consumption."

    return c ** (1 - ν) / (1 - ν)

l1, l2, l3 = 0.5, 0.05, -0.0008

@jax.jit
def l(j):
    "Age-specific wage profile."

    return l1 + l2 * j + l3 * j ** 2
```

Let's define a `Firm` namedtuple that contains parameters governing the production technology.

```
Firm = namedtuple("Firm", ("α", "Z"))

def create_firm(α=0.3, Z=1):

    return Firm(α=α, Z=Z)
```

```
firm = create_firm()
```

The following helper functions link aggregates (K, L) and prices (w, r) that emerge from the representative firm's first-order necessary conditions.

```
@jax.jit
def KL_to_r(K, L, firm):
```

(continues on next page)

(continued from previous page)

```
    α, Z = firm

    return Z * α * (K / L) ** (α - 1)

@jax.jit
def KL_to_w(K, L, firm):

    α, Z = firm

    return Z * (1 - α) * (K / L) ** α
```

We use a function `find_τ` to find flat tax rates that balance the government budget constraint given other policy variables that include debt levels, government spending, and transfers.

```
@jax.jit
def find_τ(policy, price, aggs):

    D, D_next, G, δ = policy
    r, w = price
    K, L = aggs

    num = r * D + G - D_next + D - δ.sum(axis=-1)
    denom = w * L + r * (D + K)

    return num / denom
```

We use a namedtuple `Household` to store parameters that characterize households' problems.

```
Household = namedtuple("Household", ("j_grid", "a_grid", "y_grid",
                                     "Π", "β", "init_μ", "VJ"))

def create_household(
        a_min=0., a_max=10, a_size=200,
        Π=[[0.9, 0.1], [0.1, 0.9]],
        y_grid=[0.5, 1.5],
        β=0.96, J=50
    ):

    j_grid = jnp.arange(J)

    a_grid = jnp.linspace(a_min, a_max, a_size)

    y_grid, Π = map(jnp.array, (y_grid, Π))
    y_size = len(y_grid)

    # Population distribution of new borns
    init_μ = jnp.zeros((a_size * y_size))

    # Newborns are endowed with zero asset
    # and equal probability of y
    init_μ = init_μ.at[:y_size].set(1 / y_size)

    # Terminal value V_bar(a)
    VJ = jnp.empty(a_size * y_size)
    for a_i in range(a_size):
        a = a_grid[a_i]
        VJ = VJ.at[a_i*y_size:(a_i+1)*y_size].set(V_bar(a))
```

(continues on next page)

(continued from previous page)

```
    return Household(j_grid=j_grid, a_grid=a_grid, γ_grid=γ_grid,
                     Π=Π, β=β, init_μ=init_μ, VJ=VJ)
```

```
hh = create_household()
```

We apply discrete state dynamic programming tools.

Initial steps involve preparing rewards and transition matrices R and Q for our discretized Bellman equations.

```
@jax.jit
def populate_Q(household):

    j_grid, a_grid, γ_grid, Π, β, init_μ, VJ = household

    num_state = a_grid.size * γ_grid.size
    num_action = a_grid.size

    Q = jsp.linalg.block_diag(*[Π]*a_grid.size)
    Q = Q.reshape((num_state, num_action, γ_grid.size))
    Q = jnp.tile(Q, a_grid.size).T

    return Q

@jax.jit
def populate_R(j, r, w, τ, δ, household):

    j_grid, a_grid, γ_grid, Π, β, init_μ, VJ = household

    num_state = a_grid.size * γ_grid.size
    num_action = a_grid.size

    a = jnp.reshape(a_grid, (a_grid.size, 1, 1))
    γ = jnp.reshape(γ_grid, (1, γ_grid.size, 1))
    ap = jnp.reshape(a_grid, (1, 1, a_grid.size))
    c = (1 + r*(1-τ)) * a + (1-τ) * w * l(j) * γ - δ[j] - ap

    return jnp.reshape(jnp.where(c > 0, u(c), -jnp.inf),
                       (num_state, num_action))
```

82.11 Computing a steady state

We first compute a steady state.

Given guesses of prices and taxes, we can use backwards induction to solve for value functions and optimal consumption and saving policies at all ages.

The function `backwards_opt` solves for optimal values by applying the discretized bellman operator backwards.

We use `jax.lax.scan` to facilitate sequential and recurrent computations efficiently.

```
@jax.jit
def backwards_opt(prices, taxes, household, Q):

    r, w = prices
```

(continues on next page)

(continued from previous page)

```
    τ, δ = taxes

    j_grid, a_grid, γ_grid, Π, β, init_μ, VJ = household
    J = j_grid.size

    num_state = a_grid.size * γ_grid.size
    num_action = a_grid.size

    def bellman_operator_j(V_next, j):
        "Solve household optimization problem at age j given Vj+1"

        Rj = populate_R(j, r, w, τ, δ, household)
        vals = Rj + β * Q.dot(V_next)
        σ_j = jnp.argmax(vals, axis=1)
        V_j = vals[jnp.arange(num_state), σ_j]

        return V_j, (V_j, σ_j)

    js = jnp.arange(J-1, -1, -1)
    init_V = VJ

    # Iterate from age J to 1
    _, outputs = jax.lax.scan(bellman_operator_j, init_V, js)
    V, σ = outputs
    V = V[::-1]
    σ = σ[::-1]

    return V, σ
```

```
r, w = 0.05, 1
τ, δ = 0.15, np.zeros(hh.j_grid.size)

Q = populate_Q(hh)
```

```
V, σ = backwards_opt([r, w], [τ, δ], hh, Q)
```

Let's time the computation with `block_until_ready()` to ensure that all JAX operations are complete

```
%time backwards_opt([r, w], [τ, δ], hh, Q)[0].block_until_ready();
```

```
CPU times: user 1.51 ms, sys: 0 ns, total: 1.51 ms
Wall time: 50.5 ms
```

From the optimal consumption and saving choices by each cohort, we can compute a joint probability distribution of asset levels and idiosyncratic productivity levels in a steady state.

```
@jax.jit
def popu_dist(σ, household, Q):

    j_grid, a_grid, γ_grid, Π, β, init_μ, VJ = household

    J = hh.j_grid.size
    num_state = hh.a_grid.size * hh.γ_grid.size

    def update_popu_j(μ_j, j):
```

(continues on next page)

(continued from previous page)

```
        "Update population distribution from age j to j+1"

        Qσ = Q[jnp.arange(num_state), σ[j]]
        μ_next = μ_j @ Qσ

        return μ_next, μ_next

    js = jnp.arange(J-1)

    # iterate from age 1 to J
    _, μ = jax.lax.scan(update_popu_j, init_μ, js)
    μ = jnp.concatenate([init_μ[jnp.newaxis], μ], axis=0)

    return μ
```

```
μ = popu_dist(σ, hh, Q)
```

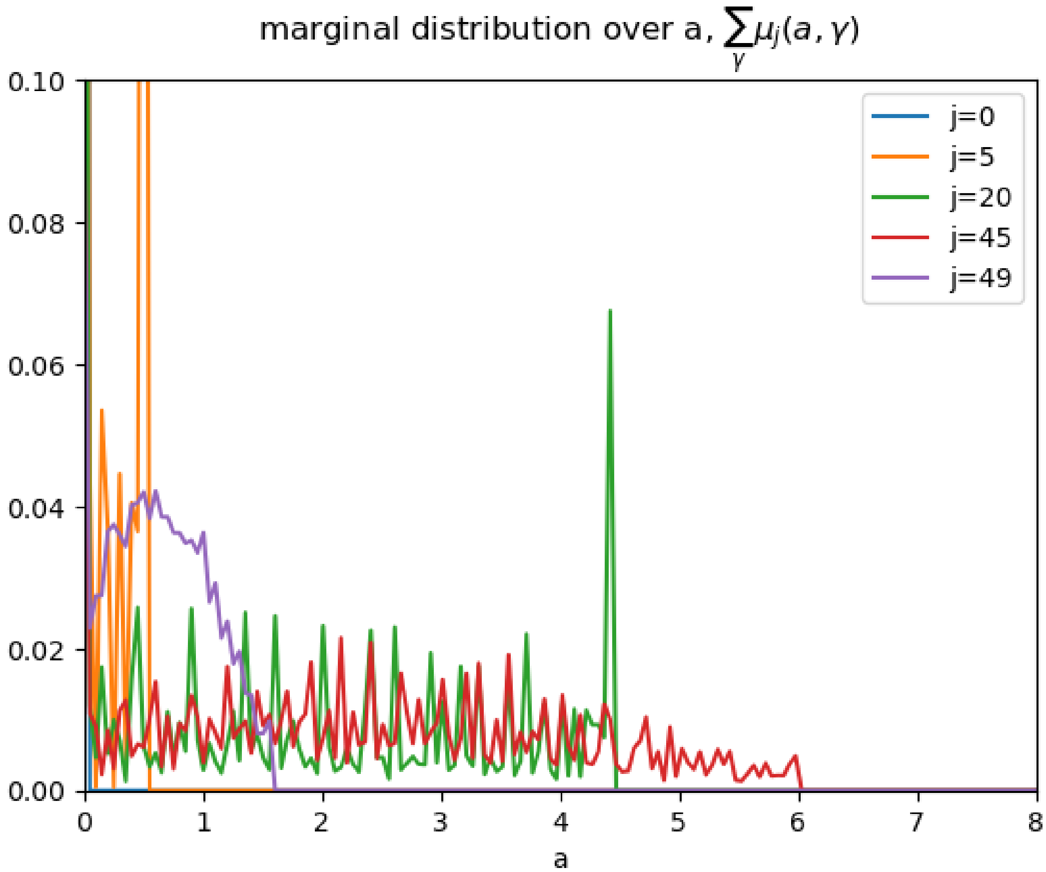
Let's time the computation

```
%time popu_dist(σ, hh, Q)[0].block_until_ready();
```

```
CPU times: user 41.4 ms, sys: 1.09 ms, total: 42.5 ms
Wall time: 55.8 ms
```

Below we plot the marginal distribution of savings for each age group.

```
for j in [0, 5, 20, 45, 49]:
    plt.plot(hh.a_grid, jnp.sum(μ[j].reshape((hh.a_grid.size, hh.y_grid.size)),␣
 ↪axis=1), label=f'j={j}')

plt.legend()
plt.xlabel('a')

plt.title(r'marginal distribution over a, $\sum_\gamma \mu_j(a, \gamma)$')
plt.xlim([0, 8])
plt.ylim([0, 0.1])

plt.show()
```



These marginal distributions confirm that new agents enter the economy with no asset holdings.

- the blue $j = 0$ distribution has mass only at $a = 0$.

As agents age, at first they gradually accumulate assets.

- the orange $j = 5$ distribution puts positive mass on positive but low asset levels
- the green $j = 20$ distribution puts positive mass on a much wider range of asset levels.
- the red $j = 45$ distribution is even wider

At a later age, they gradually deplete their asset holdings.

- the purple $j = 49$ distribution illustrates this

At the end of life, they will have drawn down all of their assets.

Let's now look at age-specific optimal saving policies that generate the preceding marginal distributions of assets at different ages.

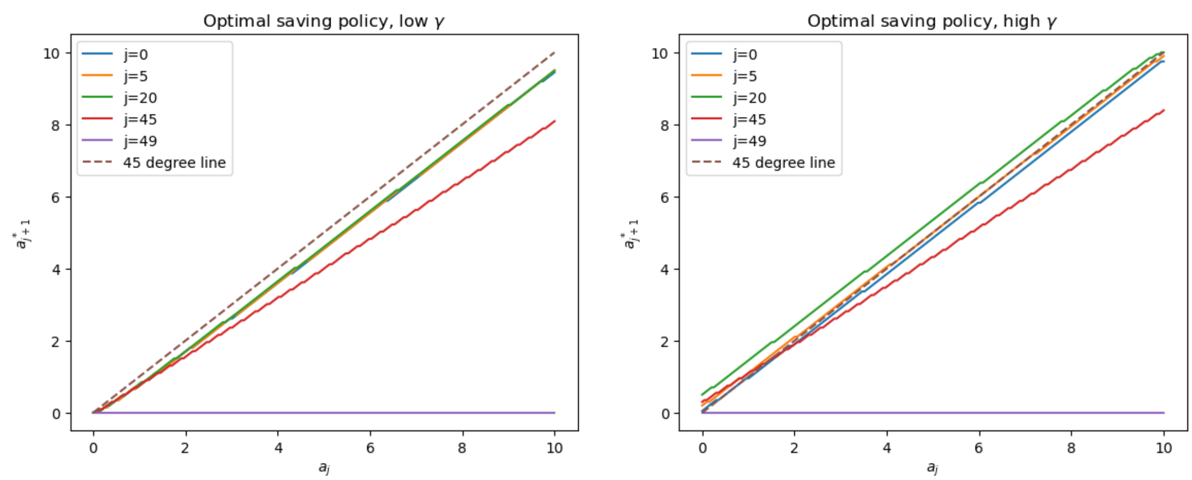
We'll plot some saving functions with the following Python code.

```
σ_reshaped = σ.reshape(hh.j_grid.size, hh.a_grid.size, hh.y_grid.size)
j_labels = [f'j={j}' for j in [0, 5, 20, 45, 49]]

fig, axs = plt.subplots(1, 2, figsize=(14, 5))

axs[0].plot(hh.a_grid, hh.a_grid[σ_reshaped[[0, 5, 20, 45, 49], :, 0].T])
axs[0].plot(hh.a_grid, hh.a_grid, '--')
```

(continues on next page)

(continued from previous page)

```
axs[0].set_xlabel("$a_{j}$")
axs[0].set_ylabel("$a^*_{j+1}$")
axs[0].legend(j_labels+['45 degree line'])
axs[0].set_title(r"Optimal saving policy, low $\gamma$")

axs[1].plot(hh.a_grid, hh.a_grid[σ_reshaped[[0, 5, 20, 45, 49], :, 1].T])
axs[1].plot(hh.a_grid, hh.a_grid, '--')
axs[1].set_xlabel("$a_{j}$")
axs[1].set_ylabel("$a^*_{j+1}$")
axs[1].legend(j_labels+['45 degree line'])
axs[1].set_title(r"Optimal saving policy, high $\gamma$")

plt.show()
```



From an implied stationary population distribution, we can compute the aggregate labor supply L and private savings A.

```
@jax.jit
def compute_aggregates(μ, household):

    j_grid, a_grid, γ_grid, Π, β, init_μ, VJ = household

    J, a_size, γ_size = j_grid.size, a_grid.size, γ_grid.size

    μ = μ.reshape((J, hh.a_grid.size, hh.γ_grid.size))

    # Compute private savings
    a = a_grid.reshape((1, a_size, 1))
    A = (a * μ).sum() / J

    γ = γ_grid.reshape((1, 1, γ_size))
    lj = l(j_grid).reshape((J, 1, 1))
    L = (lj * γ * μ).sum() / J

    return A, L
```

```
A, L = compute_aggregates(μ, hh)
A, L
```

```
(Array(1.8594263, dtype=float32), Array(1.0781993, dtype=float32))
```

The capital stock in this economy equals $A - D$.

```
D = 0
K = A - D
```

The firm's optimality conditions imply interest rate r and wage rate w.

```
KL_to_r(K, L, firm), KL_to_w(K, L, firm)
```

```
(Array(0.20485441, dtype=float32), Array(0.8243317, dtype=float32))
```

The implied prices (r, w) differ from our guesses, so we must update our guesses and iterate until we find a fixed point.

This is our outer loop.

```
@jax.jit
def find_ss(household, firm, pol_target, Q, tol=1e-6, verbose=False):

    j_grid, a_grid, γ_grid, Π, β, init_μ, VJ = household
    J = j_grid.size
    num_state = a_grid.size * γ_grid.size

    D, G, δ = pol_target

    # Initial guesses of prices
    r, w = 0.05, 1.

    # Initial guess of τ
    τ = 0.15

    def cond_fn(state):
        "The convergence criteria."

        V, σ, μ, K, L, r, w, τ, D, G, δ, r_old, w_old = state

        error = (r - r_old) ** 2 + (w - w_old) ** 2

        return error > tol

    def body_fn(state):
        "The main body of iteration."

        V, σ, μ, K, L, r, w, τ, D, G, δ, r_old, w_old = state
        r_old, w_old, τ_old = r, w, τ

        # Household optimal decisions and values
        V, σ = backwards_opt([r, w], [τ, δ], hh, Q)

        # Compute the stationary distribution
        μ = popu_dist(σ, hh, Q)

        # Compute aggregates
        A, L = compute_aggregates(μ, hh)
        K = A - D
```

(continues on next page)

(continued from previous page)

```
        # Update prices
        r, w = KL_to_r(K, L, firm), KL_to_w(K, L, firm)

        # Find τ
        D_next = D
        τ = find_τ([D, D_next, G, δ],
                   [r, w],
                   [K, L])

        r = (r + r_old) / 2
        w = (w + w_old) / 2

        return V, σ, μ, K, L, r, w, τ, D, G, δ, r_old, w_old

    # Initial state
    V = jnp.empty((J, num_state), dtype=float)
    σ = jnp.empty((J, num_state), dtype=int)
    μ = jnp.empty((J, num_state), dtype=float)

    K, L = 1., 1.
    initial_state = (V, σ, μ, K, L, r, w, τ, D, G, δ, r-1, w-1)
    V, σ, μ, K, L, r, w, τ, D, G, δ, _, _ = jax.lax.while_loop(
                                    cond_fn, body_fn, initial_state)

    return V, σ, μ, K, L, r, w, τ, D, G, δ
```

```
ss1 = find_ss(hh, firm, [0, 0.1, np.zeros(hh.j_grid.size)], Q, verbose=True)
```

Let's time the computation

```
%time find_ss(hh, firm, [0, 0.1, np.zeros(hh.j_grid.size)], Q)[0].block_until_ready();
```

```
CPU times: user 664 ms, sys: 12.4 ms, total: 676 ms
Wall time: 739 ms
```

```
hh_out_ss1 = ss1[:3]
quant_ss1 = ss1[3:5]
price_ss1 = ss1[5:7]
policy_ss1 = ss1[7:11]
```

```
# V, σ, μ
V_ss1, σ_ss1, μ_ss1 = hh_out_ss1
```

```
# K, L
K_ss1, L_ss1 = quant_ss1

K_ss1, L_ss1
```

```
(Array(6.6221957, dtype=float32), Array(1.0781994, dtype=float32))
```

```
# r, w
r_ss1, w_ss1 = price_ss1

r_ss1, w_ss1
```

```
(Array(0.08430456, dtype=float32), Array(1.2056923, dtype=float32))
```

```
# τ, D, G, δ
τ_ss1, D_ss1, G_ss1, δ_ss1 = policy_ss1

τ_ss1, D_ss1, G_ss1, δ_ss1
```

```
(Array(0.05380344, dtype=float32),
 Array(0, dtype=int32, weak_type=True),
 Array(0.1, dtype=float32, weak_type=True),
 Array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
 ↪dtype=float32))
```

82.12 Transition dynamics

We compute transition dynamics using a function `path_iteration`.

In an outer loop, we iterate over guesses of prices and taxes.

In an inner loop, we compute the optimal consumption and saving choices by each cohort j in each time t, then find the implied evolution of the joint distribution of assets and productivities.

We then update our guesses of prices and taxes given the aggregate labor supply and capital stock in the economy.

We use `solve_backwards` to solve for optimal saving choices given price and tax sequences and `simulate_forward` to compute the evolution of the joint distributions.

We require two steady states as inputs: the initial steady state to provide the initial condition for `simulate_forward`, and the final steady state to provide continuation values for `solve_backwards`.

```
@jax.jit
def bellman_operator(prices, taxes, V_next, household, Q):

    r, w = prices
    τ, δ = taxes

    j_grid, a_grid, γ_grid, Π, β, init_μ, VJ = household
    J = j_grid.size

    num_state = a_grid.size * γ_grid.size
    num_action = a_grid.size

    def bellman_operator_j(j):
        Rj = populate_R(j, r, w, τ, δ, household)
        vals = Rj + β * Q.dot(V_next[j+1])
        σ_j = jnp.argmax(vals, axis=1)
        V_j = vals[jnp.arange(num_state), σ_j]

        return V_j, σ_j

    V, σ = jax.vmap(bellman_operator_j, (0,))(jnp.arange(J-1))

    # The last life stage
```

(continues on next page)

(continued from previous page)

```
    j = J-1
    Rj = populate_R(j, r, w, τ, δ, household)
    vals = Rj + β * Q.dot(VJ)
    σ = jnp.concatenate([σ, jnp.argmax(vals, axis=1)[jnp.newaxis]])
    V = jnp.concatenate([V, vals[jnp.arange(num_state), σ[j]][jnp.newaxis]])

    return V, σ
```

```
@jax.jit
def solve_backwards(V_ss2, σ_ss2, household, firm, price_seq, pol_seq, Q):

    j_grid, a_grid, γ_grid, Π, β, init_μ, VJ = household
    J = j_grid.size
    num_state = a_grid.size * γ_grid.size

    τ_seq, D_seq, G_seq, δ_seq = pol_seq
    r_seq, w_seq = price_seq

    T = r_seq.size

    def solve_backwards_t(V_next, t):

        prices = (r_seq[t], w_seq[t])
        taxes = (τ_seq[t], δ_seq[t])
        V, σ = bellman_operator(prices, taxes, V_next, household, Q)

        return V, (V,σ)

    ts = jnp.arange(T-2, -1, -1)
    init_V = V_ss2

    _, outputs = jax.lax.scan(solve_backwards_t, init_V, ts)
    V_seq, σ_seq = outputs
    V_seq = V_seq[::-1]
    σ_seq = σ_seq[::-1]

    V_seq = jnp.concatenate([V_seq, V_ss2[jnp.newaxis]])
    σ_seq = jnp.concatenate([σ_seq, σ_ss2[jnp.newaxis]])

    return V_seq, σ_seq
```

```
@jax.jit
def population_evolution(σt, μt, household, Q):

    j_grid, a_grid, γ_grid, Π, β, init_μ, VJ = household

    J = hh.j_grid.size
    num_state = hh.a_grid.size * hh.γ_grid.size

    def population_evolution_j(j):

        Qσ = Q[jnp.arange(num_state), σt[j]]
        μ_next = μt[j] @ Qσ

        return μ_next

```

(continues on next page)

(continued from previous page)

```
    μ_next = jax.vmap(population_evolution_j, (0,))(jnp.arange(J-1))
    μ_next = jnp.concatenate([init_μ[jnp.newaxis], μ_next])

    return μ_next
```

```
@jax.jit
def simulate_forwards(σ_seq, D_seq, μ_ss1, K_ss1, L_ss1, household, Q):

    j_grid, a_grid, γ_grid, Π, β, init_μ, VJ = household

    J, num_state = μ_ss1.shape

    T = σ_seq.shape[0]

    def simulate_forwards_t(μ, t):

        μ_next = population_evolution(σ_seq[t], μ, household, Q)

        A, L = compute_aggregates(μ_next, household)
        K = A - D_seq[t+1]

        return μ_next, (μ_next, K, L)

    ts = jnp.arange(T-1)
    init_μ = μ_ss1

    _, outputs = jax.lax.scan(simulate_forwards_t, init_μ, ts)
    μ_seq, K_seq, L_seq = outputs

    μ_seq = jnp.concatenate([μ_ss1[jnp.newaxis], μ_seq])
    K_seq = jnp.concatenate([K_ss1[jnp.newaxis], K_seq])
    L_seq = jnp.concatenate([L_ss1[jnp.newaxis], L_seq])

    return μ_seq, K_seq, L_seq
```

The following algorithm describes the path iteration procedure:

Algorithm 82.12.1 (AK-Aiyagari transition path algorithm)

Inputs Given initial steady state ss_1, final steady state ss_2, time horizon T, and policy sequences (D, G, δ)

Output Compute equilibrium transition paths for value functions V, policy functions σ, distributions μ, and prices (r, w, τ)

1. Initialize from steady states:
   - $(V_1, \sigma_1, \mu_1) \leftarrow ss_1$ *(Initial steady state)*
   - $(V_2, \sigma_2, \mu_2) \leftarrow ss_2$ *(Final steady state)*
   - $(r, w, \tau) \leftarrow initialize\_prices(T)$ *(Linear interpolation)*
   - $error \leftarrow \infty, i \leftarrow 0$
2. **While** $error > \varepsilon$ or $i \leq max\_iter$:
   1. $i \leftarrow i + 1$
   2. $(r_{\text{old}}, w_{\text{old}}, \tau_{\text{old}}) \leftarrow (r, w, \tau)$

3. **Backward induction:** For $t \in [T, 1]$:
   - For $j \in [0, J-1]$ *(age groups)*:
     - $V[t, j] \leftarrow \max_{a'}\{u(c) + \beta\mathbb{E}[V[t+1, j+1]]\}$
     - $\sigma[t, j] \leftarrow \arg\max_{a'}\{u(c) + \beta\mathbb{E}[V[t+1, j+1]]\}$
4. **Forward simulation:** For $t \in [1, T]$:
   - $\mu[t] \leftarrow \Gamma(\sigma[t], \mu[t-1])$ *(Distribution evolution)*
   - $K[t] \leftarrow \int a \, d\mu[t] - D[t]$ *(Aggregate capital)*
   - $L[t] \leftarrow \int l(j)\gamma \, d\mu[t]$ *(Aggregate labor)*
   - $r[t] \leftarrow \alpha Z(K[t]/L[t])^{\alpha-1}$ *(Interest rate)*
   - $w[t] \leftarrow (1-\alpha)Z(K[t]/L[t])^{\alpha}$ *(Wage rate)*
   - $\tau[t] \leftarrow solve\_budget(r[t], w[t], K[t], L[t], D[t], G[t])$
5. Compute convergence metric:
   - $error \leftarrow \|r - r_{\text{old}}\| + \|w - w_{\text{old}}\| + \|\tau - \tau_{\text{old}}\|$
6. Update prices with dampening:
   - $r \leftarrow \lambda r + (1-\lambda) r_{\text{old}}$
   - $w \leftarrow \lambda w + (1-\lambda) w_{\text{old}}$
   - $\tau \leftarrow \lambda \tau + (1-\lambda) \tau_{\text{old}}$

3. **Return** $(V, \sigma, \mu, r, w, \tau)$

```python
def path_iteration(ss1, ss2, pol_target, household, firm, Q, tol=1e-4, verbose=False):

    # Starting point: initial steady state
    V_ss1, σ_ss1, μ_ss1 = ss1[:3]
    K_ss1, L_ss1 = ss1[3:5]
    r_ss1, w_ss1 = ss1[5:7]
    τ_ss1, D_ss1, G_ss1, δ_ss1 = ss1[7:11]

    # Ending point: converging new steady state
    V_ss2, σ_ss2, μ_ss2 = ss2[:3]
    K_ss2, L_ss2 = ss2[3:5]
    r_ss2, w_ss2 = ss2[5:7]
    τ_ss2, D_ss2, G_ss2, δ_ss2 = ss2[7:11]

    # The given policies: D, G, δ
    D_seq, G_seq, δ_seq = pol_target
    T = G_seq.shape[0]

    # Initial guesses of prices
    r_seq = jnp.linspace(0, 1, T) * (r_ss2 - r_ss1) + r_ss1
    w_seq = jnp.linspace(0, 1, T) * (w_ss2 - w_ss1) + w_ss1

    # Initial guess of policy
    τ_seq = jnp.linspace(0, 1, T) * (τ_ss2 - τ_ss1) + τ_ss1

    error = 1
    num_iter = 0
```

(continues on next page)

(continued from previous page)

```
    if verbose:
        fig, axs = plt.subplots(1, 3, figsize=(14, 3))
        axs[0].plot(jnp.arange(T), r_seq)
        axs[1].plot(jnp.arange(T), w_seq)
        axs[2].plot(jnp.arange(T), τ_seq, label=f'iter {num_iter}')

    while error > tol:
        # Repeat until finding the fixed point

        r_old, w_old, τ_old = r_seq, w_seq, τ_seq

        pol_seq = (τ_seq, D_seq, G_seq, δ_seq)
        price_seq = (r_seq, w_seq)

        # Solve optimal policies backwards
        V_seq, σ_seq = solve_backwards(
            V_ss2, σ_ss2, hh, firm, price_seq, pol_seq, Q)

        # Compute population evolution forwards
        μ_seq, K_seq, L_seq = simulate_forwards(
            σ_seq, D_seq, μ_ss1, K_ss1, L_ss1, household, Q)

        # Update prices by aggregate capital and labor supply
        r_seq = KL_to_r(K_seq, L_seq, firm)
        w_seq = KL_to_w(K_seq, L_seq, firm)

        # Find taxes that balance the government budget constraint
        τ_seq = find_τ([D_seq[:-1], D_seq[1:], G_seq, δ_seq],
                       [r_seq, w_seq],
                       [K_seq, L_seq])

        # Distance between new and old guesses
        error = jnp.sum((r_old - r_seq) ** 2) + \
                jnp.sum((w_old - w_seq) ** 2) + \
                jnp.sum((τ_old - τ_seq) ** 2)

        num_iter += 1
        if verbose:
            print(f"Iteration {num_iter:3d}: error = {error:.6e}")
            axs[0].plot(jnp.arange(T), r_seq)
            axs[1].plot(jnp.arange(T), w_seq)
            axs[2].plot(jnp.arange(T), τ_seq, label=f'iter {num_iter}')

        r_seq = (r_seq + r_old) / 2
        w_seq = (w_seq + w_old) / 2
        τ_seq = (τ_seq + τ_old) / 2

    if verbose:
        axs[0].set_xlabel('t')
        axs[1].set_xlabel('t')
        axs[2].set_xlabel('t')

        axs[0].set_title('r')
        axs[1].set_title('w')
        axs[2].set_title('τ')
```

(continues on next page)

(continued from previous page)

```
        axs[2].legend(loc='center left', bbox_to_anchor=(1, 0.5))

    return V_seq, σ_seq, μ_seq, K_seq, L_seq, r_seq, w_seq, \
            τ_seq, D_seq, G_seq, δ_seq
```

We can now compute equilibrium transitions that are ignited by fiscal policy reforms.

82.13 Experiment 1: Immediate tax cut

Assume that the government cuts the tax rate and immediately balances its budget by issuing debt.

At $t = 0$, the government unexpectedly announces an immediate tax cut.

From $t = 0$ to 19, the government issues debt, so debt D_{t+1} increases linearly for 20 periods.

The government sets a target for its new debt level $D_{20} = D_0 + 1 = \bar{D} + 1$.

Government spending $\bar{G}$ and transfers $\bar{\delta}_j$ remain constant.

The government adjusts τ_t to balance the budget along the transition.

We want to compute the equilibrium transition path.

Our first step is to prepare appropriate policy variable arrays `D_seq`, `G_seq`, `δ_seq`

We'll compute a `τ_seq` that balances government budgets.

```
T = 150

D_seq = jnp.ones(T+1) * D_ss1
D_seq = D_seq.at[:21].set(D_ss1 + jnp.linspace(0, 1, 21))
D_seq = D_seq.at[21:].set(D_seq[20])

G_seq = jnp.ones(T) * G_ss1

δ_seq = jnp.repeat(δ_ss1, T).reshape((T, δ_ss1.size))
```

In order to iterate the path, we need to first find its destination, which is the new steady state under the new fiscal policy.

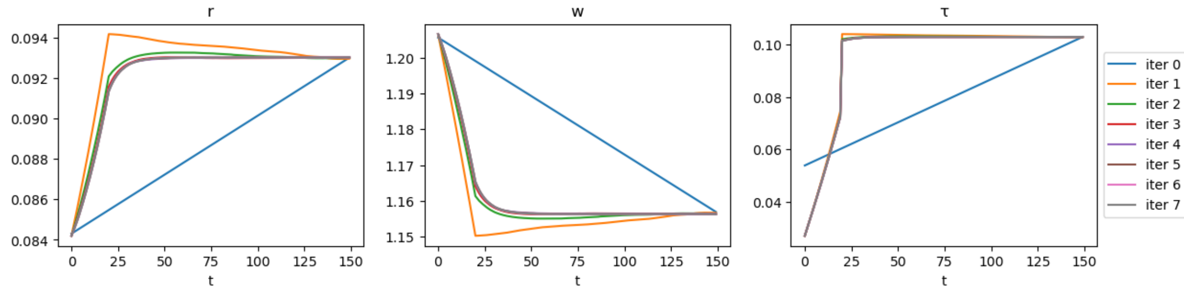
```
ss2 = find_ss(hh, firm, [D_seq[-1], G_seq[-1], δ_seq[-1]], Q)
```

We can use `path_iteration` to find equilibrium transition dynamics.

Setting the key argument `verbose=True` tells the function `path_iteration` to display convergence information.

```
paths = path_iteration(ss1, ss2, [D_seq, G_seq, δ_seq], hh, firm, Q, verbose=True)
```

```
Iteration   1: error = 2.075169e-01
Iteration   2: error = 3.733911e-02
Iteration   3: error = 7.672485e-03
Iteration   4: error = 1.819943e-03
Iteration   5: error = 4.734877e-04
Iteration   6: error = 1.216737e-04
Iteration   7: error = 2.955517e-05
```



Having successfully computed transition dynamics, let's study them.

```
V_seq, σ_seq, μ_seq = paths[:3]
K_seq, L_seq = paths[3:5]
r_seq, w_seq = paths[5:7]
τ_seq, D_seq, G_seq, δ_seq = paths[7:11]
```

```
ap = hh.a_grid[σ_seq[0]]
```

```
j = jnp.reshape(hh.j_grid, (hh.j_grid.size, 1, 1))
lj = l(j)
a = jnp.reshape(hh.a_grid, (1, hh.a_grid.size, 1))
y = jnp.reshape(hh.y_grid, (1, 1, hh.y_grid.size))
```

```
t = 0

ap = hh.a_grid[σ_seq[t]]
δ = δ_seq[t].reshape((hh.j_grid.size, 1, 1))

inc = (1 + r_seq[t]*(1-τ_seq[t])) * a \
        + (1-τ_seq[t]) * w_seq[t] * lj * y - δ
inc = inc.reshape((hh.j_grid.size, hh.a_grid.size * hh.y_grid.size))

c = inc - ap

c_mean0 = (c * μ_seq[t]).sum(axis=1)
```

We care about how the policy change affects consumption across cohorts and across time.

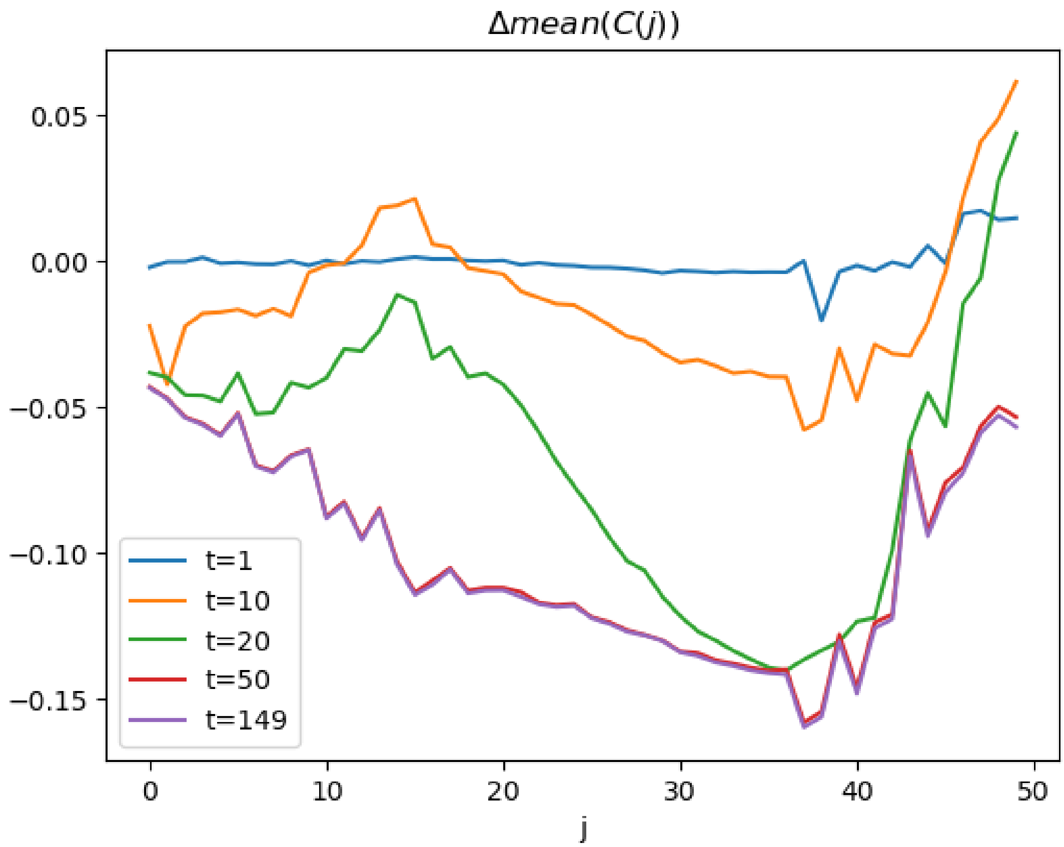
We can study age-specific average consumption levels.

```
for t in [1, 10, 20, 50, 149]:

    ap = hh.a_grid[σ_seq[t]]
    δ = δ_seq[t].reshape((hh.j_grid.size, 1, 1))

    inc = (1 + r_seq[t]*(1-τ_seq[t])) * a + (1-τ_seq[t]) * w_seq[t] * lj * y - δ
    inc = inc.reshape((hh.j_grid.size, hh.a_grid.size * hh.y_grid.size))

    c = inc - ap

    c_mean = (c * μ_seq[t]).sum(axis=1)

    plt.plot(range(hh.j_grid.size), c_mean-c_mean0, label=f't={t}')

plt.legend()
```

(continues on next page)

(continued from previous page)

```
plt.xlabel(r'j')
plt.title(r'$\Delta mean(C(j))$')
plt.show()
```



To summarize the transition, we can plot paths as we did in *Transitions in an Overlapping Generations Model*.

But unlike the setup in that two-period lived overlapping generations model, we no longer have representative old and young agents.

- now we have 50 cohorts of different ages at each time

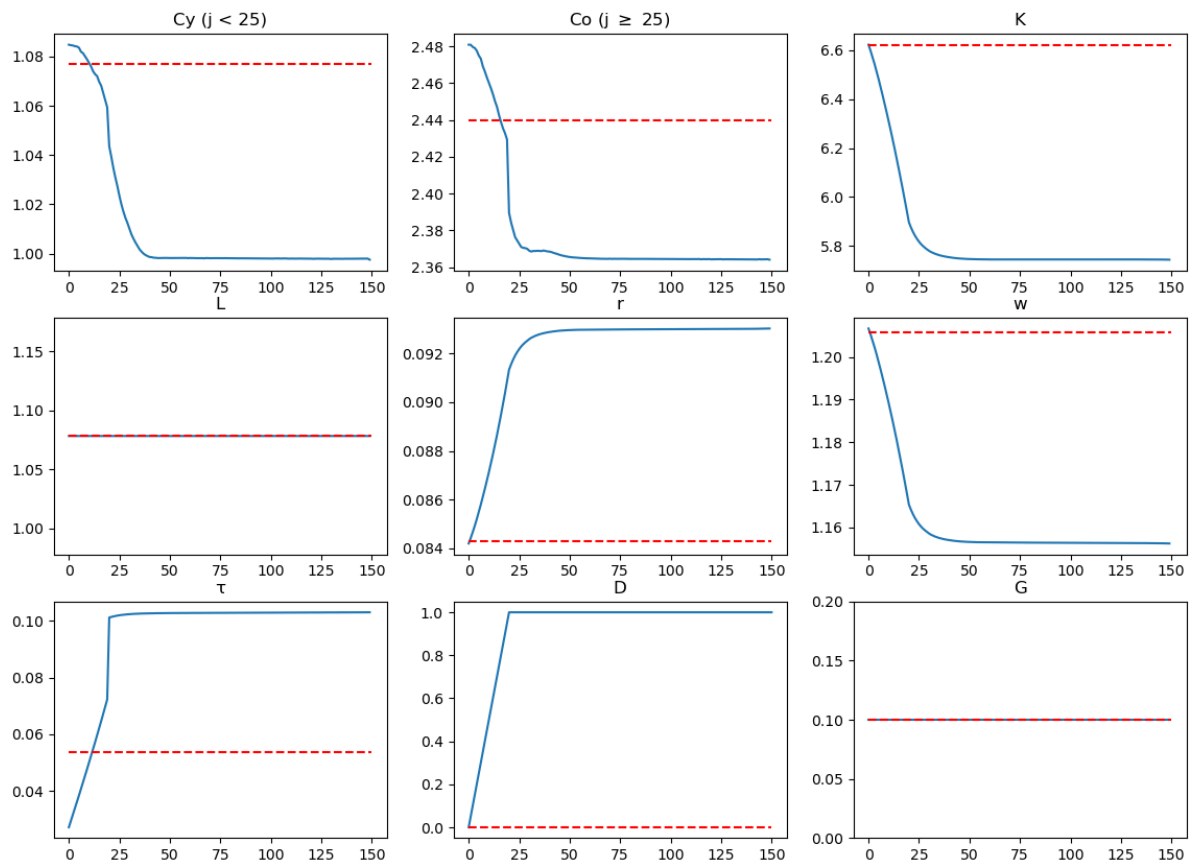
To proceed, we construct two age groups of equal size – young and old.

- at age 25, someone moves from being young to becoming old

```
ap = hh.a_grid[σ_ss1]
J = hh.j_grid.size
δ = δ_ss1.reshape((hh.j_grid.size, 1, 1))

inc = (1 + r_ss1*(1-τ_ss1)) * a + (1-τ_ss1) * w_ss1 * lj * γ - δ
inc = inc.reshape((hh.j_grid.size, hh.a_grid.size * hh.γ_grid.size))

c = inc - ap

Cy_ss1 = (c[:J//2] * μ_ss1[:J//2]).sum() / (J // 2)
Co_ss1 = (c[J//2:] * μ_ss1[J//2:]).sum() / (J // 2)
```

```
T = σ_seq.shape[0]
J = σ_seq.shape[1]

Cy_seq = np.empty(T)
Co_seq = np.empty(T)

for t in range(T):
    ap = hh.a_grid[σ_seq[t]]
    δ = δ_seq[t].reshape((hh.j_grid.size, 1, 1))

    inc = (1 + r_seq[t]*(1-τ_seq[t])) * a + (1-τ_seq[t]) * w_seq[t] * lj * γ - δ
    inc = inc.reshape((hh.j_grid.size, hh.a_grid.size * hh.γ_grid.size))

    c = inc - ap

    Cy_seq[t] = (c[:J//2] * μ_seq[t, :J//2]).sum() / (J // 2)
    Co_seq[t] = (c[J//2:] * μ_seq[t, J//2:]).sum() / (J // 2)
```

```
fig, axs = plt.subplots(3, 3, figsize=(14, 10))

# Cy (j=0-24)
axs[0, 0].plot(Cy_seq)
axs[0, 0].hlines(Cy_ss1, 0, T, color='r', linestyle='--')
axs[0, 0].set_title('Cy (j < 25)')

# Cy (j=25-49)
axs[0, 1].plot(Co_seq)
axs[0, 1].hlines(Co_ss1, 0, T, color='r', linestyle='--')
axs[0, 1].set_title(r'Co (j $\geq$ 25)')

names = ['K', 'L', 'r', 'w', 'τ', 'D', 'G']
for i in range(len(names)):
    i_var = i + 3
    i_axes = i + 2

    row_i = i_axes // 3
    col_i = i_axes % 3

    axs[row_i, col_i].plot(paths[i_var])
    axs[row_i, col_i].hlines(ss1[i_var], 0, T, color='r', linestyle='--')
    axs[row_i, col_i].set_title(names[i])

# ylims
axs[1, 0].set_ylim([ss1[4]-0.1, ss1[4]+0.1])
axs[2, 2].set_ylim([ss1[9]-0.1, ss1[9]+0.1])

plt.show()
```



Now let's compute the mean and variance of consumption conditional on age at each time t.

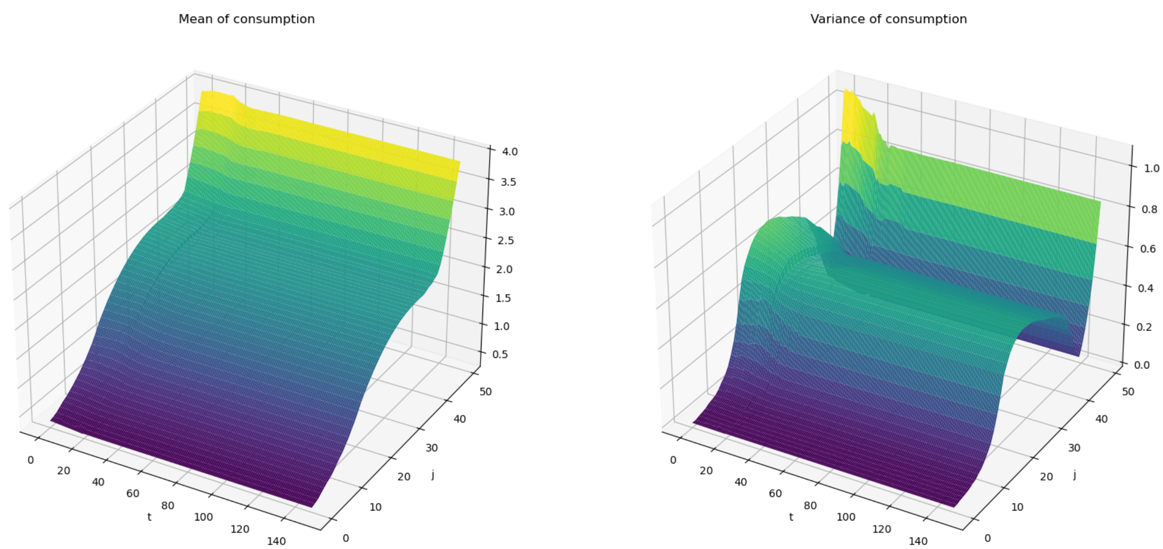
```
Cmean_seq = np.empty((T, J))
Cvar_seq = np.empty((T, J))

for t in range(T):
    ap = hh.a_grid[σ_seq[t]]
    δ = δ_seq[t].reshape((hh.j_grid.size, 1, 1))

    inc = (1 + r_seq[t]*(1-τ_seq[t])) * a + (1-τ_seq[t]) * w_seq[t] * lj * γ - δ
    inc = inc.reshape((hh.j_grid.size, hh.a_grid.size * hh.γ_grid.size))

    c = inc - ap

    Cmean_seq[t] = (c * μ_seq[t]).sum(axis=1)
    Cvar_seq[t] = ((c - Cmean_seq[t].reshape((J, 1))) ** 2 * μ_seq[t]).sum(axis=1)
```

```
J_seq, T_range = np.meshgrid(np.arange(J), np.arange(T))

fig = plt.figure(figsize=[20, 20])

# Plot the consumption mean over age and time
ax1 = fig.add_subplot(121, projection='3d')
ax1.plot_surface(T_range, J_seq, Cmean_seq, rstride=1, cstride=1,
                 cmap='viridis', edgecolor='none')
ax1.set_title(r"Mean of consumption")
ax1.set_xlabel(r"t")
```

(continues on next page)

(continued from previous page)

```
ax1.set_ylabel(r"j")

# plot the consumption variance over age and time
ax2 = fig.add_subplot(122, projection='3d')
ax2.plot_surface(T_range, J_seq, Cvar_seq, rstride=1, cstride=1,
                 cmap='viridis', edgecolor='none')
ax2.set_title(r"Variance of consumption")
ax2.set_xlabel(r"t")
ax2.set_ylabel(r"j")

plt.show()
```



82.14 Experiment 2: Preannounced tax cut

Now the government announces a permanent tax rate cut at time 0 but implements it only after 20 periods.

We will use the same key toolkit `path_iteration`.

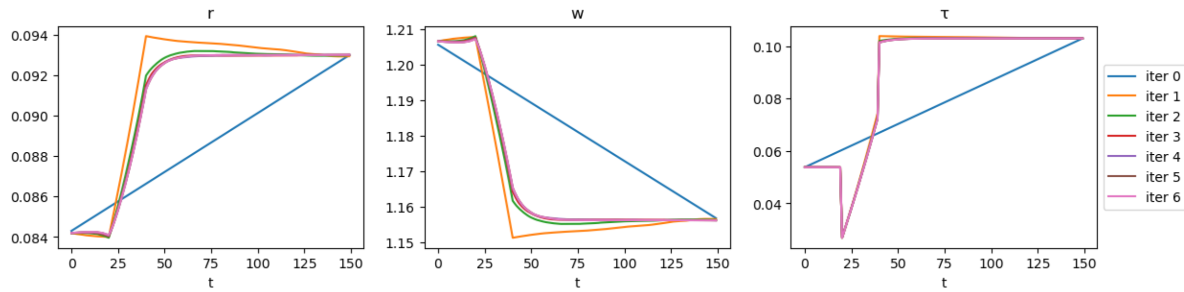
We must specify `D_seq` appropriately.

```
T = 150

D_t = 20
D_seq = jnp.ones(T+1) * D_ss1
D_seq = D_seq.at[D_t:D_t+21].set(D_ss1 + jnp.linspace(0, 1, 21))
D_seq = D_seq.at[D_t+21:].set(D_seq[D_t+20])

G_seq = jnp.ones(T) * G_ss1

δ_seq = jnp.repeat(δ_ss1, T).reshape((T, δ_ss1.size))
```

```
ss2 = find_ss(hh, firm, [D_seq[-1], G_seq[-1], δ_seq[-1]], Q)
```

```
paths = path_iteration(ss1, ss2, [D_seq, G_seq, δ_seq],
                       hh, firm, Q, verbose=True)
```

```
Iteration   1: error = 1.300627e-01
Iteration   2: error = 2.349870e-02
Iteration   3: error = 4.931191e-03
Iteration   4: error = 1.196040e-03
Iteration   5: error = 3.122933e-04
Iteration   6: error = 7.865898e-05
```




```
V_seq, σ_seq, μ_seq = paths[:3]
K_seq, L_seq = paths[3:5]
r_seq, w_seq = paths[5:7]
τ_seq, D_seq, G_seq, δ_seq = paths[7:11]
```

```
T = σ_seq.shape[0]
J = σ_seq.shape[1]

Cy_seq = np.empty(T)
Co_seq = np.empty(T)

for t in range(T):
    ap = hh.a_grid[σ_seq[t]]
    δ = δ_seq[t].reshape((hh.j_grid.size, 1, 1))

    inc = (1 + r_seq[t]*(1-τ_seq[t])) * a + (1-τ_seq[t]) * w_seq[t] * lj * γ - δ
    inc = inc.reshape((hh.j_grid.size, hh.a_grid.size * hh.γ_grid.size))

    c = inc - ap

    Cy_seq[t] = (c[:J//2] * μ_seq[t, :J//2]).sum() / (J // 2)
    Co_seq[t] = (c[J//2:] * μ_seq[t, J//2:]).sum() / (J // 2)
```

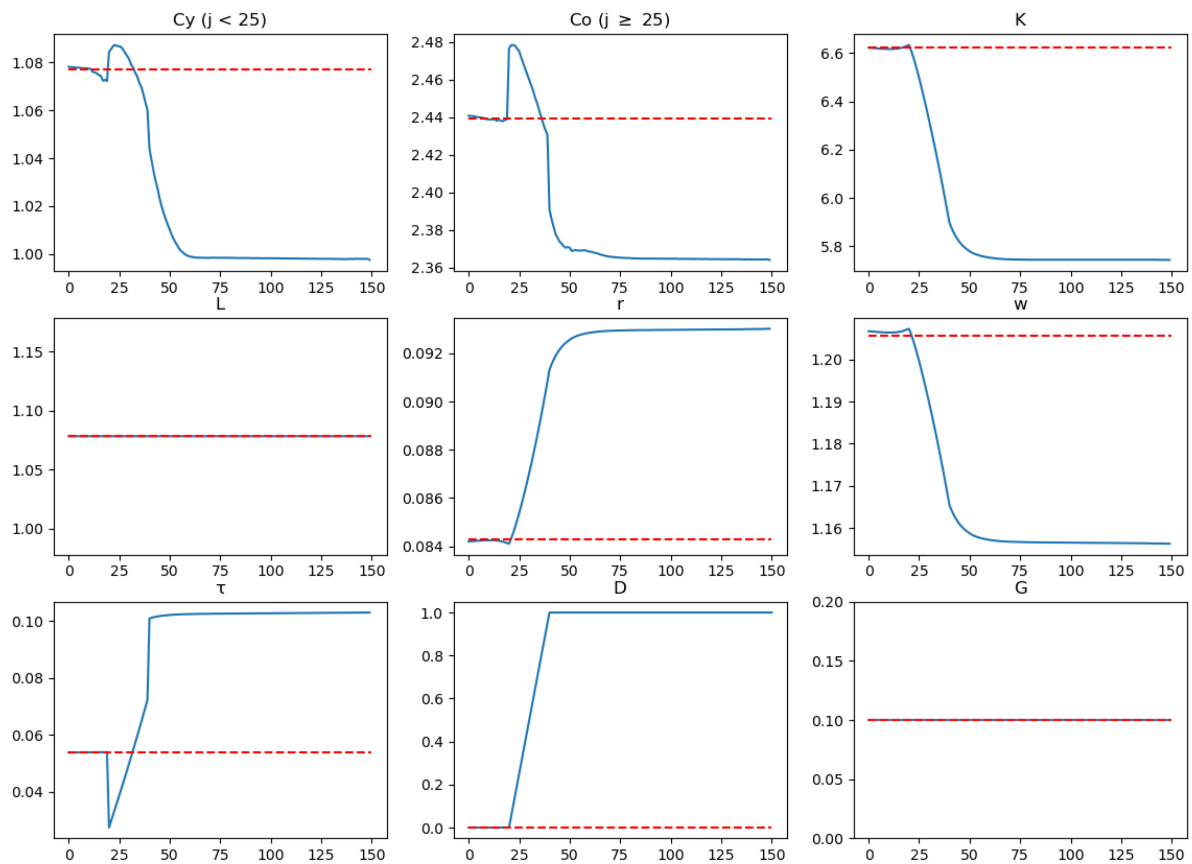
Below we plot the transition paths of the economy.

```
fig, axs = plt.subplots(3, 3, figsize=(14, 10))

# Cy (j=0-24)
axs[0, 0].plot(Cy_seq)
axs[0, 0].hlines(Cy_ss1, 0, T, color='r', linestyle='--')
axs[0, 0].set_title('Cy (j < 25)')

# Cy (j=25-49)
axs[0, 1].plot(Co_seq)
axs[0, 1].hlines(Co_ss1, 0, T, color='r', linestyle='--')
```

(continues on next page)

(continued from previous page)

```
axs[0, 1].set_title(r'Co (j $\geq$ 25)')

names = ['K', 'L', 'r', 'w', 'τ', 'D', 'G']
for i in range(len(names)):
    i_var = i + 3
    i_axes = i + 2

    row_i = i_axes // 3
    col_i = i_axes % 3

    axs[row_i, col_i].plot(paths[i_var])
    axs[row_i, col_i].hlines(ss1[i_var], 0, T, color='r', linestyle='--')
    axs[row_i, col_i].set_title(names[i])

# ylims
axs[1, 0].set_ylim([ss1[4]-0.1, ss1[4]+0.1])
axs[2, 2].set_ylim([ss1[9]-0.1, ss1[9]+0.1])

plt.show()
```



Notice how prices and quantities respond immediately to the anticipated tax rate increase.

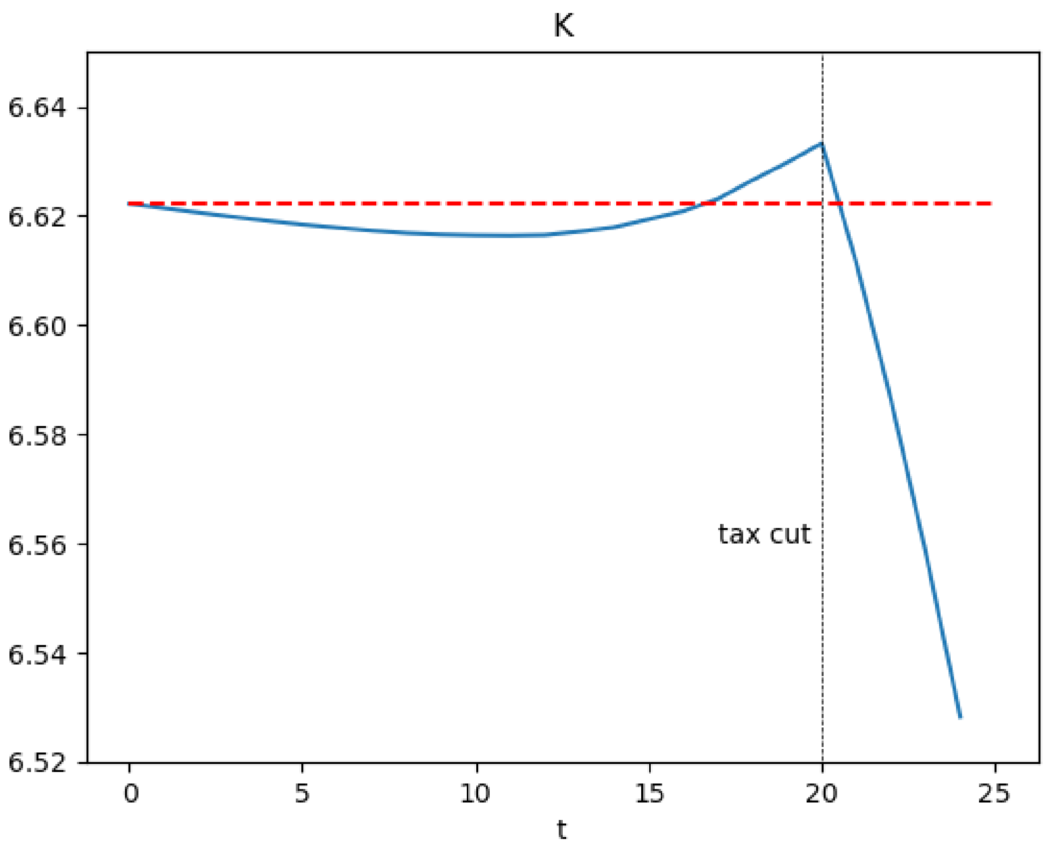
Let’s zoom in on how the capital stock responds.

```
# K
i_var = 3
```

(continues on next page)

(continued from previous page)

```
plt.plot(paths[i_var][:25])
plt.hlines(ss1[i_var], 0, 25, color='r', linestyle='--')
plt.vlines(20, 6, 7, color='k', linestyle='--', linewidth=0.5)
plt.text(17, 6.56, r'tax cut')
plt.ylim([6.52, 6.65])
plt.title("K")
plt.xlabel("t")
plt.show()
```



After the tax cut policy is implemented at $t = 20$, the aggregate capital will decrease because of the crowding out effect.

Having foreseen an increase in the interest rate, individuals a few periods before $t = 20$ start saving more.

Because that increases the capital, a temporary decrease in the interest rate ensues.

For agents living in much earlier periods, that lower interest rate causes them to save less.

We can also plot evolutions of means and variances of consumption by different cohorts along a transition path.

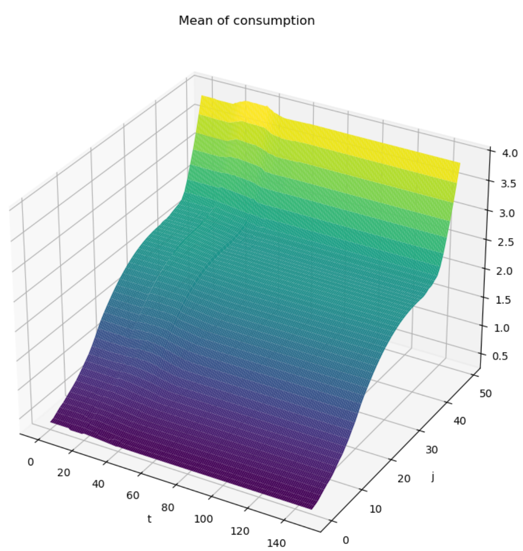
```
Cmean_seq = np.empty((T, J))
Cvar_seq = np.empty((T, J))

for t in range(T):
    ap = hh.a_grid[σ_seq[t]]
    δ = δ_seq[t].reshape((hh.j_grid.size, 1, 1))

    inc = (1 + r_seq[t]*(1-τ_seq[t])) * a + (1-τ_seq[t]) * w_seq[t] * lj * γ - δ
```

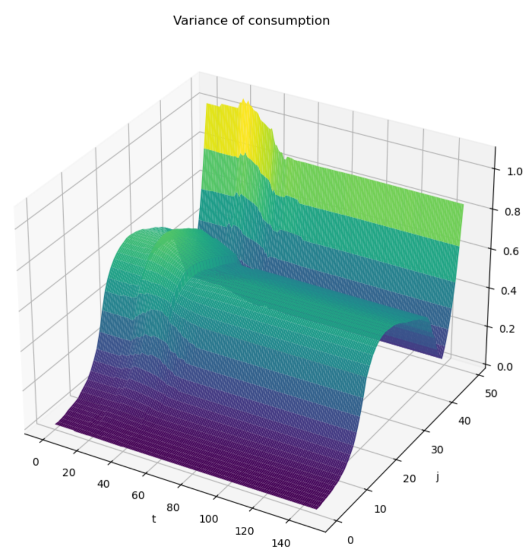
(continues on next page)

(continued from previous page)

```
    inc = inc.reshape((hh.j_grid.size, hh.a_grid.size * hh.y_grid.size))

    c = inc - ap

    Cmean_seq[t] = (c * μ_seq[t]).sum(axis=1)
    Cvar_seq[t] = (
        (c - Cmean_seq[t].reshape((J, 1))) ** 2 * μ_seq[t]).sum(axis=1)
```

```
J_seq, T_range = np.meshgrid(np.arange(J), np.arange(T))

fig = plt.figure(figsize=[20, 20])

# Plot the consumption mean over age and time
ax1 = fig.add_subplot(121, projection='3d')
ax1.plot_surface(T_range, J_seq, Cmean_seq, rstride=1, cstride=1,
                 cmap='viridis', edgecolor='none')
ax1.set_title(r"Mean of consumption")
ax1.set_xlabel(r"t")
ax1.set_ylabel(r"j")

# Plot the consumption variance over age and time
ax2 = fig.add_subplot(122, projection='3d')
ax2.plot_surface(T_range, J_seq, Cvar_seq, rstride=1, cstride=1,
                 cmap='viridis', edgecolor='none')
ax2.set_title(r"Variance of consumption")
ax2.set_xlabel(r"t")
ax2.set_ylabel(r"j")

plt.show()
```





CHAPTER

EIGHTYTHREE

TWO COMPUTATIONS TO FUND SOCIAL SECURITY

Contents

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

In addition to what's in Anaconda, this lecture will need the following library

```
!pip install jax
```

83.1 Overview

This lecture describes two computational experiments about alternative ways to move gradually from an unfunded (pay-as-you-go) to a fully funded social security system, following Huang *et al.* [1997].

As populations age, pay-as-you-go social security systems have faced financial difficulties.

This situation has led some of today's policy makers and policy advisors to think that today's citizens would be better off if earlier policy makers had set up a fully-funded retirement system.

But starting from where we are today, a transition to a fully funded system creates distributional challenges because older generations who contributed to the unfunded system could lose benefits.

To study possibilities quantitatively, this lecture employs a general equilibrium overlapping generations model that modifies the Auerbach and Kotlikoff [1987] environment by incorporating

- risk-sensitive preferences
- uncertainty about lifetimes
- uninsurable labor income risk
- a theory of consumption distributions within and across cohorts

By employing the discounted risk-sensitive linear-quadratic preferences of Hansen and Sargent [1995], the model yields linear decision rules for individual consumption and savings.

This makes it computationally feasible to track the joint distribution of consumption and wealth across cohorts.

This lecture relates to two other lectures:

- *Transitions in an Overlapping Generations Model* studies how taxes, transfers, and debt affect capital accumulation in a two-period OLG model, introducing the Auerbach and Kotlikoff [1987] framework that this lecture extends.
- *A Long-Lived, Heterogeneous Agent, Overlapping Generations Model* studies how fiscal policy interacts with precautionary savings in a long-lived OLG model, using discrete dynamic programming rather than the linear-quadratic approach adopted here.

We use the following imports and configurations

```
import jax
import jax.numpy as jnp
from jax import jit, vmap
import jax.lax as lax
import numpy as np
import matplotlib.pyplot as plt
from collections import namedtuple

# Enable 64-bit precision
jax.config.update("jax_enable_x64", True)
```

83.2 Model

83.2.1 Environment

The economy consists of overlapping generations of finitely lived individuals who may live up to $T_0 + 1$ years and an infinitely lived government.

Individual consumers and the government can invest at a constant risk-free gross rate of return.

During the first $T_1 + 1$ periods of life, consumers receive labor income that they allocate among consumption, taxes, and asset accumulation.

During the final $T_0 - T_1$ periods of life (retirement), consumers receive social security benefits and dissave by drawing down their assets.

The government taxes income from capital and labor, issues debt, purchases goods, and pays retirement benefits.

For any variable z, we use subscript t to denote age, argument s in parentheses to denote calendar time, and superscript $s - t$ to denote date of birth, so that $z^{s-t}(s) \equiv z_t(s) \equiv z_t^{s-t}(s)$.

83.2.2 Demographics

At date s, a cohort of measure $N_0(s)$ consumers is born who live during periods $s, s + 1, \ldots, s + T_0$.

As a cohort ages, its members face random survival according to age-to-age survival probabilities $\{\alpha_t\}_{t=0}^{T_0}$, where α_t is the probability of surviving from age t to $t + 1$.

Let $N_t(s)$ be the number of age-t people alive at time s and let n be the constant gross rate of population growth.

The size of age group t at time s satisfies

$$N_t(s) = \lambda_t \cdot N_0(s - t)$$

where $\lambda_t = \prod_{j=0}^{t-1} \alpha_j$ for $t = 0, \ldots, T_0$, with $\lambda_0 = 1$, and births follow $N_0(s) = n^s \cdot N_0(0)$.

The population fraction of cohort t at each time s is

$$f_t = \frac{\lambda_t \cdot n^{-t}}{\sum_{\tau=0}^{T_0} \lambda_\tau \cdot n^{-\tau}}$$

and total population at time s is

$$N(s) = N_0(0) \cdot n^s \cdot \sum_{t=0}^{T_0} n^{-t} \cdot \lambda_t.$$

83.2.3 Distributions and aggregates

Individuals face life span uncertainty and labor income shocks.

They self-insure by accumulating risk-free assets (government bonds and physical capital), while properly taking into account the social security benefits that they anticipate receiving.

Let ϵ_0^t denote the history of random shocks that an individual has received from birth to age t.

The state vector $x_t(s) = x_t(s; \epsilon_0^t, x_0)$ measures the stock of assets as well as information variables a consumer uses to forecast future preferences or opportunities.

The model delivers consumption as a time- and age-dependent linear function of the state vector

$$c_t(s;\epsilon_0^t, x_0) = \eta_{ct}(s) \cdot x_t(s;\epsilon_0^t, x_0)$$

where the state vector follows a linear law of motion

$$x_{t+1}(s+1;\epsilon_0^{t+1}, x_0) = A_t(s) \cdot x_t(s;\epsilon_0^t, x_0) + C_t(s) \cdot \epsilon_{t+1}$$

with ϵ_{t+1} a martingale difference sequence satisfying $E(\epsilon_{t+1}|J_t) = 0$, $E(\epsilon_{t+1} \cdot \epsilon_{t+1}'|J_t) = I$, and $J_t = (\epsilon_0^t, x_0)$.

The model delivers probability distributions for state vectors.

Let $\mu_t(s) = E[x_t(s)]$ and $\Sigma_t(s) = E[(x_t(s) - \mu_t(s))(x_t(s) - \mu_t(s))']$.

These moments satisfy

$$\mu_{t+1}(s+1) = A_t(s) \cdot \mu_t(s)$$

$$\Sigma_{t+1}(s+1) = A_t(s) \cdot \Sigma_t(s) \cdot A_t(s)' + C_t(s) \cdot C_t(s)'$$

Per capita aggregate consumption is

$$c(s)/N(s) = \sum_{t=0}^{T_0} \mu_{ct}(s) \cdot f_t$$

where $\mu_{ct}(s)$ is the mean consumption of age-t people at time s.

The distribution of consumption within age cohort t has mean $\mu_{ct}(s) = \eta_{ct}(s) \cdot \mu_t(s)$ and variance $\Sigma_{ct}(s) = \eta_{ct}(s) \cdot \Sigma_t(s) \cdot \eta_{ct}(s)'$.

83.2.4 Resource constraint

The economy-wide physical resource constraint is

$$g(s) \cdot N(s) + \sum_{t=0}^{T_0} c_t(s) \cdot N_t^{s-t} + K(s) = R(s-1) \cdot K(s-1) + w(s) \cdot \sum_{t=0}^{T_1} \varepsilon_t \cdot N_t^{s-t} + N_0(s) \cdot k_{-1}(s)$$

where $g(s)$ is per capita government purchases, $K(s-1)$ is physical capital, $R(s-1) = 1 + r(s-1) - \delta$ is the gross return on assets, ε_t is the exogenous efficiency endowment of age-t people, $w(s)$ is the base wage rate, $N_0(s) \cdot k_{-1}(s)$ is capital brought by newborns, and δ is the depreciation rate.

83.2.5 Factor prices

We consider two alternative assumptions about factor prices:

- *Small open economy:* $r(s-1) = r$ and $w(s) = w$ are exogenous and constant.
- *Closed economy:* Factor prices are determined by marginal products from a Cobb-Douglas production function:

$$r(s-1) = \tilde{A} \cdot \tilde{\alpha} \cdot (K(s-1)/\tilde{N}(s))^{\tilde{\alpha}-1}, \qquad w(s) = (1-\tilde{\alpha}) \cdot \tilde{A} \cdot (K(s-1)/\tilde{N}(s))^{\tilde{\alpha}}$$

where $\tilde{N}(s) = \sum_{t=0}^{T_1} \varepsilon_t \cdot N_t^{s-t}$ is aggregate labor input in efficiency units and $\tilde{\alpha}$ is capital's share of income.

83.2.6 Consumers' problems

Individual consumers face an overlapping-generations version of a classic consumption-saving problem (see *The Permanent Income Model*).

Working-age consumers ($t \leq T_1$) receive labor income $w(s) \cdot \varepsilon_t + d_t$, where d_t is an AR(1) process

$$d_t = \rho_d \cdot d_{t-1} + \xi_t$$

with ξ_t being Gaussian white noise with variance σ_d^2.

The budget constraint at age t and time s is

$$c_t(s) + a_t(s) = R(s-1) \cdot a_{t-1}(s-1) + w(s) \cdot \varepsilon_t + S_t(s) - T_t(s) + d_t \tag{83.1}$$

where $a_{t-1}(s-1)$ is asset holdings at the beginning of age t, $S_t(s)$ is social security benefits (zero while working, $S(s)$ when retired), and taxes are $T_t(s) = \tau_0(s) + \tau_\ell(s)(w(s) \cdot \varepsilon_t + d_t) + \tau_a(s)(R(s-1) - 1) \cdot a_{t-1}(s-1)$.

Following Hansen and Sargent [1995], preferences over stochastic consumption processes are defined recursively by

$$U_t = -(\pi \cdot c_t - \gamma_t)^2/2 + \beta_t \cdot \mathcal{R}_t(U_{t+1})$$

where $\mathcal{R}_t(U_{t+1}) = (2/\sigma) \cdot \log E[\exp(\sigma \cdot U_{t+1}/2)|J_t]$, σ is the risk-sensitivity parameter, and $\beta_t = \bar{\beta} \cdot \alpha_t$ is the survival-adjusted discount factor.

This preference specification delivers linear decision rules while allowing a form of risk-sensitivity that induces a type of precautionary savings.

When $\sigma < 0$, the consumer prefers early resolution of uncertainty, and decision rules depend partly on noise statistics.

83.2.7 Government

The government purchases goods, pays social security benefits, taxes capital and labor income, confiscates accidental bequests, and issues one-period bonds.

The government budget constraint is

$$\begin{aligned}
g(s) \cdot N(s) + \sum_{t=T_1+1}^{T_0} S_t(s) \cdot N_t^{s-t} + R(s-1) \cdot \sum_{t=1}^{T_0} b_{t-1}(s-1) \cdot N_t^{s-t} = \\
\sum_{t=0}^{T_0} N_t^{s-t} \{\tau_a(s)[R(s-1) - 1] \cdot a_{t-1}(s-1) + \tau_\ell(s) \cdot w(s) \cdot \varepsilon_t\} + \tau_0 \cdot N(s) + \\
\sum_{t=0}^{T_0} b_t(s) \cdot N_t^{s-t} + R(s-1) \cdot \sum_{t=0}^{T_0} (1-\alpha_t) \cdot k_t(s-1) \cdot N_t^{s-t-1}
\end{aligned}$$

where $b_t(s)$ is government debt held by age-t individuals at time s and the last term represents the bequest tax.

A transition between fiscal regimes is described by dates $0 \leq s_1 < s_2 < s_3$:

- Before $s = 0$: initial stationary equilibrium
- At $s = 0$: government announces a policy change
- $s \in [s_1, s_2)$: fiscal parameters and the social security system change
- $s \geq s_2$: new constant policy parameters
- $s \geq s_3$: final stationary equilibrium

83.2.8 Equilibrium

We define the following objects:

- An **allocation** is a stochastic process for $\{c_t(s), a_t(s)\}_{s=0}^{s_3}$ for $t = 0, \dots, T_0$, and a sequence $\{K(s)\}_{s=0}^{s_3}$.
- A **government policy** is a sequence $\{b(s), g(s), \tau_\ell(s), S(s), \tau_a(s)\}_{s=0}^{s_3}$.
- A **price system** is a sequence $\{w(s), r(s-1)\}_{s=0}^{s_3}$.

An **equilibrium** is an allocation, a price system, and a government policy such that

1. given the price sequence and government policy, the allocation solves households' optimum problems, and
2. the allocation and government policy satisfy the government budget constraint at each date s.

In a **stationary equilibrium**, all variables are independent of calendar time s, which simplifies the government budget constraint to

$$g + \sum_{t=T_1+1}^{T_0} S_t \cdot f_t + [R/n - 1] \cdot \bar{b} = \tau_a(R-1) \cdot \sum_{t=0}^{T_0} a_{t-1} \cdot f_t + \tau_\ell \cdot \sum_{t=0}^{T_1} w \cdot \varepsilon_t \cdot f_t + \frac{R}{n} \cdot \sum_{t=0}^{T_0} (1-\alpha_t) \cdot a_t \cdot f_t$$

where $\bar{b}$ is per capita government debt in steady state.

83.2.9 Transition dynamics

The following diagram shows the age-time structure during the transition.

Each horizontal line represents the lifetime of one cohort, and the vertical dashed lines mark the policy change dates s_1 and s_2.

```
fig, ax = plt.subplots(figsize=(8, 5))

birth_range = np.arange(-60, 41, 5)

ax.hlines(birth_range, birth_range, birth_range + 60, 'k', linewidth=1)
ax.vlines([0, 40], -60, 40, 'k', linestyle='--', linewidth=0.5)

ax.set_ylabel("Date born")
ax.set_xlabel("Time")

ax.text(-20, 25, r"$s_1=0$")
ax.text(43, -50, r"$s_2$")

ax.invert_yaxis()
plt.show()
```

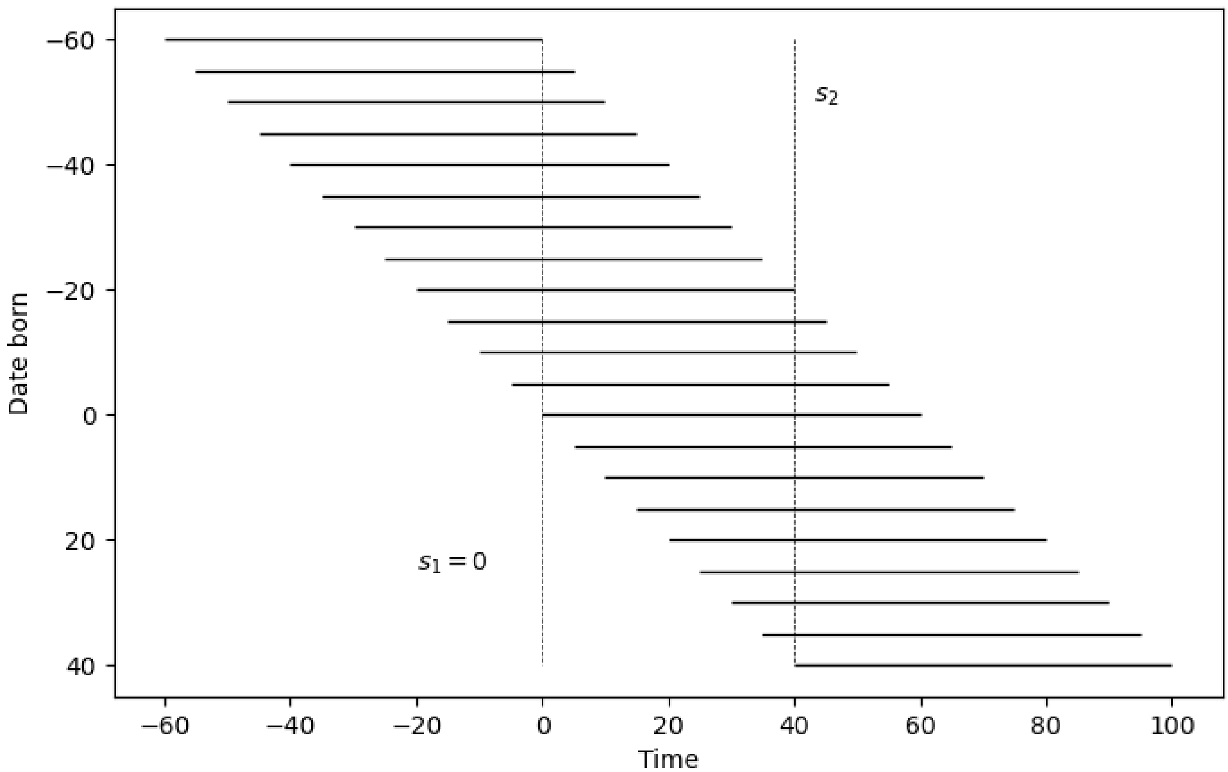


Fig. 83.1: Age-time diagram for overlapping generations

Small open economy

With fixed factor prices, the transition has a clear structure:

- Cohorts born before $s_1 - T_0$ die before the policy change and are unaffected.
- Cohorts alive at s_1 must recalculate their consumption-saving plans for their remaining lifetimes.
- Cohorts born between s_1 and s_2 face time-varying tax and benefit rates, while those born after s_2 face constant parameters.
- The transition ends at $s_3 = s_2 + T_0$ when the last cohort that experienced the policy change has died.

Because factor prices are fixed, we can compute the transition by solving decision rules for cohorts born at dates $s_1 - T_0 - 1, \ldots, s_2$.

For any date s, aggregate consumption is computed by summing across all living cohorts (along a vertical line in the age-time diagram), weighted by their population fractions.

Closed economy

With endogenous factor prices, the transition is more complex:

- Factor prices continue to evolve after policy parameters stabilize at s_2, so we follow Auerbach and Kotlikoff [1987] and truncate at $s_3 = s_2 + 2T_0$.
- The computation requires nested iteration: an inner loop determines labor income tax rates, and an outer loop adjusts interest rates to clear factor markets.
- Changes in saving behavior affect capital accumulation, which alters marginal products and feeds back into household decisions.
- Lower interest rates benefit young workers through higher wages but hurt retirees through lower returns on savings.

83.3 Two experiments

We explore two strategies for transitioning to a fully funded social security system.

In experiment 1, the government terminates social security benefits but compensates entitled generations through a one-time increase in government debt.

In experiment 2, the government retains social security benefits but temporarily raises taxes to accumulate physical capital, the returns from which eventually finance social security payments.

Both proposals finance a transition to fully funded social security while maintaining welfare across generations, but they entail different amounts of intergenerational risk-sharing.

We compute both experiments under fixed and endogenous factor prices and compare outcomes below.

83.4 Computation strategy

83.4.1 Dynamic program

An individual consumer's problem can be formulated as a discounted risk-sensitive linear control problem (see *LQ Control: Foundations*).

Let $x_t = [a_{t-1}, z_t]'$ where z_t is the vector of shocks.

The optimal value function takes the form $U_t = x_t' \cdot P_t \cdot x_t + \xi_t$.

The recursive problem is

$$U_t = \max_{u_t} \left\{ u_t' Q_t u_t + x_t' R_t x_t + \frac{2\beta_t}{\sigma} \log E_t[\exp(\sigma U_{t+1}/2)] \right\}$$

subject to $x_{t+1} = A_t x_t + B_t u_t + C_t w_{t+1}$.

We deploy two operators

$$T_t(P) = P + \sigma P C_t \left(I - \sigma C_t' P C_t\right)^{-1} C_t' P$$
$$D_t(W) = R_t + A_t' \left[\beta_t W - \beta_t^2 W B_t \left(Q_t + \beta_t B_t' W B_t\right)^{-1} B_t' W\right] A_t$$
$$\mathcal{S}_t(k, P) = \beta_t k - (\beta_t/\sigma) \log \det \left(I - \sigma C_t' P C_t\right)$$

that we use to construct a value function recursion $P_t = (D_t \circ T_t) P_{t+1}$, $\xi_t = \mathcal{S}_t(\xi_{t+1}, P_{t+1})$, and an optimal control

$$u_t = -F_t x_t, \qquad F_t = \beta_t \left[Q_t + \beta_t B_t' T_t(P_{t+1}) B_t\right]^{-1} B_t' T_t(P_{t+1}) A_t.$$

Operators $T_t, D_t, \mathcal{S}_t$ and decision rule F_t are constructed in `solve_riccati_step`.

Given value function parameters (P_{t+1}, ξ_{t+1}) at the next age, it constructs the state-space matrix A_t, applies the cross-product trick, evaluates the Riccati operators, and returns the optimal decision rule F_t, the closed-loop matrix $A_t^o = A_t - BF_t$, and the updated (P_t, ξ_t)

```python
def solve_riccati_step(
        ε_t, β_t, Ind_work_t,
        RR, w, τ_l, τ_a, τ_0, benef,
        P_next, ξ_next,
        ρ_d, σ, B, C, R, Q, H):
    """One backward step of the risk-sensitive Riccati recursion."""

    A = jnp.array([
        [RR * (1.0 - τ_a) + τ_a,
         (1.0 - τ_l) * w * ε_t
         - τ_0 + benef * (1.0 - Ind_work_t),
         (1.0 - τ_l) * Ind_work_t],
        [0.0, 1.0, 0.0],
        [0.0, 0.0, ρ_d]
    ])

    Q_scalar = Q[0, 0]
    Q_inv_scalar = 1.0 / Q_scalar
    Q_inv = jnp.array([[Q_inv_scalar]])

    # Cross-product trick: A* = A - B Q^{-1} H
    A = A - B @ Q_inv @ H
```

(continues on next page)

(continued from previous page)

```
    # T_t operator
    CTP = C.T @ P_next @ C
    PP_scalar = 1.0 - σ * CTP[0, 0]
    PP_inv_scalar = 1.0 / PP_scalar
    PC = P_next @ C
    CP = C.T @ P_next
    TP = P_next + σ * PP_inv_scalar * (PC @ CP)

    # D_t operator and decision rule F_t
    BTB_scalar = (B.T @ TP @ B)[0, 0]
    Q_BTB_scalar = Q_scalar + β_t * BTB_scalar
    Q_BTB_inv_scalar = 1.0 / Q_BTB_scalar
    BT_TP = B.T @ TP
    BT_TP_A = BT_TP @ A
    F = β_t * Q_BTB_inv_scalar * BT_TP_A

    TP_B = TP @ B
    middle = (β_t * TP
              - β_t**2 * Q_BTB_inv_scalar
              * (TP_B @ BT_TP))
    P = R + A.T @ middle @ A

    # S_t operator
    log_det_PP = jnp.log(PP_scalar)
    ξ = jnp.where(
        σ != 0.0,
        β_t * (ξ_next - log_det_PP / σ),
        β_t * (ξ_next + CTP[0, 0])
    )

    Ao = A - B @ F
    F = F + Q_inv @ H

    return F.squeeze(), Ao, P, ξ
```

83.4.2 State space preparation

The budget constraint (83.1) and the income process can be written in state-space form.

Let $x_t = [a_{t-1}(s-1),\ 1,\ d_t]'$, $u_t = c_t(s)$, and $w_{t+1} = \epsilon_{t+1}$, so that

$$x_{t+1} = A_t x_t + B u_t + C w_{t+1}$$

where

$$A_t = \begin{bmatrix} R(s-1)(1-\tau_t^a(s)) + \tau_t^a(s) & (1-\tau_\ell(s))w(s)\varepsilon_t - \tau_0(s) + S_t(s) & \mathbf{1}_t^{\text{work}}(1-\tau_\ell(s)) \\ 0 & 1 & 0 \\ 0 & 0 & \rho_d \end{bmatrix},$$

$$B = \begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix},$$

$$C = \begin{bmatrix} 0 \\ 0 \\ \sigma_d \end{bmatrix}$$

and $\mathbf{1}_t^{\text{work}}$ indicates whether the agent is of working age.

The per-period return $-\frac{1}{2}(\pi c_t - \gamma)^2$ introduces a cross-product term H between the control and the state.

This is eliminated using the cross-product trick (see *Eliminating Cross Products*):

$$A^* = A - BQ^{-1}H, \qquad R^* = R - H'Q^{-1}H.$$

83.4.3 Means and covariances

Define $A_o = A - BF$ as the closed-loop transition matrix, so that $x_{t+1} = A_o x_t + C w_{t+1}$ and unconditional moments satisfy

$$\mu_{t+1} = A_{o,t}\mu_t, \qquad \Sigma_{t+1} = A_{o,t}\Sigma_t A'_{o,t} + CC'.$$

The moment recursion is implemented as `forward_moment_step`, which propagates the mean vector and covariance matrix by one age step and computes consumption statistics as by-products.

```python
def forward_moment_step(μx_t, Σx_t, Ao_t, F_t, CCT):
    """One step of the forward moment recursion."""

    μx_next = Ao_t @ μx_t
    μc_t = -F_t @ μx_t
    Σx_next = CCT + Ao_t @ Σx_t @ Ao_t.T
    Vc_t = F_t @ Σx_t @ F_t.T
    return μx_next, μc_t, Σx_next, Vc_t
```

83.4.4 Computing transitions

A cohort born at s lives during $s, s+1, \dots, s+T_0$ and works during $s, s+1, \dots, s+T_1$.

Let $0 \leq s_1 < s_2 < s_3$.

- At $s = 0$, the government announces a policy change between $s = s_1$ and $s = s_2$.
- From $s = s_2$, government policies are constant forever.
- From $s = s_3$, convergence to the final stationary equilibrium is achieved (in the small open economy, $s_3 = s_2 + T_0$).

The affected cohorts are those born at $s = s_1 - T_0, s_1 - T_0 + 1, \dots, s_2$.

In all exercises, we set $T_0 = 65$, $T_1 = 43$, $s_1 = 0$, and $s_2 = 40$.

83.5 Calibration

The model parameters are set as follows.

83.5.1 Preference parameters

Parameter	Description	Value
$\{\alpha_t\}_{t=0}^{T_0}$	Age-to-age survival probabilities	Faber [1982]
π	Consumption preference parameter	1.0
σ	Risk-sensitivity parameter	-0.05
$\bar{\gamma}$	Preference shock parameter	7.0
$\tilde{\beta}$	Discount factor	0.986
T_0	Maximum age	65
T_1	Retirement age	43
n	Gross population growth rate	1.012

83.5.2 Technology parameters

Parameter	Description	Value
k_{-1}	Initial capital endowment	4.0
σ_d	Standard deviation of income shock	0.85
ρ_d	Persistence of income shock	0.8
δ	Depreciation rate	0.06
$\{\varepsilon_t\}_{t=0}^{T_1}$	Age-efficiency profile	Hansen [1993]
w	Base wage rate (exogenous)	5.0147
r	Return on capital (exogenous)	0.1275
$\tilde{A}$	Production function scaling (endogenous)	2.2625
$\tilde{\alpha}$	Capital share (endogenous)	0.40

```
N_GRID_SS = 10
TOL_SS = 1e-10

T0 = 65   # maximum lifespan (ages 21 to 86)
T1 = 43   # working life length (retire at 65)

UNIT_GRID = jnp.linspace(0.0, 1.0, N_GRID_SS)
AGE_INDICES = jnp.arange(T0 + 2)
```

The hidden code cell below defines the age-efficiency profile $\{\varepsilon_t\}$ and the survival probabilities $\{\alpha_t\}$ based on Faber [1982] and Hansen [1993].

```
fig, axs = plt.subplots(1, 2, figsize=(10, 6))

axs[0].plot(ε_arr)
axs[0].set_title("Working efficiency")
axs[0].set_xlabel("Age")

axs[1].plot(α_arr)
axs[1].set_title("Survival probability")
axs[1].set_xlabel("Age")

plt.tight_layout()
plt.show()
```

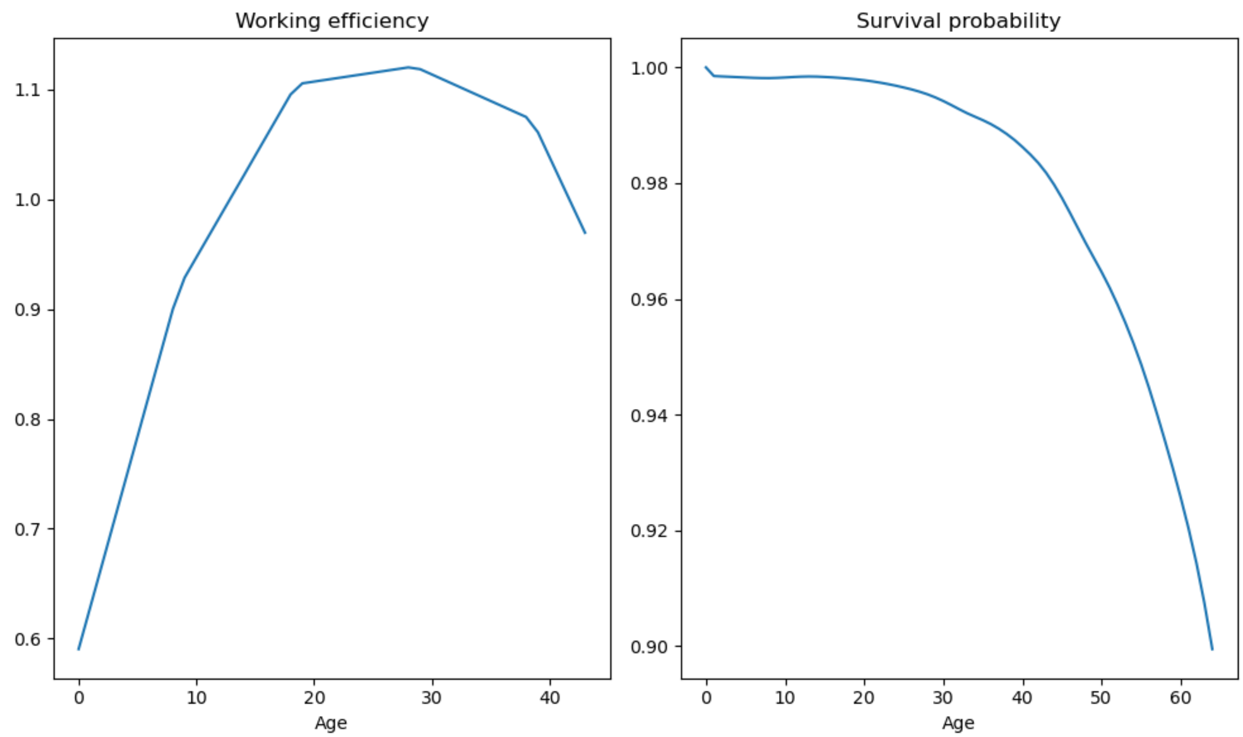


Fig. 83.2: Age-efficiency profile and survival probabilities

We impose a large penalty on terminal asset holdings to enforce the end-of-life condition, and set the initial state to $x_0 = [k_{-1},\ 1,\ 0]'$.

```
P_end = jnp.zeros((3, 3))
P_end = P_end.at[0, 0].set(-2000000.0)
ξ_end = 0.0

x0 = jnp.array([4.0, 1.0, 0.0])
Σ0 = jnp.zeros((3, 3))
```

All household parameters are collected into a named tuple.

```
Household = namedtuple('Household', (
    'α_arr', 'frac', 'n', 'π', 'σ', 'k_init', 'ε_arr', 'Ind_work',
    'σ_d', 'ρ_d', 'γ_bar', 'β_arr',
    'T0', 'T1', 'T2', 'n_x', 'n_w',
    'P_end', 'ξ_end', 'x0', 'Σ0',
    'B', 'C', 'R', 'Q', 'H'
))

def create_household(α_arr=α_arr,     # Age-to-age survival probabilities
                     n=1.012,         # Gross population growth rate
                     π=1,             # Consumption preference parameter
                     σ=-0.05,         # Risk-sensitivity parameter
                     k_init=4,        # initial capital endowment
                     ε_arr=ε_arr,     # age-efficiency profile
                     σ_d=0.85,        # std of income shock
                     ρ_d=0.8,         # persistence of income shock
```

(continues on next page)

(continued from previous page)

```
                  γ_bar=7,          # Preference shock parameter
                  β_tilde=0.986,    # Discount factor
                  T0=65,            # Maximum age
                  T1=43,            # Retirement age
                  n_x=3,            # Number of states
                  n_w=1,            # Number of shocks
                  P_end=P_end,      # Terminal value
                  ξ_end=ξ_end,      # Terminal value
                  x0=x0,            # Initial mean
                  Σ0=Σ0):           # Initial variance
    """Create a Household named tuple with derived arrays."""

    α_arr = np.concatenate([α_arr, np.array([0])])
    T2 = T0 - T1

    frac = np.ones(T0 + 1)
    frac[1:] = np.cumprod(α_arr / n)[:-1]
    frac = frac / frac.sum()

    ε_arr = np.concatenate([ε_arr, np.zeros(T0 + 1 - ε_arr.size)])

    # Indicator for working ages: 1 if working (ε > 0), 0 if retired
    Ind_work = (ε_arr != 0).astype(np.float64)

    β_arr = β_tilde * α_arr
    β_arr[-1] = β_tilde

    B = jnp.array([[-1.0, 0.0, 0.0]]).T
    C = jnp.array([[0.0, 0.0, σ_d]]).T

    Q = jnp.array([[-0.5 * π**2]])
    H = jnp.array([[0.0, 0.5 * π * γ_bar, 0.0]])

    # Apply cross-product trick: R* = R - H'Q^{-1}H
    R_base = np.array([[0.0, 0.0, 0.0],
                       [0.0, -0.5 * γ_bar**2, 0.0],
                       [0.0, 0.0, 0.0]])
    H_np = np.array([[0.0, 0.5 * π * γ_bar, 0.0]])
    Q_inv_np = np.array([[1.0 / (-0.5 * π**2)]])
    R = jnp.array(R_base - H_np.T @ Q_inv_np @ H_np)

    return Household(
        α_arr=jnp.array(α_arr), frac=jnp.array(frac), n=n, π=π, σ=σ,
        k_init=k_init, ε_arr=jnp.array(ε_arr), Ind_work=jnp.array(Ind_work),
        σ_d=σ_d, ρ_d=ρ_d, γ_bar=γ_bar, β_arr=jnp.array(β_arr),
        T0=T0, T1=T1, T2=T2, n_x=n_x, n_w=n_w,
        P_end=P_end, ξ_end=ξ_end, x0=x0, Σ0=Σ0,
        B=B, C=C, R=R, Q=Q, H=H
    )
```

```
hh = create_household()
```

The stationary population distribution follows.

```
fig, ax = plt.subplots()
ax.plot(hh.frac)
```

(continues on next page)

(continued from previous page)

```
ax.set_xlabel("Age")
ax.set_ylabel("Population fraction")
ax.set_title("Population distribution over age")
plt.show()
```

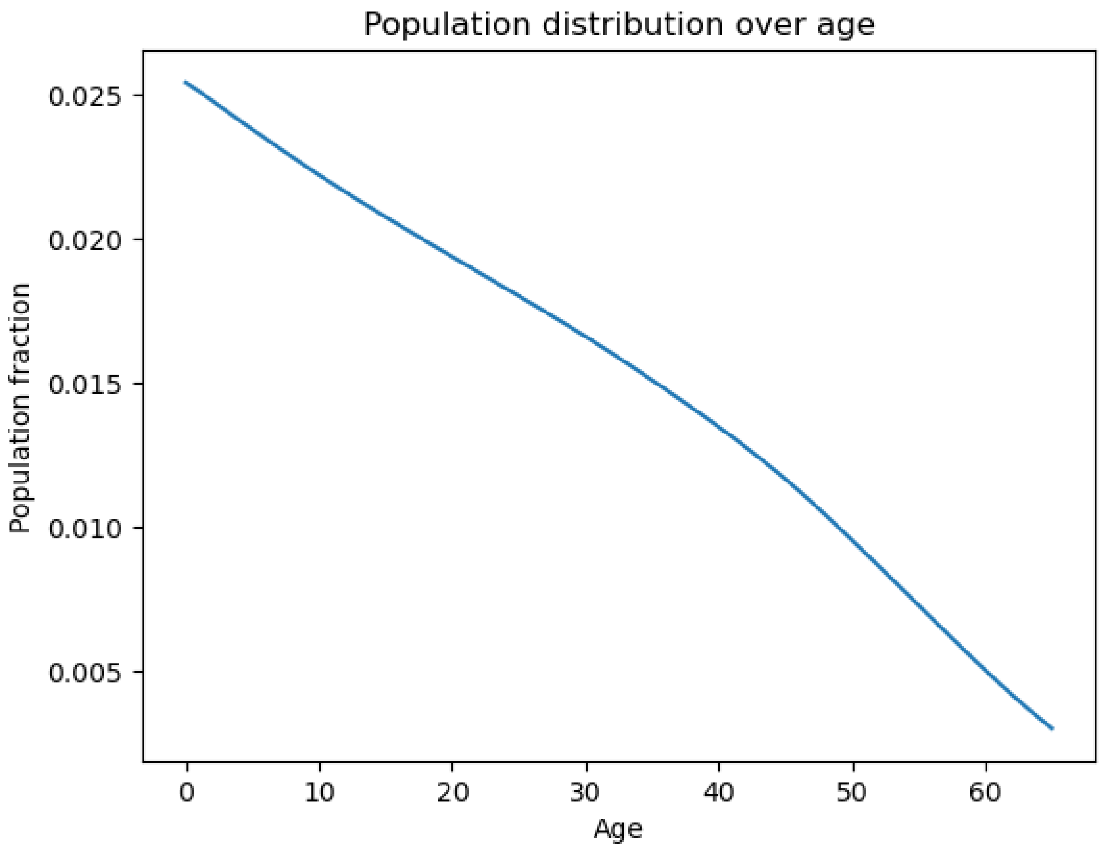


Fig. 83.3: Stationary population distribution over age

Mortality causes the population fraction to decline with age, a demographic pattern central to the intergenerational redistribution that social security reform entails.

Under the small open economy assumption, factor prices are fixed at calibrated values; under the closed economy assumption, they are determined by Cobb-Douglas marginal products

```
Tech = namedtuple('Tech', ('δ', 'w', 'r', 'RR', 'A', 'α_tilde'))

def create_Tech(δ=0.06, w=5.0147, r=0.1275,
                A=2.2625, α_tilde=0.40):
    """Create a Tech named tuple with factor price parameters."""

    RR = 1 + r - δ

    return Tech(δ=δ, w=w, r=r, RR=RR, A=A, α_tilde=α_tilde)
```

```
tech = create_Tech()
```

83.6 Individual optimality

83.6.1 Steady-state computation

With `solve_riccati_step` and `forward_moment_step` in hand, the steady-state computation proceeds in three phases.

Algorithm 83.6.1 (Steady-state computation)

1. **Backward recursion:** scan from age T_0 down to 0. At each age t, call `solve_riccati_step` to obtain decision rule F_t, closed-loop matrix A^o_t, value-function matrix P_t, and certainty-equivalent ξ_t.
2. **Forward simulation:** scan from age 0 up to T_0. At each age t, call `forward_moment_step` to propagate mean $\mu_{x,t}$ and covariance $\Sigma_{x,t}$ of the state vector, and record mean consumption $\mu_{c,t}$ and its variance $V_{c,t}$.
3. **Budget imbalance:** aggregate across cohorts. Sum tax revenues (labor, capital, lump-sum), subtract benefit payments, add accidental bequests, and return the government budget gap.

Phase 1. The backward recursion scans from age T_0 to 0, applying `solve_riccati_step` at each age

```
def _ss_backward_recursion(
        ε_arr, β_arr, Ind_work,
        RR, w, τ_l, τ_a, τ_0, benef,
        P_end, ξ_end,
        ρ_d, σ, B, C, R, Q, H):
    """Backward Riccati scan over all ages."""

    ε_rev = ε_arr[::-1]
    β_rev = β_arr[::-1]
    Ind_work_rev = Ind_work[::-1]

    def backward_step(carry, inputs):
        P_next, ξ_next = carry
        ε_t, β_t, Ind_work_t = inputs
        F, Ao, P, ξ = solve_riccati_step(
            ε_t, β_t, Ind_work_t,
            RR, w, τ_l, τ_a, τ_0, benef,
            P_next, ξ_next,
            ρ_d, σ, B, C, R, Q, H
        )
        return (P, ξ), (F, Ao, P, ξ)

    init_carry = (P_end, ξ_end)
    _, (F_rev, Ao_rev, P_rev, ξ_rev) = lax.scan(
        backward_step, init_carry,
        (ε_rev, β_rev, Ind_work_rev)
    )

    F_arr = F_rev[::-1]
    Ao_arr = Ao_rev[::-1]
    P_inner = P_rev[::-1]
```

(continues on next page)

(continued from previous page)

```
    ξ_inner = ξ_rev[::-1]

    P_arr = jnp.concatenate(
        [P_inner, P_end[None, :, :]], axis=0
    )
    ξ_arr = jnp.concatenate(
        [ξ_inner, jnp.array([ξ_end])]
    )

    return F_arr, Ao_arr, P_arr, ξ_arr
```

Phase 2. The forward recursion propagates means and covariances from age 0 to T_0

```
def _ss_forward_simulation(
        Ao_arr, F_arr, x0, Σ0, C):
    """Forward moment scan using forward_moment_step."""

    CCT = C @ C.T

    def forward_step(carry, inputs):
        μx_t, Σx_t = carry
        Ao_t, F_t = inputs
        result = forward_moment_step(
            μx_t, Σx_t, Ao_t, F_t, CCT
        )
        μx_next, μc_t, Σx_next, Vc_t = result
        return (μx_next, Σx_next), \
            (μx_next, μc_t, Σx_next, Vc_t)

    init_carry = (x0, Σ0)
    _, (μx_scn, μc_arr, Σx_scn, Vc_arr) = lax.scan(
        forward_step, init_carry, (Ao_arr, F_arr)
    )

    μx_arr = jnp.concatenate(
        [x0[None, :], μx_scn], axis=0
    )
    Σx_arr = jnp.concatenate(
        [Σ0[None, :, :], Σx_scn], axis=0
    )

    return μx_arr, μc_arr, Σx_arr, Vc_arr
```

Phase 3. Aggregating tax revenues, benefit payments, and accidental bequests across cohorts gives the government budget gap

```
def _ss_budget_imbalance(
        μx_arr, ε_arr, frac, n, α_arr,
        RR, w, τ_l, τ_a, τ_0, benef,
        G, Gb, Ind_work):
    """Aggregate tax revenues and expenditures."""

    μa_arr = μx_arr[1:, 0]
    μa_last_arr = μx_arr[:-1, 0]

    τ_l_tot = jnp.sum(τ_l * ε_arr * w * frac)
    τ_a_tot = jnp.sum(
```

(continues on next page)

(continued from previous page)

```
        τ_a * (RR - 1.0) * μa_last_arr * frac
    )
    τ_0_tot = jnp.sum(τ_0 * frac)

    retired_mask = 1.0 - Ind_work
    benef_tot = jnp.sum(benef * frac * retired_mask)

    Beq = jnp.sum(
        RR * (1.0 - α_arr) * frac * μa_arr / n
    )

    T_tot = τ_l_tot + τ_a_tot + τ_0_tot + Beq
    diff = (G + benef_tot - T_tot
            + (RR / n - 1.0) * Gb)

    return diff
```

A steady state is found when the budget gap equals zero.

`ss_imbalance` chains the three phases into a single JIT-compiled function: backward recursion, forward simulation, and budget gap

```
@jit
def ss_imbalance(price, policy, α_arr, ε_arr, frac,
                 n, β_arr, ρ_d, σ, B, C, R, Q, H,
                 P_end, ξ_end, x0, Σ0, Ind_work):
    """Backward solve, forward simulate, and return budget gap."""

    RR, w = price
    τ_l, τ_a, τ_0, benef, G, Gb = policy

    F_arr, Ao_arr, P_arr, ξ_arr = \
        _ss_backward_recursion(
            ε_arr, β_arr, Ind_work,
            RR, w, τ_l, τ_a, τ_0, benef,
            P_end, ξ_end,
            ρ_d, σ, B, C, R, Q, H
        )

    μx_arr, μc_arr, Σx_arr, Vc_arr = \
        _ss_forward_simulation(
            Ao_arr, F_arr, x0, Σ0, C
        )

    diff = _ss_budget_imbalance(
        μx_arr, ε_arr, frac, n, α_arr,
        RR, w, τ_l, τ_a, τ_0, benef,
        G, Gb, Ind_work
    )

    return (diff, P_arr, ξ_arr, Ao_arr, F_arr,
            μx_arr, μc_arr, Σx_arr, Vc_arr)
```

A named tuple `SteadyState` collects value-function parameters, decision rules, moments, and aggregate statistics into a single object that the transition solver can unpack

```
SteadyState = namedtuple("SteadyState", (
    "P_arr",      # Value function matrices by age
    "ξ_arr",      # Certainty equivalent adjustments by age
    "Ao_arr",     # Closed-loop transition matrices by age
    "F_arr",      # Decision rule matrices by age
    "μx_arr",     # Mean state vectors by age
    "μc_arr",     # Mean consumption by age
    "Σx_arr",     # Covariance matrices by age
    "Vc_arr",     # Consumption variances by age
    "debt2gdp",   # Government debt to GDP ratio
    "τ_l",        # Labor income tax rate
    "benef",      # Social security benefit level
    "Gb",         # Per-capita government debt
    "k_bar",      # Per-capita capital stock
    "RR",         # Gross return on assets
    "w",          # Wage rate
    "r",          # Interest rate (before depreciation)
    "k2gdp"       # Capital to GDP ratio
))
```

Given all other fiscal instruments, the equilibrium τ_ℓ is the value that zeroes the budget gap.

We find it by iterative grid refinement: evaluate the gap on a coarse grid, zoom into the best interval, and repeat

```
def _grid_refine(eval_fn, a_init, b_init, unit_grid, tol, max_iter):
    """Iterative grid-refinement root search.

    Must be called inside @jit functions.
    """

    n_grid = unit_grid.shape[0]

    def cond_fn(state):
        a, b, best_val, i = state
        return (jnp.abs(best_val) > tol) & (i < max_iter)

    def body_fn(state):
        a, b, _, i = state
        grid = a + (b - a) * unit_grid
        diffs = vmap(eval_fn)(grid)
        best_idx = jnp.argmin(jnp.abs(diffs))
        best_val = diffs[best_idx]
        idx_lo = jnp.maximum(best_idx - 1, 0)
        idx_hi = jnp.minimum(best_idx + 1, n_grid - 1)
        return (grid[idx_lo], grid[idx_hi], best_val, i + 1)

    grid = a_init + (b_init - a_init) * unit_grid
    diffs = vmap(eval_fn)(grid)
    best_idx = jnp.argmin(jnp.abs(diffs))
    best_val = diffs[best_idx]
    idx_lo = jnp.maximum(best_idx - 1, 0)
    idx_hi = jnp.minimum(best_idx + 1, n_grid - 1)

    init_state = (grid[idx_lo], grid[idx_hi], best_val, 0)
    final_state = lax.while_loop(cond_fn, body_fn, init_state)
    a_final, b_final, _, _ = final_state
    return (a_final + b_final) / 2.0
```

`_ss_diff_for_tau_l` evaluates the budget gap at a given τ_ℓ, and `_find_ss_tau_l` wraps it inside the grid-refine loop.

```
def _ss_diff_for_tau_l(τ_l, price_arr, policy_no_τl, α_arr, ε_arr, frac, n,
                        β_arr, ρ_d, σ, B, C, R, Q, H, P_end, ξ_end, x0, Σ0,
                        Ind_work):
    """Budget imbalance for a given τ_l."""

    τ_a, τ_0, benef, G, Gb = policy_no_τl
    policy_arr = jnp.array([τ_l, τ_a, τ_0, benef, G, Gb])
    diff, *_ = ss_imbalance(
        price_arr, policy_arr,
        α_arr, ε_arr, frac, n,
        β_arr, ρ_d, σ, B, C, R, Q, H, P_end, ξ_end, x0, Σ0,
        Ind_work
    )
    return diff
```

```
@jit
def _find_ss_tau_l(price_arr, policy_no_τl, α_arr, ε_arr, frac, n,
                    β_arr, ρ_d, σ, B, C, R, Q, H, P_end, ξ_end, x0, Σ0,
                    Ind_work, unit_grid):
    """Find τ_l that balances the steady-state budget."""

    def compute_diff(τ_l):
        return _ss_diff_for_tau_l(
            τ_l, price_arr, policy_no_τl, α_arr, ε_arr, frac, n,
            β_arr, ρ_d, σ, B, C, R, Q, H, P_end, ξ_end, x0, Σ0,
            Ind_work
        )

    return _grid_refine(compute_diff, -0.5, 1.0 - 1e-5, unit_grid, TOL_SS, 10)
```

GDP is the sum of capital and labor income shares

```
def _compute_gdp(μa_arr, frac, ε_arr, Gb, r, w, n, x0_0, frac_0):
    """Compute GDP from aggregates."""

    ε_agg = jnp.sum(frac * ε_arr)
    a_agg = jnp.sum(frac * μa_arr)
    k_agg = a_agg - Gb
    k_share = r * (k_agg / n + frac_0 * x0_0)
    l_share = w * ε_agg
    return k_agg, k_share + l_share
```

`find_ss_exo` ties the pieces together: it solves for τ_ℓ, evaluates the full steady state, and returns a `SteadyState` named tuple

```
def find_ss_exo(price, policy_target, hh, tech):
    """Find steady state with exogenous prices by solving for τ_l."""

    frac, ε_arr, n, x0 = hh.frac, hh.ε_arr, hh.n, hh.x0
    RR, w = price
    r = RR - 1 + tech.δ

    τ_a, τ_0, benef, G, Gb = policy_target
```

(continues on next page)

(continued from previous page)

```
    price_arr = jnp.array([RR, w])
    policy_no_τl = jnp.array([τ_a, τ_0, benef, G, Gb])

    τ_l = _find_ss_tau_l(
        price_arr, policy_no_τl,
        hh.α_arr, hh.ε_arr, hh.frac, hh.n,
        hh.β_arr, hh.ρ_d, hh.σ, hh.B, hh.C, hh.R, hh.Q, hh.H,
        hh.P_end, hh.ξ_end, hh.x0, hh.Σ0,
        hh.Ind_work, UNIT_GRID
    )

    price_arr = jnp.array([RR, w])
    policy_arr = jnp.array([float(τ_l), τ_a, τ_0, benef, G, Gb])
    diff, P_arr, ξ_arr, Ao_arr, F_arr, μx_arr, μc_arr, Σx_arr, Vc_arr = \
        ss_imbalance(
            price_arr, policy_arr,
            hh.α_arr, hh.ε_arr, hh.frac, hh.n,
            hh.β_arr, hh.ρ_d, hh.σ,
            hh.B, hh.C, hh.R, hh.Q, hh.H,
            hh.P_end, hh.ξ_end, hh.x0, hh.Σ0,
            hh.Ind_work
        )

    k_agg, gdp = _compute_gdp(
        μx_arr[1:, 0], frac, ε_arr,
        Gb, r, w, n, x0[0], frac[0]
    )
    debt2gdp = Gb / gdp
    k2gdp = k_agg / gdp

    return SteadyState(
        P_arr=P_arr, ξ_arr=ξ_arr, Ao_arr=Ao_arr, F_arr=F_arr,
        μx_arr=μx_arr, μc_arr=μc_arr, Σx_arr=Σx_arr, Vc_arr=Vc_arr,
        debt2gdp=float(debt2gdp), τ_l=float(τ_l), benef=benef, Gb=Gb,
        k_bar=float(k_agg), RR=RR, w=w, r=float(r), k2gdp=float(k2gdp)
    )
```

The initial fiscal policy sets a social security replacement rate of $\theta = 0.6$

```
aveinc = tech.w * sum(hh.ε_arr) / (hh.T1 + 1)
θ = 0.6
benef_0 = aveinc * θ

G_0 = 1.44          # government purchases
Gb_0 = 2.8 * G_0    # government debt
τ_l_0 = 0.3385      # labor income tax
τ_a_0 = 0.30        # capital income tax
τ_0_0 = 0           # lump-sum tax

RR, w = tech.RR, tech.w
```

83.6.2 Initial and terminal steady states

The initial steady state features a calibrated replacement rate ($\theta = 0.6$), positive social security benefits, and government expenditure and debt set to match targets.

The transition dates are $s_1 = 0$ and $s_2 = 40$, with horizons $S = 140$ (exogenous prices) and $S = 200$ (endogenous prices).

```
S_exo = 140
S_endo = 200
S1, S2 = 0, 40
S3 = S2 + 2 * hh.T0

RR_exo, w_exo = tech.RR, tech.w
```

Two helper functions build the price and policy arrays that the transition solver expects.

Under the small open economy assumption, prices are constant over time.

```
def make_exo_price_seq(S, RR, w):
    """Construct constant price sequence for small open economy."""

    return jnp.column_stack([jnp.full(S + 2, RR), jnp.full(S + 2, w)])
```

The policy sequence sets τ_ℓ to the initial steady-state value before s_1 and to the terminal value after s_2, while holding all other fiscal instruments constant.

```
def make_policy_seq(S, ss0_τl, ss1_τl, S1, S2, τ_a, τ_0, benef, G, Gb):
    """Construct policy sequence with initial/terminal
    τ_l and constant other policies."""

    policy_seq = jnp.empty((S + 2, 6))
    policy_seq = policy_seq.at[:S1 + 1, 0].set(ss0_τl)
    policy_seq = policy_seq.at[S2 + 1:, 0].set(ss1_τl)
    policy_seq = policy_seq.at[:, 1].set(τ_a)
    policy_seq = policy_seq.at[:, 2].set(τ_0)
    policy_seq = policy_seq.at[:, 3].set(benef)
    policy_seq = policy_seq.at[:, 4].set(G)
    policy_seq = policy_seq.at[:, 5].set(Gb)
    return policy_seq
```

```
ss0 = find_ss_exo((RR, w), (τ_a_0, τ_0_0, benef_0, G_0, Gb_0), hh, tech)

print(f"Initial Steady State (s < 0):")
print(f"  Labor tax τ_l = {ss0.τ_l:.4f}")
print(f"  Interest rate r - δ = {ss0.r - tech.δ:.4f}")
print(f"  Capital/GDP = {ss0.k2gdp:.4f}")
print(f"  Debt/GDP = {ss0.debt2gdp:.4f}")
```

```
Initial Steady State (s < 0):
  Labor tax τ_l = 0.3383
  Interest rate r - δ = 0.0675
  Capital/GDP = 3.1615
  Debt/GDP = 0.5899
```

The following figure traces how the equilibrium labor tax rate varies with government debt in the terminal steady state (no social security)

```
Gb_arr = np.linspace(0.5 * Gb_0, 1.5 * Gb_0, 20)
τl_arr = np.empty_like(Gb_arr)
debt2gdp_arr = np.empty_like(Gb_arr)

for i, Gb in enumerate(Gb_arr):
    ss = find_ss_exo((RR, w), (τ_a_0, τ_0_0, 0, G_0, Gb), hh, tech)
    τl_arr[i] = ss.τ_l
    debt2gdp_arr[i] = ss.debt2gdp

fig, ax = plt.subplots()
ax.plot(τl_arr, debt2gdp_arr)
ax.hlines(ss0.debt2gdp, τl_arr.min(),
          np.maximum(τl_arr.max(), ss0.τ_l), linestyle='--', color='r')
ax.scatter(ss0.τ_l, ss0.debt2gdp)
ax.text(ss0.τ_l * 0.95, ss0.debt2gdp * 0.95, "ss0")
ax.text(0.07, 0.4, r"ss1($G_b$)")
ax.set_xlabel(r'$\tau_\ell$')
ax.set_ylabel('Debt/GDP')
plt.show()
```

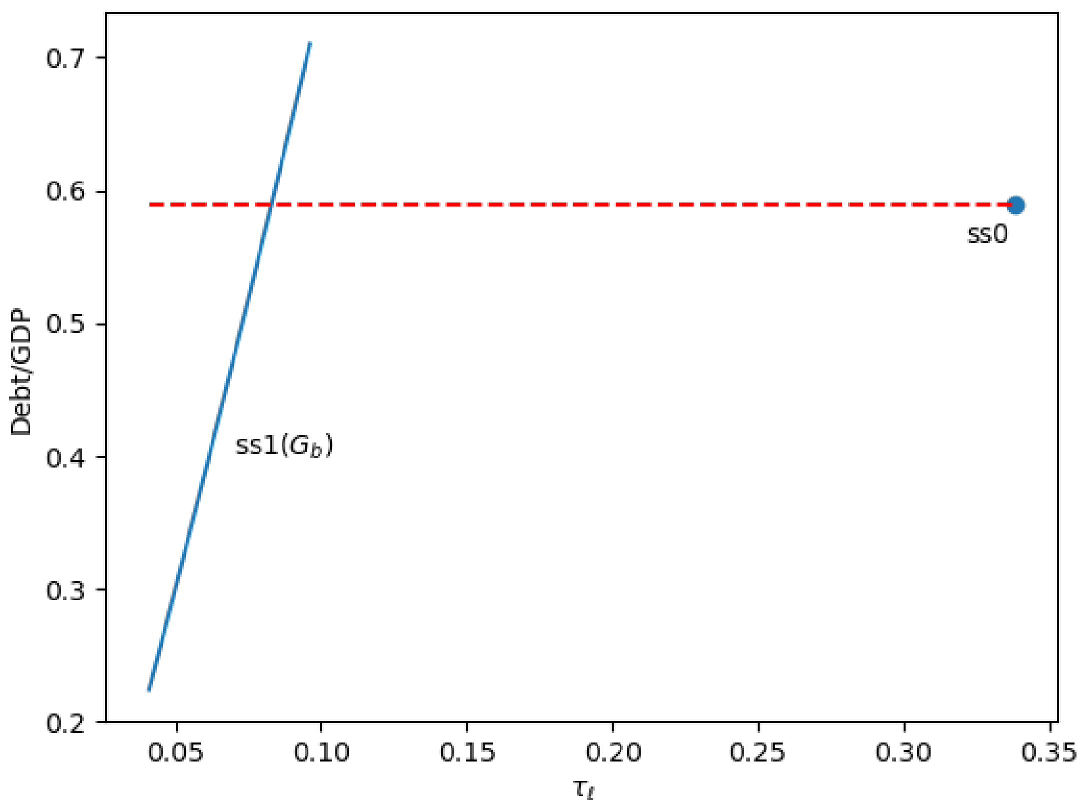


Fig. 83.4: Debt-to-GDP ratio as a function of the labor tax rate

Higher government debt requires larger interest payments, so the equilibrium labor tax rises.

The marked point shows the initial steady state.

To set the terminal steady state, we need to invert this relationship: given a target debt-to-GDP ratio, find the debt level

$\bar{b}$ and the associated τ_ℓ.

```
def _compute_debt2gdp_for_Gb(
        Gb, price_arr, policy_no_Gb,
        α_arr, ε_arr, frac, n,
        β_arr, ρ_d, σ, B, C, R, Q, H,
        P_end, ξ_end, x0, Σ0,
        δ, Ind_work, unit_grid):
    """Compute debt-to-GDP ratio for a given Gb."""

    RR, w = price_arr
    τ_a, τ_0, benef, G = policy_no_Gb
    r = RR - 1 + δ

    policy_no_τl = jnp.array([τ_a, τ_0, benef, G, Gb])

    # Reuse _find_ss_tau_l instead of duplicating grid search
    τ_l = _find_ss_tau_l(
        price_arr, policy_no_τl, α_arr, ε_arr, frac, n,
        β_arr, ρ_d, σ, B, C, R, Q, H, P_end, ξ_end, x0, Σ0,
        Ind_work, unit_grid
    )

    policy_arr = jnp.array([τ_l, τ_a, τ_0, benef, G, Gb])
    _, _, _, _, _, μx_arr, _, _, _ = ss_imbalance(
        price_arr, policy_arr,
        α_arr, ε_arr, frac, n,
        β_arr, ρ_d, σ, B, C, R, Q, H, P_end, ξ_end, x0, Σ0,
        Ind_work
    )

    _, gdp = _compute_gdp(
        μx_arr[1:, 0], frac, ε_arr,
        Gb, r, w, n, x0[0], frac[0]
    )
    return Gb / gdp
```

_find_Gb_for_debt2gdp searches over $\bar{b}$ values via grid refinement to match a target debt-to-GDP ratio.

```
@jit
def _find_Gb_for_debt2gdp(
        debt2gdp_target, price_arr, policy_no_Gb,
        α_arr, ε_arr, frac, n,
        β_arr, ρ_d, σ, B, C, R, Q, H,
        P_end, ξ_end, x0, Σ0,
        δ, Ind_work, unit_grid, unit_grid_tau):
    """Find Gb consistent with a target debt-to-GDP ratio."""

    RR, w = price_arr
    τ_a, τ_0, benef, G = policy_no_Gb

    def compute_diff_coarse(Gb):
        debt2gdp = _compute_debt2gdp_for_Gb(
            Gb, price_arr, policy_no_Gb, α_arr, ε_arr, frac, n,
            β_arr, ρ_d, σ, B, C, R, Q, H, P_end, ξ_end, x0, Σ0,
            δ, Ind_work, unit_grid_tau
        )
        return debt2gdp - debt2gdp_target
```

(continues on next page)

(continued from previous page)

```
    return _grid_refine(compute_diff_coarse, -40.0, 20.0, unit_grid, TOL_SS, 5)
```

`ss_target_debt2gdp_exo` finds the debt level consistent with the target ratio, then computes the full steady state.

```
def ss_target_debt2gdp_exo(debt2gdp_target, policy_target, price, hh, tech):
    """Find steady state with target debt-to-GDP ratio."""

    τ_a, τ_0, benef, G = policy_target
    RR, w = price

    price_arr = jnp.array([RR, w])
    policy_no_Gb = jnp.array([τ_a, τ_0, benef, G])

    Gb = _find_Gb_for_debt2gdp(
        float(debt2gdp_target), price_arr, policy_no_Gb,
        hh.α_arr, hh.ε_arr, hh.frac, hh.n,
        hh.β_arr, hh.ρ_d, hh.σ, hh.B, hh.C, hh.R, hh.Q, hh.H,
        hh.P_end, hh.ξ_end, hh.x0, hh.Σ0, tech.δ,
        hh.Ind_work, UNIT_GRID, UNIT_GRID
    )

    return find_ss_exo((RR, w), (τ_a, τ_0, benef, G, float(Gb)), hh, tech)
```

The terminal steady state eliminates social security ($\theta = 0$) while matching the initial debt-to-GDP ratio

```
ss1 = ss_target_debt2gdp_exo(
    ss0.debt2gdp, (τ_a_0, τ_0_0, 0, G_0), (RR_exo, w_exo), hh, tech
)

print(f"\nTerminal Steady State (s >= s3):")
print(f"  Labor tax τ_l = {ss1.τ_l:.4f}")
print(f"  Benefits θ = 0")
print(f"  Capital/GDP = {ss1.k2gdp:.4f}")
print(f"  Debt/GDP = {ss1.debt2gdp:.4f}")
```

```
Terminal Steady State (s >= s3):
  Labor tax τ_l = 0.0831
  Benefits θ = 0
  Capital/GDP = 4.1567
  Debt/GDP = 0.5901
```

83.7 Transition path computation

The transition path describes how the economy moves from the initial steady state (with social security) to the terminal steady state (after reform).

This is more complex than the steady-state computation because prices and policies change over time, so each cohort faces a unique lifetime sequence of tax and benefit rates.

`solve_backwards` solves the household problem backward in time during the transition, computing optimal decision rules $F_t(s)$ and closed-loop transition matrices $A_t^o(s)$ at each calendar date s and age t.

```
@jit
def solve_backwards(
        price_seq, policy_seq,
        P_arr_ss1, ξ_arr_ss1,
        ε_arr, β_arr, Ind_work,
        ρ_d, σ, B, C, R, Q, H,
        P_end, ξ_end,
        s_indices, ages):
    """Backward Riccati scan over all dates and ages."""

    # Infer dimensions from input arrays
    n_x = P_end.shape[0]
    S = s_indices.shape[0] - 1

    def solve_all_ages(P_next_all, ξ_next_all, RR_s, w_s, τ_l, τ_a, τ_0, benef):
        def solve_one_age(t, P_next, ξ_next):
            ε_t = ε_arr[t]
            β_t = β_arr[t]
            Ind_work_t = Ind_work[t]
            F, Ao, P, ξ = solve_riccati_step(
                ε_t, β_t, Ind_work_t, RR_s, w_s, τ_l, τ_a, τ_0, benef,
                P_next, ξ_next, ρ_d, σ, B, C, R, Q, H
            )
            return F, Ao, P, ξ

        P_next_shifted = P_next_all[1:]
        ξ_next_shifted = ξ_next_all[1:]

        F_all, Ao_all, P_all, ξ_all = vmap(
            solve_one_age
        )(ages, P_next_shifted, ξ_next_shifted)
        return F_all, Ao_all, P_all, ξ_all

    def scan_body(carry, s_inv):
        P_next_seq, ξ_next_seq = carry
        s = S - s_inv

        RR_s = price_seq[s, 0]
        w_s = price_seq[s, 1]
        τ_l = policy_seq[s, 0]
        τ_a = policy_seq[s, 1]
        τ_0 = policy_seq[s, 2]
        benef = policy_seq[s, 3]

        F_s, Ao_s, P_s, ξ_s = solve_all_ages(
            P_next_seq, ξ_next_seq, RR_s, w_s, τ_l, τ_a, τ_0, benef
        )

        # Build P_curr and ξ_curr using the known shapes from input arrays
        P_curr = jnp.zeros_like(P_arr_ss1)
        P_curr = P_curr.at[:T0+1].set(P_s)
        P_curr = P_curr.at[-1].set(P_end)

        ξ_curr = jnp.zeros_like(ξ_arr_ss1)
        ξ_curr = ξ_curr.at[:T0+1].set(ξ_s)
        ξ_curr = ξ_curr.at[-1].set(ξ_end)
```

(continues on next page)

(continued from previous page)

```
        output = (F_s, Ao_s, P_s, ξ_s)
        new_carry = (P_curr, ξ_curr)

        return new_carry, output

    init_carry = (P_arr_ss1, ξ_arr_ss1)

    # s_indices already has the right length
    _, outputs = lax.scan(scan_body, init_carry, s_indices)

    F_seq, Ao_seq, P_seq_inner, ξ_seq_inner = outputs

    F_seq = jnp.flip(F_seq, axis=0)
    Ao_seq = jnp.flip(Ao_seq, axis=0)

    # Build output arrays using shapes from price_seq
    P_seq = jnp.zeros((price_seq.shape[0], P_arr_ss1.shape[0], n_x, n_x))
    ξ_seq = jnp.zeros((price_seq.shape[0], ξ_arr_ss1.shape[0]))

    P_seq_inner = jnp.flip(P_seq_inner, axis=0)
    ξ_seq_inner = jnp.flip(ξ_seq_inner, axis=0)
    P_seq = P_seq.at[:S+1, :T0+1].set(P_seq_inner)
    ξ_seq = ξ_seq.at[:S+1, :T0+1].set(ξ_seq_inner)

    P_seq = P_seq.at[:, -1].set(P_end)
    ξ_seq = ξ_seq.at[:, -1].set(ξ_end)
    P_seq = P_seq.at[-1, :].set(P_arr_ss1)
    ξ_seq = ξ_seq.at[-1, :].set(ξ_arr_ss1)

    return F_seq, Ao_seq, P_seq, ξ_seq
```

`simulate_forwards` takes the computed decision rules and simulates the economy forward from the initial distribution, tracking the evolution of asset means and variances across cohorts through the transition.

```
@jit
def simulate_forwards(
        Ao_seq, F_seq, μx_init, Σx_init,
        C, x0, Σ0, s_indices, ages):
    """Forward moment scan over all dates and ages."""

    # Infer dimensions from input arrays
    n_x = x0.shape[0]
    CCT = C @ C.T
    S = s_indices.shape[0] - 1

    def simulate_all_ages(μx_curr, Σx_curr, Ao_s, F_s):
        def simulate_one_age(t, μx_t, Σx_t, Ao_t, F_t):
            return forward_moment_step(
                μx_t, Σx_t, Ao_t, F_t, CCT
            )

        μx_next_all, μc_all, Σx_next_all, Vc_all = vmap(simulate_one_age)(
            ages, μx_curr[:T0+1], Σx_curr[:T0+1], Ao_s, F_s
        )
        return μx_next_all, μc_all, Σx_next_all, Vc_all

    def scan_body(carry, s):
```

(continues on next page)

(continued from previous page)

```
        μx_curr, Σx_curr = carry

        Ao_s = Ao_seq[s]
        F_s = F_seq[s]

        μx_next_inner, μc_s, Σx_next_inner, Vc_s = simulate_all_ages(
            μx_curr, Σx_curr, Ao_s, F_s
        )

        # Use shapes from μx_init
        μx_next = jnp.zeros_like(μx_init)
        μx_next = μx_next.at[0].set(x0)
        μx_next = μx_next.at[1:T0+2].set(μx_next_inner)

        Σx_next = jnp.zeros_like(Σx_init)
        Σx_next = Σx_next.at[0].set(Σ0)
        Σx_next = Σx_next.at[1:T0+2].set(Σx_next_inner)

        output = (μx_curr, μc_s, Σx_curr, Vc_s)
        new_carry = (μx_next, Σx_next)

        return new_carry, output

    init_carry = (μx_init, Σx_init)
    final_carry, outputs = lax.scan(scan_body, init_carry, s_indices)

    μx_seq_inner, μc_seq, Σx_seq_inner, Vc_seq = outputs

    # Build output arrays using inferred sizes
    μx_seq = jnp.zeros((S + 2, T0 + 2, n_x))
    Σx_seq = jnp.zeros((S + 2, T0 + 2, n_x, n_x))

    μx_seq = μx_seq.at[:S+1].set(μx_seq_inner)
    Σx_seq = Σx_seq.at[:S+1].set(Σx_seq_inner)

    μx_seq = μx_seq.at[S+1].set(final_carry[0])
    Σx_seq = Σx_seq.at[S+1].set(final_carry[1])

    return μx_seq, μc_seq, Σx_seq, Vc_seq
```

Given a candidate transition tax rate τ_ℓ^{trans}, the function `transition_paths` constructs the complete policy sequence, solves backward, simulates forward, computes the capital and debt paths by aggregating across cohorts, and returns the terminal debt carryover that we seek to drive to zero.

```
@jit
def _transition_paths(
        τ_l_trans, price_seq, policy_seq,
        ss1_P_arr, ss1_ξ_arr, ss1_Gb,
        μx_init, Σx_init, k_bar_init,
        s_indices, age_range, S1, S2,
        ε_arr, β_arr, Ind_work,
        ρ_d, σ, B, C, R, Q, H,
        P_end, ξ_end, x0, Σ0,
        frac, n):
    """Solve backward and simulate forward for a given τ_l_trans."""
```

(continues on next page)

(continued from previous page)

```
    # Infer dimensions from input arrays
    n_x = x0.shape[0]
    S = s_indices.shape[0] - 2

    # Derive variants via slicing
    s_indices_scan = s_indices[:-1]       # arange(S+1)
    ages = age_range[:-1]                  # arange(T0+1)
    capital_indices = s_indices[1:-1]     # arange(1, S+1)

    # Update policy sequence with transition tax using dynamic indexing
    mask = (s_indices >= S1 + 1) & (s_indices <= S2)
    τ_l_col = jnp.where(mask, τ_l_trans, policy_seq[:, 0])
    policy_seq = policy_seq.at[:, 0].set(τ_l_col)

    # Solve backwards
    F_seq, Ao_seq, P_seq, ξ_seq = solve_backwards(
        price_seq, policy_seq, ss1_P_arr, ss1_ξ_arr,
        ε_arr, β_arr, Ind_work,
        ρ_d, σ, B, C, R, Q, H,
        P_end, ξ_end,
        s_indices_scan, ages
    )

    # Simulate forwards
    μx_seq, μc_seq, Σx_seq, Vc_seq = simulate_forwards(
        Ao_seq, F_seq, μx_init, Σx_init, C, x0, Σ0, s_indices_scan, ages
    )

    # Compute capital path
    ε_agg = jnp.sum(ε_arr * frac)
    frac0_x0 = frac[0] * x0[0]

    def capital_step(k_prev, s):
        RR = price_seq[s, 0]
        w = price_seq[s, 1]
        G = policy_seq[s, 4]
        c_agg = jnp.sum(μc_seq[s] * frac)
        k_new = RR * (frac0_x0 + k_prev / n) - G - c_agg + w * ε_agg
        return k_new, k_new

    # capital_indices is pre-created arange(1, S+1)
    _, k_path = lax.scan(capital_step, k_bar_init, capital_indices)
    k_seq = jnp.concatenate([jnp.array([k_bar_init]), k_path])

    # Compute debt path
    a_seq = jnp.sum(μx_seq[1:, 1:, 0] * frac, axis=1)
    Gb_seq = a_seq - k_seq

    carryover = Gb_seq[-1] - ss1_Gb

    return carryover, μx_seq, μc_seq, k_seq, Gb_seq, F_seq, Ao_seq
```

`transition_paths` unpacks the steady-state and household objects and calls the JIT-compiled inner function.

```
def transition_paths(
        τ_l_trans, price_seq, policy_seq,
        ss0, ss1, hh, tech,
```

(continues on next page)

(continued from previous page)

```
        S, S1, S2, μx_init, Σx_init):
    """Compute transition path."""

    policy_seq = jnp.asarray(policy_seq)
    price_seq = jnp.asarray(price_seq)

    # Pre-create iteration arrays (use slicing for variants)
    s_indices = jnp.arange(S + 2)

    carryover, μx_seq, μc_seq, k_seq, Gb_seq, F_seq, Ao_seq = _transition_paths(
        float(τ_l_trans), price_seq, policy_seq,
        ss1.P_arr, ss1.ξ_arr, float(ss1.Gb),
        μx_init, Σx_init, float(ss0.k_bar),
        s_indices, AGE_INDICES,
        S1, S2,
        hh.ε_arr, hh.β_arr, hh.Ind_work,
        hh.ρ_d, hh.σ, hh.B, hh.C,
        hh.R, hh.Q, hh.H,
        hh.P_end, hh.ξ_end, hh.x0, hh.Σ0,
        hh.frac, hh.n
    )

    return (float(carryover), μx_seq, μc_seq,
            k_seq, Gb_seq, F_seq, Ao_seq)
```

83.7.1 Shooting method

To find the correct transition tax rate, we use a shooting method.

If the tax rate is too low, debt explodes; if it is too high, debt falls below the target.

The equilibrium tax rate is found where the terminal debt exactly hits the target.

We start by computing two transition paths with different trial tax rates to illustrate the shooting method

```
price_seq = make_exo_price_seq(S_exo, RR, w)
policy_seq_base = make_policy_seq(S_exo, ss0.τ_l, ss1.τ_l, S1, S2,
                                  τ_a_0, τ_0_0, 0, G_0, Gb_0)

τ_l_low = 0.14
τ_l_high = 0.17

_, μx_seq1, μc_seq1, k_seq1, Gb_seq1, _, _ = transition_paths(
    τ_l_low, price_seq, policy_seq_base, ss0, ss1, hh, tech,
    S_exo, S1, S2, ss0.μx_arr, ss0.Σx_arr)

_, μx_seq2, μc_seq2, k_seq2, Gb_seq2, _, _ = transition_paths(
    τ_l_high, price_seq, policy_seq_base, ss0, ss1, hh, tech,
    S_exo, S1, S2, ss0.μx_arr, ss0.Σx_arr)
```

We can plot the resulting debt paths to see how they differ under the two trial tax rates

```
fig, ax = plt.subplots(figsize=(10, 5))
ax.plot(Gb_seq1, 'b-', linewidth=2,
        label=f'$\\tau_\\ell$ = {τ_l_low:.2f} (too low)')
ax.plot(Gb_seq2, 'r-', linewidth=2,
```

(continues on next page)

(continued from previous page)

```
        label=f'$\\tau_\\ell$ = {τ_l_high:.2f} (too high)')
ax.axhline(ss1.Gb, color='k', linestyle='--',
           label=f'Target $G_b$ = {ss1.Gb:.2f}')
ax.axvspan(S1, S2, alpha=0.1, color='yellow', label='Transition period')
ax.set_xlabel('Time')
ax.set_ylabel('Government debt')
ax.legend()
plt.show()
```

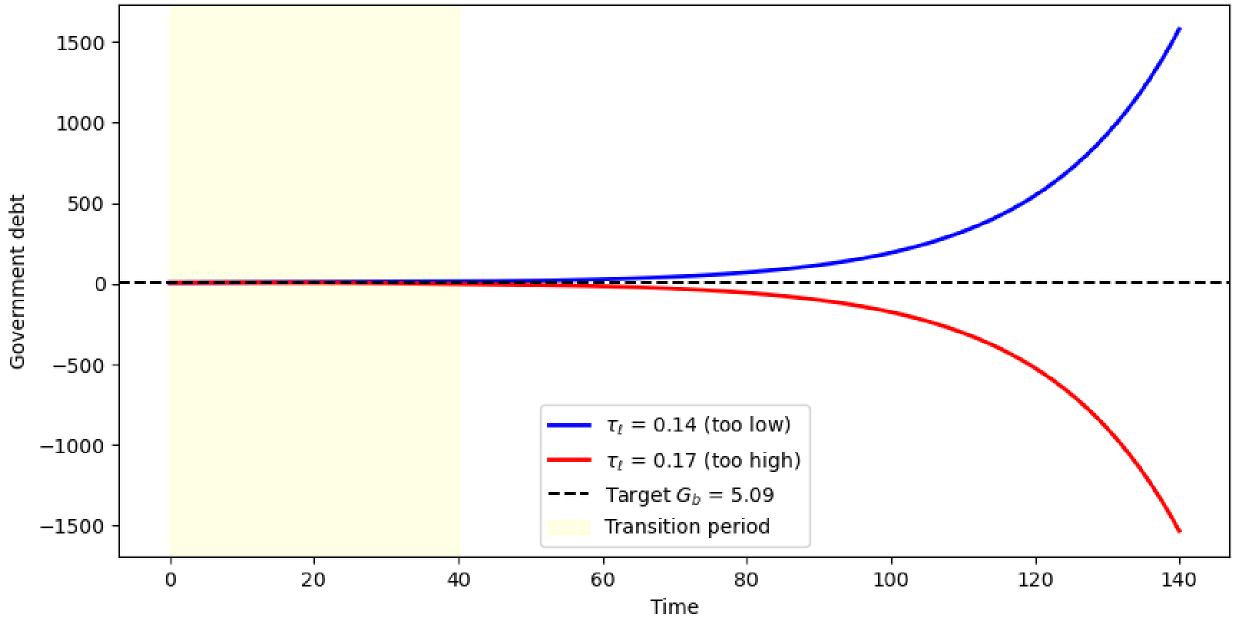


Fig. 83.5: Shooting method for finding the transition tax rate

The blue curve shows government debt increasing over time when the tax rate is too low, while the red curve shows debt falling below the target when the tax rate is too high.

The equilibrium transition tax rate lies between these two extremes, at the value where terminal debt exactly matches the target G_b.

Bisection automates this shooting procedure.

For Experiment 1, each cohort alive at s_1 must also be compensated by the present value of the social security benefits it would have received under the original system.

`_compute_compensation` evaluates this present value for a single cohort identified by its remaining lifetime.

```
def _compute_compensation(
        death_time, τ_l_seq, τ_a_seq,
        benef_diff, RR_seq, w_seq,
        ε_arr, indices, ss0_τ_l):
    """Present value of lost benefits for one cohort."""

    n_periods = death_time + 1
    age_at_0 = T0 - death_time
```

(continues on next page)

(continued from previous page)

```
    time_mask = indices < n_periods
    age_mask = indices >= age_at_0

    τ_l_cohort = jnp.where(time_mask, τ_l_seq[:T0 + 1], 0.0)
    τ_a_cohort = jnp.where(time_mask, τ_a_seq[:T0 + 1], 0.0)
    RR_cohort = jnp.where(time_mask, RR_seq[:T0 + 1], 1.0)
    w_cohort = jnp.where(time_mask, w_seq[:T0 + 1], 0.0)

    ε_masked = jnp.where(age_mask, ε_arr, 0.0)
    benef_masked = jnp.where(age_mask, benef_diff, 0.0)
    benef_masked = jnp.where(ε_masked != 0, 0.0, benef_masked)

    age_idx = jnp.clip(age_at_0 + indices, 0, T0)
    ε_cohort = jnp.where(
        time_mask, ε_arr[age_idx], 0.0
    )
    benef_cohort = jnp.where(
        time_mask, benef_diff[age_idx], 0.0
    )
    benef_cohort = jnp.where(ε_cohort != 0, 0.0, benef_cohort)

    RR_tilde_seq = RR_cohort - τ_a_cohort * (RR_cohort - 1)
    RR_tilde_seq = jnp.where(time_mask, RR_tilde_seq, 1.0)
    discount_factors = jnp.cumprod(RR_tilde_seq)

    labor_loss = w_cohort * ε_cohort * (τ_l_cohort - ss0_τ_l)

    pv_seq = jnp.where(
        time_mask,
        (benef_cohort + labor_loss) / discount_factors,
        0.0
    )

    valid = (death_time >= 0) & (death_time < T0)
    return jnp.where(valid, jnp.sum(pv_seq), 0.0)
```

`apply_compensation` vectorizes this calculation across all cohorts with `vmap` and adds the result to each cohort's initial asset holdings.

```
@jit
def apply_compensation(
        μx_arr_ss0, Σx_arr_ss0,
        τ_l_seq, τ_a_seq, benef_diff,
        RR_seq, w_seq, ε_arr, ss0_τ_l,
        ages_full, ages, x0, Σ0,
        comp_mult):
    """Vectorize compensation across cohorts and adjust initial assets."""

    def compute_comp_for_age(age):
        death_time = T0 - age
        comp = _compute_compensation(
            death_time,
            τ_l_seq, τ_a_seq, benef_diff,
            RR_seq, w_seq, ε_arr,
            ages, ss0_τ_l
        )
        valid = (age >= 1) & (age <= T0)
```

(continues on next page)

(continued from previous page)

```
        return jnp.where(valid, comp, 0.0)

    compensations = vmap(compute_comp_for_age)(ages_full)

    μx_init = jnp.zeros_like(μx_arr_ss0)
    Σx_init = jnp.zeros_like(Σx_arr_ss0)

    μx_init = μx_init.at[0].set(x0)
    Σx_init = Σx_init.at[0].set(Σ0)
    μx_init = μx_init.at[-1].set(x0)
    Σx_init = Σx_init.at[-1].set(Σ0)

    μx_init = μx_init.at[1:-1].set(μx_arr_ss0[1:-1])
    Σx_init = Σx_init.at[1:-1].set(Σx_arr_ss0[1:-1])

    # comp_mult: 0.0 = no compensation, 1.0 = full
    μx_init = μx_init.at[:, 0].add(comp_mult * compensations)

    return μx_init, Σx_init
```

`_transition_carryover` applies compensation, solves the transition, and returns the terminal debt carryover – the scalar the bisection drives to zero.

```
def _transition_carryover(
        τ_l_trans, price_seq, policy_seq,
        ss1_P_arr, ss1_ξ_arr, ss1_Gb,
        ss0_μx_arr, ss0_Σx_arr, k_bar_init,
        benef_diff, ss0_τ_l, comp_mult,
        s_indices, age_range, S1, S2,
        ε_arr, β_arr, Ind_work,
        ρ_d, σ, B, C, R, Q, H,
        P_end, ξ_end, x0, Σ0,
        frac, n):
    """Terminal debt carryover for a given transition τ_l."""

    ages = age_range[:-1]  # arange(T0+1)

    # Update policy sequence with transition tax
    mask = (s_indices >= S1 + 1) & (s_indices <= S2)
    τ_l_col = jnp.where(mask, τ_l_trans, policy_seq[:, 0])
    policy_seq_updated = policy_seq.at[:, 0].set(τ_l_col)

    # Compute initial conditions (compensation zeroed when comp_mult=0.0)
    μx_init, Σx_init = apply_compensation(
        ss0_μx_arr, ss0_Σx_arr,
        policy_seq_updated[:, 0], policy_seq_updated[:, 1], benef_diff,
        price_seq[:, 0], price_seq[:, 1],
        ε_arr, ss0_τ_l,
        age_range, ages, x0, Σ0,
        comp_mult
    )

    carryover, *_ = _transition_paths(
        τ_l_trans, price_seq, policy_seq,
        ss1_P_arr, ss1_ξ_arr, ss1_Gb,
        μx_init, Σx_init, k_bar_init,
        s_indices, age_range, S1, S2,
```

(continues on next page)

(continued from previous page)

```
        ε_arr, β_arr, Ind_work,
        ρ_d, σ, B, C, R, Q, H,
        P_end, ξ_end, x0, Σ0,
        frac, n
    )
    return carryover
```

We implement the bisection search in `_find_transition_tau_l`, which repeatedly evaluates `_transition_carryover` at the midpoint of a shrinking interval until the carryover is driven to zero.

```
@jit
def _find_transition_tau_l(
        price_seq, policy_seq, bounds,
        ss1_P_arr, ss1_ξ_arr, ss1_Gb,
        ss0_μx_arr, ss0_Σx_arr, k_bar_init,
        benef_diff, ss0_τ_l, comp_mult,
        s_indices, age_range, S1, S2,
        ε_arr, β_arr, Ind_work,
        ρ_d, σ, B, C, R, Q, H,
        P_end, ξ_end, x0, Σ0,
        frac, n):
    """Find transition τ_l using bisection.

    comp_mult controls compensation (0 or 1).
    """

    a, b = bounds[0], bounds[1]

    def compute_carryover(τ_l_trans):
        return _transition_carryover(
            τ_l_trans, price_seq, policy_seq,
            ss1_P_arr, ss1_ξ_arr, ss1_Gb,
            ss0_μx_arr, ss0_Σx_arr, k_bar_init,
            benef_diff, ss0_τ_l, comp_mult,
            s_indices, age_range,
            S1, S2,
            ε_arr, β_arr, Ind_work,
            ρ_d, σ, B, C, R, Q, H,
            P_end, ξ_end, x0, Σ0,
            frac, n
        )

    def cond_fn(state):
        a, b, fa, fb, i = state
        return (jnp.abs(b - a) > 1e-10) & (i < 100)

    def body_fn(state):
        a, b, fa, fb, i = state
        c = (a + b) / 2.0
        fc = compute_carryover(c)
        a_new = jnp.where(fa * fc > 0, c, a)
        b_new = jnp.where(fa * fc > 0, b, c)
        fa_new = jnp.where(fa * fc > 0, fc, fa)
        fb_new = jnp.where(fa * fc > 0, fb, fc)
        return (a_new, b_new, fa_new, fb_new, i + 1)

    fa, fb = compute_carryover(a), compute_carryover(b)
```

(continues on next page)

(continued from previous page)

```
    init_state = (a, b, fa, fb, 0)
    final_state = lax.while_loop(cond_fn, body_fn, init_state)
    a_final, b_final, _, _, _ = final_state

    return (a_final + b_final) / 2.0
```

The top-level wrapper `find_transition_exo` sets up the compensation parameters and calls the bisection solver, then recomputes the full transition path at the equilibrium tax rate.

```
def find_transition_exo(price_seq, policy_seq_base, ss0, ss1,
                        hh, tech, S, S1, S2,
                        compensation_data=None,
                        τl_bounds=(0.01, 0.6)):
    """Find transition tax rate under exogenous prices."""

    policy_seq = jnp.asarray(policy_seq_base)
    price_seq = jnp.asarray(price_seq)
    bounds = jnp.array([τl_bounds[0], τl_bounds[1]])
    s_indices = jnp.arange(S + 2)

    # Set up compensation parameters (default zeros when not using compensation)
    if compensation_data is not None:
        benef_diff, ss0_τ_l = compensation_data
        comp_mult = 1.0
    else:
        benef_diff = jnp.zeros(hh.T0 + 1)
        ss0_τ_l = ss0.τ_l
        comp_mult = 0.0

    # Find transition tax using unified function
    τ_l_trans = _find_transition_tau_l(
        price_seq, policy_seq, bounds,
        ss1.P_arr, ss1.ξ_arr, float(ss1.Gb),
        ss0.μx_arr, ss0.Σx_arr, float(ss0.k_bar),
        benef_diff, float(ss0_τ_l), comp_mult,
        s_indices, AGE_INDICES,
        S1, S2,
        hh.ε_arr, hh.β_arr, hh.Ind_work,
        hh.ρ_d, hh.σ, hh.B, hh.C,
        hh.R, hh.Q, hh.H,
        hh.P_end, hh.ξ_end, hh.x0, hh.Σ0,
        hh.frac, hh.n
    )
    τ_l_trans = float(τ_l_trans)

    # Compute final results with initial conditions
    mask = (s_indices >= S1 + 1) & (s_indices <= S2)
    τ_l_col = jnp.where(
        mask, τ_l_trans, policy_seq[:, 0]
    )
    policy_seq_final = policy_seq.at[:, 0].set(
        τ_l_col
    )

    μx_init, Σx_init = apply_compensation(
        ss0.μx_arr, ss0.Σx_arr,
        policy_seq_final[:, 0],
```

(continues on next page)

(continued from previous page)

```
        policy_seq_final[:, 1],
        benef_diff,
        price_seq[:, 0], price_seq[:, 1],
        hh.ε_arr, float(ss0_τ_l),
        AGE_INDICES, AGE_INDICES[:-1],
        hh.x0, hh.Σ0, comp_mult
    )

    results = transition_paths(
        τ_l_trans, price_seq, policy_seq,
        ss0, ss1, hh, tech, S, S1, S2,
        μx_init, Σx_init
    )

    return τ_l_trans, results
```

Bisection over the transition tax rate produces the equilibrium path.

```
τ_l_trans, results = find_transition_exo(
    price_seq, policy_seq_base, ss0, ss1,
    hh, tech, S_exo, S1, S2)

carryover, μx_seq, μc_seq, k_seq, Gb_seq, F_seq, Ao_seq = results
```

In the baseline case (no compensation), social security benefits are simply terminated.

```
τ_l_seq = np.zeros(S_exo + 1)
τ_l_seq[:S1 + 1] = ss0.τ_l
τ_l_seq[S1 + 1:S2 + 1] = τ_l_trans
τ_l_seq[S2 + 1:] = ss1.τ_l

fig, axes = plt.subplots(1, 2, figsize=(12, 4))

axes[0].plot(τ_l_seq, 'b-', linewidth=2)
axes[0].axhline(ss0.τ_l, color='r', linestyle=':',
                label=f'Initial $\\tau_\\ell$ = {ss0.τ_l:.4f}')
axes[0].axhline(ss1.τ_l, color='g', linestyle=':',
                label=f'Terminal $\\tau_\\ell$ = {ss1.τ_l:.4f}')
axes[0].axvspan(S1, S2, alpha=0.1, color='yellow')
axes[0].set_xlabel('Time')
axes[0].set_ylabel('Labor tax rate')
axes[0].set_title('Labor tax rate path')
axes[0].legend()


axes[1].plot(Gb_seq, 'b-', linewidth=2)
axes[1].axhline(ss0.Gb, color='r', linestyle=':',
                label=f'Initial $G_b$ = {ss0.Gb:.2f}')
axes[1].axhline(ss1.Gb, color='g', linestyle=':',
                label=f'Terminal $G_b$ = {ss1.Gb:.2f}')
axes[1].axvspan(S1, S2, alpha=0.1, color='yellow')
axes[1].set_xlabel('Time')
axes[1].set_ylabel('Government debt')
axes[1].set_title('Government debt path')
axes[1].legend()
```

(continues on next page)

(continued from previous page)

```
plt.tight_layout()
plt.show()
```

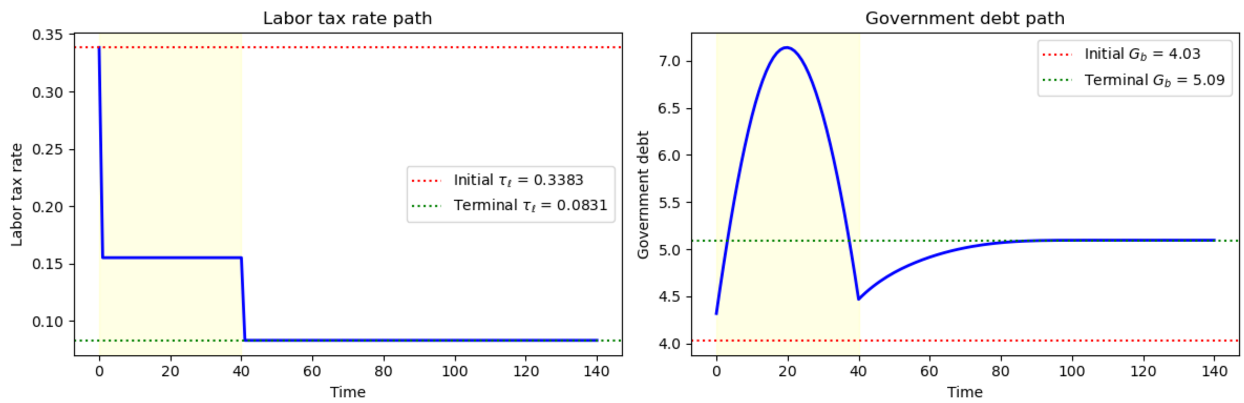


Fig. 83.6: Baseline transition path (no compensation)

The left panel shows the labor tax rate dropping during the transition because the government no longer needs to fund social security benefits.

The right panel shows government debt converging from its initial level to the new steady-state level, with the shaded region marking the transition period $[s_1, s_2)$.

83.8 Experiment 1: compensation through debt

In this experiment, the government terminates social security benefits but compensates affected generations.

Each cohort receives a transfer equal to the present value of the benefits they would have received – an actuarially fair buy-out.

Algorithm 83.8.1 (Fixed factor prices – Experiment 1 (buy-out))

Here $s_3 = s_2 + T_0 = 105$.

Step 1. Set up parameters.

Step 2. Solve the initial stationary equilibrium with constant social security benefit S: fix $\tau_a, \tau_0, S, G, \bar{b}$ and solve for $\tau_\ell = \tau_{\ell,0}$ such that the government budget balances:

```
F(τ_ℓ) = government budget imbalance
Find root of F(τ_ℓ) = 0.
```

Step 3. Solve the terminal stationary equilibrium with no social security: search over $\bar{b}$ so that the debt-to-GDP ratio matches a target:

```
H(Gb) = debt-to-GDP given Gb
        (internally solves F(τ_ℓ; Gb) = 0)
Find root of H(Gb) = target.
```

The associated labor tax is $\tau_{\ell,2}$.

Step 4. Solve the transition path: at $s = 0 = s_1$, all cohorts alive lose benefits and a cohort of age t receives a one-time compensation equal to the present value of lost benefits, discounted at the after-tax return $\tilde{R}(s) = R(s)[1 - \tau_a(s)] + \tau_a(s)$:

$$\text{comp}_t = S \sum_{j=\max(T_1 - t,\, 0)}^{T_0 - t} \prod_{i=0}^{j} \tilde{R}(s+i)^{-1}.$$

The government sets $\tau_{\ell,1}$ during $[s_1, s_2)$ and $\tau_{\ell,2}$ from s_2 onwards, with a one-time expenditure increase of $\sum f_t \, \text{comp}_t$ at s_1.

Find $\tau_{\ell,1}$ such that terminal government debt matches the target:

```
J(τ_ℓ) = terminal debt carryover
Find root of J(τ_ℓ) = Gb_terminal.
```

83.8.1 Fixed prices

We hold factor prices fixed and construct the price and policy sequences for the transition.

The benefit difference vector `benef_diff_exp1` records the per-period benefit loss for each age: retirees lose their old-regime benefits, while workers are unaffected.

```
ss1_exp1_exo = ss1

price_seq_exp1_exo = make_exo_price_seq(S_exo, RR_exo, w_exo)
policy_seq_exp1_exo = make_policy_seq(S_exo, ss0.τ_l, ss1.τ_l, S1, S2,
                                       τ_a_0, τ_0_0, 0, G_0, Gb_0)

benef_diff_exp1 = jnp.zeros(hh.T0 + 1)
benef_diff_exp1 = benef_diff_exp1.at[hh.T1 + 1:].set(ss0.benef)
```

The function `buyout_compensation_exp1_exo` computes the present-value compensation for each cohort alive at the reform date and adds it to their initial assets.

We then solve for the transition tax rate with and without the buy-out, so that we can compare the two paths.

```
def buyout_compensation_exp1_exo(τ_l_trans, policy_seq_base, price_seq):
    """Compute buy-out compensation under exogenous prices."""

    policy_seq = policy_seq_base.copy()
    policy_seq[S1 + 1:S2 + 1, 0] = τ_l_trans
    return apply_compensation(
        ss0.μx_arr, ss0.Σx_arr,
        policy_seq[:, 0], policy_seq[:, 1], benef_diff_exp1,
        price_seq[:, 0], price_seq[:, 1], hh.ε_arr, ss0.τ_l,
        AGE_INDICES, AGE_INDICES[:-1],
        hh.x0, hh.Σ0,
        1.0  # comp_mult = 1.0 for full compensation
    )

# Solve with buyout
τ_l_exp1_exo_bo, results_exp1_exo_bo = find_transition_exo(
    price_seq_exp1_exo, policy_seq_exp1_exo, ss0, ss1_exp1_exo,
    hh, tech, S_exo, S1, S2,
    compensation_data=(benef_diff_exp1, ss0.τ_l)
```

(continues on next page)

(continued from previous page)

```
)

# Solve without buyout (for comparison)
τ_l_exp1_exo_nb, results_exp1_exo_nb = find_transition_exo(
    price_seq_exp1_exo, policy_seq_exp1_exo, ss0, ss1_exp1_exo,
    hh, tech, S_exo, S1, S2
)
```

We compare the transition paths with and without buy-out compensation.

```
exp1_exo = {
    'ss0': ss0, 'ss1': ss1_exp1_exo,
    'τ_l_buyout': τ_l_exp1_exo_bo, 'τ_l_no_buyout': τ_l_exp1_exo_nb,
    'results_buyout': results_exp1_exo_bo,
    'results_no_buyout': results_exp1_exo_nb,
    'hh': hh, 'tech': tech
}
```

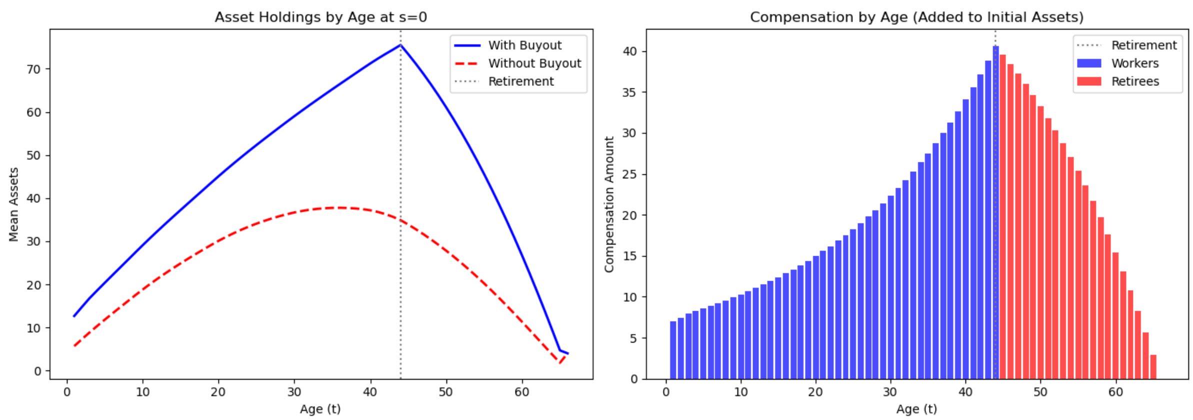
The following figure shows how the buy-out reshapes initial asset holdings across cohorts.

```
# Extract results
_, μx_seq_bo, μc_seq_bo, k_seq_bo, Gb_seq_bo, _, _ = exp1_exo['results_buyout']
results_nb = exp1_exo['results_no_buyout']
_, μx_seq_nb, μc_seq_nb, k_seq_nb, Gb_seq_nb, _, _ = results_nb

# Mean assets by age at time s=0 (with vs without buyout)
μa_bo = μx_seq_bo[0, 1:, 0]  # Assets at s=0 with buyout
μa_nb = μx_seq_nb[0, 1:, 0]  # Assets at s=0 without buyout

# Compensation = difference in initial assets
compensation_by_age = μa_bo - μa_nb

fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# Asset profiles
ages = np.arange(1, hh.T0 + 2)
axes[0].plot(ages, μa_bo, 'b-', linewidth=2, label='With Buyout')
axes[0].plot(ages, μa_nb, 'r--', linewidth=2, label='Without Buyout')
axes[0].axvline(hh.T1 + 1, color='gray', linestyle=':', label='Retirement')
axes[0].set_xlabel('Age (t)')
axes[0].set_ylabel('Mean Assets')
axes[0].set_title('Asset Holdings by Age at s=0')
axes[0].legend()


# Compensation histogram
working_ages = ages[ages <= hh.T1 + 1]
retired_ages = ages[ages > hh.T1 + 1]
comp_working = compensation_by_age[:hh.T1 + 1]
comp_retired = compensation_by_age[hh.T1 + 1:]

axes[1].bar(working_ages, comp_working,
            color='blue', alpha=0.7, label='Workers')
axes[1].bar(retired_ages, comp_retired,
            color='red', alpha=0.7, label='Retirees')
axes[1].axhline(0, color='k', linewidth=0.5)
axes[1].axvline(hh.T1 + 1, color='gray', linestyle=':', label='Retirement')
```

(continues on next page)

(continued from previous page)

```
axes[1].set_xlabel('Age (t)')
axes[1].set_ylabel('Compensation Amount')
axes[1].set_title('Compensation by Age (Added to Initial Assets)')
axes[1].legend()


plt.tight_layout()
plt.show()
```



Retirees receive the largest compensation because they were expecting benefits for the remainder of their lives.

Older workers receive significant compensation, while young workers receive little because they have their entire working lives to adjust.

The declining profile among retirees reflects the actuarial calculation: older retirees have fewer remaining years of expected benefits.

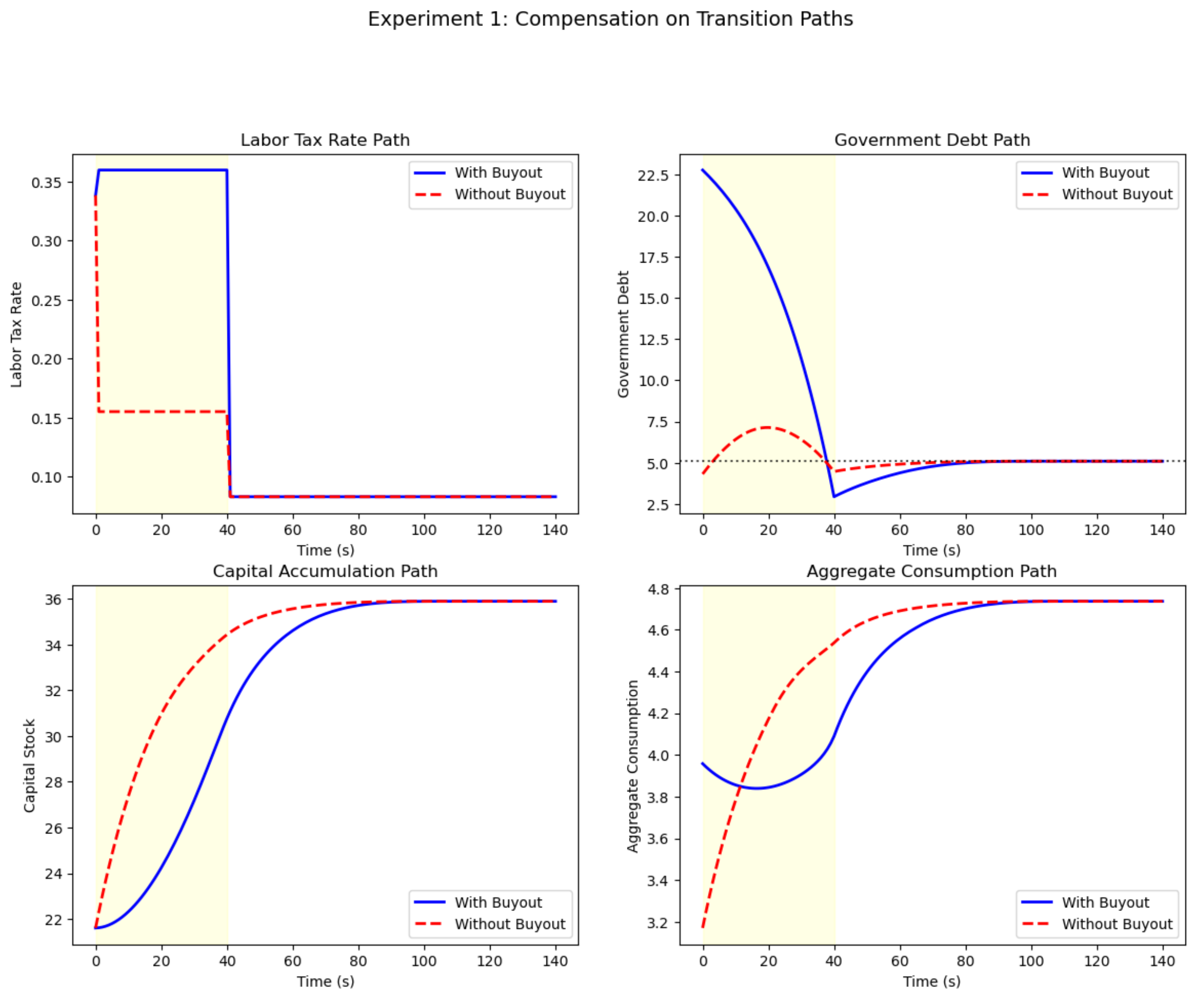
We now plot the aggregate transition paths for the labor tax, government debt, capital, and consumption under both schemes.

```
# hh, tech, ss0, ss1 already in scope — just alias from dict for readability
ss0_exp1 = exp1_exo['ss0']
ss1_exp1 = exp1_exo['ss1']

# Construct τ_l sequences
τ_l_seq_bo = np.zeros(S_exo + 1)
τ_l_seq_bo[:S1 + 1] = ss0_exp1.τ_l
τ_l_seq_bo[S1 + 1:S2 + 1] = exp1_exo['τ_l_buyout']
τ_l_seq_bo[S2 + 1:] = ss1_exp1.τ_l

τ_l_seq_nb = np.zeros(S_exo + 1)
τ_l_seq_nb[:S1 + 1] = ss0_exp1.τ_l
τ_l_seq_nb[S1 + 1:S2 + 1] = exp1_exo['τ_l_no_buyout']
τ_l_seq_nb[S2 + 1:] = ss1_exp1.τ_l

fig, axes = plt.subplots(2, 2, figsize=(14, 10))

# τ_l comparison
axes[0, 0].plot(τ_l_seq_bo, 'b-', linewidth=2, label='With Buyout')
axes[0, 0].plot(τ_l_seq_nb, 'r--', linewidth=2, label='Without Buyout')
axes[0, 0].axvspan(S1, S2, alpha=0.1, color='yellow')
axes[0, 0].set_xlabel('Time (s)')
```

(continues on next page)

(continued from previous page)

```
axes[0, 0].set_ylabel('Labor Tax Rate')
axes[0, 0].set_title('Labor Tax Rate Path')
axes[0, 0].legend()

# Gb comparison
axes[0, 1].plot(Gb_seq_bo, 'b-', linewidth=2, label='With Buyout')
axes[0, 1].plot(Gb_seq_nb, 'r--', linewidth=2, label='Without Buyout')
axes[0, 1].axhline(ss1_exp1.Gb, color='k', linestyle=':', alpha=0.7)
axes[0, 1].axvspan(S1, S2, alpha=0.1, color='yellow')
axes[0, 1].set_xlabel('Time (s)')
axes[0, 1].set_ylabel('Government Debt')
axes[0, 1].set_title('Government Debt Path')
axes[0, 1].legend()

# Capital path
axes[1, 0].plot(k_seq_bo, 'b-', linewidth=2, label='With Buyout')
axes[1, 0].plot(k_seq_nb, 'r--', linewidth=2, label='Without Buyout')
axes[1, 0].axvspan(S1, S2, alpha=0.1, color='yellow')
axes[1, 0].set_xlabel('Time (s)')
axes[1, 0].set_ylabel('Capital Stock')
axes[1, 0].set_title('Capital Accumulation Path')
axes[1, 0].legend()

# Aggregate consumption
c_agg_bo = np.array(μc_seq_bo[:S_exo + 1]) @ np.array(hh.frac)
c_agg_nb = np.array(μc_seq_nb[:S_exo + 1]) @ np.array(hh.frac)

axes[1, 1].plot(c_agg_bo, 'b-', linewidth=2, label='With Buyout')
axes[1, 1].plot(c_agg_nb, 'r--', linewidth=2, label='Without Buyout')
axes[1, 1].axvspan(S1, S2, alpha=0.1, color='yellow')
axes[1, 1].set_xlabel('Time (s)')
axes[1, 1].set_ylabel('Aggregate Consumption')
axes[1, 1].set_title('Aggregate Consumption Path')
axes[1, 1].legend()

plt.suptitle(
    'Experiment 1: Compensation on Transition Paths',
    fontsize=14, y=1.02
)
plt.show()
```



The buy-out scheme leads to a slower initial rise in private capital because the government must make large transfers.

Both schemes converge to the same terminal steady state.

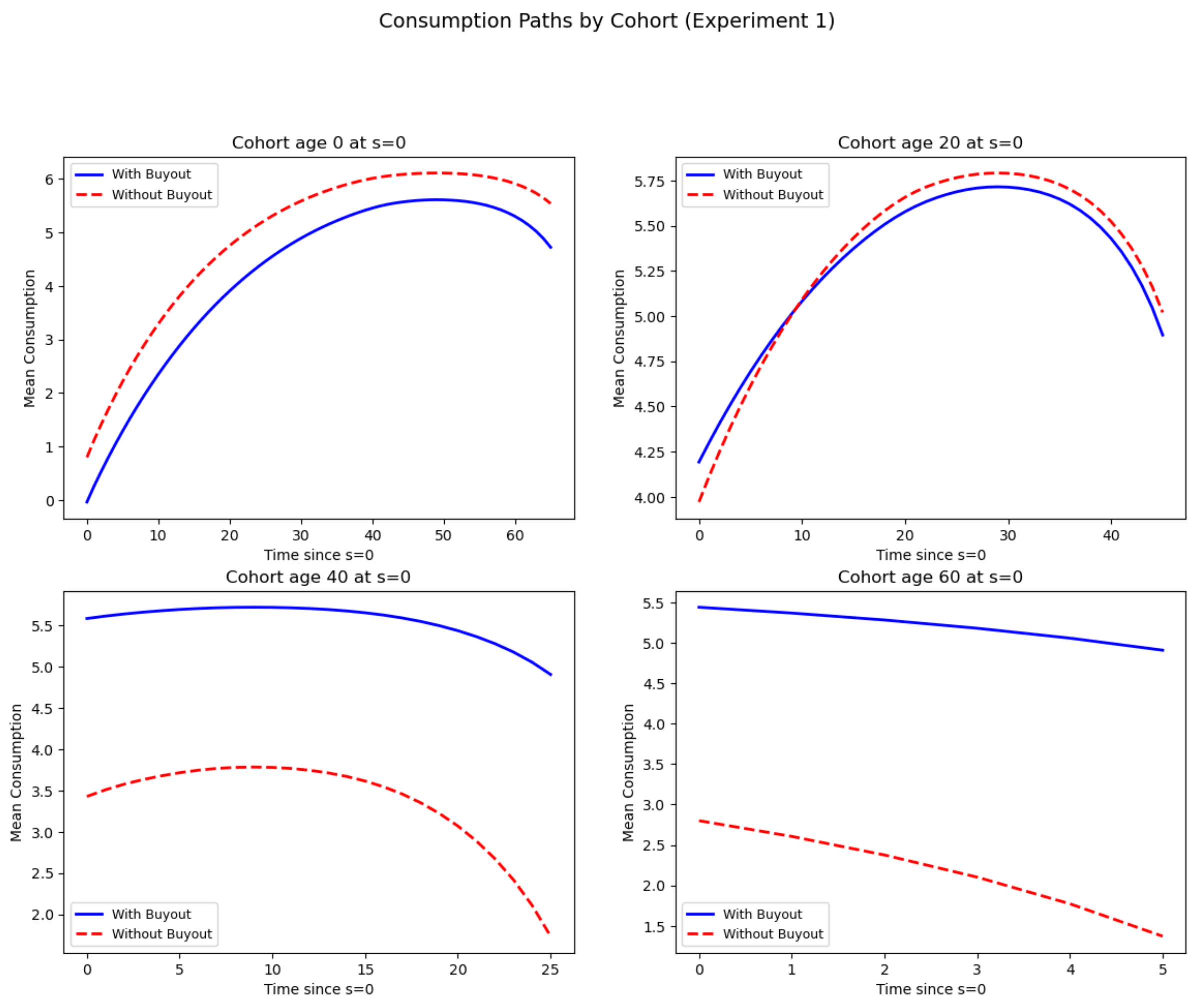
We now examine consumption paths for cohorts at different ages when the reform occurs.

```
fig, axes = plt.subplots(2, 2, figsize=(14, 10))

selected_ages = [0, 20, 40, 60]  # Cohorts at different ages at s=0

for idx, age_at_0 in enumerate(selected_ages):
    ax = axes[idx // 2, idx % 2]

    remaining_life = hh.T0 - age_at_0
    max_time = min(remaining_life + 1, S_exo + 1)

    c_bo = [μc_seq_bo[s, age_at_0 + s]
            for s in range(max_time)
            if age_at_0 + s <= hh.T0]
    c_nb = [μc_seq_nb[s, age_at_0 + s]
            for s in range(max_time)
            if age_at_0 + s <= hh.T0]

    ax.plot(c_bo, 'b-', linewidth=2, label='With Buyout')
```

(continues on next page)

(continued from previous page)

```
    ax.plot(c_nb, 'r--', linewidth=2, label='Without Buyout')
    ax.set_xlabel('Time since s=0')
    ax.set_ylabel('Mean Consumption')
    ax.set_title(f'Cohort age {age_at_0} at s=0')
    ax.legend(fontsize=9)


plt.suptitle(
    'Consumption Paths by Cohort (Experiment 1)',
    fontsize=14, y=1.02
)
plt.show()
```



Young workers (age 0) show nearly identical consumption paths because they receive little compensation and have decades to adjust their savings.

Near-retirement workers (age 40) exhibit more noticeable differences as the buyout compensation partially offsets their lost benefits.

Retirees (age 60) show the most dramatic difference: without compensation, their consumption drops sharply when benefits end, whereas the buyout scheme maintains higher consumption by replacing the lost income.

83.8.2 Endogenous prices

With endogenous factor prices, changes in saving behavior affect capital accumulation, which alters marginal products and feeds back into household decisions.

Algorithm 83.8.2 (Endogenous factor prices)

Here $s_3 = s_2 + 2T_0 = 170$.

Steps 1–3 are the same as *Algorithm 83.8.1*, with an outer fixed-point loop over factor prices to clear factor markets in steady state.

Step 4. The factor price sequences are now endogenous.

Wrap the fixed-price transition solver in a relaxation loop:

```
T(R_seq):
  1. Compute the wage sequence from R_seq via Cobb-Douglas.
  2. Taking prices as given, find root of J(τ_ℓ) = Gb_terminal.
  3. Compute the implied capital path, then the implied R*.
  Return R* - R_seq.

Iterate T(R_seq) = 0 with relaxation until convergence.
```

We iterate on the price sequence until factor markets clear.

```python
@jit
def compute_factor_prices(k_prod, ε_bar, A, α, δ):
    """Compute factor prices from Cobb-Douglas."""

    k_per_eff = k_prod / ε_bar
    r = A * α * (k_per_eff ** (α - 1))
    w = A * (1 - α) * (k_per_eff ** α)
    RR = 1 + r - δ
    return r, w, RR
```

With endogenous prices, finding a steady state requires an outer fixed-point loop: solve the household problem at given prices, compute the implied capital stock, update prices via Cobb-Douglas marginal products, and repeat until convergence.

```python
def find_ss_endo(
        debt2gdp_target, policy_target,
        hh, tech, RR_init=None, w_init=None,
        max_iter=50, tol=1e-5, verbose=False):
    """Find steady state with endogenous factor prices."""

    τ_a, τ_0, benef, G = policy_target
    ε_bar = float(jnp.sum(hh.frac * hh.ε_arr))

    RR = RR_init if RR_init else tech.RR
    w = w_init if w_init else tech.w

    relaxation = 0.3

    for iteration in range(max_iter):
        try:
            ss = ss_target_debt2gdp_exo(
```

(continues on next page)

(continued from previous page)

```
                debt2gdp_target,
                (τ_a, τ_0, benef, G),
                (RR, w), hh, tech
            )
        except ValueError:
            RR = RR * 0.99
            continue

        K_eff = ss.k_bar / hh.n + float(hh.frac[0] * hh.x0[0])
        r_new, w_new, RR_new = compute_factor_prices(
            K_eff, ε_bar,
            tech.A, tech.α_tilde, tech.δ
        )
        r_new = float(r_new)
        w_new = float(w_new)
        RR_new = float(RR_new)
        price_diff = abs(RR_new - RR) + abs(w_new - w)

        if verbose and iteration % 5 == 0:
            print(f"    SS iter {iteration}: "
                  f"RR={RR:.6f}, w={w:.4f}, "
                  f"k_bar={ss.k_bar:.4f}")

        if price_diff < tol:
            if verbose:
                print(f"    Converged at iteration {iteration}")
            break

        RR = RR + relaxation * (RR_new - RR)
        w = w + relaxation * (w_new - w)

    return ss_target_debt2gdp_exo(
        debt2gdp_target,
        (τ_a, τ_0, benef, G),
        (RR, w), hh, tech
    )
```

The price iteration also needs an initial guess for the transition price path and a way to update it after each inner solve.

The function `init_price_seq_interp` linearly interpolates between the two steady-state price vectors, while `_update_prices_from_capital` recomputes factor prices from the capital path via Cobb-Douglas marginal products.

```
def init_price_seq_interp(S, S1, S3, ss0_RR, ss0_w, ss1_RR, ss1_w):
    """Linearly interpolate price sequence between steady states."""

    s_indices = jnp.arange(S + 2)
    t_frac = jnp.clip((s_indices - S1) / (S3 - S1), 0.0, 1.0)

    RR_seq = ss0_RR + t_frac * (ss1_RR - ss0_RR)
    w_seq = ss0_w + t_frac * (ss1_w - ss0_w)

    RR_seq = jnp.where(s_indices <= S1, ss0_RR, RR_seq)
    w_seq = jnp.where(s_indices <= S1, ss0_w, w_seq)
    RR_seq = jnp.where(s_indices >= S3, ss1_RR, RR_seq)
    w_seq = jnp.where(s_indices >= S3, ss1_w, w_seq)
```

(continues on next page)

(continued from previous page)

```
    return jnp.column_stack([RR_seq, w_seq])


@jit
def _update_prices_from_capital(
        k_seq, k_bar_ss0, n, frac0_x0,
        ε_bar, A, α, δ,
        s_indices_full, ss1_RR, ss1_w, S3):
    """Compute new price sequence from the capital path."""

    k_prev = jnp.concatenate([jnp.array([k_bar_ss0]), k_seq[:-1]])
    K_eff = k_prev / n + frac0_x0

    k_per_eff = K_eff / ε_bar
    r_new = A * α * (k_per_eff ** (α - 1))
    w_new = A * (1 - α) * (k_per_eff ** α)
    RR_new = 1 + r_new - δ

    price_seq_new = jnp.column_stack([RR_new, w_new])
    price_seq_new = jnp.concatenate([price_seq_new, price_seq_new[-1:]], axis=0)

    terminal_prices = jnp.array([[ss1_RR, ss1_w]])
    mask = s_indices_full >= S3
    price_seq_new = jnp.where(mask[:, None], terminal_prices, price_seq_new)

    return price_seq_new
```

The top-level function `find_transition_endo` wraps everything in a relaxation loop: at each iteration it solves the transition under current prices, computes the implied capital path, updates prices, and checks for convergence.

```
def find_transition_endo(price_seq, policy_seq_base,
                         ss0, ss1, hh, tech, S, S1, S2, S3,
                         compensation_data=None,
                         max_iter=50, tol=1e-3,
                         relaxation=0.5, verbose=False):
    """Find transition with endogenous prices."""

    ε_bar = float(jnp.sum(hh.frac * hh.ε_arr))
    frac0_x0 = float(hh.frac[0] * hh.x0[0])

    price_seq = jnp.asarray(price_seq)
    policy_seq_base = jnp.asarray(policy_seq_base)

    # Pre-create iteration arrays for price update
    s_indices_full = jnp.arange(S + 2)

    if verbose:
        print("  Starting price iteration...")

    for iteration in range(max_iter):
        try:
            τ_l_trans, results = find_transition_exo(
                price_seq, policy_seq_base, ss0, ss1,
                hh, tech, S, S1, S2,
                compensation_data=compensation_data
            )
        except ValueError:
```

(continues on next page)

(continued from previous page)

```
            τ_l_trans = 0.35
            results = transition_paths(
                τ_l_trans, price_seq, policy_seq_base,
                ss0, ss1, hh, tech, S, S1, S2,
                ss0.μx_arr, ss0.Σx_arr
            )

        _, μx_seq, μc_seq, k_seq, Gb_seq, F_seq, Ao_seq = results

        price_seq_new = _update_prices_from_capital(
            k_seq, float(ss0.k_bar), hh.n, float(frac0_x0), float(ε_bar),
            tech.A, tech.α_tilde, tech.δ,
            s_indices_full,
            float(ss1.RR), float(ss1.w), S3
        )

        price_diff = float(jnp.max(jnp.abs(price_seq_new - price_seq)))

        if verbose:
            print(f"  Iter {iteration}: "
                  f"τ_l={τ_l_trans:.4f}, "
                  f"price_diff={price_diff:.6f}")

        if price_diff < tol:
            if verbose:
                print(f"  Converged at iteration {iteration}")
            break

        price_seq = price_seq + relaxation * (price_seq_new - price_seq)

    return τ_l_trans, price_seq, results
```

We now compute the initial and terminal steady states under endogenous prices and solve the transition with price iteration.

```
# Compute endogenous prices for initial SS
ε_bar = float(jnp.sum(hh.frac * hh.ε_arr))
K_eff_0 = ss0.k_bar / hh.n + float(hh.frac[0] * hh.x0[0])
r0_endo, w0_endo, RR0_endo = compute_factor_prices(
    K_eff_0, ε_bar,
    tech.A, tech.α_tilde, tech.δ
)
r0_endo = float(r0_endo)
w0_endo = float(w0_endo)
RR0_endo = float(RR0_endo)

ss0_exp1_endo = SteadyState(
    P_arr=ss0.P_arr, ξ_arr=ss0.ξ_arr, Ao_arr=ss0.Ao_arr, F_arr=ss0.F_arr,
    μx_arr=ss0.μx_arr, μc_arr=ss0.μc_arr, Σx_arr=ss0.Σx_arr, Vc_arr=ss0.Vc_arr,
    debt2gdp=ss0.debt2gdp, τ_l=ss0.τ_l, benef=ss0.benef, Gb=ss0.Gb,
    k_bar=ss0.k_bar, RR=RR0_endo, w=w0_endo, r=r0_endo, k2gdp=ss0.k2gdp
)

ss1_exp1_endo = find_ss_endo(
    ss0.debt2gdp, (τ_a_0, τ_0_0, 0, G_0), hh, tech,
    RR_init=tech.RR, w_init=tech.w, verbose=True
)
```

```
SS iter 0: RR=1.067500, w=5.0147, k_bar=35.8906
SS iter 5: RR=1.044957, w=5.7366, k_bar=30.4338
SS iter 10: RR=1.044407, w=5.7330, k_bar=30.1611
SS iter 15: RR=1.044405, w=5.7291, k_bar=30.1587
SS iter 20: RR=1.044406, w=5.7284, k_bar=30.1588
SS iter 25: RR=1.044406, w=5.7283, k_bar=30.1588
Converged at iteration 28
```

The initial price guess interpolates linearly between the two steady states.

```python
# Initialize price sequence
price_seq_exp1_endo = init_price_seq_interp(
    S_endo, S1, S3,
    float(ss0_exp1_endo.RR), float(ss0_exp1_endo.w),
    float(ss1_exp1_endo.RR), float(ss1_exp1_endo.w)
)

# Policy sequence
policy_seq_exp1_endo = make_policy_seq(
    S_endo,
    ss0_exp1_endo.τ_l, ss1_exp1_endo.τ_l,
    S1, S2,
    τ_a_0, τ_0_0, 0, G_0,
    ss0_exp1_endo.Gb
)
```

The benefit difference vector records the benefit loss at each age, and the price iteration finds the equilibrium transition path with buy-out compensation.

```python
# Buyout compensation
benef_diff_exp1_endo = jnp.zeros(hh.T0 + 1)
benef_diff = ss0_exp1_endo.benef - ss1_exp1_endo.benef
benef_diff_exp1_endo = benef_diff_exp1_endo.at[
    hh.T1 + 1:
].set(benef_diff)

# Solve with price iteration
print("\n  Solving transition with endogenous prices...")
endo_result = find_transition_endo(
    price_seq_exp1_endo, policy_seq_exp1_endo,
    ss0_exp1_endo, ss1_exp1_endo,
    hh, tech, S_endo, S1, S2, S3,
    compensation_data=(
        benef_diff_exp1_endo,
        ss0_exp1_endo.τ_l
    ),
    verbose=True
)
τ_l_exp1_endo_bo = endo_result[0]
price_seq_exp1_endo_conv = endo_result[1]
results_exp1_endo = endo_result[2]
(_, μx_seq_exp1_endo, μc_seq_exp1_endo,
 k_seq_exp1_endo, Gb_seq_exp1_endo,
 _, _) = results_exp1_endo
```

```
Solving transition with endogenous prices...
Starting price iteration...
```

(continues on next page)

(continued from previous page)

```
Iter 0: τ_l=0.3391, price_diff=0.490311
Iter 1: τ_l=0.3549, price_diff=0.184939
Iter 2: τ_l=0.3637, price_diff=0.064441
Iter 3: τ_l=0.3676, price_diff=0.021013
Iter 4: τ_l=0.3692, price_diff=0.006973
Iter 5: τ_l=0.3698, price_diff=0.002371
Iter 6: τ_l=0.3701, price_diff=0.000828
Converged at iteration 6
```

The endogenous-price results are stored for comparison with the fixed-price case.

```
exp1_endo = {
    'ss0': ss0_exp1_endo, 'ss1': ss1_exp1_endo,
    'τ_l_buyout': τ_l_exp1_endo_bo,
    'price_seq': price_seq_exp1_endo_conv,
    'k_seq': k_seq_exp1_endo, 'Gb_seq': Gb_seq_exp1_endo,
    'results': results_exp1_endo,
    'μc_seq': μc_seq_exp1_endo, 'μx_seq': μx_seq_exp1_endo
}
```

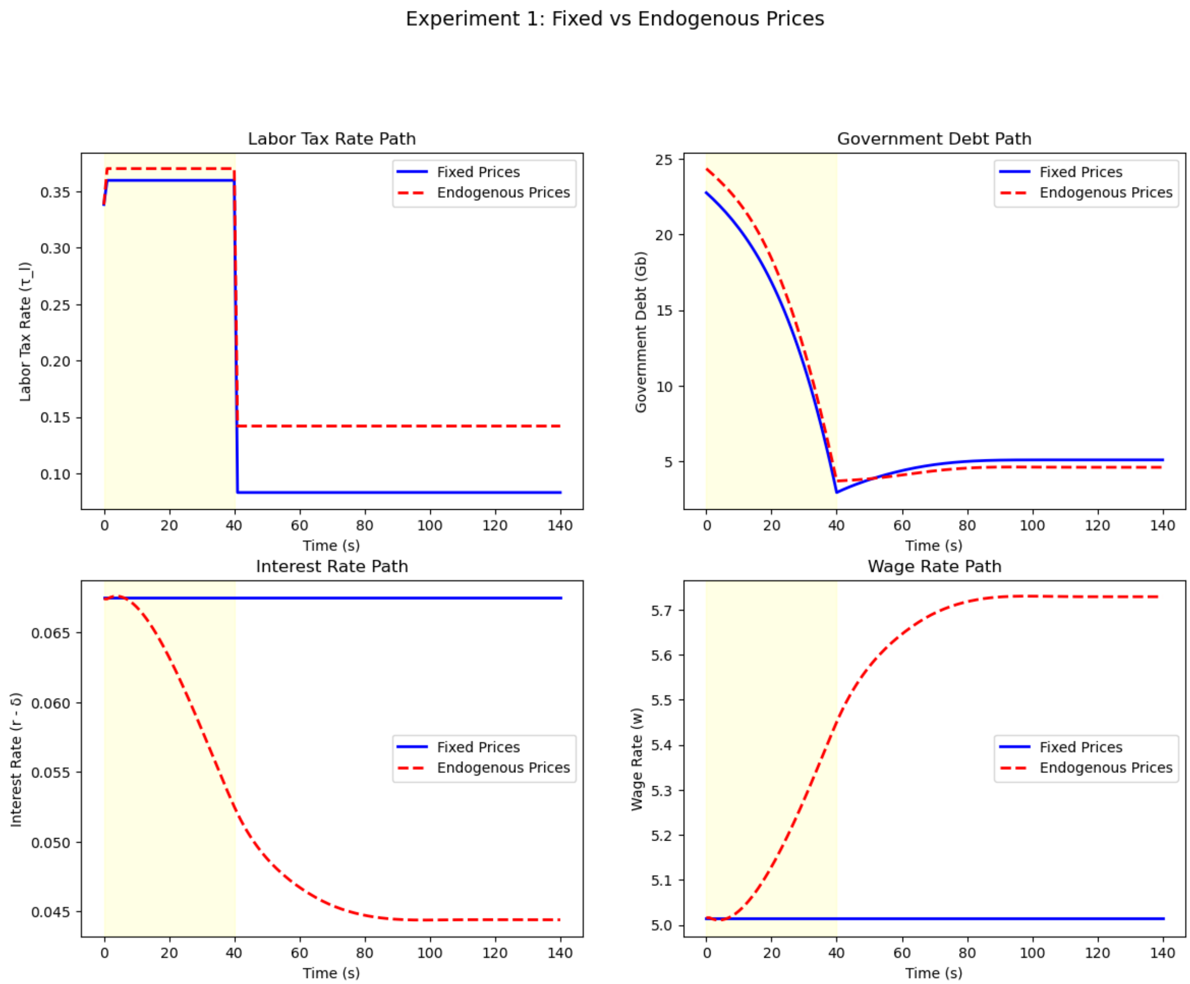
The following figure compares the transition paths under fixed and endogenous factor prices, showing how general equilibrium effects alter the tax, debt, interest rate, and wage paths.

```
# Get endogenous price sequences
price_seq_endo = exp1_endo['price_seq']
S_endo = price_seq_endo.shape[0] - 2

# Construct fixed price sequences for comparison
RR_fixed = tech.RR
w_fixed = tech.w

# For fixed prices, construct τ_l sequence
τ_l_seq_fixed = np.zeros(S_exo + 1)
τ_l_seq_fixed[:S1 + 1] = ss0_exp1.τ_l
τ_l_seq_fixed[S1 + 1:S2 + 1] = exp1_exo['τ_l_buyout']
τ_l_seq_fixed[S2 + 1:] = ss1_exp1.τ_l

# For endogenous prices
τ_l_seq_endo = np.zeros(S_endo + 1)
τ_l_seq_endo[:S1 + 1] = exp1_endo['ss0'].τ_l
τ_l_seq_endo[S1 + 1:S2 + 1] = exp1_endo['τ_l_buyout']
τ_l_seq_endo[S2 + 1:] = exp1_endo['ss1'].τ_l

fig, axes = plt.subplots(2, 2, figsize=(14, 10))

# Labor tax comparison
axes[0, 0].plot(τ_l_seq_fixed, 'b-', linewidth=2, label='Fixed Prices')
axes[0, 0].plot(τ_l_seq_endo[:len(τ_l_seq_fixed)],
                'r--', linewidth=2,
                label='Endogenous Prices')
axes[0, 0].axvspan(S1, S2, alpha=0.1, color='yellow')
axes[0, 0].set_xlabel('Time (s)')
axes[0, 0].set_ylabel('Labor Tax Rate (τ_l)')
axes[0, 0].set_title('Labor Tax Rate Path')
axes[0, 0].legend()

# Government debt comparison
```

(continues on next page)

(continued from previous page)

```
Gb_seq_fixed = Gb_seq_bo
Gb_seq_endo_exp1 = exp1_endo['Gb_seq']
axes[0, 1].plot(Gb_seq_fixed, 'b-', linewidth=2, label='Fixed Prices')
axes[0, 1].plot(
    Gb_seq_endo_exp1[:len(Gb_seq_fixed)],
    'r--', linewidth=2,
    label='Endogenous Prices'
)
axes[0, 1].axvspan(S1, S2, alpha=0.1, color='yellow')
axes[0, 1].set_xlabel('Time (s)')
axes[0, 1].set_ylabel('Government Debt (Gb)')
axes[0, 1].set_title('Government Debt Path')
axes[0, 1].legend()

# Interest rate comparison
r_fixed = np.full(S_exo + 1, tech.r - tech.δ)
r_endo = price_seq_endo[:-1, 0] - 1  # RR - 1 = r - δ
axes[1, 0].plot(r_fixed, 'b-', linewidth=2, label='Fixed Prices')
axes[1, 0].plot(
    r_endo[:len(r_fixed)],
    'r--', linewidth=2,
    label='Endogenous Prices'
)
axes[1, 0].axvspan(S1, S2, alpha=0.1, color='yellow')
axes[1, 0].set_xlabel('Time (s)')
axes[1, 0].set_ylabel('Interest Rate (r - δ)')
axes[1, 0].set_title('Interest Rate Path')
axes[1, 0].legend()

# Wage rate comparison
w_fixed_seq = np.full(S_exo + 1, tech.w)
w_endo = price_seq_endo[:-1, 1]
axes[1, 1].plot(w_fixed_seq, 'b-', linewidth=2, label='Fixed Prices')
axes[1, 1].plot(
    w_endo[:len(w_fixed_seq)],
    'r--', linewidth=2,
    label='Endogenous Prices'
)
axes[1, 1].axvspan(S1, S2, alpha=0.1, color='yellow')
axes[1, 1].set_xlabel('Time (s)')
axes[1, 1].set_ylabel('Wage Rate (w)')
axes[1, 1].set_title('Wage Rate Path')
axes[1, 1].legend()

plt.suptitle(
    'Experiment 1: Fixed vs Endogenous Prices',
    fontsize=14, y=1.02
)
plt.show()
```



The top-left panel shows that the transition tax rate is similar under both price assumptions, but the bottom panels reveal important differences in factor prices.

As the capital stock rises during the transition, the interest rate falls and the wage rises under endogenous pricing, while these remain constant under the small open economy assumption.

These price effects create additional redistributive consequences beyond those intended by the policy change: lower interest rates benefit young workers through higher wages, but hurt retirees through lower returns on savings.

83.9 Experiment 2: government capital accumulation

In Experiment 2, the government maintains social security benefits but temporarily raises taxes to accumulate physical capital.

The returns from this capital eventually finance the social security payments, so that in the terminal steady state the government is a net creditor rather than a debtor.

Unlike Experiment 1, which eliminates benefits, this approach preserves the social insurance function of social security – namely, insurance against life span risk and partial insurance against labor income volatility.

By having the government save on behalf of households, the economy can achieve higher capital accumulation while maintaining the intergenerational insurance that social security provides.

Algorithm 83.9.1 (Fixed factor prices – Experiment 2 (government funding))

Here $s_3 = s_2 + T_0 = 105$.

Steps 1–3 are the same as *Algorithm 83.8.1*, except that social security benefits are maintained and the target debt-to-GDP ratio is negative (the government becomes a net creditor).

The right target induces the government to accumulate sufficient capital to finance benefits from asset returns.

Step 4 is the same root-finding procedure over $\tau_{\ell,1}$, but without compensation payments.

We first compute the terminal steady state under fixed prices, targeting a negative debt-to-GDP ratio that makes the government a net creditor.

```
debt2gdp_target_exp2_exo = -1.1785
ss1_exp2_exo = ss_target_debt2gdp_exo(
    debt2gdp_target_exp2_exo,
    (τ_a_0, τ_0_0, benef_0, G_0),
    (RR_exo, w_exo), hh, tech
)
```

With the terminal steady state in hand, bisection over the transition tax rate gives the equilibrium path under fixed prices.

```
# Price and policy sequences
price_seq_exp2_exo = make_exo_price_seq(S_exo, RR_exo, w_exo)
policy_seq_exp2_exo = make_policy_seq(
    S_exo, ss0.τ_l, ss1_exp2_exo.τ_l,
    S1, S2,
    τ_a_0, τ_0_0, benef_0, G_0, Gb_0
)

# Solve (no compensation)
τ_l_exp2_exo, results_exp2_exo = find_transition_exo(
    price_seq_exp2_exo, policy_seq_exp2_exo, ss0, ss1_exp2_exo,
    hh, tech, S_exo, S1, S2
)
```

The results are packaged for the cross-experiment comparison below.

```
(carryover_exp2, μx_seq_exp2_exo,
 μc_seq_exp2_exo, k_seq_exp2_exo,
 Gb_seq_exp2_exo, F_seq_exp2_exo,
 Ao_seq_exp2_exo) = results_exp2_exo
exp2_exo = {
    'ss0': ss0, 'ss1': ss1_exp2_exo,
    'τ_l_trans': τ_l_exp2_exo,
    'results': results_exp2_exo,
    'k_seq': k_seq_exp2_exo, 'Gb_seq': Gb_seq_exp2_exo,
    'μc_seq': μc_seq_exp2_exo, 'μx_seq': μx_seq_exp2_exo
}
```

We repeat the computation under endogenous factor prices, using the same initial steady state as Experiment 1.

```
# Compute endogenous prices for initial SS (reuse from Exp 1)
ss0_exp2_endo = ss0_exp1_endo  # Same initial SS

# Terminal steady state with endogenous prices
```

(continues on next page)

(continued from previous page)

```
ss1_exp2_endo = find_ss_endo(
    debt2gdp_target=-1.925,
    policy_target=(τ_a_0, τ_0_0, benef_0, G_0),
    hh=hh, tech=tech,
    RR_init=tech.RR, w_init=tech.w,
    verbose=True
)
```

```
    SS iter 0: RR=1.067500, w=5.0147, k_bar=43.5570
    SS iter 5: RR=1.044472, w=5.7664, k_bar=30.0436
    SS iter 10: RR=1.044623, w=5.7272, k_bar=30.0407
    SS iter 15: RR=1.044648, w=5.7206, k_bar=30.0403
    SS iter 20: RR=1.044651, w=5.7195, k_bar=30.0409
    SS iter 25: RR=1.044651, w=5.7193, k_bar=30.0409
    Converged at iteration 29
```

Price iteration produces the endogenous-price transition path.

```
# Initialize price sequence
price_seq_exp2_endo = init_price_seq_interp(
    S_endo, S1, S3,
    float(ss0_exp2_endo.RR), float(ss0_exp2_endo.w),
    float(ss1_exp2_endo.RR), float(ss1_exp2_endo.w)
)

# Policy sequence
policy_seq_exp2_endo = make_policy_seq(
    S_endo,
    ss0_exp2_endo.τ_l, ss1_exp2_endo.τ_l,
    S1, S2,
    τ_a_0, τ_0_0, benef_0, G_0,
    ss0_exp2_endo.Gb
)

# Solve with price iteration (no compensation)
endo2 = find_transition_endo(
    price_seq_exp2_endo, policy_seq_exp2_endo,
    ss0_exp2_endo, ss1_exp2_endo,
    hh, tech, S_endo, S1, S2, S3,
    verbose=True
)
τ_l_exp2_endo = endo2[0]
price_seq_exp2_endo_conv = endo2[1]
results_exp2_endo = endo2[2]
(_, μx_seq_exp2_endo, μc_seq_exp2_endo,
 k_seq_exp2_endo, Gb_seq_exp2_endo,
 _, _) = results_exp2_endo
```

```
  Starting price iteration...
  Iter 0: τ_l=0.3660, price_diff=0.394320
  Iter 1: τ_l=0.3769, price_diff=0.202210
  Iter 2: τ_l=0.3845, price_diff=0.103709
  Iter 3: τ_l=0.3891, price_diff=0.050465
  Iter 4: τ_l=0.3916, price_diff=0.022852
  Iter 5: τ_l=0.3928, price_diff=0.009652
  Iter 6: τ_l=0.3933, price_diff=0.003839
```

(continues on next page)

(continued from previous page)

```
Iter 7: τ_l=0.3935, price_diff=0.001457
Iter 8: τ_l=0.3936, price_diff=0.000537
Converged at iteration 8
```

```
exp2_endo = {
    'ss0': ss0_exp2_endo, 'ss1': ss1_exp2_endo,
    'τ_l_trans': τ_l_exp2_endo,
    'price_seq': price_seq_exp2_endo_conv,
    'k_seq': k_seq_exp2_endo, 'Gb_seq': Gb_seq_exp2_endo,
    'results': results_exp2_endo,
    'μc_seq': μc_seq_exp2_endo, 'μx_seq': μx_seq_exp2_endo
}
```

We now compare all four reform scenarios: the buy-out scheme and the government funding scheme, each under fixed and endogenous factor prices.

```
# Get debt sequences for all cases
Gb_buyout_fixed = Gb_seq_bo
Gb_buyout_endo = exp1_endo['Gb_seq']
Gb_accum_fixed = exp2_exo['Gb_seq']
Gb_accum_endo = exp2_endo['Gb_seq']

# Get capital sequences
k_buyout_fixed = k_seq_bo
k_buyout_endo = exp1_endo['k_seq']
k_accum_fixed = exp2_exo['k_seq']
k_accum_endo = exp2_endo['k_seq']

# Common time horizon for plotting
T_plot = min(len(Gb_buyout_fixed), len(Gb_buyout_endo),
             len(Gb_accum_fixed), len(Gb_accum_endo))

fig, axes = plt.subplots(2, 3, figsize=(18, 10))

# Labels for four scenarios
lb = ['Buyout (Fixed)', 'Buyout (Endo)',
      'Gov Funding (Fixed)', 'Gov Funding (Endo)']
ls = ['b-', 'b--', 'r-', 'r--']

# Government Debt paths
ax = axes[0, 0]
for d, s, l in zip(
    [Gb_buyout_fixed, Gb_buyout_endo,
     Gb_accum_fixed, Gb_accum_endo],
    ls, lb
):
    ax.plot(d[:T_plot], s, linewidth=2, label=l)
ax.axhline(0, color='k', linestyle=':', alpha=0.5)
ax.axvspan(0, 40, alpha=0.1, color='yellow')
ax.set_xlabel('Time (s)')
ax.set_ylabel('Government Debt (Gb)')
ax.set_title('Government Debt Paths')
ax.legend(fontsize=9)


# Capital paths
```

(continues on next page)

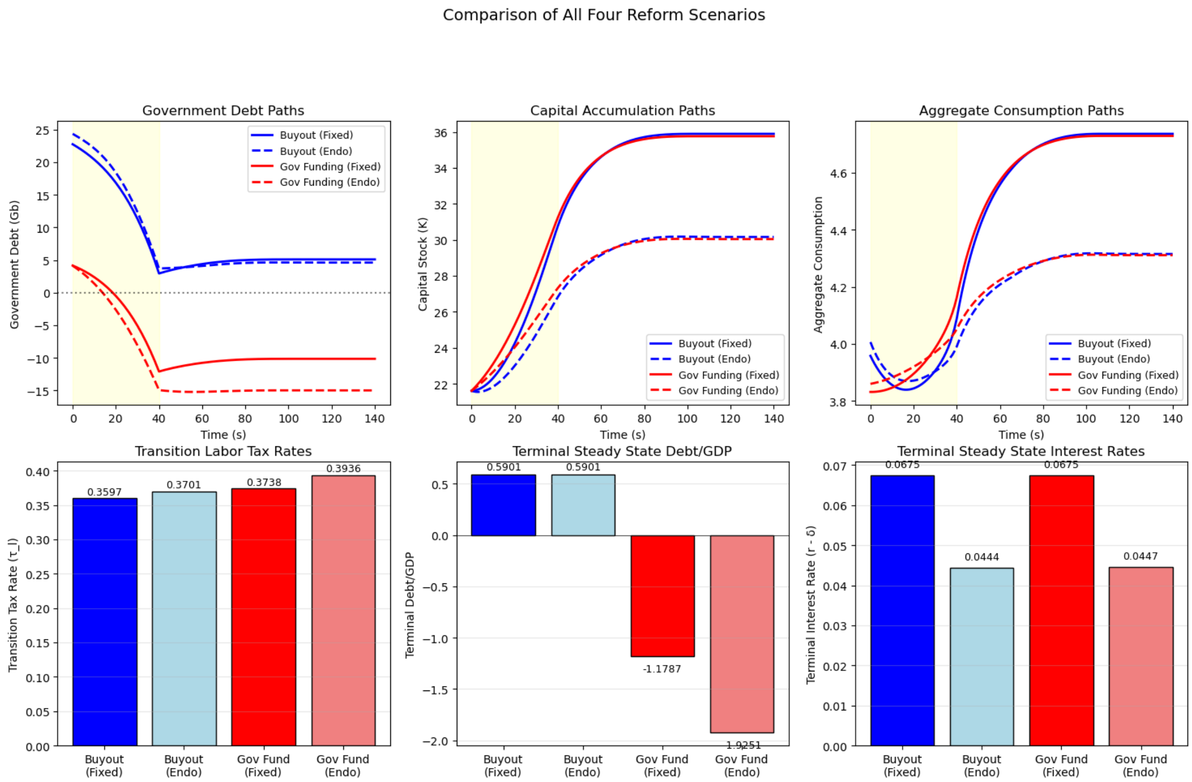
(continued from previous page)

```
ax = axes[0, 1]
for d, s, l in zip(
    [k_buyout_fixed, k_buyout_endo,
     k_accum_fixed, k_accum_endo],
    ls, lb
):
    ax.plot(d[:T_plot], s, linewidth=2, label=l)
ax.axvspan(0, 40, alpha=0.1, color='yellow')
ax.set_xlabel('Time (s)')
ax.set_ylabel('Capital Stock (K)')
ax.set_title('Capital Accumulation Paths')
ax.legend(fontsize=9)


# Aggregate consumption
res2_exo = exp2_exo['results']
res1_endo = exp1_endo['results']
res2_endo = exp2_endo['results']
_, _, μc_exp2_exo, _, _, _, _ = res2_exo
_, _, μc_exp1_endo, _, _, _, _ = res1_endo
_, _, μc_exp2_endo, _, _, _, _ = res2_endo

def _agg_c(μc, T):
    """Aggregate per-capita consumption across cohorts."""

    n = min(T, μc.shape[0])
    return np.array([
        np.sum(μc[s] * hh.frac) for s in range(n)
    ])

c_agg_buyout_fixed = c_agg_bo[:T_plot]
c_agg_buyout_endo = _agg_c(μc_exp1_endo, T_plot)
c_agg_accum_fixed = _agg_c(μc_exp2_exo, T_plot)
c_agg_accum_endo = _agg_c(μc_exp2_endo, T_plot)

ax = axes[0, 2]
for d, s, l in zip(
    [c_agg_buyout_fixed, c_agg_buyout_endo,
     c_agg_accum_fixed, c_agg_accum_endo],
    ls, lb
):
    ax.plot(d[:T_plot], s, linewidth=2, label=l)
axes[0, 2].axvspan(0, 40, alpha=0.1, color='yellow')
axes[0, 2].set_xlabel('Time (s)')
axes[0, 2].set_ylabel('Aggregate Consumption')
axes[0, 2].set_title('Aggregate Consumption Paths')
axes[0, 2].legend(fontsize=9)

# Bar chart: Transition tax rates
cases = ['Buyout\n(Fixed)', 'Buyout\n(Endo)',
         'Gov Fund\n(Fixed)', 'Gov Fund\n(Endo)']
τ_l_values = [exp1_exo['τ_l_buyout'], exp1_endo['τ_l_buyout'],
              exp2_exo['τ_l_trans'], exp2_endo['τ_l_trans']]
colors = ['blue', 'lightblue', 'red', 'lightcoral']
axes[1, 0].bar(cases, τ_l_values, color=colors, edgecolor='black')
axes[1, 0].set_ylabel('Transition Tax Rate (τ_l)')
axes[1, 0].set_title('Transition Labor Tax Rates')
```

(continues on next page)

(continued from previous page)

```
axes[1, 0].grid(True, alpha=0.3, axis='y')
for i, v in enumerate(τ_l_values):
    axes[1, 0].text(i, v + 0.005, f'{v:.4f}', ha='center', fontsize=9)

# Bar chart: Terminal debt/GDP
debt2gdp_values = [exp1_exo['ss1'].debt2gdp, exp1_endo['ss1'].debt2gdp,
                   exp2_exo['ss1'].debt2gdp, exp2_endo['ss1'].debt2gdp]
axes[1, 1].bar(cases, debt2gdp_values, color=colors, edgecolor='black')
axes[1, 1].axhline(0, color='k', linestyle='-', linewidth=0.5)
axes[1, 1].set_ylabel('Terminal Debt/GDP')
axes[1, 1].set_title('Terminal Steady State Debt/GDP')
axes[1, 1].grid(True, alpha=0.3, axis='y')
for i, v in enumerate(debt2gdp_values):
    y = v + 0.05 if v > 0 else v - 0.15
    axes[1, 1].text(
        i, y, f'{v:.4f}',
        ha='center', fontsize=9
    )

# Bar chart: Terminal interest rate
r_values = [exp1_exo['ss1'].r - tech.δ, exp1_endo['ss1'].r - tech.δ,
            exp2_exo['ss1'].r - tech.δ, exp2_endo['ss1'].r - tech.δ]
axes[1, 2].bar(cases, r_values, color=colors, edgecolor='black')
axes[1, 2].set_ylabel('Terminal Interest Rate (r - δ)')
axes[1, 2].set_title('Terminal Steady State Interest Rates')
axes[1, 2].grid(True, alpha=0.3, axis='y')
for i, v in enumerate(r_values):
    axes[1, 2].text(i, v + 0.002, f'{v:.4f}', ha='center', fontsize=9)

plt.suptitle('Comparison of All Four Reform Scenarios', fontsize=14, y=1.02)
plt.show()
```



The top row compares transition dynamics: the buy-out scheme (blue) accumulates higher debt during the transition due to compensation payments, while the government funding scheme (red) leads to large negative debt as the government becomes a net creditor.

The bottom-left bar chart shows that government funding requires higher transition tax rates than the buy-out scheme because benefit payments continue alongside capital accumulation.

Under endogenous pricing, the larger capital stock reduces the marginal product of capital and hence interest rates, as shown in the bottom-right panel.

The government-funded scheme (Experiment 2) delivers larger long-run efficiency gains because it preserves insurance against life span risk and labor income volatility that would be lost under privatization.

Higher labor income tax rates during the transition also provide implicit insurance against earnings risk, amplifying the efficiency advantage under endogenous prices.

83.10 Distribution surfaces

The 3D surface plots below show how assets and consumption evolve over both the age dimension and calendar time.

```
# Compute variances for 3D plotting
def compute_variances(results, ss0, hh):
    """Compute variance sequences from transition results."""

    _, μx_seq, μc_seq, k_seq, Gb_seq, F_seq, Ao_seq = results

    # Convert to numpy
    μx_seq = np.array(μx_seq)
    μc_seq_full = np.array(μc_seq)
```

(continues on next page)

(continued from previous page)

```
    F_seq = np.array(F_seq)
    Ao_seq = np.array(Ao_seq)
    Σx_arr_ss0 = np.array(ss0.Σx_arr)
    Σ0 = np.array(hh.Σ0)
    C = np.array(hh.C)

    # Get actual dimensions from data
    S_plus_1 = Ao_seq.shape[0]  # S+1
    T0_plus_1 = Ao_seq.shape[1]  # T0+1

    Σx_seq = np.empty((S_plus_1 + 1, T0_plus_1 + 1, hh.n_x, hh.n_x))
    Vc_seq = np.empty((S_plus_1, T0_plus_1))
    Va_seq = np.empty((S_plus_1, T0_plus_1))

    Σx_seq[:, 0] = Σ0
    Σx_seq[0, :] = Σx_arr_ss0[:T0_plus_1 + 1]

    CCT = C @ C.T
    for s in range(S_plus_1):
        Ao_s = Ao_seq[s]                    # (T0+1, n_x, n_x)
        Σx_s = Σx_seq[s, :T0_plus_1]        # (T0+1, n_x, n_x)
        F_s = F_seq[s]                       # (T0+1, n_x)
        Σx_seq[s + 1, 1:] = CCT + Ao_s @ Σx_s @ Ao_s.transpose(0, 2, 1)
        Vc_seq[s] = np.einsum('ti,tij,tj->t', F_s, Σx_s, F_s)
        Va_seq[s] = Σx_s[:, 0, 0]

    # Extract mean assets - match dimensions with Ao_seq
    μa_seq = μx_seq[:S_plus_1, :T0_plus_1, 0]
    μc_seq_out = μc_seq_full[:S_plus_1, :T0_plus_1]

    return μa_seq, Va_seq, μc_seq_out, Vc_seq

# Compute variances for each case
μa_bf, Va_bf, μc_bf, Vc_bf = compute_variances(
    exp1_exo['results_buyout'], exp1_exo['ss0'], hh
)
μa_be, Va_be, μc_be, Vc_be = compute_variances(
    exp1_endo['results'], exp1_endo['ss0'], hh
)
μa_af, Va_af, μc_af, Vc_af = compute_variances(
    exp2_exo['results'], exp2_exo['ss0'], hh
)
μa_ae, Va_ae, μc_ae, Vc_ae = compute_variances(
    exp2_endo['results'], exp2_endo['ss0'], hh
)

case_names = [
    'Buyout (Fixed)', 'Buyout (Endo)',
    'Gov Funding (Fixed)', 'Gov Funding (Endo)'
]


def plot_surface_grid(
        data_cases, case_names, zlabel,
        suptitle, cmap='viridis',
        transform=None):
    """Plot 2x2 grid of 3D surfaces for age-time data."""
```

(continues on next page)

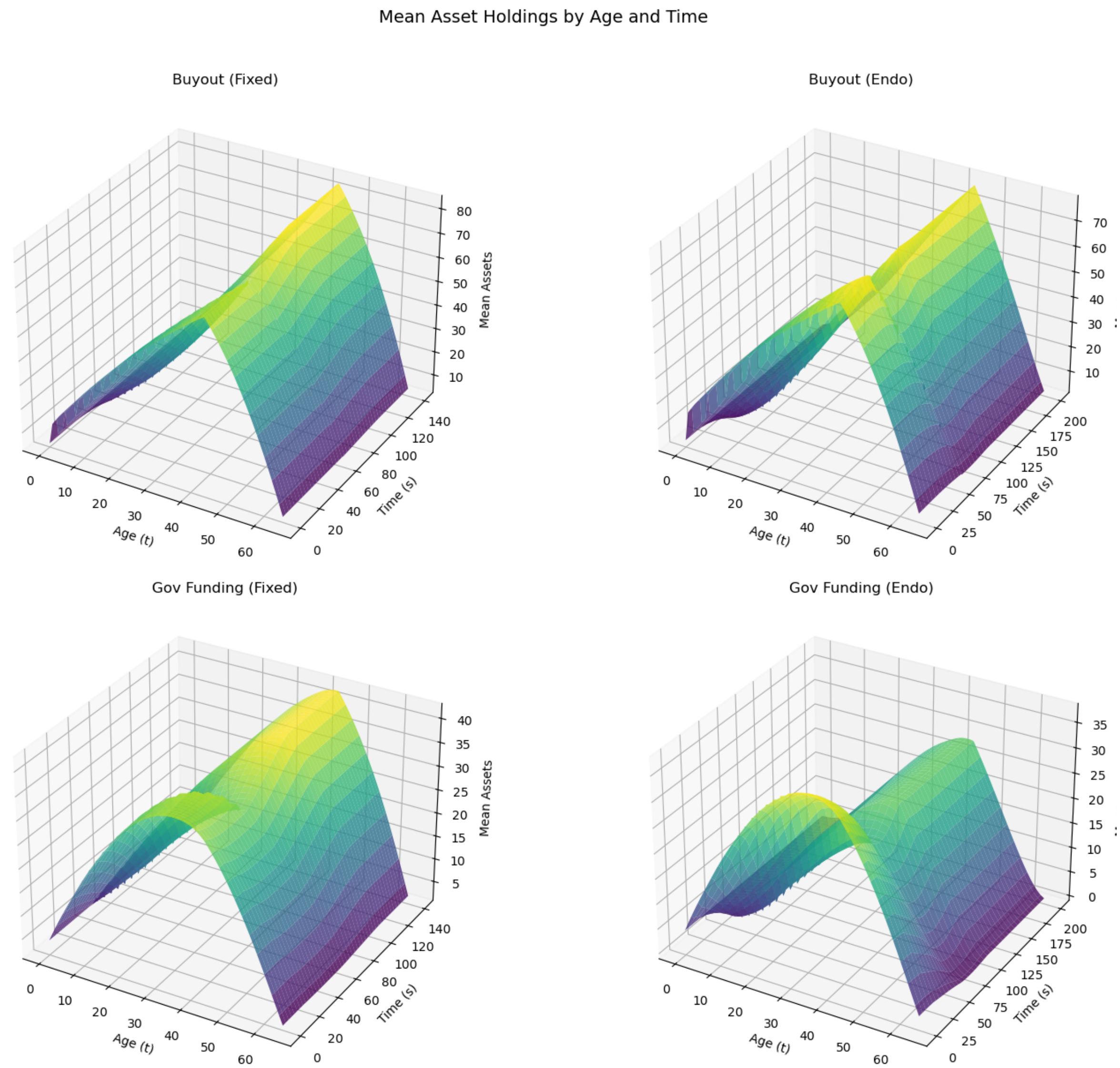
(continued from previous page)

```
    fig = plt.figure(figsize=(16, 12))
    for i, (data, name) in enumerate(zip(data_cases, case_names)):
        Z = transform(data) if transform is not None else data
        n_time, n_age = Z.shape
        X, Y = np.meshgrid(np.arange(n_age), np.arange(n_time))
        ax = fig.add_subplot(2, 2, i + 1, projection='3d')
        ax.plot_surface(X, Y, Z, cmap=cmap, edgecolor='none', alpha=0.8)
        ax.set_xlabel('Age (t)')
        ax.set_ylabel('Time (s)')
        ax.set_zlabel(zlabel)
        ax.set_title(name)
    plt.suptitle(suptitle, fontsize=14, y=1.02)
    plt.tight_layout()
    plt.show()
```

Each surface shows the joint distribution across age (t) and calendar time (s), revealing life-cycle patterns, transition dynamics, and cross-cohort heterogeneity.

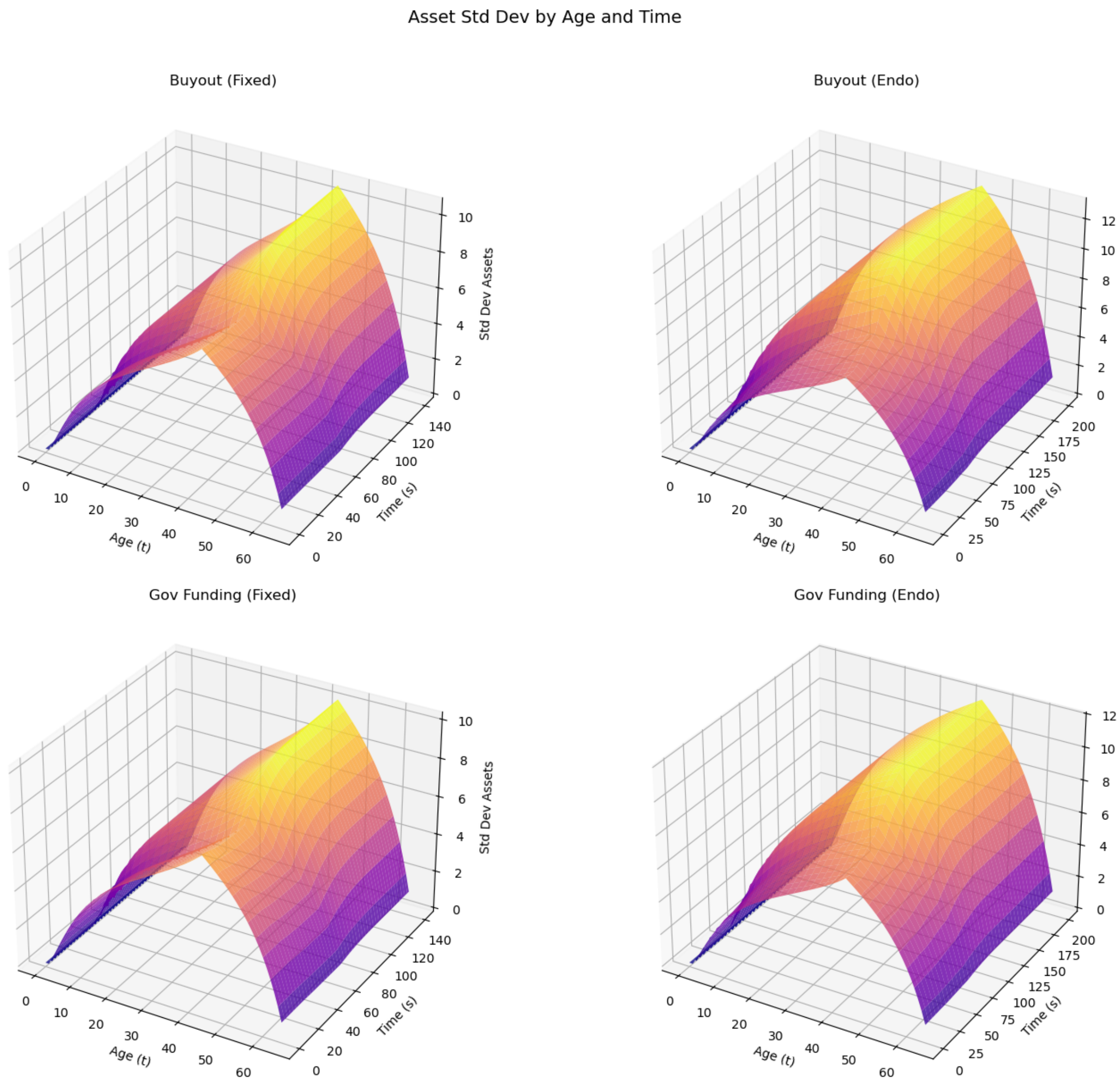
The mean asset surfaces display the hump-shaped life-cycle profile of asset holdings, with peak assets shifting as working generations increase their saving in response to the reform.

```
plot_surface_grid(
    [μa_bf, μa_be, μa_af, μa_ae],
    case_names, 'Mean Assets',
    'Mean Asset Holdings by Age and Time'
)
```



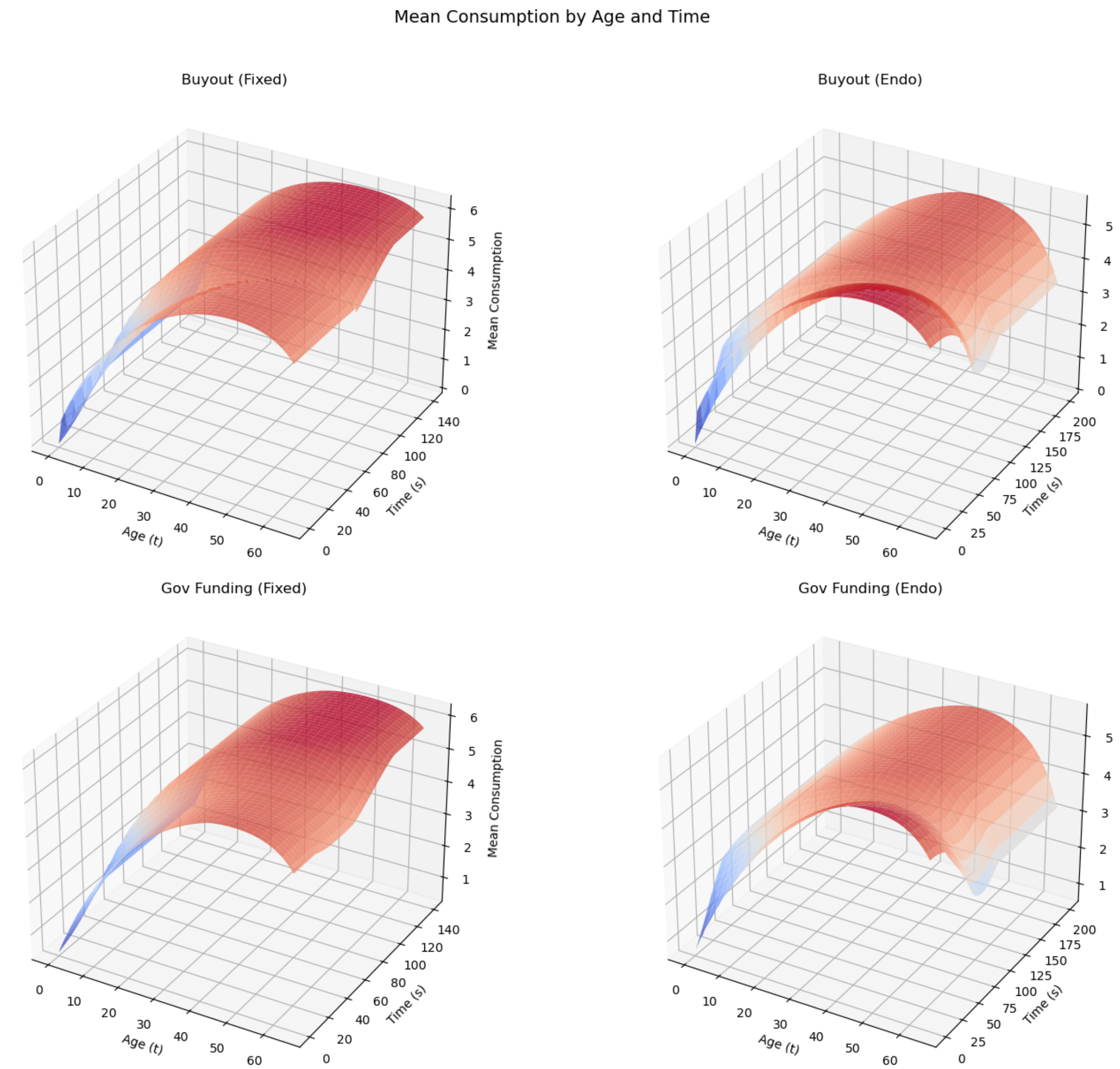
The asset variance surfaces show how cumulative income shocks cause dispersion to increase with age, with the transition potentially altering the rate of dispersion growth.

```
plot_surface_grid(
    [Va_bf, Va_be, Va_af, Va_ae],
    case_names, 'Std Dev Assets',
    'Asset Std Dev by Age and Time',
    cmap='plasma', transform=np.sqrt
)
```



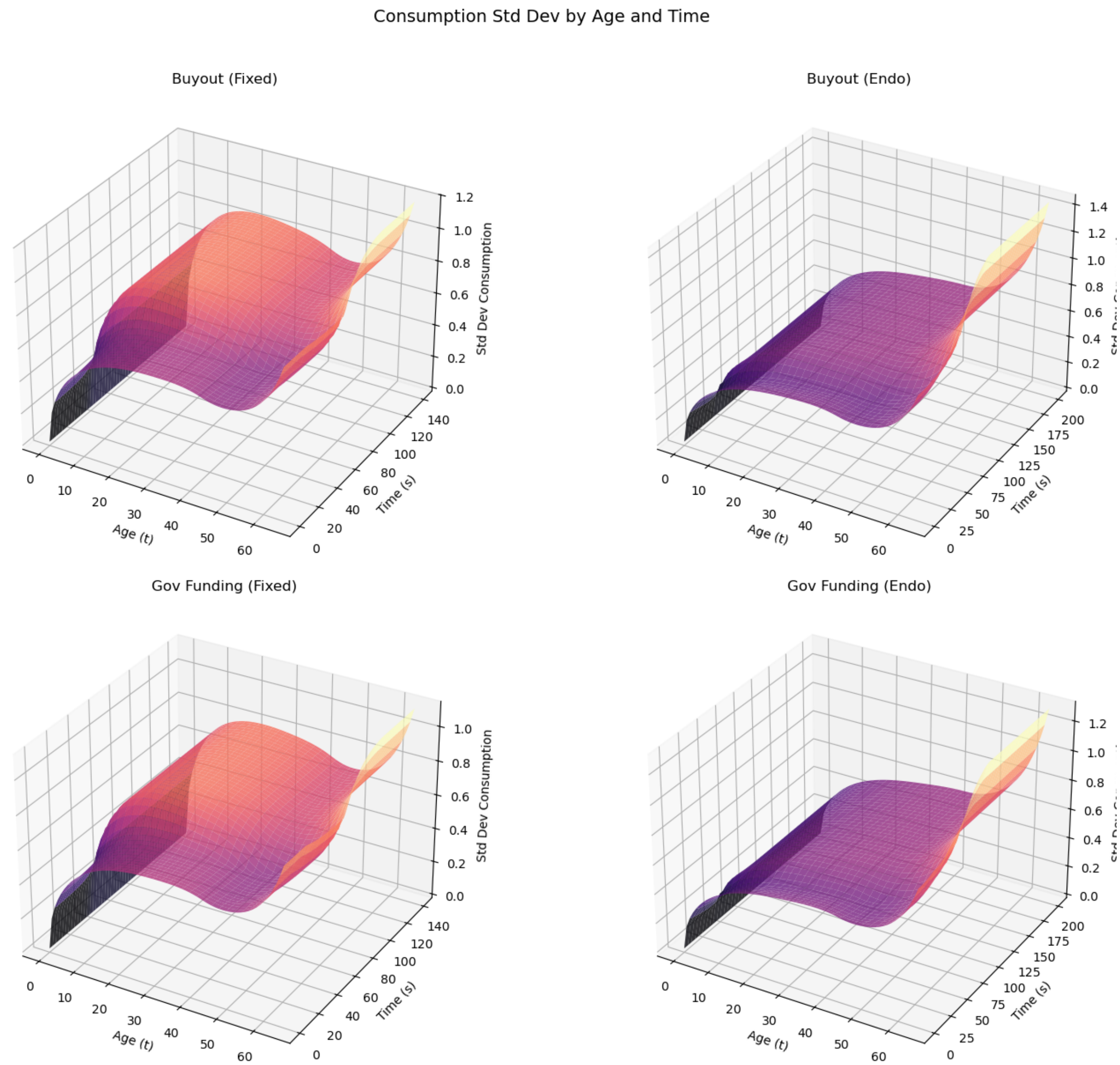
The mean consumption surfaces reflect the optimal consumption path, which should be smooth across ages due to the permanent income hypothesis underlying the model.

```
plot_surface_grid(
    [μc_bf, μc_be, μc_af, μc_ae],
    case_names, 'Mean Consumption',
    'Mean Consumption by Age and Time',
    cmap='coolwarm'
)
```



The consumption variance surfaces reveal how the certainty-equivalence property of the LQ framework shapes the within-cohort distribution of consumption over time.

```
plot_surface_grid(
    [Vc_bf, Vc_be, Vc_af, Vc_ae],
    case_names, 'Std Dev Consumption',
    'Consumption Std Dev by Age and Time',
    cmap='magma', transform=np.sqrt
)
```


Consumption Std Dev by Age and Time
Buyout (Fixed)
Buyout (Endo)
Gov Funding (Fixed)
Gov Funding (Endo)
Age (t)
Time (s)
Std Dev Consumption

Part XIII

Asset Pricing and Finance

CHAPTER

EIGHTYFOUR

ASSET PRICING: FINITE STATE MODELS

Contents

"A little knowledge of geometric series goes a long way" – Robert E. Lucas, Jr.

"Asset pricing is all about covariances" – Lars Peter Hansen

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install quantecon
```

84.1 Overview

An asset is a claim on one or more future payoffs.

The spot price of an asset depends primarily on

- the anticipated income stream
- attitudes about risk
- rates of time preference

In this lecture, we consider some standard pricing models and dividend stream specifications.

We study how prices and dividend-price ratios respond in these different scenarios.

We also look at creating and pricing *derivative* assets that repackage income streams.

Key tools for the lecture are

- Markov processses
- formulas for predicting future values of functions of a Markov state

- a formula for predicting the discounted sum of future values of a Markov state

Let's start with some imports:

```
import matplotlib.pyplot as plt
import numpy as np
import quantecon as qe
from numpy.linalg import eigvals, solve
```

84.2 Pricing Models

Let $\{d_t\}_{t \geq 0}$ be a stream of dividends

- A time-t **cum-dividend** asset is a claim to the stream $d_t, d_{t+1}, \ldots$.
- A time-t **ex-dividend** asset is a claim to the stream $d_{t+1}, d_{t+2}, \ldots$.

Let's look at some equations that we expect to hold for prices of assets under ex-dividend contracts (we will consider cum-dividend pricing in the exercises).

84.2.1 Risk-Neutral Pricing

Our first scenario is risk-neutral pricing.

Let $\beta = 1/(1+\rho)$ be an intertemporal discount **factor**, where ρ is the **rate** at which agents discount the future.

The basic risk-neutral asset pricing equation for pricing one unit of an ex-dividend asset is

$$p_t = \beta \mathbb{E}_t [d_{t+1} + p_{t+1}] \tag{84.1}$$

This is a simple "cost equals expected benefit" relationship.

Here $\mathbb{E}_t[y]$ denotes the best forecast of y, conditioned on information available at time t.

More precisely, $\mathbb{E}_t[y]$ is the mathematical expectation of y conditional on information available at time t.

84.2.2 Pricing with Random Discount Factor

What happens if for some reason traders discount payouts differently depending on the state of the world?

Michael Harrison and David Kreps [Harrison and Kreps, 1979] and Lars Peter Hansen and Scott Richard [Hansen and Richard, 1987] showed that in quite general settings the price of an ex-dividend asset obeys

$$p_t = \mathbb{E}_t \left[m_{t+1} (d_{t+1} + p_{t+1}) \right] \tag{84.2}$$

for some **stochastic discount factor** m_{t+1}.

Here the fixed discount factor β in (84.1) has been replaced by the random variable m_{t+1}.

How anticipated future payoffs are evaluated now depends on statistical properties of m_{t+1}.

The stochastic discount factor can be specified to capture the idea that assets that tend to have good payoffs in bad states of the world are valued more highly than other assets whose payoffs don't behave that way.

This is because such assets pay well when funds are more urgently wanted.

We give examples of how the stochastic discount factor has been modeled below.

84.2.3 Asset Pricing and Covariances

Recall that, from the definition of a conditional covariance $\text{cov}_t(x_{t+1}, y_{t+1})$, we have

$$\mathbb{E}_t(x_{t+1}y_{t+1}) = \text{cov}_t(x_{t+1}, y_{t+1}) + \mathbb{E}_t x_{t+1} \mathbb{E}_t y_{t+1} \tag{84.3}$$

If we apply this definition to the asset pricing equation (84.2) we obtain

$$p_t = \mathbb{E}_t m_{t+1} \mathbb{E}_t(d_{t+1} + p_{t+1}) + \text{cov}_t(m_{t+1}, d_{t+1} + p_{t+1}) \tag{84.4}$$

It is useful to regard equation (84.4) as a generalization of equation (84.1)

- In equation (84.1), the stochastic discount factor $m_{t+1} = \beta$, a constant.
- In equation (84.1), the covariance term $\text{cov}_t(m_{t+1}, d_{t+1} + p_{t+1})$ is zero because $m_{t+1} = \beta$.
- In equation (84.1), $\mathbb{E}_t m_{t+1}$ can be interpreted as the reciprocal of the one-period risk-free gross interest rate.
- When m_{t+1} covaries more negatively with the payout $p_{t+1} + d_{t+1}$, the price of the asset is lower.

Equation (84.4) asserts that the covariance of the stochastic discount factor with the one period payout $d_{t+1} + p_{t+1}$ is an important determinant of the price p_t.

We give examples of some models of stochastic discount factors that have been proposed later in this lecture and also in a later lecture.

84.2.4 The Price-Dividend Ratio

Aside from prices, another quantity of interest is the **price-dividend ratio** $v_t := p_t/d_t$.

Let's write down an expression that this ratio should satisfy.

We can divide both sides of (84.2) by d_t to get

$$v_t = \mathbb{E}_t \left[m_{t+1} \frac{d_{t+1}}{d_t} (1 + v_{t+1}) \right] \tag{84.5}$$

Below we'll discuss the implication of this equation.

84.3 Prices in the Risk-Neutral Case

What can we say about price dynamics on the basis of the models described above?

The answer to this question depends on

1. the process we specify for dividends
2. the stochastic discount factor and how it correlates with dividends

For now we'll study the risk-neutral case in which the stochastic discount factor is constant.

We'll focus on how an asset price depends on a dividend process.

84.3.1 Example 1: Constant Dividends

The simplest case is risk-neutral price of a constant, non-random dividend stream $d_t = d > 0$.

Removing the expectation from (84.1) and iterating forward gives

$$\begin{aligned} p_t &= \beta(d + p_{t+1}) \\ &= \beta(d + \beta(d + p_{t+2})) \\ &\vdots \\ &= \beta(d + \beta d + \beta^2 d + \cdots + \beta^{k-2} d + \beta^{k-1} p_{t+k}) \end{aligned}$$

If $\lim_{k \to +\infty} \beta^{k-1} p_{t+k} = 0$, this sequence converges to

$$\bar{p} := \frac{\beta d}{1 - \beta} \tag{84.6}$$

This is the equilibrium price in the constant dividend case.

Indeed, simple algebra shows that setting $p_t = \bar{p}$ for all t satisfies the difference equation $p_t = \beta(d + p_{t+1})$.

84.3.2 Example 2: Dividends with Deterministic Growth Paths

Consider a growing, non-random dividend process $d_{t+1} = g d_t$ where $0 < g\beta < 1$.

While prices are not usually constant when dividends grow over time, a price dividend-ratio can be.

If we guess this, substituting $v_t = v$ into (84.5) as well as our other assumptions, we get $v = \beta g(1 + v)$.

Since $\beta g < 1$, we have a unique positive solution:

$$v = \frac{\beta g}{1 - \beta g}$$

The price is then

$$p_t = \frac{\beta g}{1 - \beta g} d_t$$

If, in this example, we take $g = 1 + \kappa$ and let $\rho := 1/\beta - 1$, then the price becomes

$$p_t = \frac{1 + \kappa}{\rho - \kappa} d_t$$

This is called the **Gordon formula**.

84.3.3 Example 3: Markov Growth, Risk-Neutral Pricing

Next, we consider a dividend process

$$d_{t+1} = g_{t+1} d_t \tag{84.7}$$

The stochastic growth factor $\{g_t\}$ is given by

$$g_t = g(X_t), \quad t = 1, 2, \ldots$$

where

1. $\{X_t\}$ is a finite Markov chain with state space S and transition probabilities

$$P(x, y) := \mathbb{P}\{X_{t+1} = y \,|\, X_t = x\} \qquad (x, y \in S)$$

2. g is a given function on S taking nonnegative values

You can think of

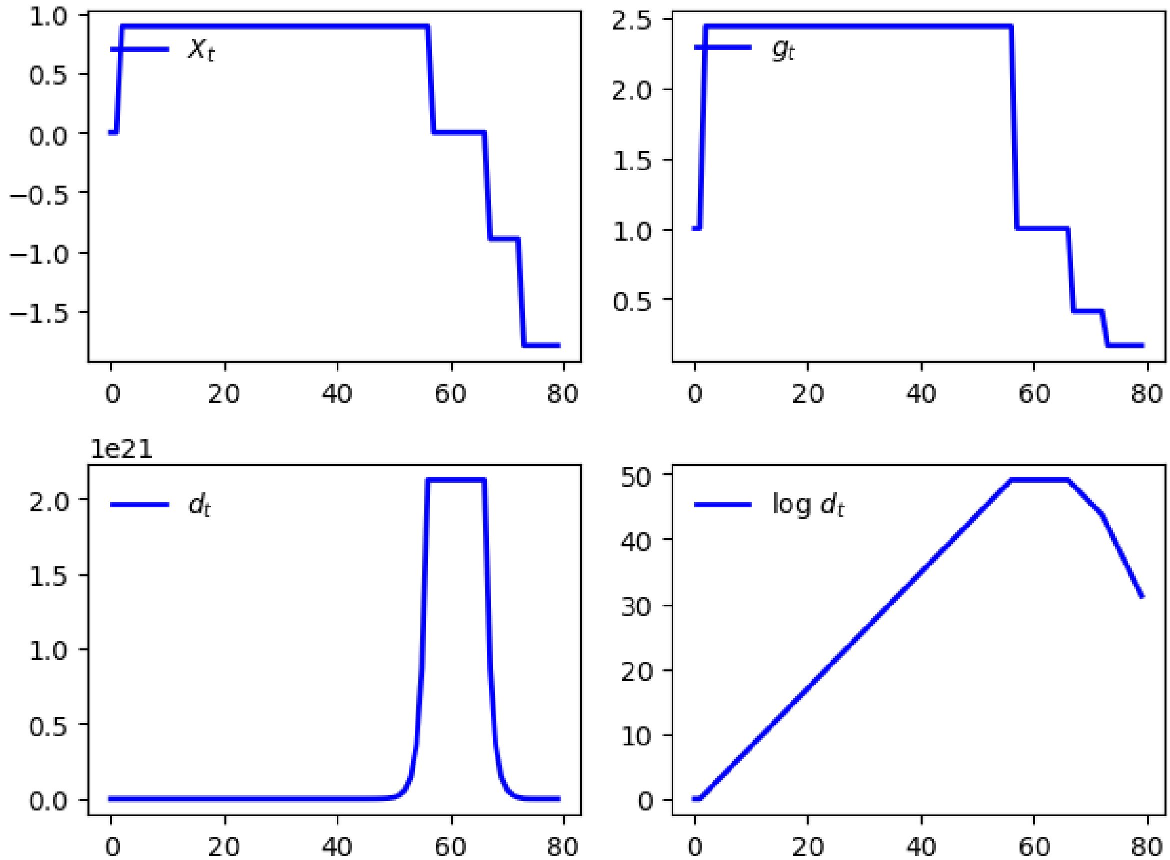
- S as n possible "states of the world" and X_t as the current state.
- g as a function that maps a given state X_t into a growth of dividends factor $g_t = g(X_t)$.
- $\ln g_t = \ln(d_{t+1}/d_t)$ is the growth rate of dividends.

(For a refresher on notation and theory for finite Markov chains see *this lecture*)

The next figure shows a simulation, where

- $\{X_t\}$ evolves as a discretized AR1 process produced using *Tauchen's method*.
- $g_t = \exp(X_t)$, so that $\ln g_t = X_t$ is the growth rate.

```
n = 7
mc = qe.tauchen(n, 0.96, 0.25)
sim_length = 80

x_series = mc.simulate(sim_length, init=np.median(mc.state_values))
g_series = np.exp(x_series)
d_series = np.cumprod(g_series) # Assumes d_0 = 1

series = [x_series, g_series, d_series, np.log(d_series)]
labels = ['$X_t$', '$g_t$', '$d_t$', r'$\log \, d_t$']

fig, axes = plt.subplots(2, 2)
for ax, s, label in zip(axes.flatten(), series, labels):
    ax.plot(s, 'b-', lw=2, label=label)
    ax.legend(loc='upper left', frameon=False)
plt.tight_layout()
plt.show()
```



Pricing Formula

To obtain asset prices in this setting, let's adapt our analysis from the case of deterministic growth.

In that case, we found that v is constant.

This encourages us to guess that, in the current case, v_t is a fixed function of the state X_t.

We seek a function v such that the price-dividend ratio satisfies $v_t = v(X_t)$.

We can substitute this guess into (84.5) to get

$$v(X_t) = \beta \mathbb{E}_t[g(X_{t+1})(1 + v(X_{t+1}))]$$

If we condition on $X_t = x$, this becomes

$$v(x) = \beta \sum_{y \in S} g(y)(1 + v(y))P(x, y)$$

or

$$v(x) = \beta \sum_{y \in S} K(x, y)(1 + v(y)) \quad \text{where} \quad K(x, y) := g(y)P(x, y) \tag{84.8}$$

Suppose that there are n possible states $x_1, \ldots, x_n$.

We can then think of (84.8) as n stacked equations, one for each state, and write it in matrix form as

$$v = \beta K(\mathbb{1} + v) \tag{84.9}$$

Here

- v is understood to be the column vector $(v(x_1), \ldots, v(x_n))'$.
- K is the matrix $(K(x_i, x_j))_{1 \leq i,j \leq n}$.
- $\mathbb{1}$ is a column vector of ones.

When does equation (84.9) have a unique solution?

From the *Neumann series lemma* and Gelfand's formula, equation (84.9) has a unique solution when βK has spectral radius strictly less than one.

Thus, we require that the eigenvalues of K be strictly less than β^{-1} in modulus.

The solution is then

$$v = (I - \beta K)^{-1} \beta K \mathbb{1} \tag{84.10}$$

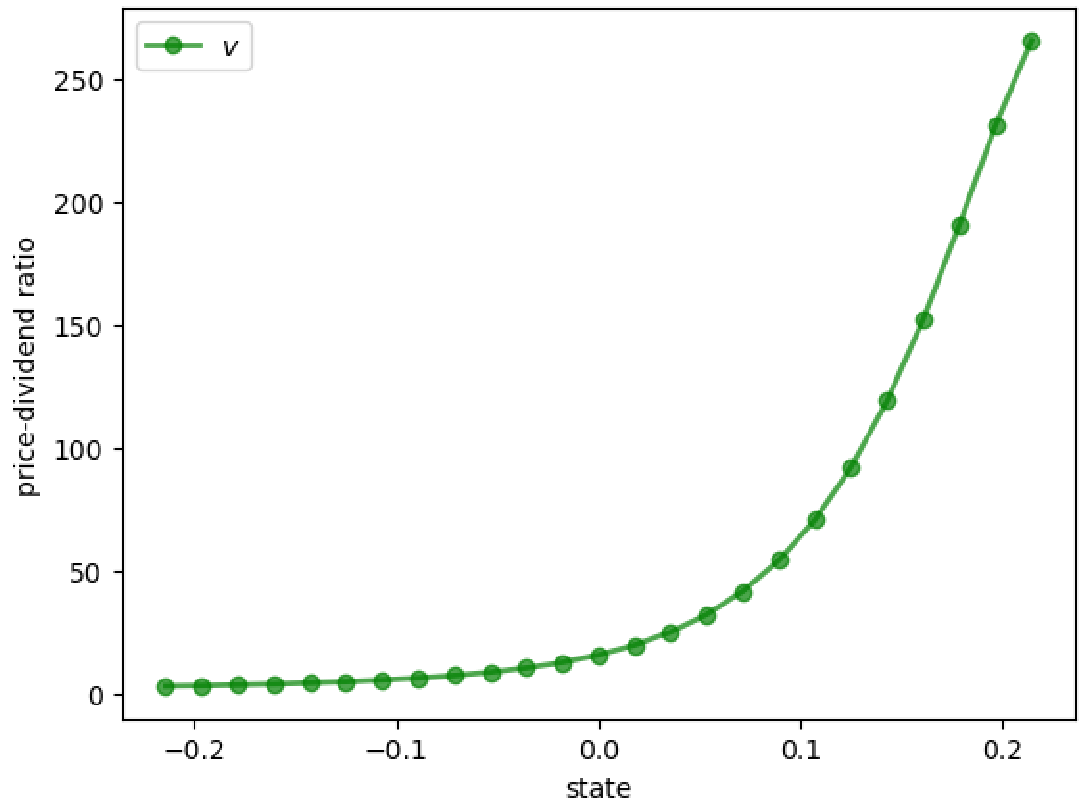
84.3.4 Code

Let's calculate and plot the price-dividend ratio at some parameters.

As before, we'll generate $\{X_t\}$ as a *discretized AR1 process* and set $g_t = \exp(X_t)$.

Here's the code, including a test of the spectral radius condition

```
n = 25  # Size of state space
β = 0.9
mc = qe.tauchen(n, 0.96, 0.02)

K = mc.P * np.exp(mc.state_values)

warning_message = "Spectral radius condition fails"
assert np.max(np.abs(eigvals(K))) < 1 / β,  warning_message

I = np.identity(n)
v = solve(I - β * K, β * K @ np.ones(n))

fig, ax = plt.subplots()
ax.plot(mc.state_values, v, 'g-o', lw=2, alpha=0.7, label='$v$')
ax.set_ylabel("price-dividend ratio")
ax.set_xlabel("state")
ax.legend(loc='upper left')
plt.show()
```



Why does the price-dividend ratio increase with the state?

The reason is that this Markov process is positively correlated, so high current states suggest high future states.

Moreover, dividend growth is increasing in the state.

The anticipation of high future dividend growth leads to a high price-dividend ratio.

84.4 Risk Aversion and Asset Prices

Now let's turn to the case where agents are risk averse.

We'll price several distinct assets, including

- An endowment stream
- A consol (a type of bond issued by the UK government in the 19th century)
- Call options on a consol

84.4.1 Pricing a Lucas Tree

Let's start with a version of the celebrated asset pricing model of Robert E. Lucas, Jr. [Lucas, 1978].

Lucas considered an abstract pure exchange economy with these features:

- a single non-storable consumption good
- a Markov process that governs the total amount of the consumption good available each period
- a single *tree* that each period yields *fruit* that equals the total amount of consumption available to the economy
- a competitive market in *shares* in the tree that entitles their owners to corresponding shares of the *dividend* stream, i.e., the *fruit* stream, yielded by the tree
- a representative consumer who in a competitive equilibrium
  - consumes the economy's entire endowment each period
  - owns 100 percent of the shares in the tree

As in [Lucas, 1978], we suppose that the stochastic discount factor takes the form

$$m_{t+1} = \beta \frac{u'(c_{t+1})}{u'(c_t)} \tag{84.11}$$

where u is a concave utility function and c_t is time t consumption of a representative consumer.

(A derivation of this expression is given in a later lecture)

Assume the existence of an endowment that follows growth process (84.7).

The asset being priced is a claim on the endowment process, i.e., the **Lucas tree** described above.

Following [Lucas, 1978], we suppose that in equilibrium the representative consumer's consumption equals the aggregate endowment, so that $d_t = c_t$ for all t.

For utility, we'll assume the **constant relative risk aversion** (CRRA) specification

$$u(c) = \frac{c^{1-\gamma}}{1-\gamma} \text{ with } \gamma > 0 \tag{84.12}$$

When $\gamma = 1$ we let $u(c) = \ln c$.

Inserting the CRRA specification into (84.11) and using $c_t = d_t$ gives

$$m_{t+1} = \beta \left(\frac{c_{t+1}}{c_t}\right)^{-\gamma} = \beta g_{t+1}^{-\gamma} \tag{84.13}$$

Substituting this into (84.5) gives the price-dividend ratio formula

$$v(X_t) = \beta \mathbb{E}_t \left[g(X_{t+1})^{1-\gamma} (1 + v(X_{t+1})) \right] \tag{84.14}$$

Conditioning on $X_t = x$, we can write this as

$$v(x) = \beta \sum_{y \in S} g(y)^{1-\gamma} (1 + v(y)) P(x, y)$$

If we let

$$J(x, y) := g(y)^{1-\gamma} P(x, y)$$

then we can rewrite equation (84.14) in vector form as

$$v = \beta J (\mathbb{1} + v)$$

Assuming that the spectral radius of J is strictly less than β^{-1}, this equation has the unique solution

$$v = (I - \beta J)^{-1} \beta J \mathbb{1} \tag{84.15}$$

We will define a function tree_price to compute v given parameters stored in the class AssetPriceModel

```python
class AssetPriceModel:
    """
    A class that stores the primitives of the asset pricing model.

    Parameters
    ----------
    β : scalar, float
        Discount factor
    mc : MarkovChain
        Contains the transition matrix and set of state values for the state
        process
    γ : scalar(float)
        Coefficient of risk aversion
    g : callable
        The function mapping states to growth rates

    """
    def __init__(self, β=0.96, mc=None, γ=2.0, g=np.exp):
        self.β, self.γ = β, γ
        self.g = g

        # A default process for the Markov chain
        if mc is None:
            self.ρ = 0.9
            self.σ = 0.02
            self.mc = qe.tauchen(n, self.ρ, self.σ)
        else:
            self.mc = mc

        self.n = self.mc.P.shape[0]

    def test_stability(self, Q):
        """
        Stability test for a given matrix Q.
        """
        sr = np.max(np.abs(eigvals(Q)))
        if not sr < 1 / self.β:
            msg = f"Spectral radius condition failed with radius = {sr}"
            raise ValueError(msg)


def tree_price(ap):
    """
    Computes the price-dividend ratio of the Lucas tree.

    Parameters
    ----------
    ap: AssetPriceModel
        An instance of AssetPriceModel containing primitives

    Returns
    -------
```

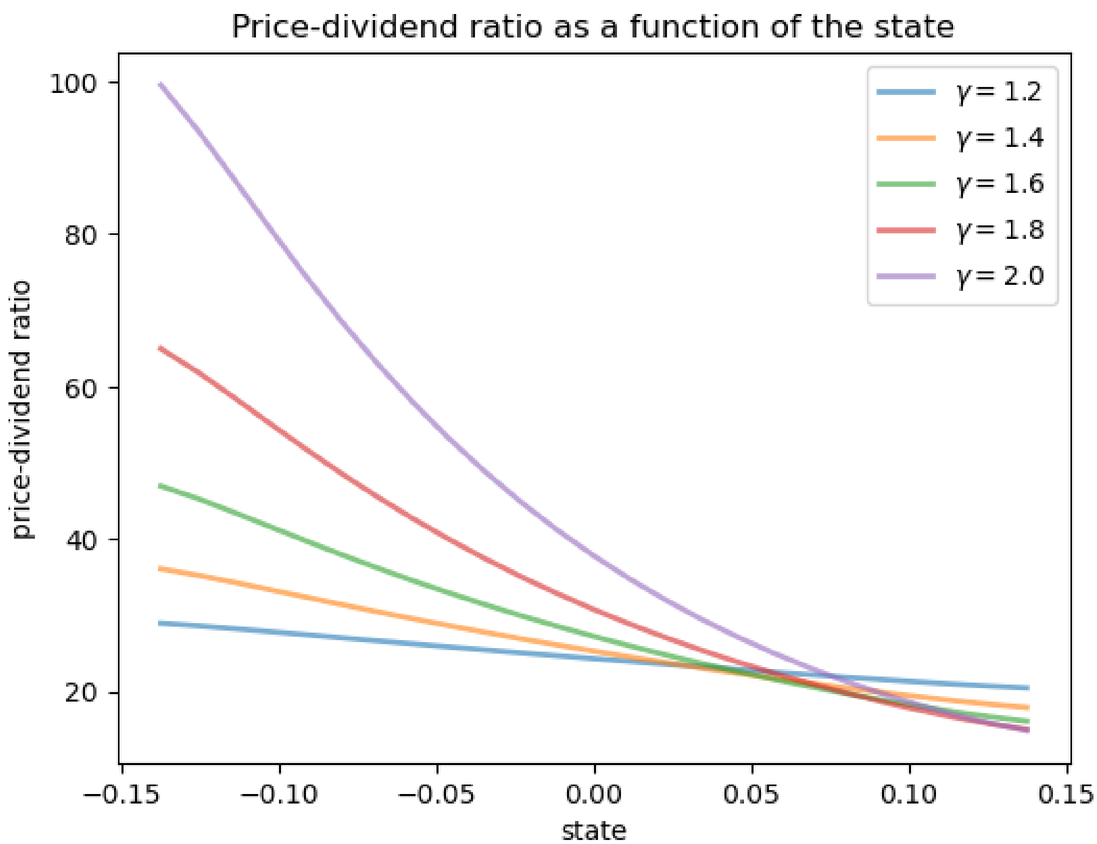
(continues on next page)

(continued from previous page)

```
    v : array_like(float)
        Lucas tree price-dividend ratio

    """
    # Simplify names, set up matrices
    β, γ, P, y = ap.β, ap.γ, ap.mc.P, ap.mc.state_values
    J = P * ap.g(y)**(1 - γ)

    # Make sure that a unique solution exists
    ap.test_stability(J)

    # Compute v
    I = np.identity(ap.n)
    Ones = np.ones(ap.n)
    v = solve(I - β * J, β * J @ Ones)

    return v
```

Here's a plot of v as a function of the state for several values of γ, with a positively correlated Markov process and $g(x) = \exp(x)$

```
γs = [1.2, 1.4, 1.6, 1.8, 2.0]
ap = AssetPriceModel()
states = ap.mc.state_values

fig, ax = plt.subplots()

for γ in γs:
    ap.γ = γ
    v = tree_price(ap)
    ax.plot(states, v, lw=2, alpha=0.6, label=rf"$\gamma = {γ}$")

ax.set_title('Price-dividend ratio as a function of the state')
ax.set_ylabel("price-dividend ratio")
ax.set_xlabel("state")
ax.legend(loc='upper right')
plt.show()
```



Notice that v is decreasing in each case.

This is because, with a positively correlated state process, higher states indicate higher future consumption growth.

With the stochastic discount factor (84.13), higher growth decreases the discount factor, lowering the weight placed on future dividends.

Special Cases

In the special case $\gamma = 1$, we have $J = P$.

Recalling that $P^i \mathbb{1} = \mathbb{1}$ for all i and applying *Neumann's geometric series lemma*, we are led to

$$v = \beta(I - \beta P)^{-1}\mathbb{1} = \beta \sum_{i=0}^{\infty} \beta^i P^i \mathbb{1} = \beta \frac{1}{1-\beta}\mathbb{1}$$

Thus, with log preferences, the price-dividend ratio for a Lucas tree is constant.

Alternatively, if $\gamma = 0$, then $J = K$ and we recover the risk-neutral solution (84.10).

This is as expected, since $\gamma = 0$ implies $u(c) = c$ (and hence agents are risk-neutral).

84.4.2 A Risk-Free Consol

Consider the same pure exchange representative agent economy.

A risk-free consol promises to pay a constant amount $\zeta > 0$ each period.

Recycling notation, let p_t now be the price of an ex-coupon claim to the consol.

An ex-coupon claim to the consol entitles an owner at the end of period t to

- ζ in period $t+1$, plus
- the right to sell the claim for p_{t+1} next period

The price satisfies (84.2) with $d_t = \zeta$, or

$$p_t = \mathbb{E}_t \left[m_{t+1} (\zeta + p_{t+1}) \right]$$

With the stochastic discount factor (84.13), this becomes

$$p_t = \mathbb{E}_t \left[\beta g_{t+1}^{-\gamma} (\zeta + p_{t+1}) \right] \tag{84.16}$$

Guessing a solution of the form $p_t = p(X_t)$ and conditioning on $X_t = x$, we get

$$p(x) = \beta \sum_{y \in S} g(y)^{-\gamma} (\zeta + p(y)) P(x, y)$$

Letting $M(x, y) = P(x, y) g(y)^{-\gamma}$ and rewriting in vector notation yields the solution

$$p = (I - \beta M)^{-1} \beta M \zeta \mathbb{1} \tag{84.17}$$

The above is implemented in the function consol_price.

```python
def consol_price(ap, ζ):
    """
    Computes price of a consol bond with payoff ζ

    Parameters
    ----------
    ap: AssetPriceModel
        An instance of AssetPriceModel containing primitives

    ζ : scalar(float)
        Coupon of the console

    Returns
    -------
    p : array_like(float)
        Console bond prices

    """
    # Simplify names, set up matrices
    β, γ, P, y = ap.β, ap.γ, ap.mc.P, ap.mc.state_values
    M = P * ap.g(y)**(- γ)

    # Make sure that a unique solution exists
    ap.test_stability(M)

    # Compute price
    I = np.identity(ap.n)
```

(continues on next page)

(continued from previous page)

```
    Ones = np.ones(ap.n)
    p = solve(I - β * M, β * ζ * M @ Ones)

    return p
```

84.4.3 Pricing an Option to Purchase the Consol

Let's now price options of various maturities.

We'll study an option that gives the owner the right to purchase a consol at a price p_S.

An Infinite Horizon Call Option

We want to price an **infinite horizon** option to purchase a consol at a price p_S.

The option entitles the owner at the beginning of a period either

1. to purchase the bond at price p_S now, or
2. not to exercise the option to purchase the asset now but to retain the right to exercise it later

Thus, the owner either *exercises* the option now or chooses *not to exercise* and wait until next period.

This is termed an infinite-horizon **call option** with **strike price** p_S.

The owner of the option is entitled to purchase the consol at price p_S at the beginning of any period, after the coupon has been paid to the previous owner of the bond.

The fundamentals of the economy are identical with the one above, including the stochastic discount factor and the process for consumption.

Let $w(X_t, p_S)$ be the value of the option when the time t growth state is known to be X_t but *before* the owner has decided whether to exercise the option at time t (i.e., today).

Recalling that $p(X_t)$ is the value of the consol when the initial growth state is X_t, the value of the option satisfies

$$w(X_t, p_S) = \max \left\{ \beta \, \mathbb{E}_t \frac{u'(c_{t+1})}{u'(c_t)} w(X_{t+1}, p_S), \; p(X_t) - p_S \right\}$$

The first term on the right is the value of waiting, while the second is the value of exercising now.

We can also write this as

$$w(x, p_S) = \max \left\{ \beta \sum_{y \in S} P(x, y) g(y)^{-\gamma} w(y, p_S), \; p(x) - p_S \right\} \tag{84.18}$$

With $M(x, y) = P(x, y) g(y)^{-\gamma}$ and w as the vector of values $(w(x_i), p_S)_{i=1}^n$, we can express (84.18) as the nonlinear vector equation

$$w = \max\{\beta M w, \; p - p_S \mathbb{1}\} \tag{84.19}$$

To solve (84.19), form an operator T that maps vector w into vector Tw via

$$Tw = \max\{\beta M w, \; p - p_S \mathbb{1}\}$$

Start at some initial w and iterate with T to convergence .

We can find the solution with the following function call_option

```python
def call_option(ap, ζ, p_s, ϵ=1e-7):
    """
    Computes price of a call option on a consol bond.

    Parameters
    ----------
    ap: AssetPriceModel
        An instance of AssetPriceModel containing primitives

    ζ : scalar(float)
        Coupon of the console

    p_s : scalar(float)
        Strike price

    ϵ : scalar(float), optional(default=1e-8)
        Tolerance for infinite horizon problem

    Returns
    -------
    w : array_like(float)
        Infinite horizon call option prices

    """
    # Simplify names, set up matrices
    β, γ, P, y = ap.β, ap.γ, ap.mc.P, ap.mc.state_values
    M = P * ap.g(y)**(- γ)

    # Make sure that a unique consol price exists
    ap.test_stability(M)

    # Compute option price
    p = consol_price(ap, ζ)
    w = np.zeros(ap.n)
    error = ϵ + 1
    while error > ϵ:
        # Maximize across columns
        w_new = np.maximum(β * M @ w, p - p_s)
        # Find maximal difference of each component and update
        error = np.amax(np.abs(w - w_new))
        w = w_new

    return w
```

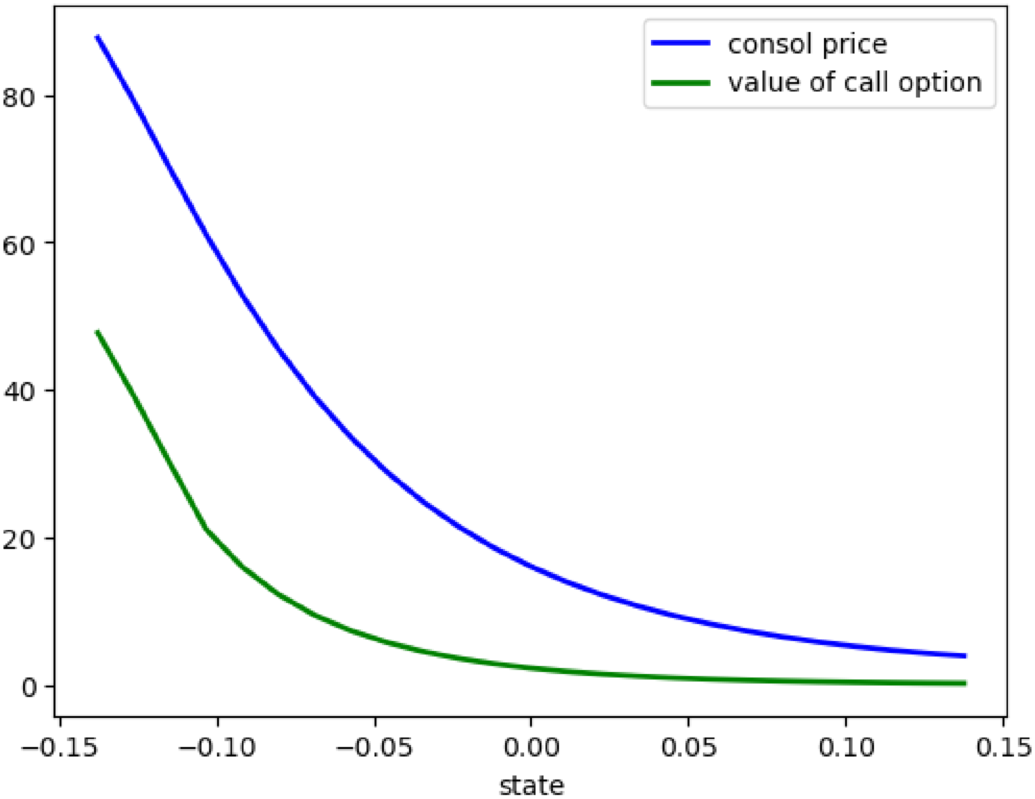
Here's a plot of w compared to the consol price when $P_S = 40$

```python
ap = AssetPriceModel(β=0.9)
ζ = 1.0
strike_price = 40

x = ap.mc.state_values
p = consol_price(ap, ζ)
w = call_option(ap, ζ, strike_price)

fig, ax = plt.subplots()
ax.plot(x, p, 'b-', lw=2, label='consol price')
ax.plot(x, w, 'g-', lw=2, label='value of call option')
```

(continues on next page)

(continued from previous page)

```
ax.set_xlabel("state")
ax.legend(loc='upper right')
plt.show()
```



In high values of the Markov growth state, the value of the option is close to zero.

This is despite the facts that the Markov chain is irreducible and that low states — where the consol prices are high — will be visited recurrently.

The reason for low valuations in high Markov growth states is that $\beta = 0.9$, so future payoffs are discounted substantially.

84.4.4 Risk-Free Rates

Let's look at risk-free interest rates over different periods.

The One-period Risk-free Interest Rate

As before, the stochastic discount factor is $m_{t+1} = \beta g_{t+1}^{-\gamma}$.

It follows that the reciprocal R_t^{-1} of the gross risk-free interest rate R_t in state x is

$$\mathbb{E}_t m_{t+1} = \beta \sum_{y \in S} P(x, y) g(y)^{-\gamma}$$

We can write this as

$$m_1 = \beta M \mathbb{1}$$

where the i-th element of m_1 is the reciprocal of the one-period gross risk-free interest rate in state x_i.

Other Terms

Let m_j be an $n \times 1$ vector whose i th component is the reciprocal of the j -period gross risk-free interest rate in state x_i.

Then $m_1 = \beta M$, and $m_{j+1} = M m_j$ for $j \geq 1$.

84.5 Exercises

Exercise 84.5.1

In the lecture, we considered **ex-dividend assets**.

A **cum-dividend** asset is a claim to the stream $d_t, d_{t+1}, \ldots$.

Following (84.1), find the risk-neutral asset pricing equation for one unit of a cum-dividend asset.

With a constant, non-random dividend stream $d_t = d > 0$, what is the equilibrium price of a cum-dividend asset?

With a growing, non-random dividend process $d_t = g d_t$ where $0 < g\beta < 1$, what is the equilibrium price of a cum-dividend asset?

Solution

For a cum-dividend asset, the basic risk-neutral asset pricing equation is

$$p_t = d_t + \beta \mathbb{E}_t [p_{t+1}]$$

With constant dividends, the equilibrium price is

$$p_t = \frac{1}{1 - \beta} d_t$$

With a growing, non-random dividend process, the equilibrium price is

$$p_t = \frac{1}{1 - \beta g} d_t$$

Exercise 84.5.2

Consider the following primitives

```
n = 5  # Size of State Space
P = np.full((n, n), 0.0125)
P[range(n), range(n)] += 1 - P.sum(1)
# State values of the Markov chain
s = np.array([0.95, 0.975, 1.0, 1.025, 1.05])
γ = 2.0
β = 0.94
```

Let g be defined by $g(x) = x$ (that is, g is the identity map).

Compute the price of the Lucas tree.

Do the same for

- the price of the risk-free consol when $\zeta = 1$
- the call option on the consol when $\zeta = 1$ and $p_S = 150.0$

Solution

First, let's enter the parameters:

```
n = 5
P = np.full((n, n), 0.0125)
P[range(n), range(n)] += 1 - P.sum(1)
s = np.array([0.95, 0.975, 1.0, 1.025, 1.05])  # State values
mc = qe.MarkovChain(P, state_values=s)

γ = 2.0
β = 0.94
ζ = 1.0
p_s = 150.0
```

Next, we'll create an instance of `AssetPriceModel` to feed into the functions

```
apm = AssetPriceModel(β=β, mc=mc, γ=γ, g=lambda x: x)
```

Now we just need to call the relevant functions on the data:

```
tree_price(apm)
```

```
array([29.47401578, 21.93570661, 17.57142236, 14.72515002, 12.72221763])
```

```
consol_price(apm, ζ)
```

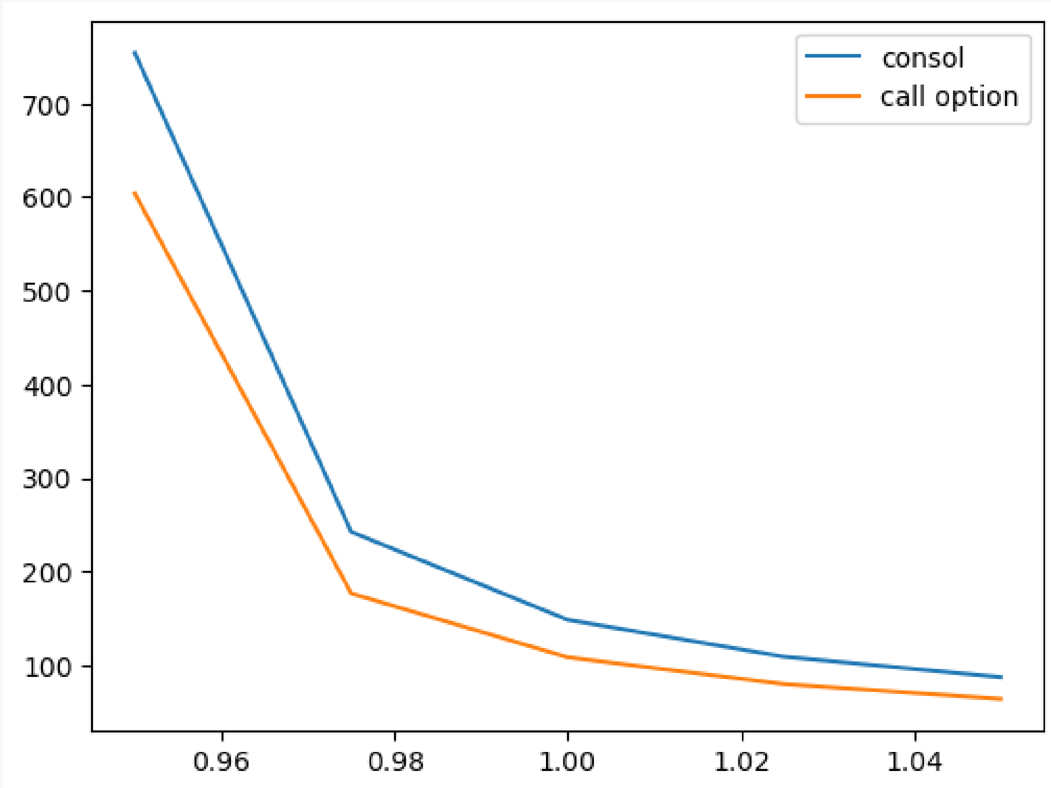
```
array([753.87100476, 242.55144082, 148.67554548, 109.25108965,
        87.56860139])
```

```
call_option(apm, ζ, p_s)
```

```
array([603.87100476, 176.8393343 , 108.67734499,  80.05179254,
        64.30843748])
```

Let's show the last two functions as a plot

```
fig, ax = plt.subplots()
ax.plot(s, consol_price(apm, ζ), label='consol')
ax.plot(s, call_option(apm, ζ, p_s), label='call option')
ax.legend()
plt.show()
```


consol
call option
700
600
500
400
300
200
100
0.96
0.98
1.00
1.02
1.04

Exercise 84.5.3

Let's consider finite horizon call options, which are more common than infinite horizon ones.

Finite horizon options obey functional equations closely related to (84.18).

A k period option expires after k periods.

If we view today as date zero, a k period option gives the owner the right to exercise the option to purchase the risk-free consol at the strike price p_S at dates $0, 1, \ldots, k-1$.

The option expires at time k.

Thus, for $k = 1, 2, \ldots$, let $w(x, k)$ be the value of a k-period option.

It obeys

$$w(x, k) = \max \left\{ \beta \sum_{y \in S} P(x, y) g(y)^{-\gamma} w(y, k-1), \; p(x) - p_S \right\}$$

where $w(x, 0) = 0$ for all x.

We can express this as a sequence of nonlinear vector equations

$$w_k = \max\{\beta M w_{k-1}, \; p - p_S \mathbb{1}\} \quad k = 1, 2, \ldots \quad \text{with } w_0 = 0$$

Write a function that computes w_k for any given k.

Compute the value of the option with `k = 5` and `k = 25` using parameter values as in Exercise 84.5.1.

Is one higher than the other? Can you give intuition?

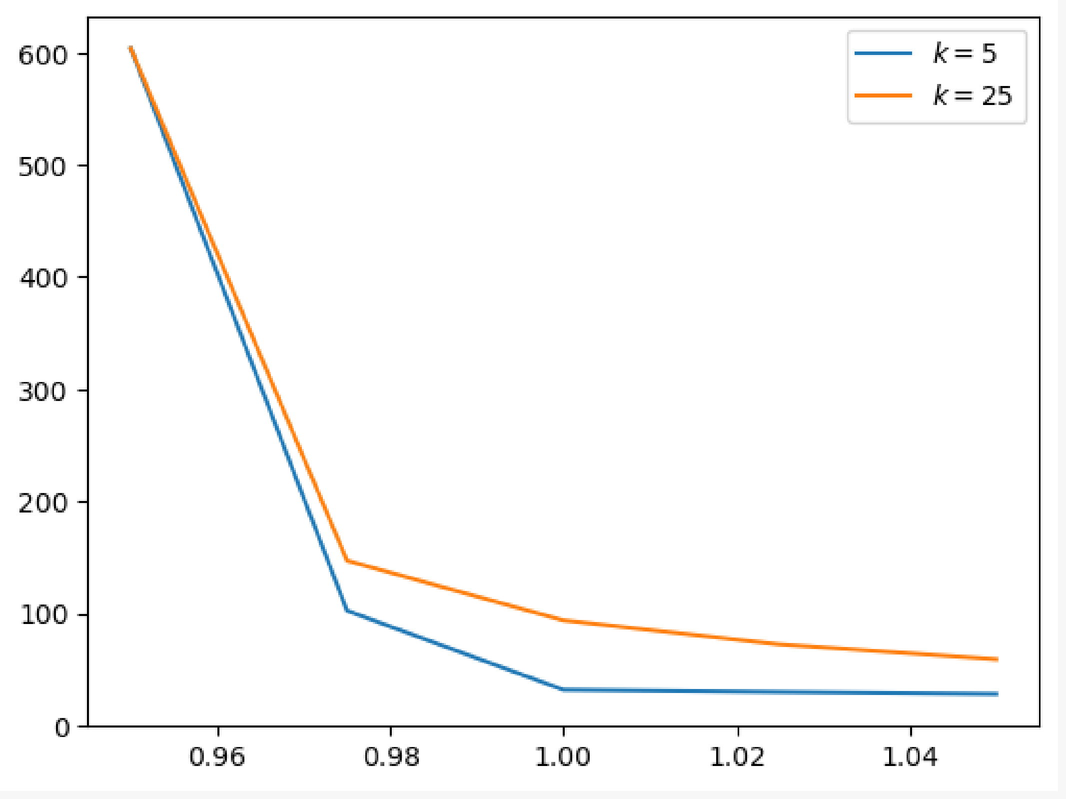
Solution

Here's a suitable function:

```python
def finite_horizon_call_option(ap, ζ, p_s, k):
    """
    Computes k period option value.
    """
    # Simplify names, set up matrices
    β, γ, P, y = ap.β, ap.γ, ap.mc.P, ap.mc.state_values
    M = P * ap.g(y)**(- γ)

    # Make sure that a unique solution exists
    ap.test_stability(M)


    # Compute option price
    p = consol_price(ap, ζ)
    w = np.zeros(ap.n)
    for i in range(k):
        # Maximize across columns
        w = np.maximum(β * M @ w, p - p_s)

    return w
```

Now let's compute the option values at `k=5` and `k=25`

```python
fig, ax = plt.subplots()
for k in [5, 25]:
    w = finite_horizon_call_option(apm, ζ, p_s, k)
    ax.plot(s, w, label=rf'$k = {k}$')
ax.legend()
plt.show()
```



Not surprisingly, options with larger k are worth more.

This is because an owner has a longer horizon over which the option can be exercised.

CHAPTER

EIGHTYFIVE

DOUBTS OR VARIABILITY?

Contents

> No one has found risk aversion parameters of 50 or 100 in the diversification of individual portfolios, in the level of insurance deductibles, in the wage premiums associated with occupations with high earnings risk, or in the revenues raised by state-operated lotteries. It would be good to have the equity premium resolved, but I think we need to look beyond high estimates of risk aversion to do it. – Robert E. Lucas Jr., [Lucas, 2003]

85.1 Overview

This lecture describes machinery that empirical macro-finance economists have used to evaluate the fits of structural statistical models that link asset prices to aggregate consumption.

The Lucas asset pricing model [Lucas, 1978] functions as a benchmark that motivates much of this work.

Note

New Keynesians call the consumption Euler equation for a one-period risk-free bond in the Lucas [Lucas, 1978] model the **IS curve**.

The distinguished **old Keynesian** disapproved of that name because the object it described was so remote from the investment function that was an important component of the IS curve of John R. Hicks [Hicks, 1937] that Tobin used.

See [Tobin, 1992].

In two classic papers, Lars Peter Hansen and Kenneth Singleton used the method of maximum likelihood [Hansen and Singleton, 1983] and a generalized method of moments [Hansen and Singleton, 1982] to investigate how well Lucas's model fit some post WWII data.

The Hansen-Singleton papers systematically organized evidence about directions in which Lucas's model misfit the data that macroeconomists subsequently called

- an **equity premium** puzzle
- a **risk-free rate** puzzle

Note

Mehra and Prescott [1985] is widely credited for naming the **equity premium** puzzle.

Weil [1989] is widely credited for naming the **risk-free rate** puzzle.

These *puzzles* are just ways of summarizing particular dimensions along which a particular asset pricing model – such as Lucas's – fails empirically.

They are thus special cases of specification failures detected by statistical diagnostics constructed earlier by [Hansen and Singleton, 1983] and [Hansen and Singleton, 1982].

Macro-finance models that purport to resolve such puzzles all do so by changing features of the economic environment assumed by Lucas [Lucas, 1978].

Many important papers have proceeded by altering the *preferences* that Lucas had imputed to a representative agent.

Hansen-Jagannathan bounds are a key tool for evaluating how well such re-specifications do in correcting those misfits of Lucas's 1978 model.

This lecture begins with a description of the [Hansen and Jagannathan, 1991] machinery.

After doing that, we proceed to describe a line of research that altered Lucas's preference specification in ways that we can think of as being designed with the Hansen-Jagannathan bounds in mind.

We'll organize much of this lecture around parts of the paper by Thomas Tallarini [Tallarini, 2000].

His paper is particularly enlightening for macro-finance researchers because it showed that a recursive preference specification could fit both the equity premium and the risk-free rate, thus *resolving* both of the puzzles mentioned above.

But like any good paper in applied economics, in answering some questions (i.e., resolving some puzzles), Tallarini's paper naturally posed new ones.

Thus, Tallarini's puzzles-resolving required setting the risk-aversion coefficient γ to around 50 for a random-walk consumption model and around 75 for a trend-stationary model, exactly the range that provoked the skepticism in the above quote from Lucas [2003].

This brings us to the next parts of this lecture.

Lucas's skeptical response to Tallarini's explanation of the two puzzles led Barillas *et al.* [2009] to ask whether those large γ values really measure aversion to atemporal risk, or whether they instead measure the agent's doubts about the underlying probability model.

Their answer, and the theme of the remaining parts of this lecture, is that much of what looks like "risk aversion" can be reinterpreted as **model uncertainty**.

The same recursion that defines Tallarini's risk-sensitive agent is observationally equivalent to another recursion that expresses an agent's concern that the probability model governing consumption growth may be wrong.

Under this reading, a parameter value that indicates extreme risk aversion in one interpretation of the recursion indicates concerns about *misspecification* in another interpretation of the same recursion.

Barillas *et al.* [2009] show that modest amounts of model uncertainty can substitute for large amounts of risk aversion in terms of choices and effects on asset prices.

This reinterpretation changes the welfare question that asset prices answer.

Do large risk premia measure the benefits from reducing well-understood aggregate fluctuations, or do they measure benefits from reducing doubts about the model describing consumption growth?

To proceed, we begin by describing Hansen and Jagannathan [1991] bounds, then specify the statistical environment, lay out four related preference specifications and the connections among them, and finally revisit Tallarini's calibration through the lens of detection-error probabilities.

Along the way, we draw on ideas and techniques from

- *Asset Pricing: Finite State Models*, where we introduce stochastic discount factors, and
- *Likelihood Ratio Processes*, where we develop the likelihood-ratio machinery that reappears here as the worst-case distortion $\hat{g}$.

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install pandas-datareader
```

We use the following imports:

```
import datetime as dt
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from pandas_datareader import data as web
from scipy.stats import norm
from scipy.optimize import brentq
```

We also set up calibration inputs and compute the covariance matrix of equity and risk-free returns from reported moments.

```
β = 0.995
T = 235

# Table 2 parameters
rw = dict(μ=0.00495, σ_ε=0.0050)
ts = dict(μ=0.00418, σ_ε=0.0050, ρ=0.980, ζ=-4.48)

# Table 1 moments, converted from percent to decimals
r_e_mean, r_e_std = 0.0227, 0.0768
r_f_mean, r_f_std = 0.0032, 0.0061
r_excess_std = 0.0767

R_mean = np.array([1.0 + r_e_mean, 1.0 + r_f_mean])
cov_erf = (r_e_std**2 + r_f_std**2 - r_excess_std**2) / 2.0
Σ_R = np.array(
    [
```

(continues on next page)

(continued from previous page)

```
        [r_e_std**2, cov_erf],
        [cov_erf, r_f_std**2],
    ]
)
Σ_R_inv = np.linalg.inv(Σ_R)
```

85.2 Asset pricing 101

85.2.1 Pricing kernel and the risk-free rate

Let's briefly review a few key concepts from *Asset Pricing: Finite State Models*.

A random variable m_{t+1} is called a **stochastic discount factor** if, for a one-period payoff y_{t+1} with time-t price p_t, it satisfies

$$p_t = E_t(m_{t+1} y_{t+1}), \tag{85.1}$$

where E_t denotes the mathematical expectation conditioned on date-t information.

For time-separable CRRA preferences with discount factor β and coefficient of relative risk aversion γ, the marginal rate of substitution gives

$$m_{t+1} = \beta \left(\frac{C_{t+1}}{C_t} \right)^{-\gamma}, \tag{85.2}$$

where C_t is consumption at time t.

Setting $y_{t+1} = 1$ (a risk-free bond) in (85.1) yields the reciprocal of the gross one-period risk-free rate:

$$\frac{1}{R_t^f} = E_t[m_{t+1}] = E_t \left[\beta \left(\frac{C_{t+1}}{C_t} \right)^{-\gamma} \right]. \tag{85.3}$$

85.2.2 Hansen–Jagannathan bounds

Let R^e_{t+1} denote the gross return on a risky asset (e.g., the market portfolio) and R^f_{t+1} the gross return on a one-period risk-free bond.

The **excess return** is

$$\xi_{t+1} = R^e_{t+1} - R^f_{t+1}.$$

An excess return is the payoff on a zero-cost portfolio that is long one dollar of the risky asset and short one dollar of the risk-free bond.

Because the portfolio costs nothing to enter, its price is $p_t = 0$, so (85.1) implies

$$0 = E_t[m_{t+1} \xi_{t+1}].$$

We can decompose the expectation of a product into a covariance plus a product of expectations:

$$E_t[m_{t+1} \xi_{t+1}] = \text{cov}_t(m_{t+1}, \xi_{t+1}) + E_t[m_{t+1}] E_t[\xi_{t+1}],$$

where cov_t denotes the conditional covariance and σ_t will denote the conditional standard deviation.

Setting the left-hand side to zero and solving for the expected excess return gives

$$E_t[\xi_{t+1}] = -\frac{\text{cov}_t(m_{t+1}, \xi_{t+1})}{E_t[m_{t+1}]}.$$

Taking absolute values and applying the **Cauchy–Schwarz inequality** $|\text{cov}(X, Y)| \leq \sigma(X)\sigma(Y)$ yields

$$\frac{|E_t[\xi_{t+1}]|}{\sigma_t(\xi_{t+1})} \leq \frac{\sigma_t(m_{t+1})}{E_t[m_{t+1}]}. \tag{85.4}$$

The left-hand side of (85.4) is the **Sharpe ratio**: the expected excess return per unit of return volatility.

The right-hand side, $\sigma_t(m)/E_t(m)$, is the **market price of risk**: the maximum Sharpe ratio attainable in the market.

In words, no asset's Sharpe ratio can exceed the market price of risk.

Unconditional version

The bound (85.4) is stated in conditional terms.

There is an unconditional counterpart that involves a vector of n gross returns R_{t+1} (e.g., equity and risk-free) with unconditional mean $E(R)$ and covariance matrix Σ_R:

$$\sigma(m) \geq \sqrt{b^\top \Sigma_R^{-1} b}, \qquad b = \mathbf{1} - E(m)E(R). \tag{85.5}$$

Exercise 1 walks through a derivation of this unconditional bound.

The function below computes the right-hand side of (85.5) for any given value of $E(m)$.

```
def hj_std_bound(E_m):
    b = np.ones(2) - E_m * R_mean
    var_lb = b @ Σ_R_inv @ b
    return np.sqrt(np.maximum(var_lb, 0.0))
```

85.2.3 Two puzzles

Reconciling formula (85.2) with the market price of risk extracted from data on asset returns (like those in Table 1 below) requires a value of γ so high that it provokes skepticism.

This is the **equity premium puzzle**.

But high values of γ bring another difficulty.

High values of γ that deliver enough volatility $\sigma(m)$ also push $E(m)$, the reciprocal of the gross risk-free rate, too far down, away from the Hansen–Jagannathan bound.

This is the **risk-free rate puzzle** of Weil [1989].

Tallarini [2000] showed that recursive preferences with IES $= 1$ can clear the HJ bar while avoiding the risk-free rate puzzle.

The figure below reproduces Tallarini's key diagnostic.

Because it motivates much of what follows, we show Tallarini's figure before developing the underlying theory.

Closed-form expressions for the Epstein–Zin SDF moments used in the plot are derived in Exercise 2.

The code below implements them alongside the corresponding CRRA moments.

```
def moments_type1_rw(γ):
    μ, σ = rw["μ"], rw["σ_ε"]
    E_m = β * np.exp(-μ + 0.5 * σ**2 * (2.0 * γ - 1.0))
    var_log_m = (σ * γ) ** 2
    mpr = np.sqrt(np.exp(var_log_m) - 1.0)
    return E_m, mpr


def moments_type1_ts(γ):
    μ, σ, ρ = ts["μ"], ts["σ_ε"], ts["ρ"]
    mean_term = 1.0 - (2.0 * (1.0 - β) * (1.0 - γ)) / (1.0 - β * ρ) \
                + (1.0 - ρ) / (1.0 + ρ)
    E_m = β * np.exp(-μ + 0.5 * σ**2 * mean_term)
    var_term = (((1.0 - β) * (1.0 - γ)) / (1.0 - β * ρ) - 1.0) ** 2 \
                + (1.0 - ρ) / (1.0 + ρ)
    var_log_m = σ**2 * var_term
    mpr = np.sqrt(np.exp(var_log_m) - 1.0)
    return E_m, mpr


def moments_crra_rw(γ):
    μ, σ = rw["μ"], rw["σ_ε"]
    var_log_m = (γ * σ) ** 2
    mean_log_m = np.log(β) - γ * μ
    E_m = np.exp(mean_log_m + 0.5 * var_log_m)
    mpr = np.sqrt(np.exp(var_log_m) - 1.0)
    return E_m, mpr
```

For each value of $\gamma \in \{1, 5, 10, \ldots, 51\}$, we plot the implied $(E(m), \sigma(m))$ pair for three combinations of specifications of preferences and consumption growth processes.

These are time-separable CRRA (crosses), Epstein–Zin preferences with random-walk consumption (circles), and Epstein–Zin preferences with trend-stationary consumption (pluses).

```
γ_grid = np.arange(1, 55, 5)

Em_rw = np.array([moments_type1_rw(γ)[0] for γ in γ_grid])
σ_m_rw = np.array(
    [moments_type1_rw(γ)[0] * moments_type1_rw(γ)[1] for γ in γ_grid])

Em_ts = np.array([moments_type1_ts(γ)[0] for γ in γ_grid])
σ_m_ts = np.array(
    [moments_type1_ts(γ)[0] * moments_type1_ts(γ)[1] for γ in γ_grid])

Em_crra = np.array([moments_crra_rw(γ)[0] for γ in γ_grid])
σ_m_crra = np.array(
    [moments_crra_rw(γ)[0] * moments_crra_rw(γ)[1] for γ in γ_grid])

Em_grid = np.linspace(0.8, 1.01, 1000)
HJ_std = np.array([hj_std_bound(x) for x in Em_grid])

fig, ax = plt.subplots(figsize=(7, 5))
ax.plot(Em_grid, HJ_std, lw=2, color="black",
                            label="Hansen-Jagannathan bound")
ax.plot(Em_rw, σ_m_rw, "o", lw=2,
                            label="Epstein-Zin, random walk")
ax.plot(Em_ts, σ_m_ts, "+", lw=2,
                            label="Epstein-Zin, trend stationary")
```

(continues on next page)

(continued from previous page)

```
ax.plot(Em_crra, σ_m_crra, "x", lw=2,
                            label="time-separable CRRA")

ax.set_xlabel(r"$E(m)$")
ax.set_ylabel(r"$\sigma(m)$")
ax.legend(frameon=False)
ax.set_xlim(0.8, 1.01)
ax.set_ylim(0.0, 0.42)

plt.tight_layout()
plt.show()
```

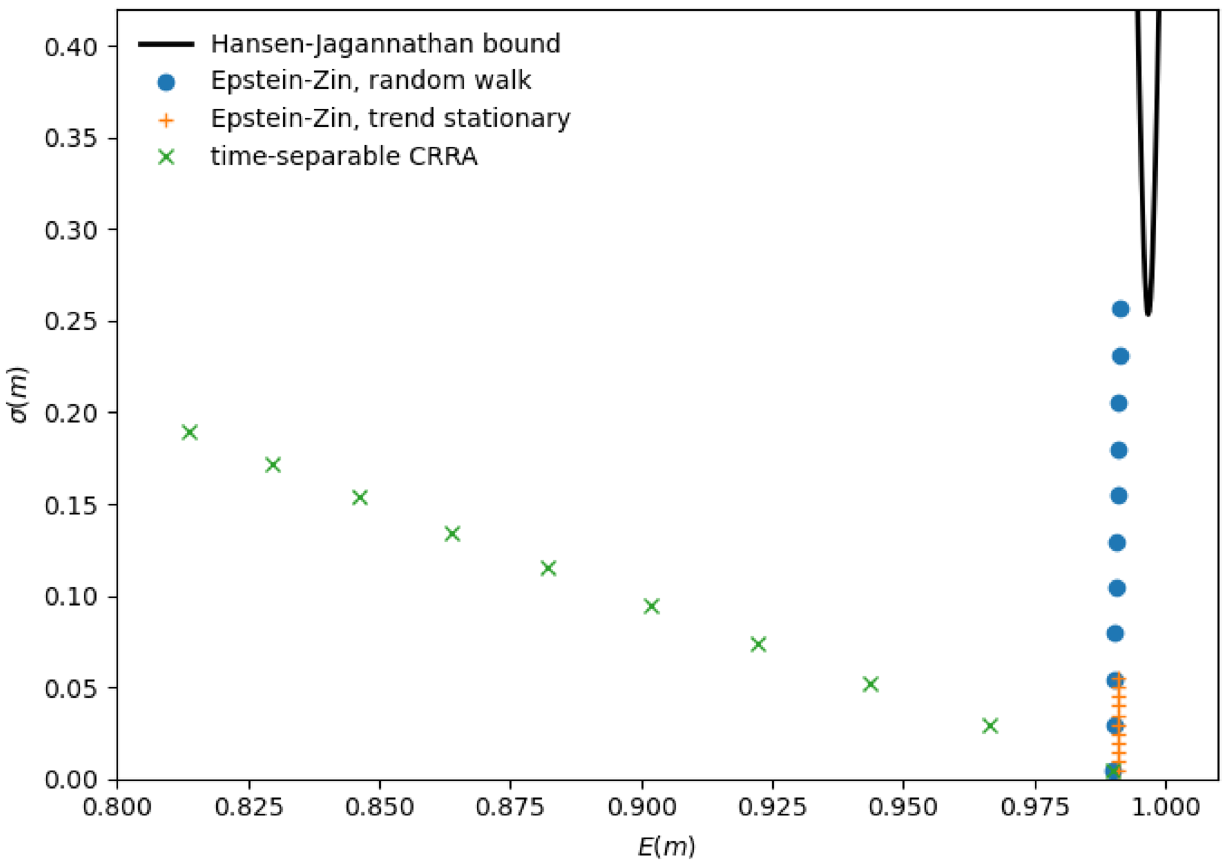


Fig. 85.1: SDF moments and Hansen-Jagannathan bound

The crosses tell the story of the risk-free-rate puzzle (Weil [1989]).

As γ rises, $\sigma(m)/E(m)$ grows but $E(m)$ drifts well below the range consistent with the observed risk-free rate.

The circles and pluses show Tallarini's way out.

Recursive utility with IES $= 1$ pushes volatility upward while keeping $E(m)$ roughly pinned near $1/(1+r^f)$.

For the random-walk model, the bound is reached at around $\gamma = 50$.

For the trend-stationary model, it is reached at around $\gamma = 75$.

The quantitative achievement is impressive, but Lucas's challenge still stands.

Where is the microeconomic evidence for $\gamma = 50$?

Barillas *et al.* [2009] argue that these large γ values are not really about risk aversion.

Instead, they reflect the agent's doubts about the probability model itself.

85.3 The choice setting

To understand their reinterpretation, we first need to describe their statistical models of consumption growth.

85.3.1 Shocks and consumption plans

We work with a general class of consumption plans.

Let x_t be an $n \times 1$ state vector and ε_{t+1} an $m \times 1$ shock.

A consumption plan belongs to the set $\mathcal{C}(A, B, H; x_0)$ if it admits the recursive representation

$$x_{t+1} = Ax_t + B\varepsilon_{t+1}, \qquad c_t = Hx_t, \tag{85.6}$$

where the eigenvalues of A are bounded in modulus by $1/\sqrt{\beta}$.

The time-t consumption can therefore be written as

$$c_t = H\left(B\varepsilon_t + AB\varepsilon_{t-1} + \cdots + A^{t-1}B\varepsilon_1\right) + HA^t x_0.$$

The equivalence theorems and Bellman equations that follow hold for arbitrary plans in $\mathcal{C}(A, B, H; x_0)$.

We focus on the random-walk and trend-stationary models as two special cases.

85.3.2 Consumption dynamics

Let $c_t = \log C_t$ be log consumption.

The *geometric-random-walk* specification is

$$c_{t+1} = c_t + \mu + \sigma_\varepsilon \varepsilon_{t+1}, \qquad \varepsilon_{t+1} \sim \mathcal{N}(0, 1).$$

Iterating forward yields

$$c_t = c_0 + t\mu + \sigma_\varepsilon(\varepsilon_t + \varepsilon_{t-1} + \cdots + \varepsilon_1), \qquad t \geq 1.$$

The *geometric-trend-stationary* specification can be written as a deterministic trend plus a stationary AR(1) component:

$$c_t = \zeta + \mu t + z_t, \qquad z_{t+1} = \rho z_t + \sigma_\varepsilon \varepsilon_{t+1}, \qquad \varepsilon_{t+1} \sim \mathcal{N}(0, 1).$$

With $z_0 = c_0 - \zeta$, this implies the representation

$$c_t = \rho^t c_0 + \mu t + (1 - \rho^t)\zeta + \sigma_\varepsilon(\varepsilon_t + \rho\varepsilon_{t-1} + \cdots + \rho^{t-1}\varepsilon_1), \qquad t \geq 1.$$

Equivalently, defining the detrended series $\tilde{c}_t := c_t - \mu t$,

$$\tilde{c}_{t+1} - \zeta = \rho(\tilde{c}_t - \zeta) + \sigma_\varepsilon \varepsilon_{t+1}.$$

The estimated parameters are $(\mu, \sigma_\varepsilon)$ for the random walk and $(\mu, \sigma_\varepsilon, \rho, \zeta)$ for the trend-stationary case.

We record these parameters and moments from the paper's tables for later reference.

```
print("Table 2 parameters")
print(f"random walk: μ={rw['μ']:.5f}, σ_ε={rw['σ_ε']:.5f}")
print(
    f"trend stationary: μ={ts['μ']:.5f}, σ_ε={ts['σ_ε']:.5f}, "
    f"ρ={ts['ρ']:.3f}, ζ={ts['ζ']:.2f}"
)
print()
print("Table 1 moments")
print(f"E[r_e]={r_e_mean:.4f}, std[r_e]={r_e_std:.4f}")
print(f"E[r_f]={r_f_mean:.4f}, std[r_f]={r_f_std:.4f}")
print(f"std[r_e-r_f]={r_excess_std:.4f}")
```

```
Table 2 parameters
random walk: μ=0.00495, σ_ε=0.00500
trend stationary: μ=0.00418, σ_ε=0.00500, ρ=0.980, ζ=-4.48

Table 1 moments
E[r_e]=0.0227, std[r_e]=0.0768
E[r_f]=0.0032, std[r_f]=0.0061
std[r_e-r_f]=0.0767
```

85.4 Preferences, distortions, and detection

85.4.1 Overview of agents I, II, III, and IV

We compare four preference specifications over consumption plans $C^\infty \in \mathcal{C}$.

Note

For origins of the names **multipler** and **constraint** preferences, see Hansen and Sargent [2001]. The risk-sensitive preference specification used here comes from Hansen and Sargent [1995], which adjusts specifications used earlier by Jacobson [1973], Whittle [1981], and Whittle [1990] to accommodate discounting in a way that preserves time-invariant optimal decision rules.

Type I agent (Kreps–Porteus–Epstein–Zin–Tallarini) with

- a discount factor $\beta \in (0, 1)$;
- an intertemporal elasticity of substitution fixed at 1;
- a risk-aversion parameter $\gamma \geq 1$; and
- an approximating conditional density $\pi(\cdot)$ for shocks and its implied joint distribution $\Pi_\infty(\cdot \mid x_0)$.

Type II agent (multiplier preferences) with

- $\beta \in (0, 1)$;
- IES $= 1$;
- unit risk aversion;
- an approximating model $\Pi_\infty(\cdot \mid x_0)$; and
- a penalty parameter $\theta > 0$ that discourages probability distortions using relative entropy.

Type III agent (constraint preferences) with

- $\beta \in (0, 1)$;
- IES $= 1$;
- unit risk aversion;
- an approximating model $\Pi_\infty(\cdot \mid x_0)$; and
- a bound η on discounted relative entropy.

Type IV agent (pessimistic ex post Bayesian) with

- $\beta \in (0, 1)$;
- IES $= 1$;
- unit risk aversion; and
- a single pessimistic joint distribution $\hat{\Pi}_\infty(\cdot \mid x_0, \theta)$ induced by the type II worst-case distortion.

Two sets of equivalence results tie these agents together.

Types I and II turn out to be observationally equivalent in a strong sense, having identical preferences over $\mathcal{C}$.

Types III and IV are equivalent in a weaker but still useful sense, delivering the same worst-case pricing implications as a type II agent for a given endowment process.

We now formalize each agent type and describe relationships among them.

For each type, we derive a Bellman equation that pins down the agent's value function and stochastic discount factor.

The stochastic discount factor for all four types takes the form

$$m_{t+1} = \beta \frac{\partial U_{t+1}/\partial c_{t+1}}{\partial U_t/\partial c_t} \hat{g}_{t+1},$$

where $\hat{g}_{t+1}$ is a likelihood-ratio distortion that we will define in each case.

Along the way, we introduce the likelihood-ratio distortion that enters the stochastic discount factor and describe detection-error probabilities that will serve as our new calibration tool.

85.4.2 Type I: Kreps–Porteus–Epstein–Zin–Tallarini preferences

The Epstein–Zin–Weil specification combines current consumption with a certainty equivalent of future utility through a CES aggregator:

$$V_t = \left[(1-\beta)C_t^\rho + \beta \mathcal{R}_t(V_{t+1})^\rho\right]^{1/\rho}, \qquad \rho := 1 - \frac{1}{\psi}, \tag{85.7}$$

where $\psi > 0$ is the intertemporal elasticity of substitution and the certainty equivalent uses the risk-aversion parameter $\gamma \geq 1$:

$$\mathcal{R}_t(V_{t+1}) = \left(E_t\left[V_{t+1}^{1-\gamma}\right]\right)^{\frac{1}{1-\gamma}}. \tag{85.8}$$

Note

For readers interested in a general class of aggregators and certainty equivalents, see Section 7.3 of Sargent and Stachurski [2025].

Let $\psi = 1$, so $\rho \to 0$.

In this limit the CES aggregator reduces to

$$V_t = C_t^{1-\beta} \cdot \mathcal{R}_t(V_{t+1})^{\beta}.$$

Taking logs and expanding the certainty equivalent (85.8) gives the *type I recursion*:

$$\log V_t = (1-\beta)c_t + \frac{\beta}{1-\gamma} \log E_t \left[(V_{t+1})^{1-\gamma} \right]. \tag{85.9}$$

A useful change of variables is to define the transformed continuation value

$$U_t \equiv \frac{\log V_t}{1-\beta} \tag{85.10}$$

and the robustness parameter

$$\theta = \frac{-1}{(1-\beta)(1-\gamma)}. \tag{85.11}$$

Substituting into (85.9) yields the *risk-sensitive recursion* (Exercise 3 asks you to verify this step)

$$U_t = c_t - \beta\theta \log E_t \left[\exp\left(\frac{-U_{t+1}}{\theta} \right) \right]. \tag{85.12}$$

When $\gamma = 1$ (equivalently $\theta = +\infty$), the $\log E \exp$ term reduces to $E_t U_{t+1}$ and the recursion becomes standard discounted expected log utility, $U_t = c_t + \beta E_t U_{t+1}$.

For consumption plans in $\mathcal{C}(A, B, H; x_0)$, the recursion (85.12) implies the Bellman equation

$$U(x) = c - \beta\theta \log \int \exp\left[\frac{-U(Ax + B\varepsilon)}{\theta} \right] \pi(\varepsilon) d\varepsilon. \tag{85.13}$$

Deriving the stochastic discount factor

The stochastic discount factor is the intertemporal marginal rate of substitution, the ratio of marginal utilities at dates $t+1$ and t.

Because c_t enters (85.12) linearly, $\partial U_t / \partial c_t = 1$.

Converting from log consumption to the consumption good gives $\partial U_t / \partial C_t = 1/C_t$.

A perturbation to c_{t+1} in a particular state feeds into U_t through the $\log E_t \exp$ term.

Differentiating (85.12):

$$\frac{\partial U_t}{\partial c_{t+1}} = -\beta\theta \frac{\exp(-U_{t+1}/\theta)(-1/\theta)}{E_t[\exp(-U_{t+1}/\theta)]} \underbrace{\frac{\partial U_{t+1}}{\partial c_{t+1}}}_{=1} = \beta \frac{\exp(-U_{t+1}/\theta)}{E_t[\exp(-U_{t+1}/\theta)]}.$$

Converting to consumption levels gives $\partial U_t / \partial C_{t+1} = \beta \frac{\exp(-U_{t+1}/\theta)}{E_t[\exp(-U_{t+1}/\theta)]} \frac{1}{C_{t+1}}$.

The ratio of these marginal utilities gives the SDF:

$$m_{t+1} = \frac{\partial U_t / \partial C_{t+1}}{\partial U_t / \partial C_t} = \beta \frac{C_t}{C_{t+1}} \frac{\exp(-U_{t+1}/\theta)}{E_t[\exp(-U_{t+1}/\theta)]}. \tag{85.14}$$

The second factor is the likelihood-ratio distortion $\hat{g}_{t+1}$: an exponential tilt that overweights states where the continuation value U_{t+1} is low.

85.4.3 Type II: multiplier preferences

We now turn to the type II (multiplier) agent.

Before writing down the preferences, we need the machinery of martingale likelihood ratios that formalizes what it means to distort a probability model.

These tools build on *Likelihood Ratio Processes*, which develops properties of likelihood ratios in detail, and *Divergence Measures*, which covers relative entropy.

Martingale likelihood ratios

Consider a nonnegative martingale G_t with $E(G_t \mid x_0) = 1$.

Its one-step increments

$$g_{t+1} = \frac{G_{t+1}}{G_t}, \qquad E_t[g_{t+1}] = 1, \quad g_{t+1} \geq 0, \qquad G_0 = 1,$$

define distorted conditional expectations: $\tilde{E}_t[b_{t+1}] = E_t[g_{t+1} b_{t+1}]$.

The conditional relative entropy of the distortion is $E_t[g_{t+1} \log g_{t+1}]$, and the discounted entropy over the entire path is $\beta E[\sum_{t=0}^{\infty} \beta^t G_t E_t(g_{t+1} \log g_{t+1}) | x_0]$.

A type II agent's *multiplier* preference ordering over consumption plans $C^\infty \in \mathcal{C}(A, B, H; x_0)$ is defined by

$$\min_{\{g_{t+1}\}} \sum_{t=0}^{\infty} E \left\{ \beta^t G_t \left[c_t + \beta\theta E_t \left(g_{t+1} \log g_{t+1} \right) \right] \Big| x_0 \right\}, \tag{85.15}$$

where $G_{t+1} = g_{t+1} G_t$, $E_t[g_{t+1}] = 1$, $g_{t+1} \geq 0$, and $G_0 = 1$.

A larger θ makes probability distortions more expensive, discouraging departures from the approximating model.

The value function satisfies the Bellman equation

$$W(x) = c + \min_{g(\varepsilon) \geq 0} \beta \int \left[g(\varepsilon) W(Ax + B\varepsilon) + \theta g(\varepsilon) \log g(\varepsilon) \right] \pi(\varepsilon) d\varepsilon \tag{85.16}$$

subject to $\int g(\varepsilon)\pi(\varepsilon) d\varepsilon = 1$.

Inside the integral, $g(\varepsilon) W(Ax + B\varepsilon)$ is the continuation value under the distorted model $g\pi$, while $\theta g(\varepsilon) \log g(\varepsilon)$ is the entropy penalty that makes large departures from the approximating model π costly.

The minimizer is (Exercise 4 derives this and verifies the equivalence $W \equiv U$)

$$\hat{g}_{t+1} = \frac{\exp(-W(Ax_t + B\varepsilon_{t+1})/\theta)}{E_t \left[\exp(-W(Ax_t + B\varepsilon_{t+1})/\theta) \right]}. \tag{85.17}$$

Notice that $g(\varepsilon)$ multiplies both the continuation value W and the entropy penalty.

This is the key structural feature that makes $\hat{g}$ a likelihood ratio.

Substituting (85.17) back into (85.16) gives

$$W(x) = c - \beta\theta \log \int \exp \left[\frac{-W(Ax + B\varepsilon)}{\theta} \right] \pi(\varepsilon) d\varepsilon,$$

which is identical to (85.13).

Therefore $W(x) \equiv U(x)$, establishing that *types I and II are observationally equivalent* over elements of $\mathcal{C}(A, B, H; x_0)$.

The mapping between parameters is

$$\theta = \left[(1 - \beta)(\gamma - 1) \right]^{-1}.$$

```python
def θ_from_γ(γ, β=β):
    if γ <= 1:
        return np.inf
    return 1.0 / ((1.0 - β) * (γ - 1.0))


def γ_from_θ(θ, β=β):
    if np.isinf(θ):
        return 1.0
    return 1.0 + 1.0 / ((1.0 - β) * θ)
```

85.4.4 Type III: constraint preferences

Type III (constraint) preferences swap the entropy penalty for a hard bound.

Rather than penalizing distortions through θ, the agent minimizes expected discounted log consumption under the worst-case model subject to a cap η on discounted relative entropy:

$$J(x_0) = \min_{\{g_{t+1}\}} \sum_{t=0}^{\infty} E\left[\beta^t G_t c_t \middle| x_0\right]$$

subject to $G_{t+1} = g_{t+1} G_t$, $E_t[g_{t+1}] = 1$, $g_{t+1} \geq 0$, $G_0 = 1$, and

$$\beta E\left[\sum_{t=0}^{\infty} \beta^t G_t E_t\left(g_{t+1} \log g_{t+1}\right) \middle| x_0\right] \leq \eta.$$

The Lagrangian for the type III problem is

$$\mathcal{L} = \sum_{t=0}^{\infty} E\left[\beta^t G_t c_t \middle| x_0\right] + \theta\left[\beta E\left(\sum_{t=0}^{\infty} \beta^t G_t E_t(g_{t+1} \log g_{t+1}) \middle| x_0\right) - \eta\right],$$

where $\theta \geq 0$ is the multiplier on the entropy constraint.

Collecting terms inside the expectation gives

$$\mathcal{L} = \sum_{t=0}^{\infty} E\left\{\beta^t G_t\left[c_t + \beta\theta E_t(g_{t+1} \log g_{t+1})\right] \middle| x_0\right\} - \theta\eta,$$

which, apart from the constant $-\theta\eta$, has the same structure as the type II objective (85.15).

The first-order condition for g_{t+1} is therefore identical, and the optimal distortion is the same $\hat{g}_{t+1}$ as in (85.17), evaluated at the θ that makes the entropy constraint bind.

The SDF is again $m_{t+1} = \beta(C_t/C_{t+1})\hat{g}_{t+1}$.

So for the particular endowment process and the θ that enforces the entropy bound, a type III agent and a type II agent assign the same shadow prices to uncertain claims.

85.4.5 Type IV: ex post Bayesian

The type IV agent is the simplest of the four: an ordinary expected-utility agent with log preferences who happens to hold a pessimistic probability model $\hat{\Pi}_\infty$:

$$\hat{E}_0 \sum_{t=0}^{\infty} \beta^t c_t.$$

$\hat{E}_0$ denotes expectation under the pessimistic model $\hat{\Pi}_\infty$.

Here $\hat{\Pi}_\infty(\cdot \mid x_0, \theta)$ is the joint distribution generated by the type II agent's worst-case distortion.

Since the agent has log utility under $\hat{\Pi}_\infty$, the Euler equation for any gross return R_{t+1} is

$$1 = \hat{E}_t \left[\beta \frac{C_t}{C_{t+1}} R_{t+1} \right].$$

To express this in terms of the approximating model Π_∞, apply a change of measure using the one-step likelihood ratio $\hat{g}_{t+1} = d\hat{\Pi}/d\Pi$:

$$1 = E_t \left[\hat{g}_{t+1} \cdot \beta \frac{C_t}{C_{t+1}} R_{t+1} \right] = E_t \left[m_{t+1} R_{t+1} \right],$$

so the effective SDF under the approximating model is $m_{t+1} = \beta (C_t / C_{t+1}) \hat{g}_{t+1}$.

For the particular A, B, H and θ used to construct $\hat{\Pi}_\infty$, the type IV value function equals $J(x)$ from type III.

85.4.6 Stochastic discount factor

Pulling together the results for all four agent types, the stochastic discount factor can be written compactly as

$$m_{t+1} = \beta \frac{C_t}{C_{t+1}} \hat{g}_{t+1}. \tag{85.18}$$

The factor $\hat{g}_{t+1}$ is a likelihood ratio between the approximating and worst-case one-step models.

With log utility, $C_t / C_{t+1} = \exp(-(c_{t+1} - c_t))$ is the usual intertemporal marginal rate of substitution.

Robustness multiplies it by $\hat{g}_{t+1}$, so uncertainty aversion enters pricing entirely through the distortion.

For the constraint-preference agent, the worst-case distortion coincides with the multiplier agent's at the θ that makes the entropy constraint bind.

For the ex post Bayesian, it is simply a change of measure from the approximating model to the pessimistic one.

85.4.7 Value function decomposition

Substituting the minimizing $\hat{g}$ back into the Bellman equation (85.16) yields a revealing decomposition of the type II value function:

$$W(x) = c + \beta \int [\hat{g}(\varepsilon) W(Ax + B\varepsilon) + \theta \hat{g}(\varepsilon) \log \hat{g}(\varepsilon)] \pi(\varepsilon) d\varepsilon. \tag{85.19}$$

Define two components:

$$J(x) = c + \beta \int \hat{g}(\varepsilon) J(Ax + B\varepsilon) \pi(\varepsilon) d\varepsilon, \tag{85.20}$$

$$N(x) = \beta \int \hat{g}(\varepsilon)[\log \hat{g}(\varepsilon) + N(Ax + B\varepsilon)]\pi(\varepsilon)d\varepsilon. \tag{85.21}$$

Then $W(x) = J(x) + \theta N(x)$.

Here $J(x_t) = \hat{E}_t \sum_{j=0}^{\infty} \beta^j c_{t+j}$ is expected discounted log consumption under the *worst-case* model.

J is the value function shared by both the type III and type IV agents.

For the type III agent, once the worst-case model is pinned down by the entropy constraint, the resulting value is simply expected discounted consumption under that model.

The type IV agent adopts the same model as a fixed belief, so she evaluates the same expectation.

The term $N(x)$ is discounted continuation entropy, measuring the total information cost of the probability distortion from date t onward.

This decomposition plays a central role in the welfare calculations of *the welfare section* below, where it explains why type III uncertainty compensation is twice that of type II.

85.4.8 Gaussian mean-shift distortions

Everything so far holds for general distortions $\hat{g}$.

We now specialize to the Gaussian case that underlies our two consumption models.

Under both models, the shock is $\varepsilon_{t+1} \sim \mathcal{N}(0, 1)$.

As we verify in the next subsection, the value function W is linear in the state, so the exponent in the worst-case distortion (85.17) is linear in ε_{t+1}.

Exponentially tilting a Gaussian by a linear function produces another Gaussian with the same variance but a shifted mean.

The worst-case model therefore keeps the variance at one but shifts the mean of ε_{t+1} to some $w < 0$.

The resulting likelihood ratio is (Exercise 5 verifies its properties)

$$\hat{g}_{t+1} = \exp\left(w\varepsilon_{t+1} - \frac{1}{2}w^2\right), \qquad E_t[\hat{g}_{t+1}] = 1. \tag{85.22}$$

Hence $\log \hat{g}_{t+1}$ is normal with mean $-w^2/2$ and variance w^2, and

$$\text{std}(\hat{g}_{t+1}) = \sqrt{e^{w^2} - 1}.$$

The mean shift w is determined by how strongly each shock ε_{t+1} affects continuation value.

From (85.17), the worst-case distortion puts $\hat{g} \propto \exp(-W(x_{t+1})/\theta)$.

If $W(x_{t+1})$ loads on ε_{t+1} with coefficient λ, then the Gaussian mean shift is $w = -\lambda/\theta$.

By guessing linear value functions and matching coefficients in the Bellman equation (Exercise 6 works out both cases), we obtain the worst-case mean shifts

$$w_{rw}(\theta) = -\frac{\sigma_\varepsilon}{(1-\beta)\theta}, \qquad w_{ts}(\theta) = -\frac{\sigma_\varepsilon}{(1-\rho\beta)\theta}. \tag{85.23}$$

The denominator $(1 - \beta)$ in the random-walk case becomes $(1 - \beta\rho)$ in the trend-stationary case.

Because the AR(1) component is persistent, each shock has a larger cumulative effect on continuation utility, so the worst-case distortion is more aggressive.

```
def w_from_θ(θ, model):
    if np.isinf(θ):
        return 0.0
    if model == "rw":
        return -rw["σ_ε"] / ((1.0 - β) * θ)
    if model == "ts":
        return -ts["σ_ε"] / ((1.0 - β * ts["ρ"]) * θ)
    raise ValueError("model must be 'rw' or 'ts'")
```

85.4.9 Discounted entropy

When the approximating and worst-case conditional densities are $\mathcal{N}(0,1)$ and $\mathcal{N}(w(\theta),1)$, the likelihood ratio is $\hat{g}(\varepsilon) = \exp(w(\theta)\varepsilon - \frac{1}{2}w(\theta)^2)$, so $\log\hat{g}(\varepsilon) = w(\theta)\varepsilon - \frac{1}{2}w(\theta)^2$.

Under the worst-case measure $\varepsilon \sim \mathcal{N}(w(\theta),1)$, so $E_{\hat{\pi}}[\varepsilon] = w(\theta)$, giving conditional relative entropy

$$E_t[\hat{g}_{t+1}\log\hat{g}_{t+1}] = w(\theta)\cdot w(\theta) - \frac{1}{2}w(\theta)^2 = \frac{1}{2}w(\theta)^2. \tag{85.24}$$

Because the distortion is i.i.d., the conditional entropy $E_t[\hat{g}_{t+1}\log\hat{g}_{t+1}] = \frac{1}{2}w(\theta)^2$ from (85.24) is constant and $N(x)$ does not depend on x.

The recursion (85.21) then reduces to $N(x) = \beta(\frac{1}{2}w(\theta)^2 + N(x))$, where we have used $\int\hat{g}(\varepsilon)\pi(\varepsilon)d\varepsilon = 1$ (since $\hat{g}$ is a likelihood ratio).

Solving for $N(x)$,

$$N(x)(1-\beta) = \frac{\beta}{2}w(\theta)^2,$$

gives discounted entropy

$$\eta = N(x) = \frac{\beta}{2(1-\beta)}w(\theta)^2. \tag{85.25}$$

```
def η_from_θ(θ, model):
    w = w_from_θ(θ, model)
    return β * w**2 / (2.0 * (1.0 - β))
```

This gives a clean mapping between θ and η that aligns multiplier and constraint preferences along an exogenous endowment process.

As we will see in the *detection-error section* below, it is more natural to hold η (or equivalently the detection-error probability p) fixed rather than θ when comparing across consumption models.

85.4.10 Value functions for random-walk consumption

We now solve the recursions (85.19), (85.20), and (85.21) in closed form for the random-walk model, where W is the type II (multiplier) value function, J is the type III/IV value function, and N is discounted continuation entropy.

Substituting $w_{rw}(\theta) = -\sigma_\varepsilon/[(1-\beta)\theta]$ from (85.23) into (85.25) gives

$$N(x) = \frac{\beta}{2(1-\beta)}w_{rw}(\theta)^2 = \frac{\beta}{2(1-\beta)}\left(\frac{-\sigma_\varepsilon}{(1-\beta)\theta}\right)^2 = \frac{\beta}{2(1-\beta)}\cdot\frac{\sigma_\varepsilon^2}{(1-\beta)^2\theta^2}$$

so that

$$N(x) = \frac{\beta\sigma_\varepsilon^2}{2(1-\beta)^3\theta^2}. \tag{85.26}$$

For W, we guess $W(x_t) = \frac{1}{1-\beta}[c_t + d]$ for some constant d and verify it in the risk-sensitive Bellman equation (85.13).

Under the random walk, $W(x_{t+1}) = \frac{1}{1-\beta}[c_t + \mu + \sigma_\varepsilon \varepsilon_{t+1} + d]$, so $-W(x_{t+1})/\theta$ is affine in the standard normal ε_{t+1}.

Using the fact that $\log E[e^Z] = \mu_Z + \frac{1}{2}\sigma_Z^2$ for a normal random variable Z, the Bellman equation (85.13) reduces to a constant-matching condition that pins down d (Exercise 7 works through the algebra):

$$W(x_t) = \frac{1}{1-\beta}\left[c_t + \frac{\beta}{1-\beta}\left(\mu - \frac{\sigma_\varepsilon^2}{2(1-\beta)\theta}\right)\right]. \tag{85.27}$$

Using $W = J + \theta N$, the type III/IV value function is

$$J(x_t) = W(x_t) - \theta N(x_t) = \frac{1}{1-\beta}\left[c_t + \frac{\beta}{1-\beta}\left(\mu - \frac{\sigma_\varepsilon^2}{(1-\beta)\theta}\right)\right]. \tag{85.28}$$

Notice that the coefficient on $\sigma_\varepsilon^2/[(1-\beta)\theta]$ doubles from $\frac{1}{2}$ in W to 1 in J.

The reason is that W includes the entropy "rebate" θN, which partially offsets the pessimistic tilt, while J evaluates consumption purely under the worst-case model with no such offset.

85.5 Detection-error probabilities

So far we have expressed SDF moments, value functions, and worst-case distortions as functions of γ (or equivalently θ).

But if γ should not be calibrated by introspection about atemporal gambles, what replaces it?

The answer proposed by Barillas *et al.* [2009] is a statistical test: how easily could an econometrician distinguish the approximating model from its worst-case alternative?

85.5.1 Likelihood-ratio testing and detection errors

Let L_T be the log likelihood ratio between the worst-case and approximating models based on a sample of length T.

Define

$$p_A = \Pr_A(L_T < 0), \qquad p_B = \Pr_B(L_T > 0),$$

where $\Pr_A$ and $\Pr_B$ denote probabilities under the approximating and worst-case models.

Then $p(\theta^{-1}) = \frac{1}{2}(p_A + p_B)$ is the average probability of choosing the wrong model.

Fix a sample size T (here 235 quarters, matching the postwar US data used in the paper).

For a given θ, compute the worst-case model and imagine that a Bayesian runs a likelihood-ratio test to distinguish it from the approximating model.

What fraction of the time would she pick the wrong one?

That fraction is the **detection-error probability** $p(\theta^{-1})$.

When p is close to 0.5 the two models are nearly indistinguishable, so the consumer's fear is hard to rule out.

When p is small the worst-case model is easy to reject and the robustness concern carries less force.

85.5.2 Market price of model uncertainty

The **market price of model uncertainty** (MPU) is the conditional standard deviation of the distortion:

$$\text{MPU} = \text{std}(\hat{g}_{t+1}) = \sqrt{e^{w(\theta)^2} - 1} \approx |w(\theta)|. \tag{85.29}$$

In the Gaussian mean-shift setting, L_T is normal with mean $\pm\frac{1}{2}w^2 T$ and variance $w^2 T$, so the detection-error probability has the closed form (Exercise 8 derives this)

$$p(\theta^{-1}) = \frac{1}{2}\left(p_A + p_B\right), \tag{85.30}$$

$$p(\theta^{-1}) = \Phi\left(-\frac{|w(\theta)|\sqrt{T}}{2}\right). \tag{85.31}$$

```python
def detection_probability(θ, model):
    w = abs(w_from_θ(θ, model))
    return norm.cdf(-0.5 * w * np.sqrt(T))


def θ_from_detection_probability(p, model):
    if p >= 0.5:
        return np.inf
    w_abs = -2.0 * norm.ppf(p) / np.sqrt(T)
    if model == "rw":
        return rw["σ_ε"] / ((1.0 - β) * w_abs)
    if model == "ts":
        return ts["σ_ε"] / ((1.0 - β * ts["ρ"]) * w_abs)
    raise ValueError("model must be 'rw' or 'ts'")
```

85.5.3 Interpreting the calibration objects

Let us trace the chain of mappings that connects preference parameters to statistical distinguishability.

The parameter θ governs how expensive it is for the minimizing player to distort the approximating model.

A small θ means cheap distortions and therefore stronger robustness concerns.

The associated $\gamma = 1 + \left[(1-\beta)\theta\right]^{-1}$ can be large even when we do not want to interpret behavior as extreme atemporal risk aversion.

The distortion magnitude $|w(\theta)|$ directly measures how pessimistically the agent tilts one-step probabilities.

The detection-error probability $p(\theta^{-1})$ translates that tilt into a statistical statement about finite-sample distinguishability.

High p means the two models are hard to tell apart, while low p means the worst case is easier to reject.

This chain bridges econometric identification and preference calibration.

Finally, recall from (85.25) that discounted entropy is $\eta = \frac{\beta}{2(1-\beta)} w(\theta)^2$, so when the distortion is a Gaussian mean shift, discounted entropy is proportional to the squared market price of model uncertainty.

85.5.4 Detection probabilities across the two models

The left panel below plots $p(\theta^{-1})$ against θ^{-1} for both consumption specifications.

Because the baseline dynamics differ, the same numerical θ implies very different detection probabilities across the two models.

The right panel resolves this by plotting detection probabilities against discounted relative entropy η, which normalizes the statistical distance.

Once indexed by η, the two curves fall on top of each other.

```
θ_inv_grid = np.linspace(0.0, 1.8, 400)
θ_grid = np.full_like(θ_inv_grid, np.inf)
mask_θ = θ_inv_grid > 0.0
θ_grid[mask_θ] = 1.0 / θ_inv_grid[mask_θ]

p_rw = np.array([detection_probability(θ, "rw") for θ in θ_grid])
p_ts = np.array([detection_probability(θ, "ts") for θ in θ_grid])

η_rw = np.array([η_from_θ(θ, "rw") for θ in θ_grid])
η_ts = np.array([η_from_θ(θ, "ts") for θ in θ_grid])

fig, axes = plt.subplots(1, 2, figsize=(12, 4))

axes[0].plot(θ_inv_grid, 100.0 * p_rw, lw=2, label="random walk")
axes[0].plot(θ_inv_grid, 100.0 * p_ts, lw=2, label="trend stationary")
axes[0].set_xlabel(r"$\theta^{-1}$")
axes[0].set_ylabel("detection error probability (percent)")
axes[0].legend(frameon=False)

axes[1].plot(η_rw, 100.0 * p_rw, lw=2, label="random walk")
axes[1].plot(η_ts, 100.0 * p_ts, lw=2, ls="--", label="trend stationary")
axes[1].set_xlabel(r"discounted entropy $\eta$")
axes[1].set_ylabel("detection error probability (percent)")
axes[1].set_xlim(0.0, 10)
axes[1].legend(frameon=False)

plt.tight_layout()
plt.show()
```

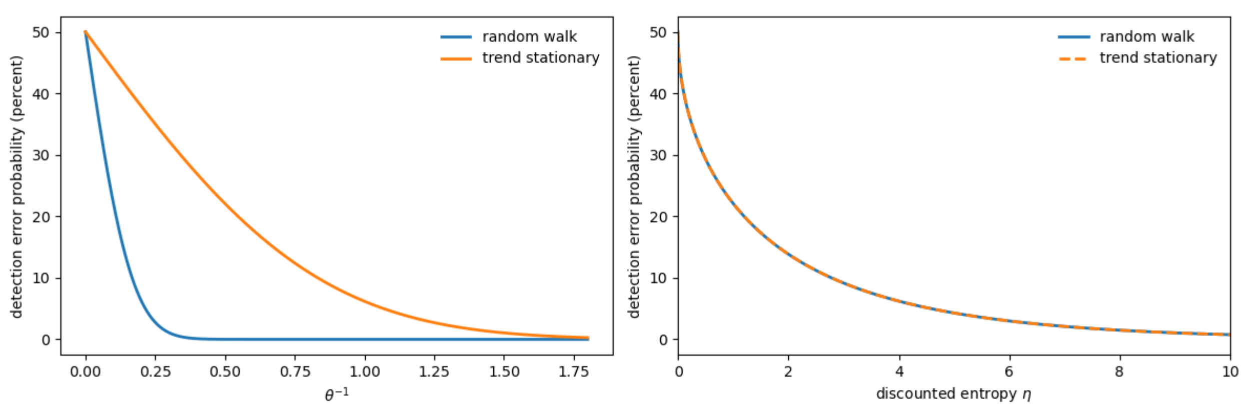


Fig. 85.2: Detection probabilities across two models

Detection-error probabilities (or equivalently, discounted entropy) are therefore the right yardstick for cross-model comparisons.

If we hold θ fixed when switching from a random walk to a trend-stationary specification, we implicitly change how much misspecification the consumer fears.

Holding η or p fixed instead keeps the statistical difficulty of detecting misspecification constant.

The explicit mapping that equates discounted entropy across models is (Exercise 9 derives it):

$$\theta_{\text{TS}} = \left(\frac{\sigma_\varepsilon^{\text{TS}}}{\sigma_\varepsilon^{\text{RW}}}\right) \frac{1-\beta}{1-\rho\beta} \theta_{\text{RW}}. \tag{85.32}$$

At our calibration $\sigma_\varepsilon^{\text{TS}} = \sigma_\varepsilon^{\text{RW}}$, this simplifies to $\theta_{\text{TS}} = \frac{1-\beta}{1-\rho\beta}\theta_{\text{RW}}$.

Because $\rho = 0.98$ and $\beta = 0.995$, the ratio $(1-\beta)/(1-\rho\beta)$ is much less than one, so holding entropy fixed requires a substantially smaller θ (stronger robustness) for the trend-stationary model than for the random walk.

85.6 Unify the two models using detection-error probabilities

With this machinery in hand, we can redraw Tallarini's figure using detection-error probabilities as the common index.

For each $p(\theta^{-1}) = 0.50, 0.45, \ldots, 0.01$, we invert to find the model-specific θ, convert to γ, and plot the implied $(E(m), \sigma(m))$ pair.

```python
p_points = np.array(
    [0.50, 0.45, 0.40, 0.35, 0.30, 0.25, 0.20, 0.15, 0.10, 0.05, 0.01])

θ_rw_points = np.array(
    [θ_from_detection_probability(p, "rw") for p in p_points])
θ_ts_points = np.array(
    [θ_from_detection_probability(p, "ts") for p in p_points])

γ_rw_points = np.array([γ_from_θ(θ) for θ in θ_rw_points])
γ_ts_points = np.array([γ_from_θ(θ) for θ in θ_ts_points])

Em_rw_p = np.array(
    [moments_type1_rw(γ)[0] for γ in γ_rw_points])
σ_m_rw_p = np.array(
    [moments_type1_rw(γ)[0] * moments_type1_rw(γ)[1] for γ in γ_rw_points])
Em_ts_p = np.array(
    [moments_type1_ts(γ)[0] for γ in γ_ts_points])
σ_m_ts_p = np.array(
    [moments_type1_ts(γ)[0] * moments_type1_ts(γ)[1] for γ in γ_ts_points])

print("p      γ_rw      γ_ts")
for p, g1, g2 in zip(p_points, γ_rw_points, γ_ts_points):
    print(f"{p:>4.2f} {g1:>9.2f} {g2:>9.2f}")
```

```
p      γ_rw      γ_ts
0.50      1.00      1.00
0.45      4.28     17.33
0.40      7.61     33.92
0.35     11.05     51.07
0.30     14.68     69.14
0.25     18.60     88.65
0.20     22.96    110.36
```

(continues on next page)

(continued from previous page)

```
0.15     28.04    135.68
0.10     34.44    167.53
0.05     43.92    214.74
0.01     61.70    303.29
```

```
# Empirical Sharpe ratio — the minimum of the HJ bound curve
sharpe = (r_e_mean - r_f_mean) / r_excess_std

def sharpe_gap(p, model):
    """Market price of risk minus Sharpe ratio, as a function of p."""
    if p >= 0.5:
        return -sharpe
    θ = θ_from_detection_probability(p, model)
    γ = γ_from_θ(θ)
    _, mpr = moments_type1_rw(γ) if model == "rw" else moments_type1_ts(γ)
    return mpr - sharpe

p_hj_rw = brentq(sharpe_gap, 1e-4, 0.49, args=("rw",))
p_hj_ts = brentq(sharpe_gap, 1e-4, 0.49, args=("ts",))

fig, ax = plt.subplots(figsize=(7, 5))
ax.plot(Em_rw_p, σ_m_rw_p, "o",
            label="random walk")
ax.plot(Em_ts_p, σ_m_ts_p, "+", markersize=12,
            label="trend stationary")
ax.plot(Em_grid, HJ_std, lw=2,
            color="black", label="Hansen-Jagannathan bound")

# Mark p where each model's market price of risk reaches the Sharpe ratio
for p_hj, model, color, name, marker in [
    (p_hj_rw, "rw", "C0", "RW", "o"),
    (p_hj_ts, "ts", "C1", "TS", "+"),
]:
    θ_hj = θ_from_detection_probability(p_hj, model)
    γ_hj = γ_from_θ(θ_hj)
    Em_hj, mpr_hj = (moments_type1_rw(γ_hj) if model == "rw"
                     else moments_type1_ts(γ_hj))
    σ_m_hj = Em_hj * mpr_hj
    ax.axhline(σ_m_hj, ls="--", lw=1, color=color,
               label=f"{name} reaches bound at $p = {p_hj:.3f}$")
    if model == "ts":
        ax.plot(Em_hj, σ_m_hj, marker, markersize=12, color=color)
    else:
        ax.plot(Em_hj, σ_m_hj, marker, color=color)

ax.set_xlabel(r"$E(m)$")
ax.set_ylabel(r"$\sigma(m)$")
ax.legend(frameon=False)
ax.set_xlim(0.96, 1.05)
ax.set_ylim(0.0, 0.34)

plt.tight_layout()
plt.show()
```

The result is striking.

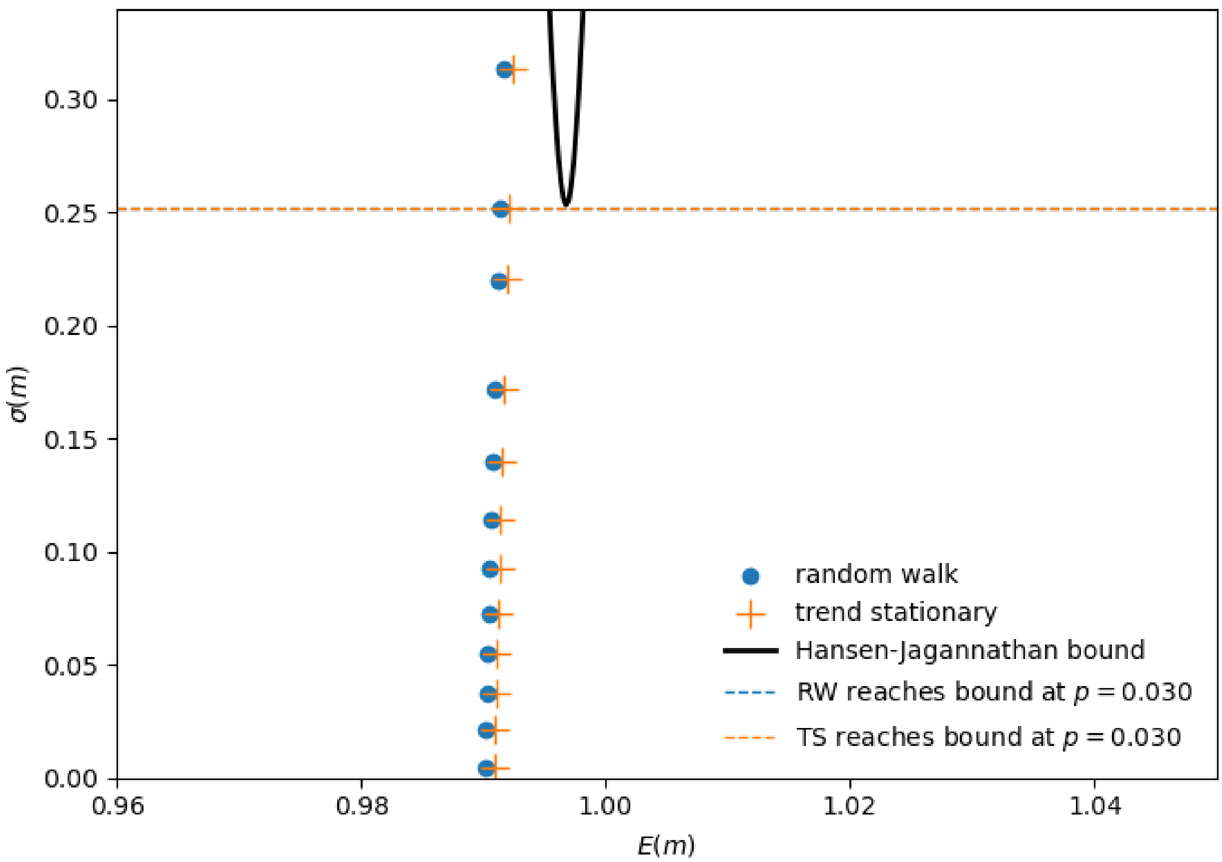


Fig. 85.3: Pricing loci from common detectability

The random-walk and trend-stationary loci nearly coincide.

Recall that under Tallarini's γ-calibration, reaching the Hansen–Jagannathan bound required $\gamma \approx 50$ for the random walk but $\gamma \approx 75$ for the trend-stationary model.

These are very different numbers for what is supposed to be the "same" preference parameter.

Under detection-error calibration, both models reach the bound at essentially the same detectability level.

The apparent model dependence was an artifact of using γ as the cross-model yardstick.

Once we measure robustness concerns in units of statistical detectability, the two consumption specifications tell a single, coherent story.

A representative consumer with moderate, difficult-to-dismiss fears about model misspecification behaves as though she has very high risk aversion.

The following figure brings together the two key ideas of this section: a small one-step density shift that is hard to detect (left panel) compounds into a large gap in expected log consumption (right panel).

At $p = 0.03$ both models share the same innovation mean shift w, and the left panel shows that the approximating and worst-case one-step densities nearly coincide.

The right panel reveals the cumulative consequence: a per-period shift that is virtually undetectable compounds into a large gap in expected log consumption, especially under random-walk dynamics where each shock has a permanent effect.

```
p_star = 0.03
θ_star = θ_from_detection_probability(p_star, "rw")
w_star = w_from_θ(θ_star, "rw")
σ_ε = rw["σ_ε"]
ρ = ts["ρ"]

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(13, 5))

ε = np.linspace(-4.5, 4.5, 500)
f0 = norm.pdf(ε, 0, 1)
fw = norm.pdf(ε, w_star, 1)

ax1.fill_between(ε, f0, alpha=0.15, color='k')
ax1.plot(ε, f0, 'k', lw=2.5,
         label=r'Approximating $\mathcal{N}(0, 1)$')
ax1.fill_between(ε, fw, alpha=0.15, color='C3')
ax1.plot(ε, fw, 'C3', lw=2, ls='--',
         label=fr'Worst case $\mathcal{{N}}({w_star:.2f},1)$')

peak = norm.pdf(0, 0, 1)
ax1.annotate('', xy=(w_star, 0.55 * peak), xytext=(0, 0.55 * peak),
             arrowprops=dict(arrowstyle='->', color='C3', lw=1.8))
ax1.text(w_star / 2, 0.59 * peak, f'$w = {w_star:.2f}$',
         ha='center', fontsize=11, color='C3')

ax1.set_xlabel(r'$\varepsilon_{t+1}$')
ax1.set_ylabel('Density')
ax1.legend(frameon=False)

quarters = np.arange(0, 241)
years = quarters / 4

gap_rw = 100 * σ_ε * w_star * quarters
gap_ts = 100 * σ_ε * w_star * (1 - ρ**quarters) / (1 - ρ)
```

(continues on next page)

(continued from previous page)

```
ax2.plot(years, gap_rw, 'C0', lw=2.5, label='Random walk')
ax2.plot(years, gap_ts, 'C1', lw=2.5, label='Trend stationary')
ax2.fill_between(years, gap_rw, alpha=0.1, color='C0')
ax2.fill_between(years, gap_ts, alpha=0.1, color='C1')
ax2.axhline(0, color='k', lw=0.5, alpha=0.3)

# Endpoint labels
ax2.text(61, gap_rw[-1], f'{gap_rw[-1]:.1f}%',
         fontsize=10, color='C0', va='center')
ax2.text(61, gap_ts[-1], f'{gap_ts[-1]:.1f}%',
         fontsize=10, color='C1', va='center')

ax2.set_xlabel('Years')
ax2.set_ylabel('Gap in expected log consumption (%)')
ax2.legend(frameon=False, loc='lower left')
ax2.set_xlim(0, 68)

plt.tight_layout()
plt.show()
```

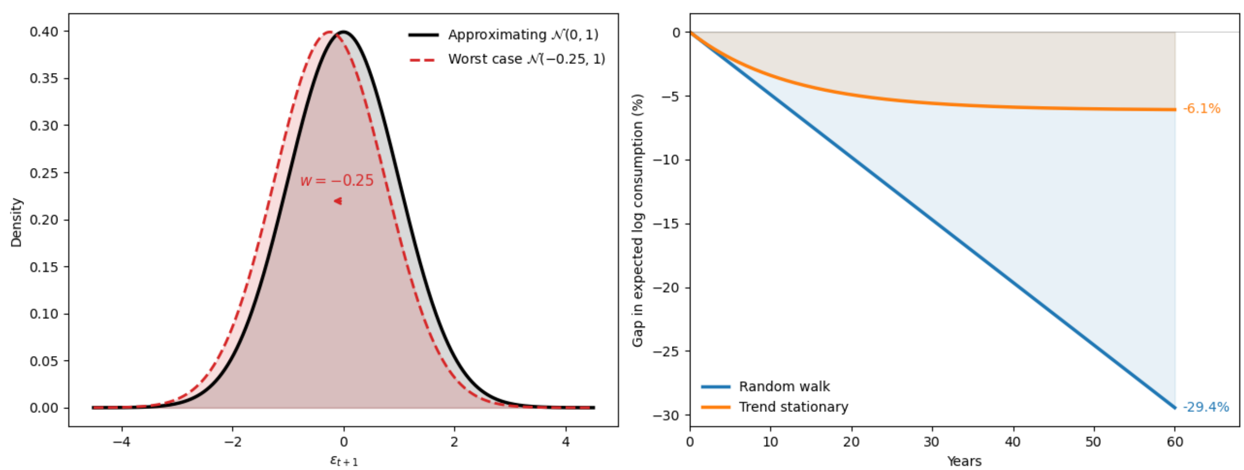


Fig. 85.4: Small one-step density shift (left) produces large cumulative consumption gap (right) at detection-error probability $p = 0.03$ with $T = 240$ quarters

The next figure poses the “doubts or variability?” question by decomposing the log SDF into two additive components.

Taking logs of (85.18) gives

$$\log m_{t+1} = \underbrace{\log \beta - \Delta c_{t+1}}_{\text{log-utility intertemporal MRS}} + \underbrace{\log \hat{g}_{t+1}}_{\text{worst-case distortion}} .$$

Under the random-walk model, $\Delta c_{t+1} = \mu + \sigma_\varepsilon \varepsilon_{t+1}$, and the Gaussian distortion (85.22) gives $\log \hat{g}_{t+1} = w\varepsilon_{t+1} - \frac{1}{2}w^2$. Substituting, we can write

$$\log m_{t+1} = (\log \beta - \mu - \tfrac{1}{2}w^2) - (\sigma_\varepsilon - w)\varepsilon_{t+1},$$

so the slope of $\log m_{t+1}$ in ε_{t+1} is $\sigma_\varepsilon - w$.

Since $w < 0$, the distortion steepens the SDF relative to what log utility alone would deliver.

The figure below reveals how little work log utility does on its own.

The intertemporal marginal rate of substitution (IMRS) is nearly flat.

At postwar calibrated volatility ($\sigma_\varepsilon = 0.005$), it contributes almost nothing to the pricing kernel's slope.

The worst-case distortion accounts for virtually all of the SDF's volatility.

What looks like extreme risk aversion ($\gamma \approx 34$) is really just log utility combined with moderate fears of model misspecification.

```
θ_cal = θ_from_detection_probability(0.10, "rw")
γ_cal = γ_from_θ(θ_cal)
w_cal = w_from_θ(θ_cal, "rw")

μ_c, σ_c = rw["μ"], rw["σ_ε"]
Δc = np.linspace(μ_c - 3.5 * σ_c, μ_c + 3.5 * σ_c, 300)
ε = (Δc - μ_c) / σ_c

log_imrs = np.log(β) - Δc
log_ghat = w_cal * ε - 0.5 * w_cal**2
log_sdf = log_imrs + log_ghat

fig, ax = plt.subplots(figsize=(8, 5))

ax.plot(100 * Δc, log_imrs, 'C1', lw=2,
        label=r'IMRS: $\log\beta - \Delta c$')
ax.plot(100 * Δc, log_ghat, 'C3', lw=2, ls='--',
        label=r'Distortion: $\log\hat{g}$')
ax.plot(100 * Δc, log_sdf, 'k', lw=2,
        label=r'SDF: $\log m = \log\mathrm{IMRS} + \log\hat{g}$')
ax.axhline(0, color='k', lw=0.5, alpha=0.3)
ax.set_xlabel(r'Consumption growth $\Delta c_{t+1}$ (%)')
ax.set_ylabel('Log SDF component')
ax.legend(frameon=False, fontsize=10, loc='upper right')

plt.show()
```

85.7 What do risk premia measure?

Lucas [2003] asked how much consumption a representative consumer would sacrifice to eliminate aggregate fluctuations.

His answer rested on the assumption that the consumer knows the true data-generating process.

The robust reinterpretation opens up a second, quite different thought experiment.

Instead of eliminating all randomness, suppose we keep the randomness but remove the consumer's fear of model misspecification (set $\theta = \infty$).

How much would she pay for that relief?

To answer this, we seek a permanent proportional reduction $c_0 - c_0^k$ in initial log consumption that leaves an agent of type k indifferent between the original risky plan and a deterministic certainty-equivalent path.

Because utility is log and the consumption process is Gaussian, these compensations can be computed in closed form.

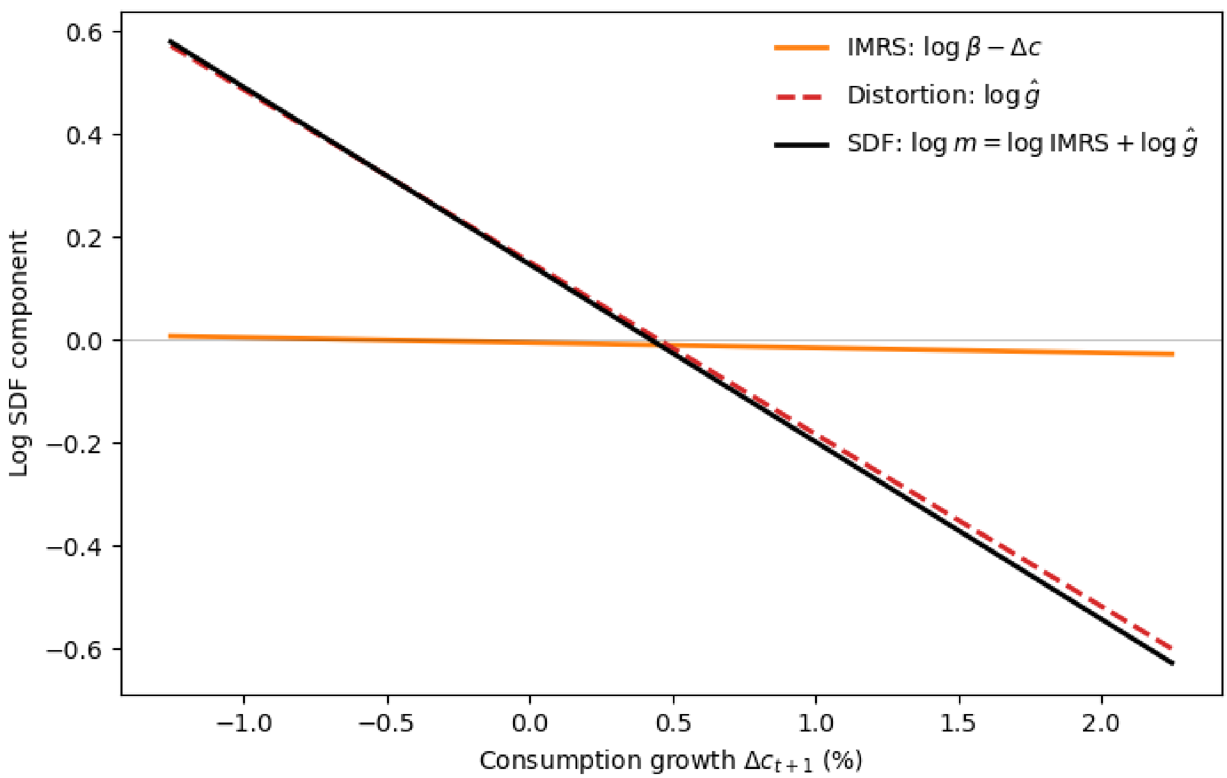


Fig. 85.5: Doubts or variability? Decomposition of the robust SDF into log-utility IMRS and worst-case distortion at $p = 0.10$

85.7.1 The certainty equivalent path

The point of comparison is the deterministic path with the same mean level of consumption as the stochastic plan:

$$c^{ce}_{t+1} - c^{ce}_t = \mu + \tfrac{1}{2}\sigma_\varepsilon^2. \tag{85.33}$$

The additional $\frac{1}{2}\sigma_\varepsilon^2$ term is a Jensen's inequality correction.

Since $E[C_t] = E[e^{c_t}] = \exp(c_0 + t\mu + \frac{1}{2}t\sigma_\varepsilon^2)$, (85.33) matches the mean *level* of consumption at every date.

85.7.2 Compensating variations from the value functions

We use the closed-form value functions derived earlier: (85.27) for the type I/II value function W and (85.28) for the type III/IV value function J.

For the certainty-equivalent path (85.33), there is no risk and no model uncertainty ($\theta = \infty$, so $\hat{g} = 1$), so the value function reduces to discounted expected log utility.

With $c^{ce}_t = c^J_0 + t(\mu + \frac{1}{2}\sigma_\varepsilon^2)$, we have

$$U^{ce}(c^J_0) = \sum_{t=0}^{\infty} \beta^t c^{ce}_t = \sum_{t=0}^{\infty} \beta^t [c^J_0 + t(\mu + \tfrac{1}{2}\sigma_\varepsilon^2)] = \frac{c^J_0}{1-\beta} + \frac{\beta(\mu + \frac{1}{2}\sigma_\varepsilon^2)}{(1-\beta)^2},$$

where we used $\sum_{t\geq 0}\beta^t = \frac{1}{1-\beta}$ and $\sum_{t\geq 0} t\beta^t = \frac{\beta}{(1-\beta)^2}$.

Factoring gives

$$U^{ce}(c^J_0) = \frac{1}{1-\beta}\left[c^J_0 + \frac{\beta}{1-\beta}\left(\mu + \tfrac{1}{2}\sigma_\varepsilon^2\right)\right].$$

85.7.3 Type I (Epstein–Zin) compensation

Setting $U^{ce}(c^I_0) = W(x_0)$ from (85.27):

$$\frac{1}{1-\beta}\left[c^I_0 + \frac{\beta}{1-\beta}\left(\mu + \tfrac{1}{2}\sigma_\varepsilon^2\right)\right] = \frac{1}{1-\beta}\left[c_0 + \frac{\beta}{1-\beta}\left(\mu - \frac{\sigma_\varepsilon^2}{2(1-\beta)\theta}\right)\right].$$

Multiplying both sides by $(1-\beta)$ and cancelling the common $\frac{\beta\mu}{1-\beta}$ terms gives

$$c^I_0 + \frac{\beta\sigma_\varepsilon^2}{2(1-\beta)} = c_0 - \frac{\beta\sigma_\varepsilon^2}{2(1-\beta)^2\theta}.$$

Solving for $c_0 - c^I_0$:

$$c_0 - c^I_0 = \frac{\beta\sigma_\varepsilon^2}{2(1-\beta)}\left(1 + \frac{1}{(1-\beta)\theta}\right) = \frac{\beta\sigma_\varepsilon^2\gamma}{2(1-\beta)}, \tag{85.34}$$

where the last step uses $\gamma = 1 + [(1-\beta)\theta]^{-1}$.

85.7.4 Type II (multiplier) decomposition

Because $W \equiv U$, we have $c_0^{II} = c_0^{I}$ and the total compensation is the same.

However, the interpretation differs because we can now decompose it into *risk* and *model uncertainty* components.

A type II agent with $\theta = \infty$ (no model uncertainty) has log preferences and requires

$$\Delta c_0^{risk} = \frac{\beta \sigma_\varepsilon^2}{2(1-\beta)}, \qquad \Delta c_0^{uncertainty} = \frac{\beta \sigma_\varepsilon^2}{2(1-\beta)^2 \theta}. \tag{85.35}$$

The risk term Δc_0^{risk} is Lucas's cost of business cycles.

At postwar consumption volatility ($\sigma_\varepsilon \approx 0.005$), it is negligibly small.

The uncertainty term $\Delta c_0^{uncertainty}$ captures the additional compensation a type II agent demands for facing model misspecification.

With θ in the denominator, this term can be first-order even when the detection-error probability is only moderate.

85.7.5 Type III (constraint) compensation

For a type III agent, we set $U^{ce}(c_0^{III}) = J(x_0)$ using the value function J from (85.28):

$$\frac{1}{1-\beta}\left[c_0^{III} + \frac{\beta}{1-\beta}\left(\mu + \tfrac{1}{2}\sigma_\varepsilon^2\right)\right] = \frac{1}{1-\beta}\left[c_0 + \frac{\beta}{1-\beta}\left(\mu - \frac{\sigma_\varepsilon^2}{(1-\beta)\theta}\right)\right].$$

Following the same algebra as for type I but with the doubled uncertainty correction in J:

$$c_0 - c_0^{III} = \frac{\beta \sigma_\varepsilon^2}{2(1-\beta)} + \frac{\beta \sigma_\varepsilon^2}{(1-\beta)^2 \theta}.$$

Using $\frac{1}{(1-\beta)\theta} = \gamma - 1$, this simplifies to

$$c_0 - c_0^{III} = \frac{\beta \sigma_\varepsilon^2}{2(1-\beta)}(2\gamma - 1). \tag{85.36}$$

The risk component is the same $\frac{\beta \sigma_\varepsilon^2}{2(1-\beta)}$ as before.

The uncertainty component alone is

$$c_0^{III}(r) - c_0^{III} = \frac{\beta \sigma_\varepsilon^2}{(1-\beta)^2 \theta},$$

which is *twice* the type II uncertainty compensation (85.35).

The factor of two traces back to the difference between W and J noted after (85.28).

The entropy rebate θN in $W = J + \theta N$ partially offsets the pessimistic tilt for the type II agent, but not for the type III agent who evaluates consumption purely under the worst-case model.

85.7.6 Type IV (ex post Bayesian) compensation

A type IV agent believes the pessimistic model, so the perceived drift is $\tilde{\mu} = \mu - \sigma_\varepsilon^2/[(1-\beta)\theta]$.

The compensation for moving to the certainty-equivalent path is the same as (85.36), because this agent ranks plans using the same value function J.

85.7.7 Comparison with a risky but free-of-model-uncertainty path

The certainty equivalents above compare a risky plan to a deterministic path, thereby eliminating both risk and uncertainty at once.

We now describe an alternative measure that isolates compensation for model uncertainty alone by keeping risk intact.

The idea is to compare two situations with identical risky consumption for all dates $t \geq 1$, concentrating all compensation for model uncertainty in a single adjustment to date-zero consumption.

Specifically, we seek $c_0^{II}(u)$ that makes a type II agent indifferent between:

1. Facing the stochastic plan under $\theta < \infty$ (fear of model misspecification), consuming c_0 at date zero.
2. Facing the *same* stochastic plan under $\theta = \infty$ (no fear of misspecification), but consuming only $c_0^{II}(u) < c_0$ at date zero.

In both cases, continuation consumptions c_t for $t \geq 1$ are generated by the random walk starting from the *same* c_0.

For the type II agent under $\theta < \infty$, the total value is $W(c_0)$ from (85.27).

For the agent liberated from model uncertainty ($\theta = \infty$), the value is

$$c_0^{II}(u) + \beta E\left[V^{\log}(c_1)\right],$$

where $V^{\log}(c_t) = \frac{1}{1-\beta}\left[c_t + \frac{\beta\mu}{1-\beta}\right]$ is the log-utility value function and $c_1 = c_0 + \mu + \sigma_\varepsilon \varepsilon_1$.

Since c_1 is built from c_0 (not $c_0^{II}(u)$), the continuation is

$$\beta E\left[V^{\log}(c_1)\right] = \frac{\beta}{1-\beta} E\left[c_1 + \frac{\beta\mu}{1-\beta}\right] = \frac{\beta}{1-\beta}\left[c_0 + \mu + \frac{\beta\mu}{1-\beta}\right] = \frac{\beta}{1-\beta}\left[c_0 + \frac{\mu}{1-\beta}\right],$$

where we used $E[c_1] = c_0 + \mu$ (the noise term has zero mean).

Expanding gives

$$\beta E\left[V^{\log}(c_1)\right] = \frac{\beta c_0}{1-\beta} + \frac{\beta\mu}{(1-\beta)^2}.$$

Setting $W(c_0)$ equal to the liberation value and simplifying:

$$\frac{c_0}{1-\beta} + \frac{\beta\mu}{(1-\beta)^2} - \frac{\beta\sigma_\varepsilon^2}{2(1-\beta)^3\theta} = c_0^{II}(u) + \frac{\beta c_0}{1-\beta} + \frac{\beta\mu}{(1-\beta)^2}.$$

Because $\frac{c_0}{1-\beta} - \frac{\beta c_0}{1-\beta} = c_0$, solving for the compensation gives

$$c_0 - c_0^{II}(u) = \frac{\beta\sigma_\varepsilon^2}{2(1-\beta)^3\theta} = \frac{\beta\sigma_\varepsilon^2(\gamma-1)}{2(1-\beta)^2}. \tag{85.37}$$

This is $\frac{1}{1-\beta}$ times the uncertainty compensation $\Delta c_0^{\text{uncertainty}}$ from (85.35).

The extra factor of $\frac{1}{1-\beta}$ arises because all compensation is packed into a single period.

Adjusting c_0 alone must offset the cumulative loss in continuation value that the uncertainty penalty imposes in every future period.

An analogous calculation for a **type III** agent, using $J(c_0)$ from (85.28), gives

$$c_0 - c_0^{III}(u) = \frac{\beta\sigma_\varepsilon^2}{(1-\beta)^3\theta} = \frac{\beta\sigma_\varepsilon^2(\gamma-1)}{(1-\beta)^2}, \tag{85.38}$$

which is $\frac{1}{1-\beta}$ times the type III uncertainty compensation and *twice* the type II compensation (85.37), again reflecting the absence of the entropy rebate in J.

85.7.8 Summary of welfare compensations (random walk)

The following table collects all compensating variations for the random walk model.

Agent	Compensation	Formula	Measures
I, II	$c_0 - c_0^{II}$	$\frac{\beta\sigma_\varepsilon^2\gamma}{2(1-\beta)}$	risk + uncertainty (vs. deterministic)
II	$c_0 - c_0^{II}(r)$	$\frac{\beta\sigma_\varepsilon^2}{2(1-\beta)}$	risk only (vs. deterministic)
II	$c_0^{II}(r) - c_0^{II}$	$\frac{\beta\sigma_\varepsilon^2}{2(1-\beta)^2\theta}$	uncertainty only (vs. deterministic)
II	$c_0 - c_0^{II}(u)$	$\frac{\beta\sigma_\varepsilon^2}{2(1-\beta)^3\theta}$	uncertainty only (vs. risky path)
III	$c_0 - c_0^{III}$	$\frac{\beta\sigma_\varepsilon^2(2\gamma-1)}{2(1-\beta)}$	risk + uncertainty (vs. deterministic)
III	$c_0^{III}(r) - c_0^{III}$	$\frac{\beta\sigma_\varepsilon^2}{(1-\beta)^2\theta}$	uncertainty only (vs. deterministic)
III	$c_0 - c_0^{III}(u)$	$\frac{\beta\sigma_\varepsilon^2}{(1-\beta)^3\theta}$	uncertainty only (vs. risky path)

The "vs. deterministic" rows use the certainty-equivalent path (85.33) as a benchmark.

The "vs. risky path" rows use the risky-but-uncertainty-free comparison of (85.37)–(85.38).

85.7.9 Trend-stationary formulas

For the trend-stationary model, the denominators $(1-\beta)$ in the uncertainty terms are replaced by $(1-\beta\rho)$, and the risk terms involve $(1-\beta\rho^2)$:

$$\Delta c_0^{risk,ts} = \frac{\beta\sigma_\varepsilon^2}{2(1-\beta\rho^2)}, \qquad \Delta c_0^{unc,ts,II} = \frac{\beta\sigma_\varepsilon^2}{2(1-\beta\rho)^2\theta}, \qquad \Delta c_0^{unc,ts,III} = \frac{\beta\sigma_\varepsilon^2}{(1-\beta\rho)^2\theta}. \tag{85.39}$$

The qualitative message carries over: the risk component is negligible, and the model-uncertainty component dominates.

85.8 Visualizing the welfare decomposition

We set $\beta = 0.995$ and calibrate θ so that $p(\theta^{-1}) = 0.10$, a conservative detection-error level.

```
p_star = 0.10
θ_star = θ_from_detection_probability(p_star, "rw")
γ_star = γ_from_θ(θ_star)
w_star = w_from_θ(θ_star, "rw")

# Type II compensations, random walk model
```

(continues on next page)

(continued from previous page)

```
comp_risk_only = β * rw["σ_ε"]**2 / (2.0 * (1.0 - β))
comp_risk_unc = comp_risk_only + β * rw["σ_ε"]**2 / (2.0 * (1.0 - β)**2 * θ_star)

# Two useful decompositions in levels
risk_only_pct = 100.0 * (np.exp(comp_risk_only) - 1.0)
risk_unc_pct = 100.0 * (np.exp(comp_risk_unc) - 1.0)
uncertainty_only_pct = 100.0 * (np.exp(comp_risk_unc - comp_risk_only) - 1.0)

print(f"p*={p_star:.2f}, θ*={θ_star:.4f}, γ*={γ_star:.2f}, w*={w_star:.4f}")
print(f"risk only compensation (log units): {comp_risk_only:.6f}")
print(f"risk + uncertainty compensation (log units): {comp_risk_unc:.6f}")
print(f"risk only compensation (percent): {risk_only_pct:.3f}%")
print(f"risk + uncertainty compensation (percent): {risk_unc_pct:.3f}%")
print(f"uncertainty component alone (percent): {uncertainty_only_pct:.3f}%")

h = 250
t = np.arange(h + 1)

# Baseline approximating model fan
mean_base = rw["μ"] * t
std_base = rw["σ_ε"] * np.sqrt(t)

# Certainty equivalent line from Eq. (47), shifted by compensating variations
certainty_slope = rw["μ"] + 0.5 * rw["σ_ε"]**2
ce_risk = -comp_risk_only + certainty_slope * t
ce_risk_unc = -comp_risk_unc + certainty_slope * t

# Alternative models from the ambiguity set in panel B
mean_low = (rw["μ"] + rw["σ_ε"] * w_star) * t
mean_high = (rw["μ"] - rw["σ_ε"] * w_star) * t
```

```
p*=0.10, θ*=5.9809, γ*=34.44, w*=-0.1672
risk only compensation (log units): 0.002487
risk + uncertainty compensation (log units): 0.085669
risk only compensation (percent): 0.249%
risk + uncertainty compensation (percent): 8.945%
uncertainty component alone (percent): 8.674%
```

```
fig, axes = plt.subplots(1, 2, figsize=(12, 4))

# Panel A
ax = axes[0]
ax.fill_between(t, mean_base - std_base, mean_base + std_base,
                alpha=0.25, color="tab:blue")
ax.plot(t, ce_risk_unc, lw=2, ls="--", color="black",
                label="certainty equivalent: risk + uncertainty")
ax.plot(t, ce_risk, lw=2, color="tab:orange",
                label="certainty equivalent: risk only")
ax.plot(t, mean_base, lw=2,
                color="tab:blue", label="approximating-model mean")
ax.set_xlabel("quarters")
ax.set_ylabel("log consumption")
ax.legend(frameon=False, fontsize=8, loc="upper left")

# Panel B
ax = axes[1]
```

(continues on next page)

(continued from previous page)

```
ax.fill_between(t, mean_base - std_base, mean_base + std_base,
                            alpha=0.20, color="tab:blue")
ax.fill_between(t, mean_low - std_base, mean_low + std_base,
                            alpha=0.20, color="tab:red")
ax.fill_between(t, mean_high - std_base, mean_high + std_base,
                            alpha=0.20, color="tab:green")
ax.plot(t, ce_risk_unc, lw=2, ls="--", color="black",
                        label="certainty equivalent: risk + uncertainty")
ax.plot(t, mean_base, lw=2, color="tab:blue", label="approximating-model mean")
ax.plot(t, mean_low, lw=2, color="tab:red", label="worst-case-leaning mean")
ax.plot(t, mean_high, lw=2, color="tab:green", label="best-case-leaning mean")
ax.set_xlabel("quarters")
ax.set_ylabel("log consumption")
ax.legend(frameon=False, fontsize=8, loc="upper left")

plt.tight_layout()
plt.show()
```

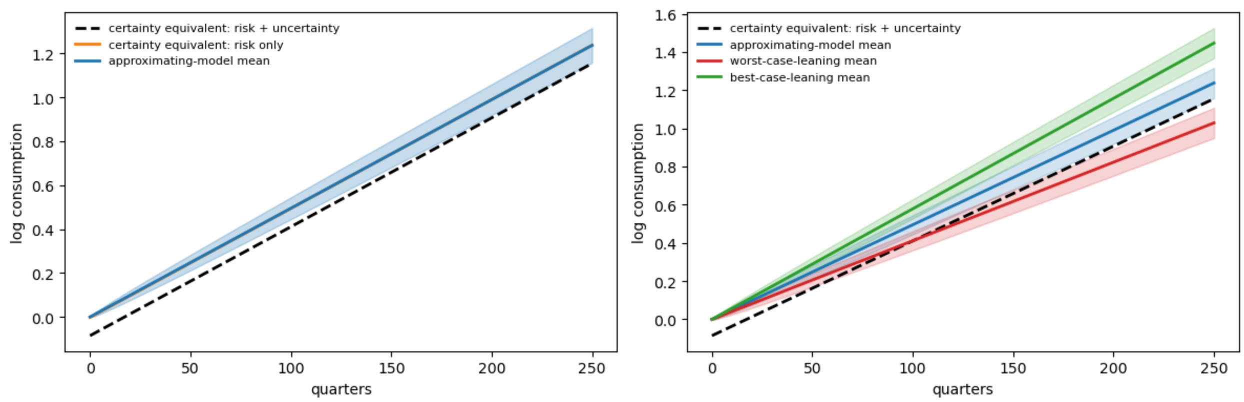


Fig. 85.6: Certainty equivalents under robustness

The left panel illustrates the elimination of model uncertainty and risk for a type II agent.

The shaded fan shows a one-standard-deviation band for the j-step-ahead conditional distribution of c_t under the calibrated random-walk model.

The dashed line c^{II} shows the certainty-equivalent path whose date-zero consumption is reduced by $c_0 - c_0^{II}$, making the type II agent indifferent between this deterministic trajectory and the stochastic plan.

It compensates for bearing both risk and model ambiguity.

The solid line c^r shows the certainty equivalent for a type II agent without model uncertainty ($\theta = \infty$), initialized at $c_0 - c_0^{II}(r)$.

At postwar calibrated values this gap is small, so c^r sits just below the center of the fan.

Consistent with Lucas [2003], the welfare gains from eliminating well-understood risk are very small.

The large welfare gains found by Tallarini [2000] can be reinterpreted as arising not from reducing risk, but from reducing model uncertainty.

The right panel shows the set of nearby models that the robust consumer guards against.

Each shaded fan depicts a one-standard-deviation band for a different model in the ambiguity set.

The models are statistically close to the baseline, with detection-error probability $p = 0.10$, but imply very different long-run consumption levels.

The consumer's caution against such alternatives accounts for the large certainty-equivalent gap in the left panel.

85.9 Welfare gains from removing model uncertainty

A type III (constraint-preference) agent evaluates the worst model inside an entropy ball of radius η.

As η grows the set of plausible misspecifications expands, and with it the welfare cost of confronting model uncertainty.

Since η itself is not easy to interpret, we instead index these costs by the associated detection-error probability $p(\eta)$.

The figure below plots the compensation for removing model uncertainty, measured as a proportion of consumption, against $p(\eta)$.

```
η_grid = np.linspace(0.0, 5.0, 300)

# Use w and η relation, then convert to θ model by model
w_abs_grid = np.sqrt(2.0 * (1.0 - β) * η_grid / β)

θ_rw_from_η = np.full_like(w_abs_grid, np.inf)
θ_ts_from_η = np.full_like(w_abs_grid, np.inf)
mask_w = w_abs_grid > 0.0
θ_rw_from_η[mask_w] = rw["σ_ε"] / ((1.0 - β) * w_abs_grid[mask_w])
θ_ts_from_η[mask_w] = ts["σ_ε"] / ((1.0 - β * ts["ρ"]) * w_abs_grid[mask_w])

# Type III uncertainty terms from Table 3
gain_rw = np.where(
    np.isinf(θ_rw_from_η),
    0.0,
    β * rw["σ_ε"]**2 / ((1.0 - β)**2 * θ_rw_from_η),
)
gain_ts = np.where(
    np.isinf(θ_ts_from_η),
    0.0,
    β * ts["σ_ε"]**2 / ((1.0 - β * ts["ρ"])**2 * θ_ts_from_η),
)

# Convert log compensation to percent of initial consumption in levels
gain_rw_pct = 100.0 * (np.exp(gain_rw) - 1.0)
gain_ts_pct = 100.0 * (np.exp(gain_ts) - 1.0)

# Detection error probabilities implied by η
p_eta_pct = 100.0 * norm.cdf(-0.5 * w_abs_grid * np.sqrt(T))
order = np.argsort(p_eta_pct)
p_plot = p_eta_pct[order]
gain_rw_plot = gain_rw_pct[order]
gain_ts_plot = gain_ts_pct[order]
```

```
fig, ax = plt.subplots(figsize=(7, 4))
ax.plot(p_plot, gain_rw_plot, lw=2, label="RW type III")
ax.plot(p_plot, gain_ts_plot, lw=2, label="TS type III")
ax.set_xlabel(r"detection error probability $p(\eta)$ (percent)")
ax.set_ylabel("proportion of consumption (percent)")
ax.legend(frameon=False)
```

(continues on next page)

(continued from previous page)

```
plt.tight_layout()
plt.show()
```

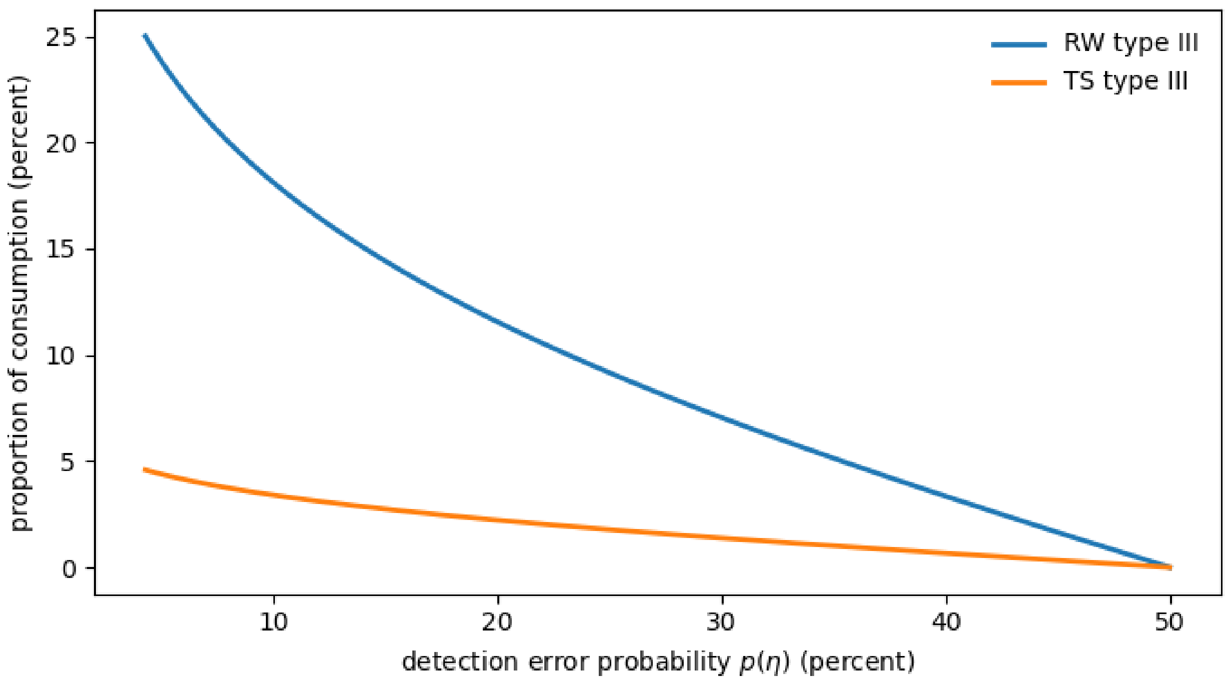


Fig. 85.7: Type III uncertainty compensation curve

The random-walk model implies somewhat larger costs than the trend-stationary model at the same detection-error probability, but both curves dwarf the classic Lucas cost of business cycles.

To put the magnitudes in perspective, Lucas estimated that eliminating all aggregate consumption risk is worth roughly 0.05% of consumption.

At detection-error probabilities of 10–20%, the model-uncertainty compensation alone runs to several percent, orders of magnitude larger.

Under the robust reading, the large risk premia that Tallarini matched with high γ are really compensations for bearing model uncertainty, and the implied welfare gains from resolving that uncertainty are correspondingly large.

The following contour plot shows how type II (multiplier) compensation varies over two dimensions: the detection-error probability p and the consumption volatility σ_ε.

The cross marks the calibrated point ($p = 0.10$, $\sigma_\varepsilon = 0.5\%$).

At the calibrated volatility, moving left (lower p, stronger robustness concerns) increases compensation dramatically, while the classic risk-only cost (the $p = 50\%$ edge) remains negligible.

A comparison of the two panels reveals that the random-walk model generates much larger welfare costs than the trend-stationary model at the same (p, σ_ε), because permanent shocks compound the worst-case drift indefinitely.

```
p_grid = np.linspace(0.02, 0.49, 300)
σ_grid = np.linspace(0.001, 0.015, 300)
P, Σ = np.meshgrid(p_grid, σ_grid)
```

(continues on next page)

(continued from previous page)

```
W_abs = -2 * norm.ppf(P) / np.sqrt(T)

# RW: total type II = βσ²γ / [2(1-β)]
Γ_rw = 1 + W_abs / Σ
comp_rw = 100 * (np.exp(β * Σ**2 * Γ_rw / (2 * (1 - β))) - 1)

# TS: risk + uncertainty
ρ_val = ts["ρ"]
risk_ts = β * Σ**2 / (2 * (1 - β * ρ_val**2))
unc_ts = β * Σ * W_abs / (2 * (1 - β * ρ_val))
comp_ts = 100 * (np.exp(risk_ts + unc_ts) - 1)

levels = [0.1, 0.2, 0.5, 1, 2, 5, 10, 20, 50]

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(13, 5.5), sharey=True)

for ax, comp, title in [(ax1, comp_rw, 'Random walk'),
                         (ax2, comp_ts, 'Trend stationary')]:
    cf = ax.contourf(100 * P, 100 * Σ, comp, levels=levels,
                     cmap='Blues', extend='both')
    cs = ax.contour(100 * P, 100 * Σ, comp, levels=levels,
                    colors='k', linewidths=0.5)
    ax.clabel(cs, fmt='%g%%', fontsize=8)
    ax.plot(10, 0.5, 'x', markersize=14, color='w',
            mec='k', mew=1, zorder=5)
    ax.set_xlabel(r'Detection-error probability $p$ (%)')
    ax.set_title(title)

ax1.set_ylabel(r'Consumption volatility $\sigma_\varepsilon$ (%)')

plt.tight_layout()
plt.show()
```

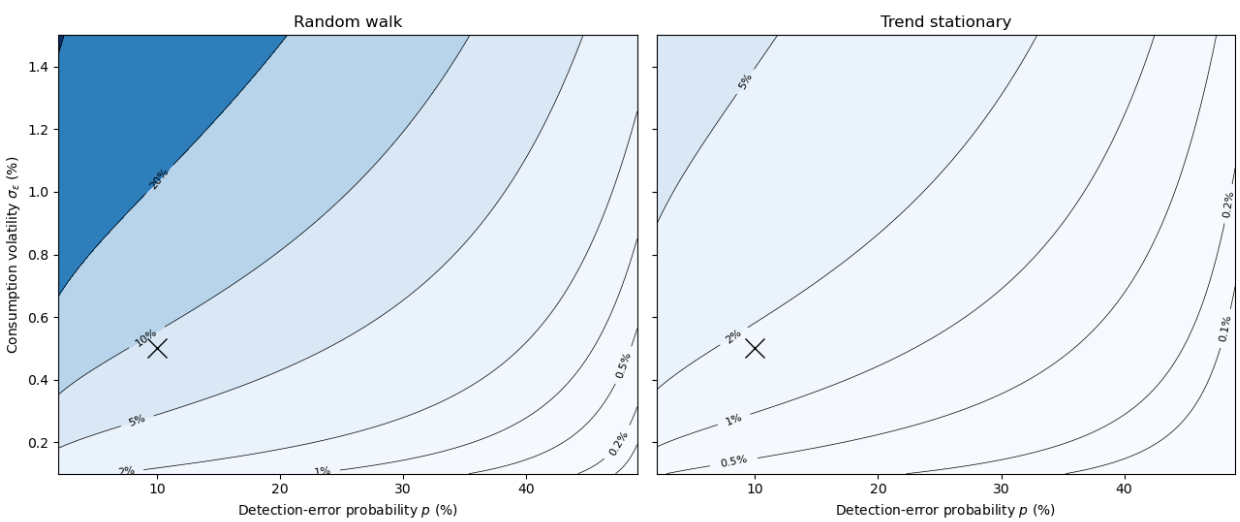


Fig. 85.8: Type II compensation across detection-error probability and consumption volatility

85.10 Learning doesn't eliminate misspecification fears

A reasonable question arises: if the consumer has 235 quarters of data, can't she learn enough to dismiss the worst-case model?

The answer is no.

This is because the drift is a low-frequency feature that is very hard to pin down.

Estimating the mean of a random walk to the precision needed to reject small but economically meaningful shifts requires far more data than estimating volatility precisely does.

The following figure makes this point concrete.

We measure consumption as real personal consumption expenditures on nondurable goods and services, deflated by its implicit chain price deflator and expressed in per-capita terms using the civilian noninstitutional population aged 16+.

The construction uses four FRED series:

FRED series	Description
`PCND`	Nominal PCE: nondurable goods (billions of \$, SAAR, quarterly)
`PCESV`	Nominal PCE: services (billions of \$, SAAR, quarterly)
`DPCERD3Q086SBEA`	PCE implicit price deflator (index 2017 = 100, quarterly)
`CNP16OV`	Civilian noninstitutional population, 16+ (thousands, monthly)

We use nominal rather than chained-dollar components because chained-dollar series are not additive.

Chain-weighted indices update their base-period expenditure weights every period, so components deflated with different price changes do not sum to the separately chained aggregate.

Adding nominal series and deflating the sum with a single price index avoids this problem.

The processing pipeline is:

1. Add nominal nondurables and services: $C_t^{nom} = C_t^{nd} + C_t^{sv}$.
2. Deflate by the PCE price index: $C_t^{real} = C_t^{nom}/(P_t/100)$.
3. Convert to per-capita: divide by the quarterly average of the monthly population series.
4. Compute log consumption: $c_t = \log C_t^{real,pc}$.

When we plot *levels* of log consumption, we align the time index to 1948Q1–2006Q4, which yields $T+1 = 236$ quarterly observations.

```
start_date = dt.datetime(1947, 1, 1)
end_date = dt.datetime(2007, 1, 1)


def _read_fred_series(series_id, start_date, end_date):
    series = web.DataReader(series_id, "fred", start_date, end_date)[series_id]
    series = pd.to_numeric(series, errors="coerce").dropna().sort_index()
    if series.empty:
        raise ValueError(f"FRED series '{series_id}' returned no data in sample window
↪")
    return series


# Fetch nominal PCE components, deflator, and population from FRED
```

(continues on next page)

(continued from previous page)

```
nom_nd = _read_fred_series("PCND", start_date, end_date)         # quarterly, 1947-
nom_sv = _read_fred_series("PCESV", start_date, end_date)        # quarterly, 1947-
defl = _read_fred_series("DPCERD3Q086SBEA", start_date, end_date)  # quarterly, 1947-
pop_m = _read_fred_series("CNP16OV", start_date, end_date)       # monthly, 1948-

# Step 1: add nominal nondurables + services
nom_total = nom_nd + nom_sv

# Step 2: deflate by PCE implicit price deflator (index 2017=100)
real_total = nom_total / (defl / 100.0)

# Step 3: convert to per-capita (population is monthly, so average to quarterly)
pop_q = pop_m.resample("QS").mean()
real_pc = (real_total / pop_q).dropna()

# Restrict to sample period 1948Q1-2006Q4
real_pc = real_pc.loc["1948-01-01":"2006-12-31"].dropna()

if real_pc.empty:
    raise RuntimeError(
        "FRED returned no usable observations after alignment/filtering")

# Step 4: log consumption
log_c_data = np.log(real_pc.to_numpy(dtype=float).reshape(-1))
years_data = (
    real_pc.index.year
    + (real_pc.index.month - 1) / 12.0).to_numpy(dtype=float)

print(f"Fetched {len(log_c_data)} quarterly observations from FRED")
print(f"Sample: {years_data[0]:.1f} - {years_data[-1] + 0.25:.1f}")
print(f"Observations: {len(log_c_data)}")
```

```
Fetched 236 quarterly observations from FRED
Sample: 1948.0 - 2007.0
Observations: 236
```

We can verify Table 2 by computing sample moments of log consumption growth from our FRED data:

```
# Growth rates: 1948Q2 to 2006Q4 (T = 235 quarters)
diff_c = np.diff(log_c_data)

μ_hat = diff_c.mean()
σ_hat = diff_c.std(ddof=1)

print("Sample estimates from FRED data vs Table 2:")
print(f"  μ   = {μ_hat:.5f}   (Table 2 RW: {rw['μ']:.5f})")
print(f"  σ_ε = {σ_hat:.4f}    (Table 2: {rw['σ_ε']:.4f})")
print(f"  T   = {len(diff_c)} quarters")
```

```
Sample estimates from FRED data vs Table 2:
  μ   = 0.00550   (Table 2 RW: 0.00495)
  σ_ε = 0.0054    (Table 2: 0.0050)
  T   = 235 quarters
```

```
p_fig6 = 0.20
```

(continues on next page)

(continued from previous page)

```
rw_fig6 = dict(μ=μ_hat, σ_ε=σ_hat)
w_fig6 = 2.0 * norm.ppf(p_fig6) / np.sqrt(T)

c = log_c_data
years = years_data

t6 = np.arange(T + 1)
μ_approx = rw_fig6["μ"]
μ_worst = rw_fig6["μ"] + rw_fig6["σ_ε"] * w_fig6

a_approx = (c - μ_approx * t6).mean()
a_worst = (c - μ_worst * t6).mean()
line_approx = a_approx + μ_approx * t6
line_worst = a_worst + μ_worst * t6

p_right = np.linspace(0.01, 0.50, 500)
w_right = 2.0 * norm.ppf(p_right) / np.sqrt(T)
μ_worst_right = rw_fig6["μ"] + rw_fig6["σ_ε"] * w_right

μ_se = rw_fig6["σ_ε"] / np.sqrt(T)
upper_band = rw_fig6["μ"] + 2.0 * μ_se
lower_band = rw_fig6["μ"] - 2.0 * μ_se
```

```
fig, axes = plt.subplots(1, 2, figsize=(12, 4))

ax = axes[0]
ax.plot(years, c, lw=2, color="tab:blue", label="log consumption")
ax.plot(years, line_approx, lw=2, ls="--",
            color="black", label="approximating model")
ax.plot(
    years,
    line_worst,
    lw=2,
    ls=":",
    color="black",
    label=rf"wc model $p(\theta^{{-1}})={p_fig6:.1f}$",
)
ax.set_xlabel("year")
ax.set_ylabel("log consumption")
ax.legend(frameon=False, fontsize=8, loc="upper left")

ax = axes[1]
ax.plot(
    100.0 * p_right,
    1_000.0 * μ_worst_right,
    lw=2,
    color="tab:red",
    label=r"$\mu + \sigma_\varepsilon w(\theta)$",
)
ax.axhline(1_000.0 * rw_fig6["μ"], lw=2, color="black", label=r"$\hat\mu$")
ax.axhline(1_000.0 * upper_band, lw=2, ls="--",
                    color="gray", label=r"$\hat\mu \pm 2\hat s.e.$")
ax.axhline(1_000.0 * lower_band, lw=2, ls="--", color="gray")
ax.set_xlabel("detection error probability (percent)")
ax.set_ylabel(r"mean consumption growth ($\times 10^{-3}$)")
ax.legend(frameon=False, fontsize=8)
```

(continues on next page)

(continued from previous page)

```
ax.set_xlim(0.0, 50.0)

plt.tight_layout()
plt.show()
```

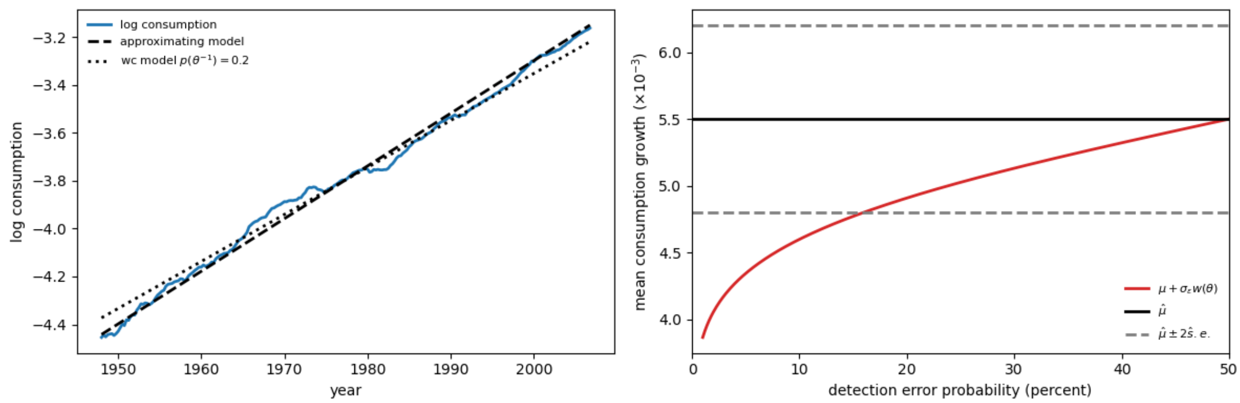


Fig. 85.9: Drift distortion and sampling uncertainty

In the left panel, postwar U.S. log consumption is shown alongside two deterministic trend lines: the approximating-model drift μ and the worst-case drift $\mu + \sigma_\varepsilon w(\theta)$ for $p(\theta^{-1}) = 0.20$.

The two trends are close enough that, even with six decades of data, it is hard to distinguish them by eye.

In the right panel, as the detection-error probability rises (the two models become harder to tell apart), the worst-case mean growth rate drifts back toward $\hat{\mu}$.

The dashed gray lines mark a two-standard-error band around the maximum-likelihood estimate of μ.

Even at detection probabilities in the 5–20% range, the worst-case drift remains inside (or very near) this confidence band.

Drift distortions that are economically large, large enough to generate substantial model-uncertainty premia, are statistically small relative to sampling uncertainty in $\hat{\mu}$.

Robustness concerns persist despite long histories precisely because the low-frequency features that matter most for pricing are the hardest to estimate precisely.

85.11 Concluding remarks

The title of this lecture poses a question: are large risk premia prices of *variability* (atemporal risk aversion) or prices of *doubts* (model uncertainty)?

Asset-pricing data alone cannot settle the question, because the two interpretations are observationally equivalent.

But the choice of interpretation matters for the conclusions we draw.

Under the risk-aversion reading, high Sharpe ratios imply that consumers would pay a great deal to smooth known aggregate consumption fluctuations.

Under the robustness reading, those same Sharpe ratios tell us that consumers would pay a great deal to resolve uncertainty about which probability model actually governs consumption growth.

Three features of the analysis support the robustness reading:

1. Detection-error probabilities provide a more stable calibration language than γ.
   - The two consumption models that required very different γ values to match the data yield nearly identical pricing implications when indexed by detectability.
2. The welfare gains implied by asset prices decompose overwhelmingly into a model-uncertainty component, with the pure risk component remaining small, consistent with Lucas's original finding.
3. The drift distortions that drive pricing are small enough to hide inside standard-error bands, so finite-sample learning cannot eliminate the consumer's fears.

Whether one ultimately prefers the risk or the uncertainty interpretation, the framework clarifies that the question is not about the size of risk premia but about the economic object those premia measure.

85.12 Exercises

The following exercises ask you to fill in several derivation steps.

Exercise 85.12.1

Let R_{t+1} be an $n \times 1$ vector of gross returns with unconditional mean $E(R)$ and covariance matrix Σ_R.

Let m_{t+1} be a stochastic discount factor satisfying $\mathbf{1} = E[m_{t+1} R_{t+1}]$.

1. Use the covariance decomposition $E[mR] = E[m]E[R] + \operatorname{cov}(m, R)$ to show that $\operatorname{cov}(m, R) = \mathbf{1} - E[m]E[R] =: b$.
2. For a portfolio with weight vector α and return $R^p = \alpha^\top R$, show that $\operatorname{cov}(m, R^p) = \alpha^\top b$.
3. Apply the Cauchy–Schwarz inequality to the pair (m, R^p) to obtain $|\alpha^\top b| \leq \sigma(m)\sqrt{\alpha^\top \Sigma_R \alpha}$.
4. Maximize the ratio $|\alpha^\top b| / \sqrt{\alpha^\top \Sigma_R \alpha}$ over α and show that the maximum is $\sqrt{b^\top \Sigma_R^{-1} b}$, attained at $\alpha^\star = \Sigma_R^{-1} b$.
5. Conclude that $\sigma(m) \geq \sqrt{b^\top \Sigma_R^{-1} b}$, which is (85.5).

Solution

Part 1. From $\mathbf{1} = E[mR] = E[m]E[R] + \operatorname{cov}(m, R)$, rearranging gives $\operatorname{cov}(m, R) = \mathbf{1} - E[m]E[R] = b$.

Part 2. The portfolio return is $R^p = \alpha^\top R$, so

$$\operatorname{cov}(m, R^p) = \operatorname{cov}(m, \alpha^\top R) = \alpha^\top \operatorname{cov}(m, R) = \alpha^\top b.$$

Part 3. Applying the Cauchy–Schwarz inequality to (m, R^p):

$$|\alpha^\top b| = |\operatorname{cov}(m, R^p)| \leq \sigma(m)\sigma(R^p) = \sigma(m)\sqrt{\alpha^\top \Sigma_R \alpha}.$$

Part 4. Rearranging Part 3 gives

$$\frac{|\alpha^\top b|}{\sqrt{\alpha^\top \Sigma_R \alpha}} \leq \sigma(m).$$

To maximize the left-hand side over α, define the Σ_R-inner product $\langle u, v \rangle_\Sigma = u^\top \Sigma_R v$.

Inserting $I = \Sigma_R \Sigma_R^{-1}$ gives

$$\alpha^\top b = \alpha^\top (\Sigma_R \Sigma_R^{-1}) b = (\alpha^\top \Sigma_R)(\Sigma_R^{-1} b) = \langle \alpha, \Sigma_R^{-1} b \rangle_\Sigma.$$

Cauchy–Schwarz in this inner product gives

$$|\langle \alpha, \Sigma_R^{-1} b \rangle_\Sigma| \leq \sqrt{\langle \alpha, \alpha \rangle_\Sigma} \sqrt{\langle \Sigma_R^{-1} b, \Sigma_R^{-1} b \rangle_\Sigma} = \sqrt{\alpha^\top \Sigma_R \alpha} \sqrt{b^\top \Sigma_R^{-1} b},$$

with equality when $\alpha \propto \Sigma_R^{-1} b$.

Substituting $\alpha^\star = \Sigma_R^{-1} b$ verifies

$$\max_\alpha \frac{|\alpha^\top b|}{\sqrt{\alpha^\top \Sigma_R \alpha}} = \sqrt{b^\top \Sigma_R^{-1} b}.$$

Part 5. Combining Parts 3 and 4 gives $\sigma(m) \geq \sqrt{b^\top \Sigma_R^{-1} b}$, which is (85.5).

Exercise 85.12.2

Combine the SDF representation (85.18) with the random-walk consumption dynamics and the Gaussian mean-shift distortion to derive closed-form SDF moments.

1. Show that $\log m_{t+1}$ is normally distributed under the approximating model and compute its mean and variance in terms of $(\beta, \mu, \sigma_\varepsilon, w)$.
2. Use lognormal moments to derive expressions for $E[m]$ and $\sigma(m)/E[m]$.
3. Use the parameter mapping $\theta = [(1-\beta)(\gamma - 1)]^{-1}$ and the associated w to obtain closed-form expressions for the random-walk model.
4. Explain why $E[m]$ stays roughly constant while $\sigma(m)/E[m]$ grows linearly with γ.

Solution

Under the random walk,

$$c_{t+1} - c_t = \mu + \sigma_\varepsilon \varepsilon_{t+1}$$

with $\varepsilon_{t+1} \sim \mathcal{N}(0, 1)$ under the approximating model.

Using (85.18) and the Gaussian distortion

$$\hat{g}_{t+1} = \exp\left(w \varepsilon_{t+1} - \tfrac{1}{2} w^2\right),$$

we get

$$m_{t+1} = \beta \exp\left(-(c_{t+1} - c_t)\right) \hat{g}_{t+1} = \beta \exp\left(-\mu - \sigma_\varepsilon \varepsilon_{t+1}\right) \exp\left(w \varepsilon_{t+1} - \frac{1}{2} w^2\right).$$

Therefore

$$\log m_{t+1} = \log \beta - \mu - \frac{1}{2} w^2 + (w - \sigma_\varepsilon) \varepsilon_{t+1},$$

which is normal with mean

$$E[\log m] = \log\beta - \mu - \tfrac{1}{2}w^2$$

and variance

$$\mathrm{Var}(\log m) = (w - \sigma_\varepsilon)^2.$$

For a lognormal random variable,

$$E[m] = \exp(E[\log m] + \tfrac{1}{2}\mathrm{Var}(\log m))$$

and

$$\sigma(m)/E[m] = \sqrt{e^{\mathrm{Var}(\log m)} - 1}.$$

Hence

$$E[m] = \beta \exp\left(-\mu - \frac{1}{2}w^2 + \frac{1}{2}(w - \sigma_\varepsilon)^2\right) = \beta \exp\left(-\mu + \frac{\sigma_\varepsilon^2}{2} - \sigma_\varepsilon w\right),$$

and

$$\frac{\sigma(m)}{E[m]} = \sqrt{\exp\left((w - \sigma_\varepsilon)^2\right) - 1}.$$

Now use $w_{\mathrm{RW}}(\theta) = -\sigma_\varepsilon/[(1-\beta)\theta]$ from (85.23) and $\theta = [(1-\beta)(\gamma-1)]^{-1}$ to get $w = -\sigma_\varepsilon(\gamma - 1)$. Then

$$-\sigma_\varepsilon w = \sigma_\varepsilon^2(\gamma - 1)$$

and

$$(w - \sigma_\varepsilon)^2 = (-\sigma_\varepsilon\gamma)^2 = \sigma_\varepsilon^2\gamma^2.$$

Substituting gives the closed-form expressions for the random-walk model:

$$E[m] = \beta \exp\left[-\mu + \frac{\sigma_\varepsilon^2}{2}(2\gamma - 1)\right], \tag{85.40}$$

$$\frac{\sigma(m)}{E[m]} = \sqrt{\exp\left(\sigma_\varepsilon^2\gamma^2\right) - 1}. \tag{85.41}$$

Notice that in (85.40), because σ_ε is small (≈ 0.005), the term $\frac{\sigma_\varepsilon^2}{2}(2\gamma - 1)$ grows slowly with γ, keeping $E[m]$ roughly constant near $1/(1 + r^f)$.

Meanwhile (85.41) shows that $\sigma(m)/E[m] \approx \sigma_\varepsilon\gamma$ grows linearly with γ.

This is how Epstein–Zin preferences push volatility toward the HJ bound without distorting the risk-free rate.

An analogous calculation for the trend-stationary model yields:

$$E[m] = \beta \exp\left[-\mu + \frac{\sigma_\varepsilon^2}{2}\left(1 - \frac{2(1-\beta)(1-\gamma)}{1-\beta\rho} + \frac{1-\rho}{1+\rho}\right)\right], \tag{85.42}$$

$$\frac{\sigma(m)}{E[m]} = \sqrt{\exp\left[\sigma_\varepsilon^2\left(\left(\frac{(1-\beta)(1-\gamma)}{1-\beta\rho} - 1\right)^2 + \frac{1-\rho}{1+\rho}\right)\right] - 1}. \tag{85.43}$$

Exercise 85.12.3

Starting from the type I recursion (85.9) and the definitions of U_t and θ in (85.10)–(85.11), derive the risk-sensitive recursion (85.12).

Verify that as $\gamma \to 1$ (equivalently $\theta \to \infty$), the recursion converges to standard discounted expected log utility $U_t = c_t + \beta E_t U_{t+1}$.

Solution

Start from the type I recursion (85.9) and write

$$(V_{t+1})^{1-\gamma} = \exp((1-\gamma)\log V_{t+1}).$$

Using $\log V_t = (1-\beta)U_t$ from (85.10), we obtain

$$(1-\beta)U_t = (1-\beta)c_t + \frac{\beta}{1-\gamma}\log E_t\left[\exp((1-\gamma)(1-\beta)U_{t+1})\right].$$

Divide by $(1-\beta)$ and use (85.11),

$$\theta = -\left[(1-\beta)(1-\gamma)\right]^{-1}.$$

Then $(1-\gamma)(1-\beta) = -1/\theta$ and $\beta/[(1-\beta)(1-\gamma)] = -\beta\theta$, so

$$U_t = c_t - \beta\theta\log E_t\left[\exp\left(-\frac{U_{t+1}}{\theta}\right)\right],$$

which is (85.12).

For $\theta \to \infty$ (equivalently $\gamma \to 1$), use the expansion

$$\exp(-U_{t+1}/\theta) = 1 - U_{t+1}/\theta + o(1/\theta).$$

Taking expectations,

$$E_t[\exp(-U_{t+1}/\theta)] = 1 - E_t[U_{t+1}]/\theta + o(1/\theta).$$

Applying $\log(1+x) = x + o(x)$ with $x = -E_t[U_{t+1}]/\theta + o(1/\theta)$,

$$\log E_t[\exp(-U_{t+1}/\theta)] = -E_t[U_{t+1}]/\theta + o(1/\theta),$$

so $-\theta\log E_t[\exp(-U_{t+1}/\theta)] \to E_t[U_{t+1}]$ as $\theta \to \infty$ and the recursion converges to

$$U_t = c_t + \beta E_t U_{t+1}.$$

Exercise 85.12.4

Consider the type II Bellman equation (85.16).

1. Use a Lagrange multiplier to impose the normalization constraint $\int g(\varepsilon)\pi(\varepsilon)d\varepsilon = 1$.
2. Derive the first-order condition for $g(\varepsilon)$ and show that the minimizer is the exponential tilt in (85.17).

3. Substitute your minimizing g back into (85.16) to recover the risk-sensitive Bellman equation (85.13).

Conclude that $W(x) \equiv U(x)$ for consumption plans in $\mathcal{C}(A, B, H; x_0)$.

Solution

Fix x and write $W'(\varepsilon) := W(Ax + B\varepsilon)$ for short.

Form the Lagrangian

$$\mathcal{L}[g, \lambda] = \beta \int \Big[g(\varepsilon) W'(\varepsilon) + \theta g(\varepsilon) \log g(\varepsilon) \Big] \pi(\varepsilon) d\varepsilon + \lambda \left(\int g(\varepsilon) \pi(\varepsilon) d\varepsilon - 1 \right).$$

The pointwise first-order condition for $g(\varepsilon)$ is

$$0 = \frac{\partial \mathcal{L}}{\partial g(\varepsilon)} = \beta \Big[W'(\varepsilon) + \theta(1 + \log g(\varepsilon)) \Big] \pi(\varepsilon) + \lambda \pi(\varepsilon),$$

so (dividing by $\beta\pi(\varepsilon)$)

$$\log g(\varepsilon) = -\frac{W'(\varepsilon)}{\theta} - 1 - \frac{\lambda}{\beta\theta}.$$

Exponentiating yields $g(\varepsilon) = K \exp(-W'(\varepsilon)/\theta)$ where $K = \exp(-1 - \lambda/(\beta\theta))$ is a constant that does not depend on ε.

To pin down K, impose the normalization $\int g(\varepsilon)\pi(\varepsilon) d\varepsilon = 1$:

$$1 = K \int \exp\left(-\frac{W(Ax + B\varepsilon)}{\theta} \right) \pi(\varepsilon) d\varepsilon,$$

so

$$K^{-1} = \int \exp\left(-\frac{W(Ax + B\varepsilon)}{\theta} \right) \pi(\varepsilon) d\varepsilon.$$

Substituting K^{-1} into the denominator of $g = K\exp(-W'/\theta)$ gives the minimizer:

$$g^*(\varepsilon) = \frac{\exp\left(-W(Ax + B\varepsilon)/\theta\right)}{\int \exp\left(-W(Ax + B\tilde{\varepsilon})/\theta\right) \pi(\tilde{\varepsilon}) d\tilde{\varepsilon}}.$$

This has exactly the same form as the distortion $\hat{g}_{t+1} = \exp(-U_{t+1}/\theta)/E_t[\exp(-U_{t+1}/\theta)]$ that appears in the type I SDF (85.14), with W in place of U.

Once we verify below that $W \equiv U$, the minimizer g^* and the SDF distortion $\hat{g}$ coincide, which is (85.17).

To substitute back, define

$$Z(x) := \int \exp(-W(Ax + B\varepsilon)/\theta) \pi(\varepsilon) d\varepsilon.$$

Then $\hat{g}(\varepsilon) = \exp(-W(Ax + B\varepsilon)/\theta)/Z(x)$ and

$$\log \hat{g}(\varepsilon) = -W(Ax + B\varepsilon)/\theta - \log Z(x).$$

Hence

$$\int \Big[\hat{g}(\varepsilon) W(Ax + B\varepsilon) + \theta \hat{g}(\varepsilon) \log \hat{g}(\varepsilon) \Big] \pi(\varepsilon) d\varepsilon = -\theta \log Z(x),$$

because the W terms cancel and $\int \hat{g}\pi = 1$.

Plugging this into (85.16) gives

$$W(x) = c - \beta\theta \log Z(x) = c - \beta\theta \log \int \exp\left(-\frac{W(Ax + B\varepsilon)}{\theta}\right) \pi(\varepsilon) d\varepsilon,$$

which is (85.13). Therefore $W(x) \equiv U(x)$.

Exercise 85.12.5

Let $\varepsilon \sim \mathcal{N}(0, 1)$ under the approximating model and define

$$\hat{g}(\varepsilon) = \exp\left(w\varepsilon - \frac{1}{2}w^2\right)$$

as in the Gaussian mean-shift section.

1. Show that $E[\hat{g}(\varepsilon)] = 1$.
2. Show that for any bounded measurable function f,

$$E[\hat{g}(\varepsilon)f(\varepsilon)]$$

equals the expectation of f under $\mathcal{N}(w, 1)$.

3. Compute the mean and variance of $\log \hat{g}(\varepsilon)$ and use these to derive

$$\text{std}(\hat{g}) = \sqrt{e^{w^2} - 1}.$$

4. Compute the conditional relative entropy $E[\hat{g} \log \hat{g}]$ and verify that it equals $\frac{1}{2}w^2$.

Solution

1. Using the moment generating function of a standard normal,

$$E[\hat{g}(\varepsilon)] = e^{-w^2/2} E[e^{w\varepsilon}] = e^{-w^2/2} e^{w^2/2} = 1.$$

2. Let $\varphi(\varepsilon) = (2\pi)^{-1/2} e^{-\varepsilon^2/2}$ be the $\mathcal{N}(0, 1)$ density.

Then

$$\hat{g}(\varepsilon)\varphi(\varepsilon) = \frac{1}{\sqrt{2\pi}} \exp\left(w\varepsilon - \frac{1}{2}w^2 - \frac{1}{2}\varepsilon^2\right) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2}(\varepsilon - w)^2\right),$$

which is the $\mathcal{N}(w, 1)$ density.

Therefore, for bounded measurable f,

$$E[\hat{g}(\varepsilon)f(\varepsilon)] = \int f(\varepsilon)\hat{g}(\varepsilon)\varphi(\varepsilon) d\varepsilon$$

equals the expectation of f under $\mathcal{N}(w, 1)$.

3. Since $\log \hat{g}(\varepsilon) = w\varepsilon - \frac{1}{2}w^2$ and $\varepsilon \sim \mathcal{N}(0, 1)$,

$$E[\log \hat{g}] = -\frac{1}{2}w^2, \qquad \text{Var}(\log \hat{g}) = w^2.$$

Moreover, $\mathrm{Var}(\hat{g}) = E[\hat{g}^2] - 1$ because $E[\hat{g}] = 1$.

Now

$$E[\hat{g}^2] = E\left[\exp\left(2w\varepsilon - w^2\right)\right] = e^{-w^2} E[e^{2w\varepsilon}] = e^{-w^2} e^{(2w)^2/2} = e^{w^2},$$

so $\mathrm{std}(\hat{g}) = \sqrt{e^{w^2} - 1}$.

4. Using part 2 with $f(\varepsilon) = \log \hat{g}(\varepsilon) = w\varepsilon - \frac{1}{2}w^2$,

$$E[\hat{g}\log\hat{g}] = E_{\mathcal{N}(w,1)}\left[w\varepsilon - \frac{1}{2}w^2\right] = w \cdot E_{\mathcal{N}(w,1)}[\varepsilon] - \frac{1}{2}w^2 = w^2 - \frac{1}{2}w^2 = \frac{1}{2}w^2.$$

Exercise 85.12.6

Derive the worst-case mean shifts (85.23) for both consumption models.

From (85.17), $\hat{g}_{t+1} \propto \exp(-W(x_{t+1})/\theta)$.

When W is linear in the state, the exponent is linear in ε_{t+1}, and the Gaussian mean shift is $w = -\lambda/\theta$ where λ is the coefficient on ε_{t+1} in $W(x_{t+1})$.

1. Random-walk model: Guess $W(x_t) = \frac{1}{1-\beta}[c_t + d]$. Using $c_{t+1} = c_t + \mu + \sigma_\varepsilon \varepsilon_{t+1}$, find λ and show that $w = -\sigma_\varepsilon/[(1-\beta)\theta]$.
2. Trend-stationary model: Write $z_t = \tilde{c}_t - \zeta$ and guess $W(x_t) = \frac{1}{1-\beta}[c_t + \alpha_1 z_t + \alpha_0]$. Show that:
   - The coefficient on ε_{t+1} in $W(x_{t+1})$ is $(1+\alpha_1)\sigma_\varepsilon/(1-\beta)$.
   - Matching coefficients on z_t in the Bellman equation gives $\alpha_1 = \beta(\rho-1)/(1-\beta\rho)$.
   - Therefore $1 + \alpha_1 = (1-\beta)/(1-\beta\rho)$ and $w = -\sigma_\varepsilon/[(1-\beta\rho)\theta]$.

Solution

Part 1. Under the guess $W(x_t) = \frac{1}{1-\beta}[c_t + d]$ and $c_{t+1} = c_t + \mu + \sigma_\varepsilon\varepsilon_{t+1}$,

$$W(x_{t+1}) = \frac{1}{1-\beta}[c_t + \mu + \sigma_\varepsilon\varepsilon_{t+1} + d].$$

The coefficient on ε_{t+1} is $\lambda = \sigma_\varepsilon/(1-\beta)$, so $w = -\lambda/\theta = -\sigma_\varepsilon/[(1-\beta)\theta]$.

Part 2. Under the guess $W(x_t) = \frac{1}{1-\beta}[c_t + \alpha_1 z_t + \alpha_0]$ with $c_{t+1} = c_t + \mu + (\rho-1)z_t + \sigma_\varepsilon\varepsilon_{t+1}$ and $z_{t+1} = \rho z_t + \sigma_\varepsilon\varepsilon_{t+1}$,

$$W(x_{t+1}) = \tfrac{1}{1-\beta}[c_t + \mu + (\rho-1)z_t + \sigma_\varepsilon\varepsilon_{t+1} + \alpha_1(\rho z_t + \sigma_\varepsilon\varepsilon_{t+1}) + \alpha_0].$$

The coefficient on ε_{t+1} is $(1+\alpha_1)\sigma_\varepsilon/(1-\beta)$.

To find α_1, substitute the guess into the Bellman equation.

The factors of $\frac{1}{1-\beta}$ cancel on both sides, and matching coefficients on z_t gives

$$\alpha_1 = \beta[(\rho-1) + \alpha_1\rho] \quad\Rightarrow\quad \alpha_1(1-\beta\rho) = \beta(\rho-1) \quad\Rightarrow\quad \alpha_1 = \frac{\beta(\rho-1)}{1-\beta\rho}.$$

Therefore

$$1 + \alpha_1 = \frac{1-\beta\rho + \beta(\rho-1)}{1-\beta\rho} = \frac{1-\beta}{1-\beta\rho},$$

and the coefficient on ε_{t+1} becomes $(1+\alpha_1)\sigma_\varepsilon/(1-\beta) = \sigma_\varepsilon/(1-\beta\rho)$, giving $w = -\sigma_\varepsilon/[(1-\beta\rho)\theta]$.

Exercise 85.12.7

Verify the closed-form value function (85.27) for the random-walk model by substituting a guess of the form $W(x_t) = \frac{1}{1-\beta}[c_t + d]$ into the risk-sensitive Bellman equation (85.13).

1. Under the random walk $c_{t+1} = c_t + \mu + \sigma_\varepsilon \varepsilon_{t+1}$, show that $W(Ax_t + B\varepsilon) = \frac{1}{1-\beta}[c_t + \mu + \sigma_\varepsilon \varepsilon_{t+1} + d]$.
2. Substitute into the $\log E \exp$ term, using the fact that for $Z \sim \mathcal{N}(\mu_Z, \sigma_Z^2)$ we have $\log E[\exp(Z)] = \mu_Z + \frac{1}{2}\sigma_Z^2$.
3. Solve for d and confirm that it matches (85.27).

Solution

Part 1. Under the random walk, $c_{t+1} = c_t + \mu + \sigma_\varepsilon \varepsilon_{t+1}$. Substituting the guess $W(x) = \frac{1}{1-\beta}[Hx + d]$ with $Hx_t = c_t$:

$$W(Ax_t + B\varepsilon_{t+1}) = \frac{1}{1-\beta}[c_t + \mu + \sigma_\varepsilon \varepsilon_{t+1} + d].$$

Part 2. The Bellman equation (85.13) requires computing

$$-\beta\theta \log E_t \left[\exp\left(\frac{-W(Ax_t + B\varepsilon_{t+1})}{\theta}\right)\right].$$

Substituting the guess:

$$\frac{-W(Ax_t + B\varepsilon_{t+1})}{\theta} = \frac{-1}{(1-\beta)\theta}[c_t + \mu + d + \sigma_\varepsilon \varepsilon_{t+1}].$$

This is an affine function of the standard normal ε_{t+1}, so the argument of the $\log E \exp$ is normal with

$$\mu_Z = \frac{-(c_t + \mu + d)}{(1-\beta)\theta}, \qquad \sigma_Z^2 = \frac{\sigma_\varepsilon^2}{(1-\beta)^2\theta^2}.$$

Using $\log E[e^Z] = \mu_Z + \frac{1}{2}\sigma_Z^2$:

$$-\beta\theta \left[\frac{-(c_t + \mu + d)}{(1-\beta)\theta} + \frac{\sigma_\varepsilon^2}{2(1-\beta)^2\theta^2}\right] = \frac{\beta}{1-\beta}\left[c_t + \mu + d - \frac{\sigma_\varepsilon^2}{2(1-\beta)\theta}\right].$$

Part 3. The Bellman equation becomes

$$\frac{1}{1-\beta}[c_t + d] = c_t + \frac{\beta}{1-\beta}\left[c_t + \mu + d - \frac{\sigma_\varepsilon^2}{2(1-\beta)\theta}\right].$$

Expanding the right-hand side:

$$c_t + \frac{\beta c_t}{1-\beta} + \frac{\beta(\mu + d)}{1-\beta} - \frac{\beta\sigma_\varepsilon^2}{2(1-\beta)^2\theta} = \frac{c_t}{1-\beta} + \frac{\beta(\mu + d)}{1-\beta} - \frac{\beta\sigma_\varepsilon^2}{2(1-\beta)^2\theta}.$$

Equating both sides and cancelling $\frac{c_t}{1-\beta}$:

$$\frac{d}{1-\beta} = \frac{\beta(\mu + d)}{1-\beta} - \frac{\beta\sigma_\varepsilon^2}{2(1-\beta)^2\theta}.$$

Solving: $d - \beta d = \beta\mu - \frac{\beta\sigma_\varepsilon^2}{2(1-\beta)\theta}$, so

$$d = \frac{\beta}{1-\beta}\left(\mu - \frac{\sigma_\varepsilon^2}{2(1-\beta)\theta}\right),$$

which matches (85.27).

Exercise 85.12.8

In the Gaussian mean-shift setting of Exercise 5, let L_T be the log likelihood ratio between the worst-case and approximating models based on T observations.

1. Show that L_T is normal under each model.
2. Compute its mean and variance under the approximating and worst-case models.
3. Using the definition of detection-error probability in (85.30), derive the closed-form expression (85.31).

Solution

Let the approximating model be $\varepsilon_i \sim \mathcal{N}(0,1)$ and the worst-case model be $\varepsilon_i \sim \mathcal{N}(w,1)$, i.i.d. for $i = 1, \ldots, T$.

Take the log likelihood ratio in the direction that matches the definitions in the text:

$$L_T = \log \frac{\prod_{i=1}^T \varphi(\varepsilon_i)}{\prod_{i=1}^T \varphi(\varepsilon_i - w)} = \sum_{i=1}^T \ell(\varepsilon_i),$$

where φ is the $\mathcal{N}(0,1)$ density and

$$\ell(\varepsilon) = \log\varphi(\varepsilon) - \log\varphi(\varepsilon - w) = -\frac{1}{2}\left[\varepsilon^2 - (\varepsilon - w)^2\right] = -w\varepsilon + \frac{1}{2}w^2.$$

Therefore

$$L_T = -w\sum_{i=1}^T \varepsilon_i + \tfrac{1}{2}w^2 T.$$

Under the approximating model, $\sum_{i=1}^T \varepsilon_i \sim \mathcal{N}(0,T)$, so

$$L_T \sim \mathcal{N}\left(\frac{1}{2}w^2 T, w^2 T\right).$$

Under the worst-case model, $\sum_{i=1}^T \varepsilon_i \sim \mathcal{N}(wT,T)$, so

$$L_T \sim \mathcal{N}\left(-\frac{1}{2}w^2 T, w^2 T\right).$$

Now

$$p_A = \Pr_A(L_T < 0) = \Phi\left(\frac{0 - \frac{1}{2}w^2 T}{|w|\sqrt{T}}\right) = \Phi\left(-\frac{|w|\sqrt{T}}{2}\right),$$

and

$$p_B = \Pr_B(L_T > 0) = 1 - \Phi\left(\frac{0 - (-\frac{1}{2}w^2 T)}{|w|\sqrt{T}}\right) = 1 - \Phi\left(\frac{|w|\sqrt{T}}{2}\right) = \Phi\left(-\frac{|w|\sqrt{T}}{2}\right).$$

Therefore

$$p(\theta^{-1}) = \tfrac{1}{2}(p_A + p_B) = \Phi\left(-\tfrac{|w|\sqrt{T}}{2}\right),$$

which is (85.31).

Exercise 85.12.9

Using the formulas for $w(\theta)$ in (85.23) and the definition of discounted entropy

$$\eta = \frac{\beta}{1-\beta} \cdot \frac{w(\theta)^2}{2},$$

show that holding η fixed across the random-walk and trend-stationary consumption specifications implies the mapping (85.32).

Specialize your result to the case $\sigma_\varepsilon^{\mathrm{TS}} = \sigma_\varepsilon^{\mathrm{RW}}$ and interpret the role of ρ.

Solution

Because η depends on θ only through $w(\theta)^2$, holding η fixed across models is equivalent to holding $|w(\theta)|$ fixed.

Using (85.23),

$$|w_{\mathrm{RW}}(\theta_{\mathrm{RW}})| = \frac{\sigma_\varepsilon^{\mathrm{RW}}}{(1-\beta)\theta_{\mathrm{RW}}}, \qquad |w_{\mathrm{TS}}(\theta_{\mathrm{TS}})| = \frac{\sigma_\varepsilon^{\mathrm{TS}}}{(1-\beta\rho)\theta_{\mathrm{TS}}}.$$

Equating these magnitudes and solving for θ_{TS} gives

$$\theta_{\mathrm{TS}} = \left(\frac{\sigma_\varepsilon^{\mathrm{TS}}}{\sigma_\varepsilon^{\mathrm{RW}}}\right) \frac{1-\beta}{1-\beta\rho}\theta_{\mathrm{RW}},$$

which is (85.32).

If $\sigma_\varepsilon^{\mathrm{TS}} = \sigma_\varepsilon^{\mathrm{RW}}$, then

$$\theta_{\mathrm{TS}} = \frac{1-\beta}{1-\beta\rho}\theta_{\mathrm{RW}}.$$

Since $\rho \in (0, 1)$ implies $1 - \beta\rho > 1 - \beta$, the ratio $(1-\beta)/(1-\beta\rho)$ is less than one.

To hold entropy fixed, the trend-stationary model therefore requires a smaller θ (i.e., a cheaper distortion and stronger robustness) than the random-walk model.

Exercise 85.12.10

For type II (multiplier) preferences under random-walk consumption growth, derive the compensating-variation formulas in (85.35).

In particular, derive

1. the *risk* term by comparing the stochastic economy to a deterministic consumption path with the same mean level of consumption (Lucas's thought experiment), and

2. the *uncertainty* term by comparing a type II agent with parameter θ to the expected-utility case $\theta = \infty$, holding the stochastic environment fixed.

Solution

Write the random walk as

$$c_t = c_0 + t\mu + \sigma_\varepsilon \sum_{j=1}^{t} \varepsilon_j$$

with $\varepsilon_j \overset{iid}{\sim} \mathcal{N}(0,1)$.

Risk term:

The mean level of consumption is

$$E[C_t] = E[e^{c_t}] = \exp(c_0 + t\mu + \tfrac{1}{2}t\sigma_\varepsilon^2),$$

so the deterministic path with the same mean levels is

$$\bar{c}_t = c_0 + t(\mu + \tfrac{1}{2}\sigma_\varepsilon^2).$$

Under expected log utility ($\theta = \infty$), discounted expected utility is

$$\sum_{t\geq 0} \beta^t E[c_t] = \frac{c_0}{1-\beta} + \frac{\beta\mu}{(1-\beta)^2},$$

while for the deterministic mean-level path it is

$$\sum_{t\geq 0} \beta^t \bar{c}_t = \frac{c_0}{1-\beta} + \frac{\beta(\mu + \frac{1}{2}\sigma_\varepsilon^2)}{(1-\beta)^2}.$$

If we reduce initial consumption by Δc_0^{risk} (so $\bar{c}_t$ shifts down by Δc_0^{risk} for all t), utility falls by $\Delta c_0^{risk}/(1-\beta)$.

Equating the two utilities gives

$$\frac{\Delta c_0^{risk}}{1-\beta} = \frac{\beta(\frac{1}{2}\sigma_\varepsilon^2)}{(1-\beta)^2} \quad \Rightarrow \quad \Delta c_0^{risk} = \frac{\beta\sigma_\varepsilon^2}{2(1-\beta)}.$$

Uncertainty term:

For type II multiplier preferences, the minimizing distortion is a Gaussian mean shift with parameter w and per-period relative entropy $\frac{1}{2}w^2$.

Under the distorted model, $E[\varepsilon] = w$, so

$$E[c_t] = c_0 + t(\mu + \sigma_\varepsilon w).$$

Plugging this into the type II objective (and using $E_t[g \log g] = \frac{1}{2}w^2$) gives the discounted objective as a function of w:

$$J(w) = \sum_{t\geq 0} \beta^t \Big(c_0 + t(\mu + \sigma_\varepsilon w)\Big) + \sum_{t\geq 0} \beta^{t+1}\theta \cdot \frac{w^2}{2}.$$

Using $\sum_{t\geq 0} \beta^t = 1/(1-\beta)$ and $\sum_{t\geq 0} t\beta^t = \beta/(1-\beta)^2$,

$$J(w) = \frac{c_0}{1-\beta} + \frac{\beta(\mu + \sigma_\varepsilon w)}{(1-\beta)^2} + \frac{\beta\theta}{1-\beta} \cdot \frac{w^2}{2}.$$

Minimizing over w yields

$$0 = \frac{\partial J}{\partial w} = \frac{\beta \sigma_\varepsilon}{(1-\beta)^2} + \frac{\beta\theta}{1-\beta} w \quad \Rightarrow \quad w^* = -\frac{\sigma_\varepsilon}{(1-\beta)\theta},$$

which matches (85.23).

Substituting w^* back in gives

$$J(w^*) = \frac{c_0}{1-\beta} + \frac{\beta\mu}{(1-\beta)^2} - \frac{\beta\sigma_\varepsilon^2}{2(1-\beta)^3\theta}.$$

When $\theta = \infty$ (no model uncertainty), the last term disappears. Thus the utility gain from removing model uncertainty at fixed $(\mu, \sigma_\varepsilon)$ is

$$\beta\sigma_\varepsilon^2/[2(1-\beta)^3\theta].$$

To offset this by a permanent upward shift in initial log consumption, we need

$$\Delta c_0^{uncertainty}/(1-\beta) = \beta\sigma_\varepsilon^2/[2(1-\beta)^3\theta],$$

so

$$\Delta c_0^{uncertainty} = \frac{\beta\sigma_\varepsilon^2}{2(1-\beta)^2\theta}.$$

Together these reproduce (85.35).

Exercise 85.12.11

Derive the trend-stationary risk compensation $\Delta c_0^{risk,ts}$ in (85.39).

For the trend-stationary model with $\tilde{c}_{t+1} - \zeta = \rho(\tilde{c}_t - \zeta) + \sigma_\varepsilon \varepsilon_{t+1}$, where $\tilde{c}_t = c_t - \mu t$, compute the risk compensation $\Delta c_0^{risk,ts}$ by comparing expected log utility under the stochastic plan to the deterministic certainty-equivalent path, and show that

$$\Delta c_0^{risk,ts} = \frac{\beta\sigma_\varepsilon^2}{2(1-\beta\rho^2)}.$$

Hint: You will need $\mathrm{Var}(z_t) = \sigma_\varepsilon^2(1 + \rho^2 + \cdots + \rho^{2(t-1)})$ and the formula $\sum_{t\geq 1} \beta^t \sum_{j=0}^{t-1} \rho^{2j} = \frac{\beta}{(1-\beta)(1-\beta\rho^2)}$.

Solution

Under the trend-stationary model with $z_0 = 0$, $c_t = c_0 + \mu t + z_t$ and $E[c_t] = c_0 + \mu t$ (since $E[z_t] = 0$).

The deterministic certainty-equivalent path matches $E[C_t] = \exp(c_0 + \mu t + \frac{1}{2}\mathrm{Var}(z_t))$, so its log is $c_0^{ce} + \mu t + \frac{1}{2}\mathrm{Var}(z_t)$.

Under expected log utility ($\theta = \infty$), the value of the stochastic plan is

$$\sum_{t\geq 0} \beta^t E[c_t] = \frac{c_0}{1-\beta} + \frac{\beta\mu}{(1-\beta)^2}.$$

The value of the certainty-equivalent path (matching mean levels) starting from $c_0 - \Delta c_0^{risk}$ is

$$\sum_{t\geq 0} \beta^t \big[c_0 - \Delta c_0^{risk} + \mu t + \tfrac{1}{2}\,\mathrm{Var}(z_t)\big].$$

Since $\mathrm{Var}(z_t) = \sigma_\varepsilon^2 \sum_{j=0}^{t-1} \rho^{2j}$, the extra term sums to

$$\sum_{t\geq 1} \beta^t \cdot \frac{\sigma_\varepsilon^2}{2} \sum_{j=0}^{t-1} \rho^{2j} = \frac{\sigma_\varepsilon^2}{2} \cdot \frac{\beta}{(1-\beta)(1-\beta\rho^2)}.$$

Equating values and solving:

$$\frac{\Delta c_0^{risk}}{1-\beta} = \frac{\beta \sigma_\varepsilon^2}{2(1-\beta)(1-\beta\rho^2)} \quad \Rightarrow \quad \Delta c_0^{risk,ts} = \frac{\beta \sigma_\varepsilon^2}{2(1-\beta\rho^2)}.$$

The uncertainty compensation follows from the value function: $\Delta c_0^{unc,ts,II} = \frac{\beta\sigma_\varepsilon^2}{2(1-\beta\rho)^2\theta}$, with the $(1-\beta)$ factors replaced by $(1-\beta\rho)$ because the worst-case mean shift scales with $1/(1-\beta\rho)$ rather than $1/(1-\beta)$.

CHAPTER

EIGHTYSIX

COMPETITIVE EQUILIBRIA WITH ARROW SECURITIES

86.1 Introduction

This lecture presents Python code for experimenting with competitive equilibria of an infinite-horizon pure exchange economy with

- Heterogeneous agents
- Endowments of a single consumption that are person-specific functions of a common Markov state
- Complete markets in one-period Arrow state-contingent securities
- Discounted expected utility preferences of a kind often used in macroeconomics and finance
- Common expected utility preferences across agents
- Common beliefs among agents
- A constant relative risk aversion (CRRA) one-period utility function that implies the existence of a representative consumer whose consumption process can be plugged into a formula for the pricing kernel for one-step Arrow securities and thereby determine equilibrium prices before determining an equilibrium distribution of wealth
- Differences in their endowments make individuals want to reallocate consumption goods across time and Markov states

We impose restrictions that allow us to **Bellmanize** competitive equilibrium prices and quantities

We use Bellman equations to describe

- asset prices
- continuation wealth levels for each person
- state-by-state natural debt limits for each person

In the course of presenting the model we shall encounter these important ideas

- a **resolvent operator** widely used in this class of models
- absence of **borrowing limits** in finite horizon economies
- state-by-state **borrowing limits** required in infinite horizon economies
- a counterpart of the **law of iterated expectations** known as a **law of iterated values**
- a **state-variable degeneracy** that prevails within a competitive equilibrium and that opens the way to various appearances of resolvent operators

86.2 The setting

In effect, this lecture implements a Python version of the model presented in section 9.3.3 of Ljungqvist and Sargent [Ljungqvist and Sargent, 2018].

86.2.1 Preferences and endowments

In each period $t \geq 0$, a stochastic event $s_t \in \mathbf{S}$ is realized.

Let the history of events up until time t be denoted $s^t = [s_0, s_1, \ldots, s_{t-1}, s_t]$.

(Sometimes we inadvertently reverse the recording order and denote a history as $s^t = [s_t, s_{t-1}, \ldots, s_1, s_0]$.)

The unconditional probability of observing a particular sequence of events s^t is given by a probability measure $\pi_t(s^t)$.

For $t > \tau$, we write the probability of observing s^t conditional on the realization of s^τ as $\pi_t(s^t|s^\tau)$.

We assume that trading occurs after observing s_0, which we capture by setting $\pi_0(s_0) = 1$ for the initially given value of s_0.

In this lecture we shall follow much macroeconomics and econometrics and assume that $\pi_t(s^t)$ is induced by a Markov process.

There are K consumers named $k = 1, \ldots, K$.

Consumer k owns a stochastic endowment of one good $y_t^k(s^t)$ that depends on the history s^t.

The history s^t is publicly observable.

Consumer k purchases a history-dependent consumption plan $c^k = \{c_t^k(s^t)\}_{t=0}^\infty$

Consumer k orders consumption plans by

$$U_k(c^k) = \sum_{t=0}^{\infty} \sum_{s^t} \beta^t u_k[c_t^k(s^t)]\pi_t(s^t),$$

where $0 < \beta < 1$.

The right side is equal to $E_0 \sum_{t=0}^\infty \beta^t u_k(c_t^k)$, where E_0 is the mathematical expectation operator, conditioned on s_0.

Here $u_k(c)$ is an increasing, twice continuously differentiable, strictly concave function of consumption $c \geq 0$ of one good.

The utility function of person k satisfies the Inada condition

$$\lim_{c \downarrow 0} u'_k(c) = +\infty.$$

This condition implies that each agent chooses strictly positive consumption for every date-history pair (t, s^t).

Those interior solutions enable us to confine our analysis to Euler equations that hold with equality and also guarantee that **natural debt limits** don't bind in economies like ours with sequential trading of Arrow securities.

We adopt the assumption, routinely employed in much of macroeconomics, that consumers share probabilities $\pi_t(s^t)$ for all t and s^t.

A **feasible allocation** satisfies

$$\sum_i c_t^k(s^t) \leq \sum_i y_t^k(s^t)$$

for all t and for all s^t.

86.3 Recursive Formulation

Following descriptions in section 9.3.3 of Ljungqvist and Sargent [Ljungqvist and Sargent, 2018] chapter 9, we set up a competitive equilibrium of a pure exchange economy with complete markets in one-period Arrow securities.

When endowments $y^k(s)$ are all functions of a common Markov state s, the pricing kernel takes the form $Q(s'|s)$, where $Q(s'|s)$ is the price of one unit of consumption in state s' at date $t+1$ when the Markov state at date t is s.

These enable us to provide a recursive formulation of a consumer's optimization problem.

Consumer k's state at time t is its financial wealth a_t^k and Markov state s_t.

Let $v^k(a,s)$ be the optimal value of consumer k's problem starting from state (a,s).

- $v^k(a,s)$ is the maximum expected discounted utility that consumer k with current financial wealth a can attain in Markov state s.

The optimal value function satisfies the Bellman equation

$$v^k(a,s) = \max_{c,\hat{a}(s')} \left\{ u_k(c) + \beta \sum_{s'} v^k[\hat{a}(s'), s'] \pi(s'|s) \right\}$$

where maximization is subject to the budget constraint

$$c + \sum_{s'} \hat{a}(s') Q(s'|s) \leq y^k(s) + a$$

and also the constraints

$$\begin{aligned} c &\geq 0, \\ -\hat{a}(s') &\leq \bar{A}^k(s'), \qquad \forall s' \in \mathbf{S} \end{aligned}$$

with the second constraint evidently being a set of state-by-state debt limits.

Note that the value function and decision rule that solve the Bellman equation implicitly depend on the pricing kernel $Q(\cdot|\cdot)$ because it appears in the agent's budget constraint.

Use the first-order conditions for the problem on the right of the Bellman equation and a Benveniste-Scheinkman formula and rearrange to get

$$Q(s_{t+1}|s_t) = \frac{\beta u_k'(c_{t+1}^k) \pi(s_{t+1}|s_t)}{u_k'(c_t^k)},$$

where it is understood that $c_t^k = c^k(s_t)$ and $c_{t+1}^k = c^k(s_{t+1})$.

A **recursive competitive equilibrium** is an initial distribution of wealth $\vec{a}_0$, a set of borrowing limits $\{\bar{A}^k(s)\}_{k=1}^K$, a pricing kernel $Q(s'|s)$, sets of value functions $\{v^k(a,s)\}_{k=1}^K$, and decision rules $\{c^k(s), \hat{a}^k(s)\}_{k=1}^K$ such that

- The state-by-state borrowing constraints satisfy the recursion

$$\bar{A}^k(s) = y^k(s) + \sum_{s'} Q(s'|s) \bar{A}^k(s')$$

- For all k, given a_0^k, $\bar{A}^k(s)$, and the pricing kernel, the value functions and decision rules solve the consumers' problems;
- For all realizations of $\{s_t\}_{t=0}^\infty$, the consumption and asset portfolios $\{\{c_t^k, \{\hat{a}_{t+1}^k(s')\}_{s'}\}_k\}_t$ satisfy $\sum_k c_t^k = \sum_k y^k(s_t)$ and $\sum_k \hat{a}_{t+1}^k(s') = 0$ for all t and s'.
- The initial financial wealth vector $\vec{a}_0$ satisfies $\sum_{k=1}^K a_0^k = 0$.

The third condition asserts that there are zero net aggregate claims in all Markov states.

The fourth condition asserts that the economy is closed and starts from a situation in which there are zero net aggregate claims.

86.4 State Variable Degeneracy

Please see Ljungqvist and Sargent [Ljungqvist and Sargent, 2018] for a description of timing protocol for trades consistent with an Arrow-Debreu vision in which

- at time 0 there are complete markets in a complete menu of history s^t-contingent claims on consumption at all dates that all trades occur at time zero
- all trades occur once and for all at time 0

If an allocation and pricing kernel Q in a recursive competitive equilibrium are to be consistent with the equilibrium allocation and price system that prevail in a corresponding complete markets economy with such history-contingent commodities and all trades occurring at time 0, we must impose that $a_0^k = 0$ for $k = 1, \ldots, K$.

That is what assures that at time 0 the present value of each agent's consumption equals the present value of his endowment stream, the single budget constraint in arrangement with all trades occurring at time 0.

Starting the system with $a_0^k = 0$ for all i has a striking implication that we call **state variable degeneracy**.

Here is what we mean by **state variable degeneracy**:

Although two state variables a, s appear in the value function $v^k(a, s)$, within a recursive competitive equilibrium starting from $a_0^k = 0 \ \forall i$ at initial Markov state s_0, two outcomes prevail:

- $a_0^k = 0$ for all i whenever the Markov state s_t returns to s_0.
- Financial wealth a is an exact function of the Markov state s.

The first finding asserts that each household recurrently visits the zero financial wealth state with which it began life.

The second finding asserts that within a competitive equilibrium the exogenous Markov state is all we require to track an individual.

Financial wealth turns out to be redundant because it is an exact function of the Markov state for each individual.

This outcome depends critically on there being complete markets in Arrow securities.

For example, it does not prevail in the incomplete markets setting of this lecture *The Aiyagari Model*

86.5 Markov Asset Prices

Let's start with a brief summary of formulas for computing asset prices in a Markov setting.

The setup assumes the following infrastructure

- Markov states: $s \in S = [\bar{s}_1, \ldots, \bar{s}_n]$ governed by an n-state Markov chain with transition probability

$$P_{ij} = \Pr\left\{s_{t+1} = \bar{s}_j \mid s_t = \bar{s}_i\right\}$$

- A collection $h = 1, \ldots, H$ of $n \times 1$ vectors of H assets that pay off $d^h(s)$ in state s
- An $n \times n$ matrix pricing kernel Q for one-period Arrow securities, where Q_{ij} = price at time t in state $s_t = \bar{s}_i$ of one unit of consumption when $s_{t+1} = \bar{s}_j$ at time $t+1$:

$$Q_{ij} = \text{Price}\left\{s_{t+1} = \bar{s}_j \mid s_t = \bar{s}_i\right\}$$

- The price of risk-free one-period bond in state i is $R_i^{-1} = \sum_j Q_{i,j}$
- The gross rate of return on a one-period risk-free bond Markov state $\bar{s}_i$ is $R_i = (\sum_j Q_{i,j})^{-1}$

86.5.1 Exogenous Pricing Kernel

At this point, we'll take the pricing kernel Q as exogenous, i.e., determined outside the model

Two examples would be

- $Q = \beta P$ where $\beta \in (0, 1)$
- $Q = SP$ where S is an $n \times n$ matrix of *stochastic discount factors*

We'll write down implications of Markov asset pricing in a nutshell for two types of assets

- the price in Markov state s at time t of a **cum dividend** stock that entitles the owner at the beginning of time t to the time t dividend and the option to sell the asset at time $t + 1$. The price evidently satisfies $p^h(\bar{s}_i) = d^h(\bar{s}_i) + \sum_j Q_{ij} p^h(\bar{s}_j)$, which implies that the vector p^h satisfies $p^h = d^h + Qp^h$ which implies the formula

$$p^h = (I - Q)^{-1} d^h$$

- the price in Markov state s at time t of an **ex dividend** stock that entitles the owner at the end of time t to the time $t + 1$ dividend and the option to sell the stock at time $t + 1$. The price is

$$p^h = (I - Q)^{-1} Q d^h$$

Note

The matrix geometric sum $(I - Q)^{-1} = I + Q + Q^2 + \cdots$ is an example of a **resolvent operator**.

Below, we describe an equilibrium model with trading of one-period Arrow securities in which the pricing kernel is endogenous.

In constructing our model, we'll repeatedly encounter formulas that remind us of our asset pricing formulas.

86.5.2 Multi-Step-Forward Transition Probabilities and Pricing Kernels

The (i, j) component of the ℓ-step ahead transition probability P^ℓ is

$$Prob(s_{t+\ell} = \bar{s}_j | s_t = \bar{s}_i) = P^\ell_{i,j}$$

The (i, j) component of the ℓ-step ahead pricing kernel Q^ℓ is

$$Q^{(\ell)}(s_{t+\ell} = \bar{s}_j | s_t = \bar{s}_i) = Q^\ell_{i,j}$$

We'll use these objects to state a useful property in asset pricing theory.

86.5.3 Laws of Iterated Expectations and Iterated Values

A **law of iterated values** has a mathematical structure that parallels a **law of iterated expectations**

We can describe its structure readily in the Markov setting of this lecture

Recall the following recursion satisfied by j step ahead transition probabilites for our finite state Markov chain:

$$P_j(s_{t+j} | s_t) = \sum_{s_{t+1}} P_{j-1}(s_{t+j} | s_{t+1}) P(s_{t+1} | s_t)$$

We can use this recursion to verify the law of iterated expectations applied to computing the conditional expectation of a random variable $d(s_{t+j})$ conditioned on s_t via the following string of equalities

$$\begin{aligned}
E\left[Ed(s_{t+j})|s_{t+1}\right]|s_t &= \sum_{s_{t+1}}\left[\sum_{s_{t+j}} d(s_{t+j})P_{j-1}(s_{t+j}|s_{t+1})\right]P(s_{t+1}|s_t) \\
&= \sum_{s_{t+j}} d(s_{t+j})\left[\sum_{s_{t+1}} P_{j-1}(s_{t+j}|s_{t+1})P(s_{t+1}|s_t)\right] \\
&= \sum_{s_{t+j}} d(s_{t+j})P_j(s_{t+j}|s_t) \\
&= Ed(s_{t+j})|s_t
\end{aligned}$$

The pricing kernel for j step ahead Arrow securities satisfies the recursion

$$Q_j(s_{t+j}|s_t) = \sum_{s_{t+1}} Q_{j-1}(s_{t+j}|s_{t+1})Q(s_{t+1}|s_t)$$

The time t **value** in Markov state s_t of a time $t+j$ payout $d(s_{t+j})$ is

$$V(d(s_{t+j})|s_t) = \sum_{s_{t+j}} d(s_{t+j})Q_j(s_{t+j}|s_t)$$

The **law of iterated values** states

$$V\left[V(d(s_{t+j})|s_{t+1})\right]|s_t = V(d(s_{t+j}))|s_t$$

We verify it by pursuing the following a string of inequalities that are counterparts to those we used to verify the law of iterated expectations:

$$\begin{aligned}
V\left[V(d(s_{t+j})|s_{t+1})\right]|s_t &= \sum_{s_{t+1}}\left[\sum_{s_{t+j}} d(s_{t+j})Q_{j-1}(s_{t+j}|s_{t+1})\right]Q(s_{t+1}|s_t) \\
&= \sum_{s_{t+j}} d(s_{t+j})\left[\sum_{s_{t+1}} Q_{j-1}(s_{t+j}|s_{t+1})Q(s_{t+1}|s_t)\right] \\
&= \sum_{s_{t+j}} d(s_{t+j})Q_j(s_{t+j}|s_t) \\
&= EV(d(s_{t+j}))|s_t
\end{aligned}$$

86.6 General Equilibrium

Now we are ready to do some fun calculations.

We find it interesting to think in terms of analytical **inputs** into and **outputs** from our general equilibrium theorizing.

86.6.1 Inputs

- Markov states: $s \in S = [\bar{s}_1, \dots, \bar{s}_n]$ governed by an n-state Markov chain with transition probability

$$P_{ij} = \Pr\left\{s_{t+1} = \bar{s}_j \mid s_t = \bar{s}_i\right\}$$

- A collection of $K \times 1$ vectors of individual k endowments: $y^k(s), k = 1, \dots, K$

- An $n \times 1$ vector of aggregate endowment: $y(s) \equiv \sum_{k=1}^{K} y^k(s)$
- A collection of $K \times 1$ vectors of individual k consumptions: $c^k(s), k = 1, \dots, K$
- A collection of restrictions on feasible consumption allocations for $s \in S$:

$$c(s) = \sum_{k=1}^{K} c^k(s) \leq y(s)$$

- Preferences: a common utility functional across agents $E_0 \sum_{t=0}^{\infty} \beta^t u(c_t^k)$ with CRRA one-period utility function $u(c)$ and discount factor $\beta \in (0, 1)$

The one-period utility function is

$$u(c) = \frac{c^{1-\gamma}}{1-\gamma}$$

so that

$$u'(c) = c^{-\gamma}$$

86.6.2 Outputs

- An $n \times n$ matrix pricing kernel Q for one-period Arrow securities, where Q_{ij} = price at time t in state $s_t = \bar{s}_i$ of one unit of consumption when $s_{t+1} = \bar{s}_j$ at time $t + 1$
- pure exchange so that $c(s) = y(s)$
- a $K \times 1$ vector distribution of wealth vector α, $\alpha_k \geq 0, \sum_{k=1}^{K} \alpha_k = 1$
- A collection of $n \times 1$ vectors of individual k consumptions: $c^k(s), k = 1, \dots, K$

86.6.3 Q is the Pricing Kernel

For any agent $k \in [1, \dots, K]$, at the equilibrium allocation, the one-period Arrow securities pricing kernel satisfies

$$Q_{ij} = \beta \left(\frac{c^k(\bar{s}_j)}{c^k(\bar{s}_i)} \right)^{-\gamma} P_{ij}$$

where Q is an $n \times n$ matrix

This follows from agent k's first-order necessary conditions.

But with the CRRA preferences that we have assumed, individual consumptions vary proportionately with aggregate consumption and therefore with the aggregate endowment.

- This is a consequence of our preference specification implying that **Engle curves** are affine in wealth and therefore satisfy conditions for **Gorman aggregation**

Thus,

$$c^k(s) = \alpha_k c(s) = \alpha_k y(s)$$

for an arbitrary **distribution of wealth** in the form of an $K \times 1$ vector α that satisfies

$$\alpha_k \in (0, 1), \quad \sum_{k=1}^{K} \alpha_k = 1$$

This means that we can compute the pricing kernel from

$$Q_{ij} = \beta \left(\frac{y_j}{y_i}\right)^{-\gamma} P_{ij} \tag{86.1}$$

Note that Q_{ij} is independent of vector α.

Key finding: We can compute competitive equilibrium **prices** prior to computing a **distribution of wealth**.

86.6.4 Values

Having computed an equilibrium pricing kernel Q, we can compute several **values** that are required to pose or represent the solution of an individual household's optimum problem.

We denote an $K \times 1$ vector of state-dependent values of agents' endowments in Markov state s as

$$A(s) = \begin{bmatrix} A^1(s) \\ \vdots \\ A^K(s) \end{bmatrix}, \quad s \in [\bar{s}_1, \ldots, \bar{s}_n]$$

and an $n \times 1$ vector of continuation endowment values for each individual k as

$$A^k = \begin{bmatrix} A^k(\bar{s}_1) \\ \vdots \\ A^k(\bar{s}_n) \end{bmatrix}, \quad k \in [1, \ldots, K]$$

A^k of consumer k satisfies

$$A^k = [I - Q]^{-1} [y^k]$$

where

$$y^k = \begin{bmatrix} y^k(\bar{s}_1) \\ \vdots \\ y^k(\bar{s}_n) \end{bmatrix} \equiv \begin{bmatrix} y_1^k \\ \vdots \\ y_n^k \end{bmatrix}$$

In a competitive equilibrium of an **infinite horizon** economy with sequential trading of one-period Arrow securities, $A^k(s)$ serves as a state-by-state vector of **debt limits** on the quantities of one-period Arrow securities paying off in state s at time $t+1$ that individual k can issue at time t.

These are often called **natural debt limits**.

Evidently, they equal the maximum amount that it is feasible for individual k to repay even if he consumes zero goods forevermore.

Remark: If we have an Inada condition at zero consumption or just impose that consumption be nonnegative, then in a **finite horizon** economy with sequential trading of one-period Arrow securities there is no need to impose natural debt limits. See the section on a Finite Horizon Economy below.

86.6.5 Continuation Wealth

Continuation wealth plays an important role in Bellmanizing a competitive equilibrium with sequential trading of a complete set of one-period Arrow securities.

We denote an $K \times 1$ vector of state-dependent continuation wealths in Markov state s as

$$\psi(s) = \begin{bmatrix} \psi^1(s) \\ \vdots \\ \psi^K(s) \end{bmatrix}, \quad s \in [\bar{s}_1, \ldots, \bar{s}_n]$$

and an $n \times 1$ vector of continuation wealths for each individual k as

$$\psi^k = \begin{bmatrix} \psi^k(\bar{s}_1) \\ \vdots \\ \psi^k(\bar{s}_n) \end{bmatrix}, \quad k \in [1, \ldots, K]$$

Continuation wealth ψ^k of consumer k satisfies

$$\psi^k = [I - Q]^{-1} [\alpha_k y - y^k] \tag{86.2}$$

where

$$y^k = \begin{bmatrix} y^k(\bar{s}_1) \\ \vdots \\ y^k(\bar{s}_n) \end{bmatrix}, \quad y = \begin{bmatrix} y(\bar{s}_1) \\ \vdots \\ y(\bar{s}_n) \end{bmatrix}$$

Note that $\sum_{k=1}^K \psi^k = 0_{n\times 1}$.

Remark: At the initial state $s_0 \in [\bar{s}_1, \ldots, \bar{s}_n]$, the continuation wealth $\psi^k(s_0) = 0$ for all agents $k = 1, \ldots, K$. This indicates that the economy begins with all agents being debt-free and financial-asset-free at time 0, state s_0.

Remark: Note that all agents' continuation wealths recurrently return to zero when the Markov state returns to whatever value s_0 it had at time 0.

86.6.6 Optimal Portfolios

A nifty feature of the model is that an optimal portfolio of a type k agent equals the continuation wealth that we just computed.

Thus, agent k's state-by-state purchases of Arrow securities next period depend only on next period's Markov state and equal

$$a_k(s) = \psi^k(s), \quad s \in [\bar{s}_1, \ldots, \bar{s}_n] \tag{86.3}$$

86.6.7 Equilibrium Wealth Distribution α

With the initial state being a particular state $s_0 \in [\bar{s}_1, \ldots, \bar{s}_n]$, we must have

$$\psi^k(s_0) = 0, \quad k = 1, \ldots, K$$

which means the equilibrium distribution of wealth satisfies

$$\alpha_k = \frac{V_z y^k}{V_z y} \tag{86.4}$$

where $V \equiv [I - Q]^{-1}$ and z is the row index corresponding to the initial state s_0.

Since $\sum_{k=1}^K V_z y^k = V_z y$, $\sum_{k=1}^K \alpha_k = 1$.

In summary, here is the logical flow of an algorithm to compute a competitive equilibrium:

- compute Q from the aggregate allocation and formula (86.1)
- compute the distribution of wealth α from the formula (86.4)
- Using α assign each consumer k the share α_k of the aggregate endowment at each state
- return to the α-dependent formula (86.2) and compute continuation wealths

- via formula (86.3) equate agent k's portfolio to its continuation wealth state by state

We can also add formulas for optimal value functions in a competitive equilibrium with trades in a complete set of one-period state-contingent Arrow securities.

Call the optimal value functions J^k for consumer k.

For the infinite horizon economy now under study, the formula is

$$J^k = (I - \beta P)^{-1} u(\alpha_k y), \quad u(c) = \frac{c^{1-\gamma}}{1-\gamma}$$

where it is understood that $u(\alpha_k y)$ is a vector.

86.7 Finite Horizon

We now describe a finite-horizon version of the economy that operates for $T+1$ periods $t \in \mathbf{T} = \{0, 1, \dots, T\}$.

Consequently, we'll want $T+1$ counterparts to objects described above, with one important exception: we won't need **borrowing limits**.

- borrowing limits aren't required for a finite horizon economy in which a one-period utility function $u(c)$ satisfies an Inada condition that sets the marginal utility of consumption at zero consumption to zero.
- Nonnegativity of consumption choices at all $t \in \mathbf{T}$ automatically limits borrowing.

86.7.1 Continuation Wealths

We denote a $K \times 1$ vector of state-dependent continuation wealths in Markov state s at time t as

$$\psi_t(s) = \begin{bmatrix} \psi^1(s) \\ \vdots \\ \psi^K(s) \end{bmatrix}, \quad s \in [\bar{s}_1, \dots, \bar{s}_n]$$

and an $n \times 1$ vector of continuation wealths for each individual k as

$$\psi_t^k = \begin{bmatrix} \psi_t^k(\bar{s}_1) \\ \vdots \\ \psi_t^k(\bar{s}_n) \end{bmatrix}, \quad k \in [1, \dots, K]$$

Continuation wealths ψ^k of consumer k satisfy

$$\begin{aligned} \psi_T^k &= \left[\alpha_k y - y^k\right] \\ \psi_{T-1}^k &= \left[I + Q\right]\left[\alpha_k y - y^k\right] \\ \vdots \quad & \quad \vdots \\ \psi_0^k &= \left[I + Q + Q^2 + \cdots + Q^T\right]\left[\alpha_k y - y^k\right] \end{aligned} \tag{86.5}$$

where

$$y^k = \begin{bmatrix} y^k(\bar{s}_1) \\ \vdots \\ y^k(\bar{s}_n) \end{bmatrix}, \quad y = \begin{bmatrix} y(\bar{s}_1) \\ \vdots \\ y(\bar{s}_n) \end{bmatrix}$$

Note that $\sum_{k=1}^K \psi_t^k = 0_{n \times 1}$ for all $t \in \mathbf{T}$.

Remark: At the initial state $s_0 \in [\bar{s}_1, \dots, \bar{s}_n]$, for all agents $k = 1, \dots, K$, continuation wealth $\psi_0^k(s_0) = 0$. This indicates that the economy begins with all agents being debt-free and financial-asset-free at time 0, state s_0.

Remark: Note that all agents' continuation wealths return to zero when the Markov state returns to whatever value s_0 it had at time 0. This will recur if the Markov chain makes the initial state s_0 recurrent.

With the initial state being a particular state $s_0 \in [\bar{s}_1, \ldots, \bar{s}_n]$, we must have

$$\psi_0^k(s_0) = 0, \quad k = 1, \ldots, K$$

which means the equilibrium distribution of wealth satisfies

$$\alpha_k = \frac{V_z y^k}{V_z y} \tag{86.6}$$

where now in our finite-horizon economy

$$V = [I + Q + Q^2 + \cdots + Q^T] \tag{86.7}$$

and z is the row index corresponding to the initial state s_0.

Since $\sum_{k=1}^K V_z y^k = V_z y$, $\sum_{k=1}^K \alpha_k = 1$.

In summary, here is the logical flow of an algorithm to compute a competitive equilibrium with Arrow securities in our finite-horizon Markov economy:

- compute Q from the aggregate allocation and formula (86.1)
- compute the distribution of wealth α from formulas (86.6) and (86.7)
- using α, assign each consumer k the share α_k of the aggregate endowment at each state
- return to the α-dependent formula (86.5) for continuation wealths and compute continuation wealths
- equate agent k's portfolio to its continuation wealth state by state

While for the infinite horizon economy, the formula for value functions is

$$J^k = (I - \beta P)^{-1} u(\alpha_k y), \quad u(c) = \frac{c^{1-\gamma}}{1-\gamma}$$

for the finite horizon economy the formula is

$$J_0^k = (I + \beta P + \cdots + \beta^T P^T) u(\alpha_k y),$$

where it is understood that $u(\alpha_k y)$ is a vector.

86.8 Python Code

We are ready to dive into some Python code.

As usual, we start with Python imports.

```python
import numpy as np
import matplotlib.pyplot as plt
```

```python
np.set_printoptions(suppress=True)
```

First, we create a Python class to compute the objects that comprise a competitive equilibrium with sequential trading of one-period Arrow securities.

In addition to infinite-horizon economies, the code is set up to handle finite-horizon economies indexed by horizon T.

We'll study examples of finite horizon economies after we first look at some infinite-horizon economies.

```
class RecurCompetitive:
    """
    A class that represents a recursive competitive economy
    with one-period Arrow securities.
    """

    def __init__(self,
                 s,         # state vector
                 P,         # transition matrix
                 ys,        # endowments ys = [y1, y2, .., yI]
                 γ=0.5,     # risk aversion
                 β=0.98,    # discount rate
                 T=None):   # time horizon, none if infinite

        # preference parameters
        self.γ = γ
        self.β = β

        # variables dependent on state
        self.s = s
        self.P = P
        self.ys = ys
        self.y = np.sum(ys, 1)

        # dimensions
        self.n, self.K = ys.shape

        # compute pricing kernel
        self.Q = self.pricing_kernel()

        # compute price of risk-free one-period bond
        self.PRF = self.price_risk_free_bond()

        # compute risk-free rate
        self.R = self.risk_free_rate()

        # V = [I - Q]^{-1} (infinite case)
        if T is None:
            self.T = None
            self.V = np.empty((1, n, n))
            self.V[0] = np.linalg.inv(np.eye(n) - self.Q)
        # V = [I + Q + Q^2 + ... + Q^T] (finite case)
        else:
            self.T = T
            self.V = np.empty((T+1, n, n))
            self.V[0] = np.eye(n)

            Qt = np.eye(n)
            for t in range(1, T+1):
                Qt = Qt.dot(self.Q)
                self.V[t] = self.V[t-1] + Qt

        # natural debt limit
        self.A = self.V[-1] @ ys

    def u(self, c):
        "The CRRA utility"
```

(continues on next page)

(continued from previous page)

```
        return c ** (1 - self.γ) / (1 - self.γ)

    def u_prime(self, c):
        "The first derivative of CRRA utility"

        return c ** (-self.γ)

    def pricing_kernel(self):
        "Compute the pricing kernel matrix Q"

        c = self.y

        n = self.n
        Q = np.empty((n, n))
        for i in range(n):
            for j in range(n):
                ratio = self.u_prime(c[j]) / self.u_prime(c[i])
                Q[i, j] = self.β * ratio * P[i, j]

        self.Q = Q

        return Q

    def wealth_distribution(self, s0_idx):
        "Solve for wealth distribution α"

        # set initial state
        self.s0_idx = s0_idx

        # simplify notations
        n = self.n
        Q = self.Q
        y, ys = self.y, self.ys

        # row of V corresponding to s0
        Vs0 = self.V[-1, s0_idx, :]
        α = Vs0 @ self.ys / (Vs0 @ self.y)

        self.α = α

        return α

    def continuation_wealths(self):
        "Given α, compute the continuation wealths ψ"

        diff = np.empty((n, K))
        for k in range(K):
            diff[:, k] = self.α[k] * self.y - self.ys[:, k]

        ψ = self.V @ diff
        self.ψ = ψ

        return ψ

    def price_risk_free_bond(self):
        "Give Q, compute price of one-period risk free bond"
```

(continues on next page)

(continued from previous page)

```
        PRF = np.sum(self.Q, axis=1)
        self.PRF = PRF

        return PRF

    def risk_free_rate(self):
        "Given Q, compute one-period gross risk-free interest rate R"

        R = np.sum(self.Q, axis=1)
        R = np.reciprocal(R)
        self.R = R

        return R

    def value_functionss(self):
        "Given α, compute the optimal value functions J in equilibrium"

        n, T = self.n, self.T
        β = self.β
        P = self.P

        # compute (I - βP)^(-1) in infinite case
        if T is None:
            P_seq = np.empty((1, n, n))
            P_seq[0] = np.linalg.inv(np.eye(n) - β * P)
        # and (I + βP + ... + β^T P^T) in finite case
        else:
            P_seq = np.empty((T+1, n, n))
            P_seq[0] = np.eye(n)

            Pt = np.eye(n)
            for t in range(1, T+1):
                Pt = Pt.dot(P)
                P_seq[t] = P_seq[t-1] + Pt * β ** t

        # compute the matrix [u(α_1 y), ..., u(α_K, y)]
        flow = np.empty((n, K))
        for k in range(K):
            flow[:, k] = self.u(self.α[k] * self.y)

        J = P_seq @ flow

        self.J = J

        return J
```

86.9 Examples

We'll use our code to construct equilibrium objects in several example economies.

Our first several examples will be infinite horizon economies.

Our final example will be a finite horizon economy.

86.9.1 Example 1

Please read the preceding class for default parameter values and the following Python code for the fundamentals of the economy.

Here goes.

```
# dimensions
K, n = 2, 2

# states
s = np.array([0, 1])

# transition
P = np.array([[.5, .5], [.5, .5]])

# endowments
ys = np.empty((n, K))
ys[:, 0] = 1 - s         # y1
ys[:, 1] = s             # y2
```

```
ex1 = RecurCompetitive(s, P, ys)
```

```
# endowments
ex1.ys
```

```
array([[1., 0.],
       [0., 1.]])
```

```
# pricing kernal
ex1.Q
```

```
array([[0.49, 0.49],
       [0.49, 0.49]])
```

```
# Risk free rate R
ex1.R
```

```
array([1.02040816, 1.02040816])
```

```
# natural debt limit, A = [A1, A2, ..., AI]
ex1.A
```

```
array([[25.5, 24.5],
       [24.5, 25.5]])
```

```
# when the initial state is state 1
print(f'α = {ex1.wealth_distribution(s0_idx=0)}')
print(f'ψ = \n{ex1.continuation_wealths()}')
print(f'J = \n{ex1.value_functionss()}')
```

```
α = [0.51 0.49]
ψ =
[[[-0.  0.]
  [ 1. -1.]]]
J =
[[[71.41428429 70.        ]
  [71.41428429 70.        ]]]
```

```
# when the initial state is state 2
print(f'α = {ex1.wealth_distribution(s0_idx=1)}')
print(f'ψ = \n{ex1.continuation_wealths()}')
print(f'J = \n{ex1.value_functionss()}')
```

```
α = [0.49 0.51]
ψ =
[[[-1.  1.]
  [ 0. -0.]]]
J =
[[[70.         71.41428429]
  [70.         71.41428429]]]
```

86.9.2 Example 2

```
# dimensions
K, n = 2, 2

# states
s = np.array([1, 2])

# transition
P = np.array([[.5, .5], [.5, .5]])

# endowments
ys = np.empty((n, K))
ys[:, 0] = 1.5          # y1
ys[:, 1] = s            # y2
```

```
ex2 = RecurCompetitive(s, P, ys)
```

```
# endowments

print("ys = \n", ex2.ys)

# pricing kernal
print ("Q = \n", ex2.Q)

# Risk free rate R
print("R = ", ex2.R)
```

```
ys =
 [[1.5 1. ]
 [1.5 2. ]]
Q =
 [[0.49       0.41412558]
 [0.57977582 0.49      ]]
R =  [1.10604104 0.93477529]
```

```
# pricing kernal
ex2.Q
```

```
array([[0.49      , 0.41412558],
       [0.57977582, 0.49      ]])
```

Note that the pricing kernal in example economies 1 and 2 differ.

This comes from differences in the aggregate endowments in state 1 and 2 in example 1.

```
ex2.β * ex2.u_prime(3.5) / ex2.u_prime(2.5) * ex2.P[0,1]
```

```
np.float64(0.4141255848169731)
```

```
ex2.β * ex2.u_prime(2.5) / ex2.u_prime(3.5) * ex2.P[1,0]
```

```
np.float64(0.5797758187437624)
```

```
# Risk free rate R
ex2.R
```

```
array([1.10604104, 0.93477529])
```

```
# natural debt limit, A = [A1, A2, ..., AI]
ex2.A
```

```
array([[69.30941886, 66.91255848],
       [81.73318641, 79.98879094]])
```

```
# when the initial state is state 1
print(f'α = {ex2.wealth_distribution(s0_idx=0)}')
print(f'ψ = \n{ex2.continuation_wealths()}')
print(f'J = \n{ex2.value_functionss()}')
```

```
α = [0.50879763 0.49120237]
ψ =
[[[ 0.          0.        ]
  [ 0.55057195 -0.55057195]]]
J =
[[[122.907875   120.76397493]
  [123.32114686 121.17003803]]]
```

```
# when the initial state is state 1
print(f'α = {ex2.wealth_distribution(s0_idx=1)}')
print(f'ψ = \n{ex2.continuation_wealths()}')
print(f'J = \n{ex2.value_functionss()}')
```

```
α = [0.50539319 0.49460681]
ψ =
[[[-0.46375886  0.46375886]
  [-0.          -0.        ]]]
J =
[[[122.49598809 121.18174895]
  [122.907875   121.58921679]]]
```

86.9.3 Example 3

```
# dimensions
K, n = 2, 2

# states
s = np.array([1, 2])

# transition
λ = 0.9
P = np.array([[1-λ, λ], [0, 1]])

# endowments
ys = np.empty((n, K))
ys[:, 0] = [1, 0]          # y1
ys[:, 1] = [0, 1]          # y2
```

```
ex3 = RecurCompetitive(s, P, ys)
```

```
# endowments

print("ys = ", ex3.ys)

# pricing kernel
print ("Q = ", ex3.Q)

# Risk free rate R
print("R = ", ex3.R)
```

```
ys =  [[1. 0.]
 [0. 1.]]
Q =  [[0.098 0.882]
 [0.    0.98 ]]
R =  [1.02040816 1.02040816]
```

```
# pricing kernel
ex3.Q
```

```
array([[0.098, 0.882],
       [0.   , 0.98 ]])
```

```
# natural debt limit, A = [A1, A2, ..., AI]
ex3.A
```

```
array([[ 1.10864745, 48.89135255],
       [ 0.        , 50.        ]])
```

Note that the natural debt limit for agent 1 in state 2 is 0.

```
# when the initial state is state 1
print(f'α = {ex3.wealth_distribution(s0_idx=0)}')
print(f'ψ = \n{ex3.continuation_wealths()}')
print(f'J = \n{ex3.value_functionss()}')
```

```
α = [0.02217295 0.97782705]
ψ =
[[[ 0.          -0.        ]
  [ 1.10864745 -1.10864745]]]
J =
[[[14.89058394 98.88513796]
  [14.89058394 98.88513796]]]
```

```
# when the initial state is state 1
print(f'α = {ex3.wealth_distribution(s0_idx=1)}')
print(f'ψ = \n{ex3.continuation_wealths()}')
print(f'J = \n{ex3.value_functionss()}')
```

```
α = [0. 1.]
ψ =
[[[-1.10864745  1.10864745]
  [ 0.          0.        ]]]
J =
[[[  0. 100.]
  [  0. 100.]]]
```

For the specification of the Markov chain in example 3, let's take a look at how the equilibrium allocation changes as a function of transition probability λ.

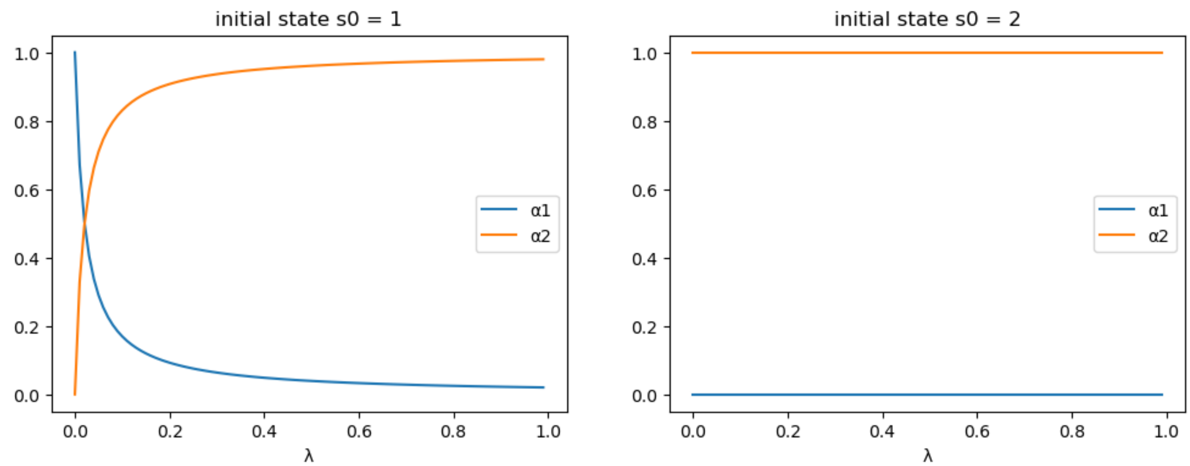
```
λ_seq = np.linspace(0, 0.99, 100)

# prepare containers
αs0_seq = np.empty((len(λ_seq), 2))
αs1_seq = np.empty((len(λ_seq), 2))

for i, λ in enumerate(λ_seq):
    P = np.array([[1-λ, λ], [0, 1]])
    ex3 = RecurCompetitive(s, P, ys)

    # initial state s0 = 1
    α = ex3.wealth_distribution(s0_idx=0)
    αs0_seq[i, :] = α

    # initial state s0 = 2
    α = ex3.wealth_distribution(s0_idx=1)
    αs1_seq[i, :] = α
```

```
fig, axs = plt.subplots(1, 2, figsize=(12, 4))

for i, αs_seq in enumerate([αs0_seq, αs1_seq]):
    for j in range(2):
```

(continues on next page)

(continued from previous page)

```
        axs[i].plot(λ_seq, αs_seq[:, j], label=f'α{j+1}')
        axs[i].set_xlabel('λ')
        axs[i].set_title(f'initial state s0 = {s[i]}')
        axs[i].legend()

plt.show()
```



86.9.4 Example 4

```
# dimensions
K, n = 2, 3

# states
s = np.array([1, 2, 3])

# transition
λ = .9
μ = .9
δ = .05

# prosperous, moderate, and recession states
P = np.array([[1-λ, λ, 0], [μ/2, μ, μ/2], [(1-δ)/2, (1-δ)/2, δ]])

# endowments
ys = np.empty((n, K))
ys[:, 0] = [.25, .75, .2]       # y1
ys[:, 1] = [1.25, .25, .2]      # y2
```

```
ex4 = RecurCompetitive(s, P, ys)
```

```
# endowments
print("ys = \n", ex4.ys)

# pricing kernal
print ("Q = \n", ex4.Q)

# Risk free rate R
```

(continues on next page)

(continued from previous page)

```
print("R = ", ex4.R)

# natural debt limit, A = [A1, A2, ..., AI]
print("A = \n", ex4.A)

print('')

for i in range(1, 4):
    # when the initial state is state i
    print(f"when the initial state is state {i}")
    print(f'α = {ex4.wealth_distribution(s0_idx=i-1)}')
    print(f'ψ = \n{ex4.continuation_wealths()}')
    print(f'J = \n{ex4.value_functionss()}\n')
```

```
ys =
 [[0.25 1.25]
 [0.75 0.25]
 [0.2  0.2 ]]
Q =
 [[0.098      1.08022498 0.        ]
 [0.36007499 0.882      0.69728222]
 [0.24038317 0.29440805 0.049     ]]
R =  [0.84873434 0.51563476 1.71294115]
A =
 [[-1.4141307  -0.45854174]
 [-1.4122483  -1.54005386]
 [-0.58434331 -0.3823659 ]]

when the initial state is state 1
α = [0.75514045 0.24485955]
ψ =
[[[ 0.          0.        ]
  [-0.81715447  0.81715447]
  [-0.14565791  0.14565791]]]
J =
[[[-2.65741909 -1.51322919]
  [-5.13103133 -2.92179221]
  [-2.65649938 -1.51270548]]]

when the initial state is state 2
α = [0.47835493 0.52164507]
ψ =
[[[ 0.5183286  -0.5183286 ]
  [ 0.         -0.        ]
  [ 0.12191319 -0.12191319]]]
J =
[[[-2.11505328 -2.20868477]
  [-4.08381377 -4.26460049]
  [-2.11432128 -2.20792037]]]

when the initial state is state 3
α = [0.60446648 0.39553352]
ψ =
[[[ 0.28216299 -0.28216299]
  [-0.37231938  0.37231938]
  [ 0.         -0.        ]]]
```

(continues on next page)

(continued from previous page)

```
J =
[[[-2.37756442 -1.92325926]
  [-4.59067883 -3.71349163]
  [-2.37674158 -1.92259365]]]
```

86.9.5 Finite Horizon Example

We now revisit the economy defined in example 1, but set the time horizon to be $T = 10$.

```
# dimensions
K, n = 2, 2

# states
s = np.array([0, 1])

# transition
P = np.array([[.5, .5], [.5, .5]])

# endowments
ys = np.empty((n, K))
ys[:, 0] = 1 - s       # y1
ys[:, 1] = s           # y2
```

```
ex1_finite = RecurCompetitive(s, P, ys, T=10)
```

```
# (I + Q + Q^2 + ... + Q^T)
ex1_finite.V[-1]
```

```
array([[5.48171623, 4.48171623],
       [4.48171623, 5.48171623]])
```

```
# endowments
ex1_finite.ys
```

```
array([[1., 0.],
       [0., 1.]])
```

```
# pricing kernal
ex1_finite.Q
```

```
array([[0.49, 0.49],
       [0.49, 0.49]])
```

```
# Risk free rate R
ex1_finite.R
```

```
array([1.02040816, 1.02040816])
```

In the finite time horizon case, ψ and J are returned as sequences.

Components are ordered from $t = T$ to $t = 0$.

```python
# when the initial state is state 2
print(f'α = {ex1_finite.wealth_distribution(s0_idx=0)}')
print(f'ψ = \n{ex1_finite.continuation_wealths()}\n')
print(f'J = \n{ex1_finite.value_functionss()}')
```

```
α = [0.55018351 0.44981649]
ψ =
[[[-0.44981649  0.44981649]
  [ 0.55018351 -0.55018351]]

 [[-0.40063665  0.40063665]
  [ 0.59936335 -0.59936335]]

 [[-0.35244041  0.35244041]
  [ 0.64755959 -0.64755959]]

 [[-0.30520809  0.30520809]
  [ 0.69479191 -0.69479191]]

 [[-0.25892042  0.25892042]
  [ 0.74107958 -0.74107958]]

 [[-0.21355851  0.21355851]
  [ 0.78644149 -0.78644149]]

 [[-0.16910383  0.16910383]
  [ 0.83089617 -0.83089617]]

 [[-0.12553824  0.12553824]
  [ 0.87446176 -0.87446176]]

 [[-0.08284397  0.08284397]
  [ 0.91715603 -0.91715603]]

 [[-0.04100358  0.04100358]
  [ 0.95899642 -0.95899642]]

 [[-0.          -0.        ]
  [ 1.          -1.        ]]]

J =
[[[ 1.48348712  1.3413672 ]
  [ 1.48348712  1.3413672 ]]

 [[ 2.9373045   2.65590706]
  [ 2.9373045   2.65590706]]

 [[ 4.36204553  3.94415611]
  [ 4.36204553  3.94415611]]

 [[ 5.75829174  5.20664019]
  [ 5.75829174  5.20664019]]

 [[ 7.12661302  6.44387459]
  [ 7.12661302  6.44387459]]

 [[ 8.46756788  7.6563643 ]
```

(continues on next page)

(continued from previous page)

```
 [ 8.46756788  7.6563643 ]]

[[ 9.78170364  8.84460421]
 [ 9.78170364  8.84460421]]

[[11.06955669 10.00907933]
 [11.06955669 10.00907933]]

[[12.33165268 11.15026494]
 [12.33165268 11.15026494]]

[[13.56850674 12.26862684]
 [13.56850674 12.26862684]]

[[14.78062373 13.3646215 ]
 [14.78062373 13.3646215 ]]]
```

```python
# when the initial state is state 2
print(f'α = {ex1_finite.wealth_distribution(s0_idx=1)}')
print(f'ψ = \n{ex1_finite.continuation_wealths()}\n')
print(f'J = \n{ex1_finite.value_functionss()}')
```

```
α = [0.44981649 0.55018351]
ψ =
[[[-0.55018351  0.55018351]
  [ 0.44981649 -0.44981649]]

 [[-0.59936335  0.59936335]
  [ 0.40063665 -0.40063665]]

 [[-0.64755959  0.64755959]
  [ 0.35244041 -0.35244041]]

 [[-0.69479191  0.69479191]
  [ 0.30520809 -0.30520809]]

 [[-0.74107958  0.74107958]
  [ 0.25892042 -0.25892042]]

 [[-0.78644149  0.78644149]
  [ 0.21355851 -0.21355851]]

 [[-0.83089617  0.83089617]
  [ 0.16910383 -0.16910383]]

 [[-0.87446176  0.87446176]
  [ 0.12553824 -0.12553824]]

 [[-0.91715603  0.91715603]
  [ 0.08284397 -0.08284397]]

 [[-0.95899642  0.95899642]
  [ 0.04100358 -0.04100358]]

 [[-1.          1.        ]
  [-0.         -0.        ]]]
```

(continues on next page)

(continued from previous page)

```
J =
[[[ 1.3413672   1.48348712]
  [ 1.3413672   1.48348712]]

 [[ 2.65590706  2.9373045 ]
  [ 2.65590706  2.9373045 ]]

 [[ 3.94415611  4.36204553]
  [ 3.94415611  4.36204553]]

 [[ 5.20664019  5.75829174]
  [ 5.20664019  5.75829174]]

 [[ 6.44387459  7.12661302]
  [ 6.44387459  7.12661302]]

 [[ 7.6563643   8.46756788]
  [ 7.6563643   8.46756788]]

 [[ 8.84460421  9.78170364]
  [ 8.84460421  9.78170364]]

 [[10.00907933 11.06955669]
  [10.00907933 11.06955669]]

 [[11.15026494 12.33165268]
  [11.15026494 12.33165268]]

 [[12.26862684 13.56850674]
  [12.26862684 13.56850674]]

 [[13.3646215  14.78062373]
  [13.3646215  14.78062373]]]
```

We can check the results with finite horizon converges to the ones with infinite horizon as $T \to \infty$.

```
ex1_large = RecurCompetitive(s, P, ys, T=10000)
ex1_large.wealth_distribution(s0_idx=1)
```

```
array([0.49, 0.51])
```

```
ex1.V, ex1_large.V[-1]
```

```
(array([[[25.5, 24.5],
         [24.5, 25.5]]]),
 array([[25.5, 24.5],
        [24.5, 25.5]]))
```

```
ex1_large.continuation_wealths()
ex1.ψ, ex1_large.ψ[-1]
```

```
(array([[[-1.,  1.],
         [ 0., -0.]]]),
 array([[-1.,  1.],
```

(continues on next page)

(continued from previous page)

```
        [ 0., -0.]]))
```

```
ex1_large.value_functionss()
ex1.J, ex1_large.J[-1]
```

```
(array([[[70.        , 71.41428429],
          [70.        , 71.41428429]]]),
 array([[70.        , 71.41428429],
        [70.        , 71.41428429]]))
```

CHAPTER

EIGHTYSEVEN

HETEROGENEOUS BELIEFS AND BUBBLES

Contents

In addition to what's in Anaconda, this lecture uses following libraries:

```
!pip install quantecon
```

87.1 Overview

This lecture describes a version of a model of Harrison and Kreps [Harrison and Kreps, 1978].

The model determines the price of a dividend-yielding asset that is traded by two types of self-interested investors.

The model features

- heterogeneous beliefs
- incomplete markets
- short sales constraints, and possibly …
- (leverage) limits on an investor's ability to borrow in order to finance purchases of a risky asset

Let's start with some standard imports:

```
import numpy as np
import quantecon as qe
import scipy.linalg as la
```

87.1.1 References

Prior to reading the following, you might like to review our lectures on

- *Markov chains*
- *Asset pricing with finite state space*

87.1.2 Bubbles

Economists differ in how they define a *bubble*.

The Harrison-Kreps model illustrates the following notion of a bubble that attracts many economists:

> *A component of an asset price can be interpreted as a bubble when all investors agree that the current price of the asset exceeds what they believe the asset's underlying dividend stream justifies.*

87.2 Structure of the Model

The model simplifies things by ignoring alterations in the distribution of wealth among investors who have hard-wired different beliefs about the fundamentals that determine asset payouts.

There is a fixed number A of shares of an asset.

Each share entitles its owner to a stream of dividends $\{d_t\}$ governed by a Markov chain defined on a state space $S \in \{0, 1\}$.

The dividend obeys

$$d_t = \begin{cases} 0 & \text{if } s_t = 0 \\ 1 & \text{if } s_t = 1 \end{cases}$$

An owner of a share at the end of time t and the beginning of time $t+1$ is entitled to the dividend paid at time $t+1$.

Thus, the stock is traded **ex dividend**.

An owner of a share at the beginning of time $t+1$ is also entitled to sell the share to another investor during time $t+1$.

Two types $h = a, b$ of investors differ only in their beliefs about a Markov transition matrix P with typical element

$$P(i, j) = \mathbb{P}\{s_{t+1} = j \mid s_t = i\}$$

Investors of type a believe the transition matrix

$$P_a = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{2}{3} & \frac{1}{3} \end{bmatrix}$$

Investors of type b think the transition matrix is

$$P_b = \begin{bmatrix} \frac{2}{3} & \frac{1}{3} \\ \frac{1}{4} & \frac{3}{4} \end{bmatrix}$$

Thus, in state 0, a type a investor is more optimistic about next period's dividend than is investor b.

But in state 1, a type a investor is more pessimistic about next period's dividend than is investor b.

The stationary (i.e., invariant) distributions of these two matrices can be calculated as follows:

```
qa = np.array([[1/2, 1/2], [2/3, 1/3]])
qb = np.array([[2/3, 1/3], [1/4, 3/4]])
mca = qe.MarkovChain(qa)
mcb = qe.MarkovChain(qb)
mca.stationary_distributions
```

```
array([[0.57142857, 0.42857143]])
```

```
mcb.stationary_distributions
```

```
array([[0.42857143, 0.57142857]])
```

The stationary distribution of P_a is approximately $\pi_a = \begin{bmatrix} .57 & .43 \end{bmatrix}$.

The stationary distribution of P_b is approximately $\pi_b = \begin{bmatrix} .43 & .57 \end{bmatrix}$.

Thus, a type a investor is more pessimistic on average.

87.2.1 Ownership Rights

An owner of the asset at the end of time t is entitled to the dividend at time $t + 1$ and also has the right to sell the asset at time $t + 1$.

Both types of investors are risk-neutral and both have the same fixed discount factor $\beta \in (0, 1)$.

In our numerical example, we'll set $\beta = .75$, just as Harrison and Kreps [Harrison and Kreps, 1978] did.

We'll eventually study the consequences of two alternative assumptions about the number of shares A relative to the resources that our two types of investors can invest in the stock.

1. Both types of investors have enough resources (either wealth or the capacity to borrow) so that they can purchase the entire available stock of the asset[1].
2. No single type of investor has sufficient resources to purchase the entire stock.

Case 1 is the case studied in Harrison and Kreps.

In case 2, both types of investors always hold at least some of the asset.

87.2.2 Short Sales Prohibited

No short sales are allowed.

This matters because it limits how pessimists can express their opinions.

- They **can** express themselves by selling their shares.
- They **cannot** express themsevles more loudly by artificially "manufacturing shares" – that is, they cannot borrow shares from more optimistic investors and then immediately sell them.

[1] By assuming that both types of agents always have "deep enough pockets" to purchase all of the asset, the model takes wealth dynamics off the table. The Harrison-Kreps model generates high trading volume when the state changes either from 0 to 1 or from 1 to 0.

87.2.3 Optimism and Pessimism

The above specifications of the perceived transition matrices P_a and P_b, taken directly from Harrison and Kreps, build in stochastically alternating temporary optimism and pessimism.

Remember that state 1 is the high dividend state.

- In state 0, a type a agent is more optimistic about next period's dividend than a type b agent.
- In state 1, a type b agent is more optimistic about next period's dividend than a type a agaub is.

However, the stationary distributions $\pi_a = \begin{bmatrix} .57 & .43 \end{bmatrix}$ and $\pi_b = \begin{bmatrix} .43 & .57 \end{bmatrix}$ tell us that a type b person is more optimistic about the dividend process in the long run than is a type a person.

87.2.4 Information

Investors know a price function mapping the state s_t at t into the equilibrium price $p(s_t)$ that prevails in that state.

This price function is endogenous and to be determined below.

When investors choose whether to purchase or sell the asset at t, they also know s_t.

87.3 Solving the Model

Now let's turn to solving the model.

We'll determine equilibrium prices under a particular specification of beliefs and constraints on trading selected from one of the specifications described above.

We shall compare equilibrium price functions under the following alternative assumptions about beliefs:

1. There is only one type of agent, either a or b.
2. There are two types of agents differentiated only by their beliefs. Each type of agent has sufficient resources to purchase all of the asset (Harrison and Kreps's setting).
3. There are two types of agents with different beliefs, but because of limited wealth and/or limited leverage, both types of investors hold the asset each period.

87.3.1 Summary Table

The following table gives a summary of the findings obtained in the remainder of the lecture (in an exercise you will be asked to recreate the table and also reinterpret parts of it).

The table reports implications of Harrison and Kreps's specifications of P_a, P_b, β.

s_t	0	1
p_a	1.33	1.22
p_b	1.45	1.91
p_o	1.85	2.08
p_p	1	1
$\hat{p}_a$	1.85	1.69
$\hat{p}_b$	1.69	2.08

Here

- p_a is the equilibrium price function under homogeneous beliefs P_a
- p_b is the equilibrium price function under homogeneous beliefs P_b
- p_o is the equilibrium price function under heterogeneous beliefs with optimistic marginal investors
- p_p is the equilibrium price function under heterogeneous beliefs with pessimistic marginal investors
- $\hat{p}_a$ is the amount type a investors are willing to pay for the asset
- $\hat{p}_b$ is the amount type b investors are willing to pay for the asset

We'll explain these values and how they are calculated one row at a time.

The row corresponding to p_o applies when both types of investor have enough resources to purchase the entire stock of the asset and strict short sales constraints prevail so that temporarily optimistic investors always price the asset.

The row corresponding to p_p would apply if neither type of investor has enough resources to purchase the entire stock of the asset and both types must hold the asset.

The row corresponding to p_p would also apply if both types have enough resources to buy the entire stock of the asset but short sales are also possible so that temporarily pessimistic investors price the asset.

87.3.2 Single Belief Prices

We'll start by pricing the asset under homogeneous beliefs.

(This is the case treated in *the lecture* on asset pricing with finite Markov states)

Suppose that there is only one type of investor, either of type a or b, and that this investor always "prices the asset".

Let $p_h = \begin{bmatrix} p_h(0) \\ p_h(1) \end{bmatrix}$ be the equilibrium price vector when all investors are of type h.

The price today equals the expected discounted value of tomorrow's dividend and tomorrow's price of the asset:

$$p_h(s) = \beta \left(P_h(s,0)(0 + p_h(0)) + P_h(s,1)(1 + p_h(1)) \right), \quad s = 0, 1 \tag{87.1}$$

These equations imply that the equilibrium price vector is

$$\begin{bmatrix} p_h(0) \\ p_h(1) \end{bmatrix} = \beta [I - \beta P_h]^{-1} P_h \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{87.2}$$

The first two rows of the table report $p_a(s)$ and $p_b(s)$.

Here's a function that can be used to compute these values

```
def price_single_beliefs(transition, dividend_payoff, β=.75):
    """
    Function to Solve Single Beliefs
    """
    # First compute inverse piece
    imbq_inv = la.inv(np.eye(transition.shape[0]) - β * transition)

    # Next compute prices
    prices = β * imbq_inv @ transition @ dividend_payoff

    return prices
```

Single Belief Prices as Benchmarks

These equilibrium prices under homogeneous beliefs are important benchmarks for the subsequent analysis.

- $p_h(s)$ tells what a type h investor thinks is the "fundamental value" of the asset.
- Here "fundamental value" means the expected discounted present value of future dividends.

We will compare these fundamental values of the asset with equilibrium values when traders have different beliefs.

87.3.3 Pricing under Heterogeneous Beliefs

There are several cases to consider.

The first is when both types of agents have sufficient wealth to purchase all of the asset themselves.

In this case, the marginal investor who prices the asset is the more optimistic type so that the equilibrium price $\bar p$ satisfies Harrison and Kreps's key equation:

$$\bar p(s) = \beta \max \left\{ P_a(s,0) \bar p(0) + P_a(s,1)(1 + \bar p(1)),\ P_b(s,0) \bar p(0) + P_b(s,1)(1 + \bar p(1)) \right\} \tag{87.3}$$

for $s = 0, 1$.

In the above equation, the max on the right side is over the two prospective values of next period's payout from owning the asset.

The marginal investor who prices the asset in state s is of type a if

$$P_a(s,0) \bar p(0) + P_a(s,1)(1 + \bar p(1)) > P_b(s,0) \bar p(0) + P_b(s,1)(1 + \bar p(1))$$

The marginal investor is of type b if

$$P_a(s,1) \bar p(0) + P_a(s,1)(1 + \bar p(1)) < P_b(s,1) \bar p(0) + P_b(s,1)(1 + \bar p(1))$$

Thus the marginal investor is the (temporarily) optimistic type.

Equation (87.3) is a functional equation that, like a Bellman equation, can be solved by

- starting with a guess for the price vector $\bar p$ and
- iterating to convergence on the operator that maps a guess $\bar p^j$ into an updated guess $\bar p^{j+1}$ defined by the right side of (87.3), namely

$$\bar p^{j+1}(s) = \beta \max \left\{ P_a(s,0) \bar p^j(0) + P_a(s,1)(1 + \bar p^j(1)),\ P_b(s,0) \bar p^j(0) + P_b(s,1)(1 + \bar p^j(1)) \right\} \tag{87.4}$$

for $s = 0, 1$.

The third row of the table labeled p_o reports equilibrium prices that solve the functional equation when $\beta = .75$.

Here the type that is optimistic about s_{t+1} prices the asset in state s_t.

It is instructive to compare these prices with the equilibrium prices for the homogeneous belief economies that solve under beliefs P_a and P_b reported in the rows labeled p_a and p_b, respectively.

Equilibrium prices p_o in the heterogeneous beliefs economy evidently exceed what any prospective investor regards as the fundamental value of the asset in each possible state.

Nevertheless, the economy recurrently visits a state that makes each investor want to purchase the asset for more than he believes its future dividends are worth.

An investor is willing to pay more than what he believes is warranted by fundamental value of the prospective dividend stream because he expects to have the option later to sell the asset to another investor who will value the asset more highly than he will then.

- Investors of type a are willing to pay the following price for the asset

$$\hat{p}_a(s) = \begin{cases} \bar{p}(0) & \text{if } s_t = 0 \\ \beta(P_a(1,0)\bar{p}(0) + P_a(1,1)(1+\bar{p}(1))) & \text{if } s_t = 1 \end{cases}$$

- Investors of type b are willing to pay the following price for the asset

$$\hat{p}_b(s) = \begin{cases} \beta(P_b(0,0)\bar{p}(0) + P_b(0,1)(1+\bar{p}(1))) & \text{if } s_t = 0 \\ \bar{p}(1) & \text{if } s_t = 1 \end{cases}$$

Evidently, $\hat{p}_a(1) < \bar{p}(1)$ and $\hat{p}_b(0) < \bar{p}(0)$.

Investors of type a want to sell the asset in state 1 while investors of type b want to sell it in state 0.

- The asset changes hands whenever the state changes from 0 to 1 or from 1 to 0.
- The valuations $\hat{p}_a(s)$ and $\hat{p}_b(s)$ are displayed in the fourth and fifth rows of the table.
- Even pessimistic investors who don't buy the asset think that it is worth more than they think future dividends are worth.

Here's code to solve for $\bar{p}$, $\hat{p}_a$ and $\hat{p}_b$ using the iterative method described above

```
def price_optimistic_beliefs(transitions, dividend_payoff, β=.75,
                             max_iter=50000, tol=1e-16):
    """
    Function to Solve Optimistic Beliefs
    """
    # We will guess an initial price vector of [0, 0]
    p_new = np.array([[0], [0]])
    p_old = np.array([[10.], [10.]])

    # We know this is a contraction mapping, so we can iterate to conv
    for i in range(max_iter):
        p_old = p_new
        p_new = β * np.max([q @ p_old
                            + q @ dividend_payoff for q in transitions],
                            axis=0)

        # If we succeed in converging, break out of for loop
        if np.max(np.sqrt((p_new - p_old)**2)) < tol:
            break

    ptwiddle = β * np.min([q @ p_old
                          + q @ dividend_payoff for q in transitions],
                          axis=0)

    phat_a = np.array([p_new[0], ptwiddle[1]])
    phat_b = np.array([ptwiddle[0], p_new[1]])

    return p_new, phat_a, phat_b
```

87.3.4 Insufficient Funds

Outcomes differ when the more optimistic type of investor has insufficient wealth — or insufficient ability to borrow enough — to hold the entire stock of the asset.

In this case, the asset price must adjust to attract pessimistic investors.

Instead of equation (87.3), the equilibrium price satisfies

$$\check p(s) = \beta \min \left\{ P_a(s,0) \check p(0) + P_a(s,1) (1 + \check p(1)) ,\ P_b(s,0) \check p(0) + P_b(s,1) (1 + \check p(1)) \right\} \tag{87.5}$$

and the marginal investor who prices the asset is always the one that values it *less* highly than does the other type.

Now the marginal investor is always the (temporarily) pessimistic type.

Notice from the sixth row of that the pessimistic price p_o is lower than the homogeneous belief prices p_a and p_b in both states.

When pessimistic investors price the asset according to (87.5), optimistic investors think that the asset is underpriced.

If they could, optimistic investors would willingly borrow at a one-period risk-free gross interest rate β^{-1} to purchase more of the asset.

Implicit constraints on leverage prohibit them from doing so.

When optimistic investors price the asset as in equation (87.3), pessimistic investors think that the asset is overpriced and would like to sell the asset short.

Constraints on short sales prevent that.

Here's code to solve for $\check p$ using iteration

```
def price_pessimistic_beliefs(transitions, dividend_payoff, β=.75,
                              max_iter=50000, tol=1e-16):
    """
    Function to Solve Pessimistic Beliefs
    """
    # We will guess an initial price vector of [0, 0]
    p_new = np.array([[0], [0]])
    p_old = np.array([[10.], [10.]])

    # We know this is a contraction mapping, so we can iterate to conv
    for i in range(max_iter):
        p_old = p_new
        p_new = β * np.min([q @ p_old
                            + q @ dividend_payoff for q in transitions],
                           axis=0)

        # If we succeed in converging, break out of for loop
        if np.max(np.sqrt((p_new - p_old)**2)) < tol:
            break

    return p_new
```

87.3.5 Further Interpretation

Jose Scheinkman [Scheinkman, 2014] interprets the Harrison-Kreps model as a model of a bubble — a situation in which an asset price exceeds what every investor thinks is merited by his or her beliefs about the value of the asset's underlying dividend stream.

Scheinkman stresses these features of the Harrison-Kreps model:

- High volume occurs when the Harrison-Kreps pricing formula (87.3) prevails.
- Type a investors sell the entire stock of the asset to type b investors every time the state switches from $s_t = 0$ to $s_t = 1$.
- Type b investors sell the asset to type a investors every time the state switches from $s_t = 1$ to $s_t = 0$.

Scheinkman takes this as a strength of the model because he observes high volume during *famous bubbles*.

- If the *supply* of the asset is increased sufficiently either physically (more "houses" are built) or artificially (ways are invented to short sell "houses"), bubbles end when the asset supply has grown enough to outstrip optimistic investors' resources for purchasing the asset.
- If optimistic investors finance their purchases by borrowing, tightening leverage constraints can extinguish a bubble.

Scheinkman extracts insights about the effects of financial regulations on bubbles.

He emphasizes how limiting short sales and limiting leverage have opposite effects.

Exercise 87.3.1

This exercise invites you to recreate the summary table using the functions we have built above.

s_t	0	1
p_a	1.33	1.22
p_b	1.45	1.91
p_o	1.85	2.08
p_p	1	1
$\hat{p}_a$	1.85	1.69
$\hat{p}_b$	1.69	2.08

You will want first to define the transition matrices and dividend payoff vector.

In addition, below we'll add an interpretation of the row corresponding to p_o by inventing two additional types of agents, one of whom is **permanently optimistic**, the other who is **permanently pessimistic**.

We construct subjective transition probability matrices for our permanently optimistic and permanently pessimistic investors as follows.

The permanently optimistic investors(i.e., the investor with the most optimistic beliefs in each state) believes the transition matrix

$$P_o = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{4} & \frac{3}{4} \end{bmatrix}$$

The permanently pessimistic investor believes the transition matrix

$$P_p = \begin{bmatrix} \frac{2}{3} & \frac{1}{3} \\ \frac{2}{3} & \frac{1}{3} \end{bmatrix}$$

We'll use these transition matrices when we present our solution of exercise 1 below.

Solution

First, we will obtain equilibrium price vectors with homogeneous beliefs, including when all investors are optimistic or pessimistic.

```
qa = np.array([[1/2, 1/2], [2/3, 1/3]])    # Type a transition matrix
qb = np.array([[2/3, 1/3], [1/4, 3/4]])    # Type b transition matrix
# Optimistic investor transition matrix
qopt = np.array([[1/2, 1/2], [1/4, 3/4]])
# Pessimistic investor transition matrix
qpess = np.array([[2/3, 1/3], [2/3, 1/3]])

dividendreturn = np.array([[0], [1]])

transitions = [qa, qb, qopt, qpess]
labels = ['p_a', 'p_b', 'p_optimistic', 'p_pessimistic']

for transition, label in zip(transitions, labels):
    print(label)
    print("=" * 20)
    s0, s1 = np.round(price_single_beliefs(transition, dividendreturn), 2)
    print(f"State 0: {s0}")
    print(f"State 1: {s1}")
    print("-" * 20)
```

```
p_a
====================
State 0: [1.33]
State 1: [1.22]
--------------------
p_b
====================
State 0: [1.45]
State 1: [1.91]
--------------------
p_optimistic
====================
State 0: [1.85]
State 1: [2.08]
--------------------
p_pessimistic
====================
State 0: [1.]
State 1: [1.]
--------------------
```

We will use the price_optimistic_beliefs function to find the price under heterogeneous beliefs.

```python
opt_beliefs = price_optimistic_beliefs([qa, qb], dividendreturn)
labels = ['p_optimistic', 'p_hat_a', 'p_hat_b']

for p, label in zip(opt_beliefs, labels):
    print(label)
    print("=" * 20)
    s0, s1 = np.round(p, 2)
    print(f"State 0: {s0}")
    print(f"State 1: {s1}")
    print("-" * 20)
```

```
p_optimistic
====================
State 0: [1.85]
State 1: [2.08]
--------------------
p_hat_a
====================
State 0: [1.85]
State 1: [1.69]
--------------------
p_hat_b
====================
State 0: [1.69]
State 1: [2.08]
--------------------
```

Notice that the equilibrium price with heterogeneous beliefs is equal to the price under single beliefs with **permanently optimistic** investors - this is due to the marginal investor in the heterogeneous beliefs equilibrium always being the type who is temporarily optimistic.

CHAPTER

EIGHTYEIGHT

SPECULATIVE BEHAVIOR WITH BAYESIAN LEARNING

Contents

88.1 Overview

This lecture describes how Morris [1996] extended the Harrison–Kreps model [Harrison and Kreps, 1978] of speculative asset pricing.

Like Harrison and Kreps's model, Morris's model determines the price of a dividend-yielding asset that is traded by risk-neutral investors who have heterogeneous beliefs.

The Harrison-Kreps model assumes that the traders have dogmatic, hard-wired beliefs about the asset's dividend stream.

Morris replaced Harrison and Kreps's traders with hard-wired beliefs about the dividend stream with traders who use Bayes' Law to update their beliefs about prospective dividends as new dividend data arrive.

Note

Morris's traders don't use data on past prices of the asset to update their beliefs about the dividend process.

Key features of the environment in Morris's model include:

- All traders share a set of statistical models for prospective dividends

- A single parameter indexes the set of statistical models
- All traders observe the same dividend history
- All traders use Bayes' Law to update beliefs
- Traders have different initial *prior distributions* over the parameter
- Traders' *posterior distributions* over the parameter eventually merge
- Before their posterior distributions merge, traders disagree about the predictive density over prospective dividends
  - therefore they disagree about the value of the asset

Just as in the hard-wired beliefs model of Harrison and Kreps, those differences of opinion induce investors to engage in *speculative behavior* in the following sense:

- sometimes they are willing to pay more for the asset than what they think is its "fundamental" value, i.e., the expected discounted value of its prospective dividend stream

Prior to reading this lecture, you might want to review the following quantecon lectures:

- *Harrison-Kreps model*
- *Likelihood ratio processes*
- *Bayesian versus frequentist statistics*

Let's start with some standard imports:

```
import numpy as np
import matplotlib.pyplot as plt
```

88.2 Structure of the model

There is a fixed supply of shares of an asset.

Each share entitles its owner to a stream of *binary* i.i.d. dividends $\{d_t\}$ where

$$d_{t+1} \in \{0, 1\}$$

The dividend at time t equals 1 with unknown probability $\theta \in (0, 1)$ and equals 0 with probability $1 - \theta$.

Unlike [Harrison and Kreps, 1978] where traders have hard-wired beliefs about a Markov transition matrix, in Morris's model:

- The true dividend probability θ is unknown
- Traders have *prior beliefs* about θ
- Traders observe dividend realizations and update beliefs via Bayes' Law

There is a finite set $\mathcal{I}$ of *risk-neutral* traders.

All traders have the same discount factor $\beta \in (0, 1)$.

- You can think of β as being related to a net risk-free interest rate r by $\beta = 1/(1 + r)$.

Owning the asset at the end of period t entitles the owner to dividends at time $t + 1, t + 2, \ldots$.

Because the dividend process is i.i.d., trader i thinks that the fundamental value of the asset is the capitalized value of the dividend stream, namely, $\sum_{j=1}^{\infty} \beta^j \hat{\theta}_i = \frac{\hat{\theta}_i}{r}$, where $\hat{\theta}_i$ is the mean of the trader's posterior distribution over θ.

88.2.1 Possible trades

Traders buy and sell the risky asset in competitive markets each period $t = 0, 1, 2, \ldots$ after dividends are paid.

As in Harrison-Kreps:

- The asset is traded *ex dividend*
- An owner of a share at the end of time t is entitled to the dividend at time $t + 1$
- An owner of a share at the end of period t also has the right to sell the share at time $t + 1$ after having received the dividend at time $t + 1$.

Short sales are prohibited.

This matters because it limits how pessimists can express their opinions:

- They *can* express themselves by selling their shares
- They *cannot* express themselves more emphatically by borrowing shares and immediately selling them

All traders have sufficient wealth to purchase the risky asset.

88.3 Information and beliefs

At time $t \geq 1$, all traders observe $(d_1, d_2, \ldots, d_t)$.

All traders update their subjective distribution over θ by applying Bayes' rule.

Traders have *heterogeneous priors* over the unknown dividend probability θ.

This heterogeneity in priors produces heterogeneous posterior beliefs.

88.4 Source of heterogeneous priors

Imputing different statistical models to agents inside a model is controversial.

Many game theorists and rational expectations applied economists think it is a bad idea.

While these economists often construct models in which agents have different *information*, they prefer to assume that all of the agents inside their model always share the same statistical model – i.e., the same joint probability distribution over the random process being modeled.

For a statistician or an economic theorist, a statistical model is a joint probability distribution that is characterized by a known parameter vector.

When working with a *set* of statistical models swept out by parameters, say θ in a known set Θ, economic theorists reduce the set of models to a single model by imputing to all agents inside the model the same prior probability distribution over θ.

> **Note**
>
> A set of statistical models that has a particular geometric structure is called a manifold of statistical models. Morris endows traders with a shared manifold of statistical models.

Proceeding in this way adheres to the *Harsanyi Common Priors Doctrine*.

[Harsanyi, 1967], [Harsanyi, 1968], [Harsanyi, 1968] argued that if two rational agents have the same information and the same reasoning capabilities, they will have the same joint probability distribution over outcomes of interest.

Harsanyi interpreted disagreements about prospective outcomes as arising from differences in agents' information sets, not differences in their statistical models.

Evidently, [Harrison and Kreps, 1978] departed from the Harsanyi common statistical model assumption when they hard-wired dogmatic disparate beliefs.

Morris [1996] abandons the Harsanyi doctrine less completely than Harrison and Kreps had.

- Morris does assume that agents share the same set of statistical models, but …
- Morris assumes that they have different initial prior distributions over the parameter that indexes the models

Morris's agents express their initial ignorance about the parameter differently – they have different priors.

Morris defends his assumption by alluding to the apparent "mispricing" of initial public offerings presented by [Miller, 1977].

Miller described a situation in which agents have access to little or no data about a new enterprise.

Morris wanted his traders to be open to changing their opinions as information about the parameter arrives.

Knowledgeable statisticians have been known to disagree about an appropriate prior.

For example, Morris described *different* respectable ways to express "maximal ignorance" about the parameter of a Bernoulli distribution

- a uniform distribution on $[0, 1]$
- a Jeffreys prior [Jeffreys, 1946] that is invariant to reparameterization; in the present situation, the Jeffreys prior takes the form of a Beta distribution with parameters $.5, .5$

Is one of these priors more "rational" than the other?

Morris thinks not.

88.5 Beta priors

For tractability, assume trader i has a Beta prior over the dividend probability

$$\theta \sim \text{Beta}(a_i, b_i)$$

where $a_i, b_i > 0$ are the prior parameters.

> **Note**
>
> The Beta distribution also appears in these quantecon lectures *Statistical Divergence Measures*, *Likelihood Ratio Processes*, *Job Search VIII: Search with Learning*.

Suppose trader i observes a history of t periods in which a total of s dividends are paid (i.e., s successes with dividend and $t - s$ failures without dividend).

By Bayes' rule, the posterior density over θ is:

$$\pi_i(\theta \mid s, t) = \frac{\theta^s (1-\theta)^{t-s} \pi_i(\theta)}{\int_0^1 \theta^s (1-\theta)^{t-s} \pi_i(\theta) d\theta}$$

where $\pi_i(\theta)$ is trader i's prior density.

Note

The Beta distribution is the conjugate prior for the Binomial likelihood. This means that when the prior is Beta(a_i, b_i) and we observe s successes in t trials, the posterior is Beta$(a_i + s, b_i + t - s)$.

The posterior mean (or expected dividend probability) is:

$$\mu_i(s,t) = \int_0^1 \theta \pi_i(\theta \mid s,t) d\theta = \mathbb{E}[\text{Beta}(a_i + s, b_i + t - s)] = \frac{a_i + s}{a_i + b_i + t}$$

Morris refers to $\mu_i(s,t)$ as trader i's **fundamental valuation** of the asset after history (s,t).

This is the probability trader i assigns to receiving a dividend next period.

It embeds trader i's updated belief about θ.

88.6 Market prices with learning

Fundamental valuations equal expected present values of dividends that our heterogeneous traders attach to the option of holding the asset *forever*.

The equilibrium price process is determined by the condition that the asset is held at time t by the trader who attaches the highest valuation to the asset at time t.

An owner of the asset has the option to sell it after receiving that period's dividend.

Traders take that into account.

That opens the possibility that a trader will be willing to pay more for the asset than that trader's fundamental valuation.

Definition 88.6.1 (Most Optimistic Valuation)

After history (s,t), the *most optimistic fundamental valuation* is:

$$\mu^*(s,t) = \max_{i \in \mathcal{I}} \mu_i(s,t)$$

Definition 88.6.2 (Equilibrium Asset Price)

Write $\tilde{p}(s,t,r)$ for the competitive equilibrium price of the risky asset (in current dollars) after history (s,t) when the interest rate is r.

The equilibrium price satisfies:

$$\tilde{p}(s,t,r) = \frac{1}{1+r}\Big[\mu^*(s,t)\{1 + \tilde{p}(s+1,t+1,r)\} + (1 - \mu^*(s,t))\tilde{p}(s,t+1,r)\Big]$$

The equilibrium price equals the highest expected discounted return among all traders from holding the asset to the next period.

Definition 88.6.3 (Normalized Price)

Define the normalized price as:

$$p(s,t,r) = r\tilde{p}(s,t,r)$$

Since the current "dollar" price of the riskless asset is $1/r$, this represents the price of the risky asset in terms of the riskless asset.

Substituting the preceding formula into the equilibrium condition gives:

$$p(s,t,r) = \frac{r}{1+r}\mu^*(s,t) + \frac{1}{1+r}\Big[\mu^*(s,t)p(s+1,t+1,r) + (1-\mu^*(s,t))p(s,t+1,r)\Big]$$

or equivalently:

$$p(s,t,r) = \mu^*(s,t) + \frac{r}{1+r}\Big[\mu^*(s,t)p(s+1,t+1,r) + (1-\mu^*(s,t))p(s,t+1,r) - \mu^*(s,t)\Big]$$

A price function that satisfies the equilibrium condition can be computed recursively.

Set $p^0(s,t,r) = 0$ for all (s,t,r), and define $p^{n+1}(s,t,r)$ by:

$$p^{n+1}(s,t,r) = \frac{r}{1+r}\mu^*(s,t) + \frac{1}{1+r}\Big[\mu^*(s,t)p^n(s+1,t+1,r) + (1-\mu^*(s,t))p^n(s,t+1,r)\Big]$$

The sequence $\{p^n(s,t,r)\}$ converges to the equilibrium price $p(s,t,r)$.

Definition 88.6.4 (Speculative Premium)

When the identity of the most optimistic trader can switch with future dividend realizations, the market price exceeds *every* trader's fundamental valuation.

In normalized units:

$$p(s,t,r) > \mu_i(s,t) \quad \text{for all } i \in \mathcal{I}$$

Define the **speculative premium** as:

$$p(s,t,r) - \mu^*(s,t) > 0$$

88.7 Two Traders

We now focus on an example with two traders with Beta priors with parameters (a_1, b_1) and (a_2, b_2).

Definition 88.7.1 (Rate Dominance (Beta Priors))

Trader 1 **rate-dominates** trader 2 if:

$$a_1 \geq a_2 \quad \text{and} \quad b_1 \leq b_2$$

Theorem 88.7.1 (Global Optimist (Two Traders))

For two traders with Beta priors:

1. If trader 1 rate-dominates trader 2, then trader 1 is a **global optimist**: $\mu_1(s,t) \geq \mu_2(s,t)$ for all histories (s,t)
2. In this case where $p(s,t,r) = \mu_1(s,t)$ for all (s,t,r), there is *no speculative premium*.

When neither trader rate-dominates the other, the identity of the most optimistic trader can switch as dividends accrue.

Along a history in which perpetual switching occurs, the price of the asset strictly exceeds both traders' fundamental valuations so long as traders continue to disagree:

$$p(s,t,r) > \max\{\mu_1(s,t), \mu_2(s,t)\}$$

Thus, along such a history, there is a persistent speculative premium.

88.7.1 Implementation

For computational tractability, let's work with a finite horizon T and solve by backward induction.

Note

On page 1122, Morris [1996] provides an argument that the limit as $T \to +\infty$ of such finite-horizon economies provides a useful selection algorithm that excludes additional equilibria that involve a Ponzi-scheme price component that Morris dismisses as fragile.

Following *Definition 88.6.2*, we use the discount factor parameterization $\beta = 1/(1+r)$ and compute dollar prices $\tilde{p}(s,t)$ via:

$$\tilde{p}(s,t) = \beta \max_{i\in\{1,2\}} \Big[\mu_i(s,t)\{1+\tilde{p}(s+1,t+1)\} + (1-\mu_i(s,t))\tilde{p}(s,t+1)\Big]$$

We set the terminal price $\tilde{p}(s,T)$ equal to the perpetuity value under the most optimistic belief.

```python
def posterior_mean(a, b, s, t):
    """
    Compute posterior mean μ_i(s,t) for Beta(a, b) prior.
    """
    return (a + s) / (a + b + t)

def perpetuity_value(a, b, s, t, β=.75):
    """
    Compute perpetuity value (β/(1-β)) * μ_i(s,t).
    """
    return (β / (1 - β)) * posterior_mean(a, b, s, t)

def price_learning_two_agents(prior1, prior2, β=.75, T=200):
    """
    Compute \tilde p(s,t) for two Beta-prior traders via backward induction.
    """
    a1, b1 = prior1
    a2, b2 = prior2
    price_array = np.zeros((T+1, T+1))
```

(continues on next page)

(continued from previous page)

```
    # Terminal condition: set to perpetuity value under max belief
    for s in range(T+1):
        perp1 = perpetuity_value(a1, b1, s, T, β)
        perp2 = perpetuity_value(a2, b2, s, T, β)
        price_array[s, T] = max(perp1, perp2)

    # Backward induction
    for t in range(T-1, -1, -1):
        for s in range(t, -1, -1):
            μ1 = posterior_mean(a1, b1, s, t)
            μ2 = posterior_mean(a2, b2, s, t)

            # One-step continuation values under each trader's beliefs
            cont1 = μ1 * (1.0 + price_array[s+1, t+1]) \
                    + (1.0 - μ1) * price_array[s, t+1]
            cont2 = μ2 * (1.0 + price_array[s+1, t+1]) \
                    + (1.0 - μ2) * price_array[s, t+1]
            price_array[s, t] = β * max(cont1, cont2)

    def μ1_fun(s, t):
        return posterior_mean(a1, b1, s, t)
    def μ2_fun(s, t):
        return posterior_mean(a2, b2, s, t)

    return price_array, μ1_fun, μ2_fun
```

88.7.2 Case A: global optimist (no premium)

Pick priors with rate dominance, e.g., trader 1: Beta$(a_1, b_1) = (2, 1)$ and trader 2: $(a_2, b_2) = (1, 2)$.

Trader 1 is the global optimist, so the normalized price equals trader 1's fundamental valuation: $p(s, t, r) = \mu_1(s, t)$.

```
β = 0.75
price_go, μ1_go, μ2_go = price_learning_two_agents(
        (2,1), (1,2), β=β, T=200)

perpetuity_1 = (β / (1 - β)) * μ1_go(0, 0)
perpetuity_2 = (β / (1 - β)) * μ2_go(0, 0)

print("Price at (0, 0) =", price_go[0,0])
print("Valuation of trader 1 at (0, 0) =", perpetuity_1)
print("Valuation of trader 2 at (0, 0) =", perpetuity_2)
```

```
Price at (0, 0) = 2.0
Valuation of trader 1 at (0, 0) = 2.0
Valuation of trader 2 at (0, 0) = 1.0
```

The price equals trader 1's perpetuity value.

88.7.3 Case B: perpetual switching (positive premium)

Now assume trader 1 has $\mathrm{Beta}(1, 1)$, trader 2 has $\mathrm{Beta}(1/2, 1/2)$.

These produce crossing posteriors, so there is no global optimist and the price exceeds both fundamentals early on.

```
price_ps, μ1_ps, μ2_ps = price_learning_two_agents(
                                (1,1), (0.5,0.5), β=β, T=200)

price_00 = price_ps[0,0]
μ1_00 = μ1_ps(0,0)
μ2_00 = μ2_ps(0,0)

perpetuity_1 = (β / (1 - β)) * μ1_ps(0, 0)
perpetuity_2 = (β / (1 - β)) * μ2_ps(0, 0)

print("Price at (0, 0) =", np.round(price_00, 6))
print("Valuation of trader 1 at (0, 0) =", perpetuity_1)
print("Valuation of trader 2 at (0, 0) =", perpetuity_2)
```

```
Price at (0, 0) = 1.599322
Valuation of trader 1 at (0, 0) = 1.5
Valuation of trader 2 at (0, 0) = 1.5
```

The resulting premium reflects the option value of reselling to whichever trader becomes temporarily more optimistic as dividends arrive sequentially.

Within this setting, we can reproduce two key figures reported in Morris [1996]

```
def normalized_price_two_agents(prior1, prior2, r, T=250):
    """Return p(s,t,r) = r \tilde p(s,t,r) for two traders."""
    β = 1.0 / (1.0 + r)
    price_array, *_ = price_learning_two_agents(prior1, prior2, β=β, T=T)
    return r * price_array

# Figure I: p*(0,0,r) as a function of r
r_grid = np.linspace(1e-3, 5.0, 200)
priors = ((1,1), (0.5,0.5))
p00 = np.array([normalized_price_two_agents(
                priors[0], priors[1], r, T=300)[0,0]
                for r in r_grid])

fig, ax = plt.subplots()
ax.plot(r_grid, p00, lw=2)
ax.set_xlabel(r'$r$')
ax.set_ylabel(r'$p^*(0,0,r)$')
ax.axhline(0.5, color='C1', linestyle='--')
plt.show()
```

In the first figure, notice that:

- The resale option pushes the normalized price $p^*(0, 0, r)$ above fundamentals (0.5) for any finite r.
- As r increases (β decreases), the option value fades and $p^*(0, 0, r) \to 0.5$.
- At $r = 0.05$ the premium is about 8–9%, consistent with Morris (1996, Section IV).

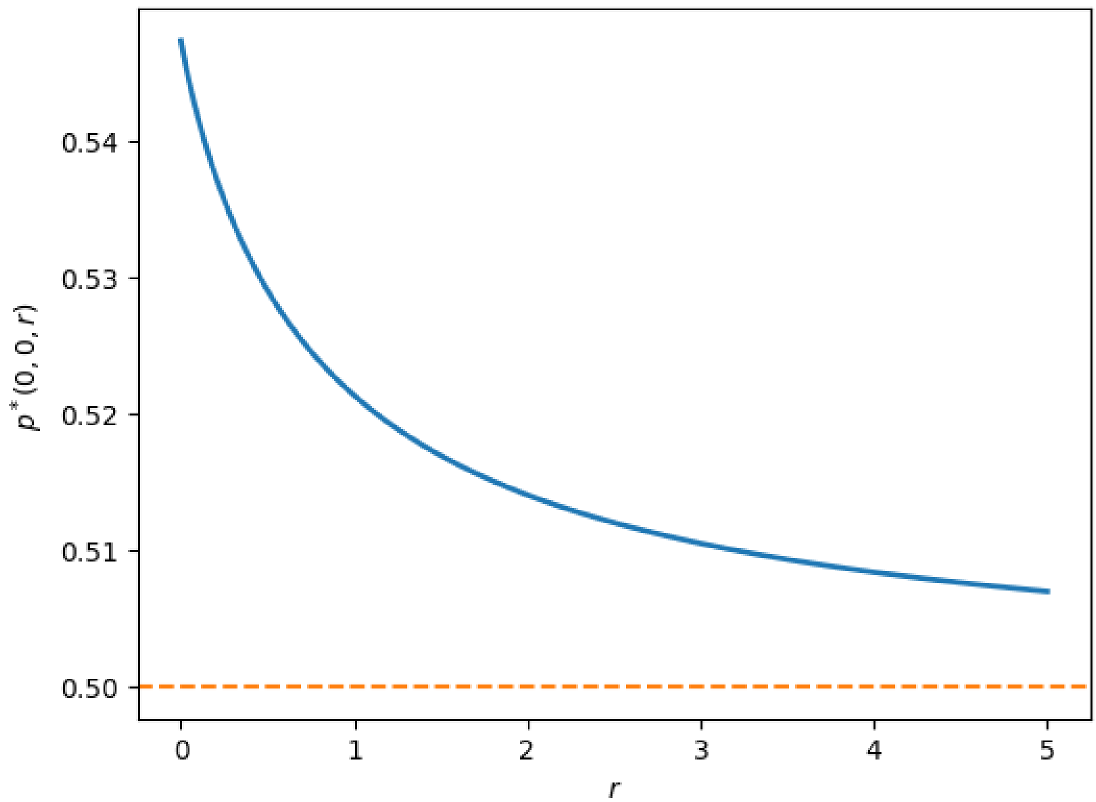


Fig. 88.1: Normalized price against interest rate

```python
# Figure II: p*(t/2,t,0.05) as a function of t
r = 0.05
T = 60
p_mat = normalized_price_two_agents(priors[0], priors[1], r, T=T)
t_vals = np.arange(0, 54, 2)
s_vals = t_vals // 2
y = np.array([p_mat[s, t] for s, t in zip(s_vals, t_vals)])

fig, ax = plt.subplots()
ax.plot(t_vals, y, lw=2)
ax.set_xlabel(r'$t$')
ax.set_ylabel(r'$p^*(t/2,t,0.05)$')
ax.axhline(0.5, color='C1', linestyle='--')
plt.show()

p0 = p_mat[0,0]
μ0 = 0.5
print("Initial normalized premium at r=0.05 (%):",
      np.round(100 * (p0 / μ0 - 1.0), 2))
```

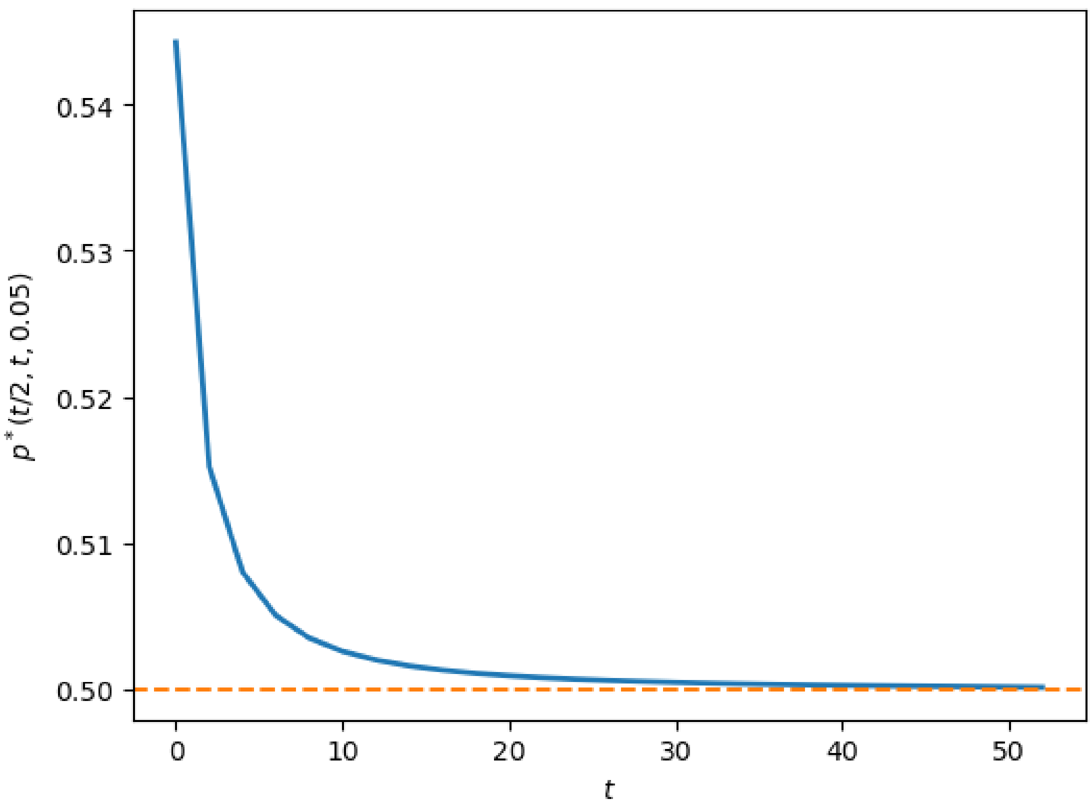


Fig. 88.2: Normalized price against time

```
Initial normalized premium at r=0.05 (%): 8.85
```

In the second figure, notice that:

- Along the symmetric path $s = t/2$, both traders' fundamental valuations equal 0.5 at every t, yet the price starts above 0.5 and declines toward 0.5 as learning reduces disagreement and the resale option loses value.

88.7.4 General N–trader extension

The same recursion extends to any finite set of Beta priors $\{(a_i, b_i)\}_{i=1}^N$ by taking a max over i each period.

```
def price_learning(priors, β=0.75, T=200):
    """
    N-trader version with heterogeneous Beta priors.
    """
    price_array = np.zeros((T+1, T+1))

    def perp_i(i, s, t):
        a, b = priors[i]
        return perpetuity_value(a, b, s, t, β)

    # Terminal condition
    for s in range(T+1):
        price_array[s, T] = max(
            perp_i(i, s, T) for i in range(len(priors)))

    # Backward induction
    for t in range(T-1, -1, -1):
        for s in range(t, -1, -1):
            conts = []
            for (a, b) in priors:
                μ = posterior_mean(a, b, s, t)
                conts.append(μ *
                (1.0 + price_array[s+1, t+1])
                     + (1.0 - μ) * price_array[s, t+1])
            price_array[s, t] = β * max(conts)

    return price_array

β = 0.75
priors = [(1,1), (0.5,0.5), (3,2)]
price_N = price_learning(priors, β=β, T=150)

# Compute valuations for each trader at (0,0)
μ_vals = [posterior_mean(a, b, 0, 0) for a, b in priors]
perp_vals = [(β / (1 - β)) * μ for μ in μ_vals]

print("Three-trader example at (s,t)=(0,0):")
print(f"Price at (0,0) = {np.round(price_N[0,0], 6)}")
print(f"\nTrader valuations:")
for i, (μ, perp) in enumerate(zip(μ_vals, perp_vals), 1):
    print(f"  Trader {i} = {np.round(perp, 6)}")
```

```
Three-trader example at (s,t)=(0,0):
Price at (0,0) = 1.937972

Trader valuations:
  Trader 1 = 1.5
  Trader 2 = 1.5
  Trader 3 = 1.8
```

Note that the asset price is above all traders' valuations.

Morris tells us that no rate dominance exists in this case.

Let's verify this using the code below

```
dominant = None
for i in range(len(priors)):
    is_dom = all(
        priors[i][0] >= priors[j][0] and priors[i][1] <= priors[j][1]
                 for j in range(len(priors)) if i != j)
    if is_dom:
        dominant = i
        break

if dominant is not None:
    print(f"\nTrader {dominant+1} is the global optimist (rate-dominant)")
else:
    print(f"\nNo global optimist and speculative premium exists")
```

```
No global optimist and speculative premium exists
```

Indeed, there is no global optimist and a speculative premium exists.

88.8 Concluding remarks

Morris [1996] uses his model to interpret a "hot issue" anomaly described by [Miller, 1977] according to which opening market prices of initial public offerings seem higher than values prices that emerge later.

88.9 Exercise

Exercise 88.9.1

Morris [Morris, 1996] provides a sharp characterization of when speculative bubbles arise.

The key condition is that there is no *global optimist*.

In this exercise, you will verify this condition for the following sets of traders with Beta priors:

1. Trader 1: $\mathrm{Beta}(2, 1)$, Trader 2: $\mathrm{Beta}(1, 2)$
2. Trader 1: $\mathrm{Beta}(1, 1)$, Trader 2: $\mathrm{Beta}(1/2, 1/2)$
3. Trader 1: $\mathrm{Beta}(3, 1)$, Trader 2: $\mathrm{Beta}(2, 1)$, Trader 3: $\mathrm{Beta}(1, 2)$
4. Trader 1: $\mathrm{Beta}(1, 1)$, Trader 2: $\mathrm{Beta}(1/2, 1/2)$, Trader 3: $\mathrm{Beta}(3/2, 3/2)$

Solution

Here is one solution:

```
def check_rate_dominance(priors):
    """
    Check if any trader rate-dominates all others.
    """
    N = len(priors)

    for i in range(N):
        a_i, b_i = priors[i]
```

```
        is_dominant = True

        for j in range(N):
            if i == j:
                continue
            a_j, b_j = priors[j]

            # Check rate dominance condition
            if not (a_i >= a_j and b_i <= b_j):
                is_dominant = False
                break

        if is_dominant:
            return i

    return None

# Test cases
test_cases = [
    ([(2, 1), (1, 2)], "Global optimist exists"),
    ([(1, 1), (0.5, 0.5)], "Perpetual switching"),
    ([(3, 1), (2, 1), (1, 2)], "Three traders with dominant"),
    ([(1, 1), (0.5, 0.5), (1.5, 1.5)], "Three traders, no dominant")
]

for priors, description in test_cases:
    dominant = check_rate_dominance(priors)

    print(f"\n{description}")
    print(f"Priors: {priors}")
    print("=="*8)
    if dominant is not None:
        print(f"Trader {dominant+1} is the global optimist (rate-dominant)")
    else:
        print(f"No global optimist exists")
    print("=="*8 + "\n")
```

```
Global optimist exists
Priors: [(2, 1), (1, 2)]
================
Trader 1 is the global optimist (rate-dominant)
================


Perpetual switching
Priors: [(1, 1), (0.5, 0.5)]
================
No global optimist exists
================


Three traders with dominant
Priors: [(3, 1), (2, 1), (1, 2)]
================
Trader 1 is the global optimist (rate-dominant)
================
```

```
Three traders, no dominant
Priors: [(1, 1), (0.5, 0.5), (1.5, 1.5)]
================
No global optimist exists
================
```

Part XIV

Data and Empirics

CHAPTER

EIGHTYNINE

PANDAS FOR PANEL DATA

Contents

89.1 Overview

In an earlier lecture on pandas, we looked at working with simple data sets.

Econometricians often need to work with more complex data sets, such as panels.

Common tasks include

- Importing data, cleaning it and reshaping it across several axes.
- Selecting a time series or cross-section from a panel.
- Grouping and summarizing data.

`pandas` (derived from 'panel' and 'data') contains powerful and easy-to-use tools for solving exactly these kinds of problems.

In what follows, we will use a panel data set of real minimum wages from the OECD to create:

- summary statistics over multiple dimensions of our data
- a time series of the average minimum wage of countries in the dataset
- kernel density estimates of wages by continent

We will begin by reading in our long format panel data from a CSV file and reshaping the resulting `DataFrame` with `pivot_table` to build a `MultiIndex`.

Additional detail will be added to our `DataFrame` using pandas' `merge` function, and data will be summarized with the `groupby` function.

89.2 Slicing and Reshaping Data

We will read in a dataset from the OECD of real minimum wages in 32 countries and assign it to `realwage`.

The dataset can be accessed with the following link:

```
url1 = 'https://raw.githubusercontent.com/QuantEcon/lecture-python/master/source/_
↪static/lecture_specific/pandas_panel/realwage.csv'
```

```
import pandas as pd

# Display 6 columns for viewing purposes
pd.set_option('display.max_columns', 6)

# Reduce decimal points to 2
pd.options.display.float_format = '{:,.2f}'.format

realwage = pd.read_csv(url1)
```

Let's have a look at what we've got to work with

```
realwage.head()  # Show first 5 rows
```

```
   Unnamed: 0        Time  Country                                    Series  \
0           0  2006-01-01  Ireland  In 2015 constant prices at 2015 USD PPPs
1           1  2007-01-01  Ireland  In 2015 constant prices at 2015 USD PPPs
2           2  2008-01-01  Ireland  In 2015 constant prices at 2015 USD PPPs
3           3  2009-01-01  Ireland  In 2015 constant prices at 2015 USD PPPs
4           4  2010-01-01  Ireland  In 2015 constant prices at 2015 USD PPPs

  Pay period     value
0     Annual 17,132.44
1     Annual 18,100.92
2     Annual 17,747.41
3     Annual 18,580.14
4     Annual 18,755.83
```

The data is currently in long format, which is difficult to analyze when there are several dimensions to the data.

We will use `pivot_table` to create a wide format panel, with a `MultiIndex` to handle higher dimensional data.

`pivot_table` arguments should specify the data (values), the index, and the columns we want in our resulting dataframe.

By passing a list in columns, we can create a `MultiIndex` in our column axis

```
realwage = realwage.pivot_table(values='value',
                                index='Time',
                                columns=['Country', 'Series', 'Pay period'])
realwage.head()
```

```
Country                                   Australia         \
Series     In 2015 constant prices at 2015 USD PPPs
```

(continues on next page)

(continued from previous page)

```
Pay period                                   Annual Hourly
Time
2006-01-01                                 20,410.65  10.33
2007-01-01                                 21,087.57  10.67
2008-01-01                                 20,718.24  10.48
2009-01-01                                 20,984.77  10.62
2010-01-01                                 20,879.33  10.57

Country                                                       ...  \
Series     In 2015 constant prices at 2015 USD exchange rates  ...
Pay period                                             Annual  ...
Time                                                           ...
2006-01-01                                          23,826.64  ...
2007-01-01                                          24,616.84  ...
2008-01-01                                          24,185.70  ...
2009-01-01                                          24,496.84  ...
2010-01-01                                          24,373.76  ...

Country                                  United States  \
Series     In 2015 constant prices at 2015 USD PPPs
Pay period                                      Hourly
Time
2006-01-01                                        6.05
2007-01-01                                        6.24
2008-01-01                                        6.78
2009-01-01                                        7.58
2010-01-01                                        7.88

Country
Series     In 2015 constant prices at 2015 USD exchange rates
Pay period                                             Annual Hourly
Time
2006-01-01                                          12,594.40   6.05
2007-01-01                                          12,974.40   6.24
2008-01-01                                          14,097.56   6.78
2009-01-01                                          15,756.42   7.58
2010-01-01                                          16,391.31   7.88

[5 rows x 128 columns]
```

To more easily filter our time series data, later on, we will convert the index into a `DateTimeIndex`

```
realwage.index = pd.to_datetime(realwage.index)
type(realwage.index)
```

```
pandas.core.indexes.datetimes.DatetimeIndex
```

The columns contain multiple levels of indexing, known as a `MultiIndex`, with levels being ordered hierarchically (Country > Series > Pay period).

A `MultiIndex` is the simplest and most flexible way to manage panel data in pandas

```
type(realwage.columns)
```

```
pandas.core.indexes.multi.MultiIndex
```

```
realwage.columns.names
```

```
FrozenList(['Country', 'Series', 'Pay period'])
```

Like before, we can select the country (the top level of our `MultiIndex`)

```
realwage['United States'].head()
```

```
Series      In 2015 constant prices at 2015 USD PPPs         \
Pay period                                    Annual Hourly
Time
2006-01-01                                 12,594.40   6.05
2007-01-01                                 12,974.40   6.24
2008-01-01                                 14,097.56   6.78
2009-01-01                                 15,756.42   7.58
2010-01-01                                 16,391.31   7.88

Series      In 2015 constant prices at 2015 USD exchange rates
Pay period                                              Annual Hourly
Time
2006-01-01                                           12,594.40   6.05
2007-01-01                                           12,974.40   6.24
2008-01-01                                           14,097.56   6.78
2009-01-01                                           15,756.42   7.58
2010-01-01                                           16,391.31   7.88
```

Stacking and unstacking levels of the `MultiIndex` will be used throughout this lecture to reshape our dataframe into a format we need.

`.stack()` rotates the lowest level of the column `MultiIndex` to the row index (`.unstack()` works in the opposite direction - try it out)

```
realwage.stack(future_stack=True).head()
```

```
Country                                               Australia  \
Series                  In 2015 constant prices at 2015 USD PPPs
Time       Pay period
2006-01-01 Annual                                     20,410.65
           Hourly                                         10.33
2007-01-01 Annual                                     21,087.57
           Hourly                                         10.67
2008-01-01 Annual                                     20,718.24

Country                                                                   \
Series                  In 2015 constant prices at 2015 USD exchange rates
Time       Pay period
2006-01-01 Annual                                               23,826.64
           Hourly                                                   12.06
2007-01-01 Annual                                               24,616.84
           Hourly                                                   12.46
2008-01-01 Annual                                               24,185.70

Country                                                 Belgium  ...   \
Series                  In 2015 constant prices at 2015 USD PPPs  ...
Time       Pay period                                            ...
2006-01-01 Annual                                     21,042.28  ...
```

(continues on next page)

(continued from previous page)

```
             Hourly                                       10.09  ...
2007-01-01 Annual                                      21,310.05  ...
             Hourly                                       10.22  ...
2008-01-01 Annual                                      21,416.96  ...

Country                                                    United Kingdom  \
Series                 In 2015 constant prices at 2015 USD exchange rates
Time        Pay period
2006-01-01 Annual                                               20,376.32
             Hourly                                                  9.81
2007-01-01 Annual                                               20,954.13
             Hourly                                                 10.07
2008-01-01 Annual                                               20,902.87

Country                                         United States  \
Series                 In 2015 constant prices at 2015 USD PPPs
Time        Pay period
2006-01-01 Annual                                      12,594.40
             Hourly                                         6.05
2007-01-01 Annual                                      12,974.40
             Hourly                                         6.24
2008-01-01 Annual                                      14,097.56

Country
Series                 In 2015 constant prices at 2015 USD exchange rates
Time        Pay period
2006-01-01 Annual                                               12,594.40
             Hourly                                                  6.05
2007-01-01 Annual                                               12,974.40
             Hourly                                                  6.24
2008-01-01 Annual                                               14,097.56

[5 rows x 64 columns]
```

We can also pass in an argument to select the level we would like to stack

```
realwage.stack(level='Country', future_stack=True).head()
```

```
Series                 In 2015 constant prices at 2015 USD PPPs         \
Pay period                                            Annual Hourly
Time        Country
2006-01-01 Australia                                20,410.65  10.33
             Belgium                                  21,042.28  10.09
             Brazil                                    3,310.51   1.41
             Canada                                   13,649.69   6.56
             Chile                                     5,201.65   2.22

Series                 In 2015 constant prices at 2015 USD exchange rates
Pay period                                                      Annual Hourly
Time        Country
2006-01-01 Australia                                          23,826.64  12.06
             Belgium                                            20,228.74   9.70
             Brazil                                              2,032.87   0.87
             Canada                                             14,335.12   6.89
             Chile                                               3,333.76   1.42
```

Using a `DatetimeIndex` makes it easy to select a particular time period.

Selecting one year and stacking the two lower levels of the MultiIndex creates a cross-section of our panel data

```
realwage.loc['2015'].stack(level=(1, 2), future_stack=True).transpose().head()
```

```
Time                                       2015-01-01         \
Series      In 2015 constant prices at 2015 USD PPPs
Pay period                                     Annual Hourly
Country
Australia                                   21,715.53  10.99
Belgium                                     21,588.12  10.35
Brazil                                       4,628.63   2.00
Canada                                      16,536.83   7.95
Chile                                        6,633.56   2.80

Time
Series      In 2015 constant prices at 2015 USD exchange rates
Pay period                                               Annual Hourly
Country
Australia                                             25,349.90  12.83
Belgium                                               20,753.48   9.95
Brazil                                                 2,842.28   1.21
Canada                                                17,367.24   8.35
Chile                                                  4,251.49   1.81
```

For the rest of lecture, we will work with a dataframe of the hourly real minimum wages across countries and time, measured in 2015 US dollars.

To create our filtered dataframe (realwage_f), we can use the xs method to select values at lower levels in the multiindex, while keeping the higher levels (countries in this case)

```
realwage_f = realwage.xs(('Hourly', 'In 2015 constant prices at 2015 USD exchange␣
↪rates'),
                         level=('Pay period', 'Series'), axis=1)
realwage_f.head()
```

```
Country     Australia  Belgium  Brazil  ...  Turkey  United Kingdom  \
Time                                    ...
2006-01-01      12.06     9.70    0.87  ...    2.27            9.81
2007-01-01      12.46     9.82    0.92  ...    2.26           10.07
2008-01-01      12.24     9.87    0.96  ...    2.22           10.04
2009-01-01      12.40    10.21    1.03  ...    2.28           10.15
2010-01-01      12.34    10.05    1.08  ...    2.30            9.96

Country     United States
Time
2006-01-01           6.05
2007-01-01           6.24
2008-01-01           6.78
2009-01-01           7.58
2010-01-01           7.88

[5 rows x 32 columns]
```

89.3 Merging Dataframes and Filling NaNs

Similar to relational databases like SQL, pandas has built in methods to merge datasets together.

Using country information from WorldData.info, we'll add the continent of each country to `realwage_f` with the `merge` function.

The dataset can be accessed with the following link:

```
url2 = 'https://raw.githubusercontent.com/QuantEcon/lecture-python/master/source/_
↪static/lecture_specific/pandas_panel/countries.csv'
```

```
worlddata = pd.read_csv(url2, sep=';')
worlddata.head()
```

```
    Country (en) Country (de)        Country (local)  ... Deathrate  \
0    Afghanistan  Afghanistan  Afganistan/Afqanestan  ...     13.70
1          Egypt      Ägypten                   Misr  ...      4.70
2  Åland Islands  Ålandinseln                  Åland  ...      0.00
3        Albania     Albanien               Shqipëria  ...      6.70
4        Algeria     Algerien     Al-Jaza’ir/Algérie  ...      4.30

  Life expectancy                                                Url
0           51.30  https://www.laenderdaten.info/Asien/Afghanista...
1           72.70  https://www.laenderdaten.info/Afrika/Aegypten/...
2            0.00  https://www.laenderdaten.info/Europa/Aland/ind...
3           78.30  https://www.laenderdaten.info/Europa/Albanien/...
4           76.80  https://www.laenderdaten.info/Afrika/Algerien/...

[5 rows x 17 columns]
```

First, we'll select just the country and continent variables from `worlddata` and rename the column to 'Country'

```
worlddata = worlddata[['Country (en)', 'Continent']]
worlddata = worlddata.rename(columns={'Country (en)': 'Country'})
worlddata.head()
```

```
         Country Continent
0    Afghanistan      Asia
1          Egypt    Africa
2  Åland Islands    Europe
3        Albania    Europe
4        Algeria    Africa
```

We want to merge our new dataframe, `worlddata`, with `realwage_f`.

The pandas `merge` function allows dataframes to be joined together by rows.

Our dataframes will be merged using country names, requiring us to use the transpose of `realwage_f` so that rows correspond to country names in both dataframes

```
realwage_f.transpose().head()
```

```
Time       2006-01-01  2007-01-01  2008-01-01  ...  2014-01-01  2015-01-01  \
Country                                        ...
Australia       12.06       12.46       12.24  ...       12.67       12.83
Belgium          9.70        9.82        9.87  ...       10.01        9.95
```

(continues on next page)

(continued from previous page)

```
Brazil           0.87        0.92        0.96  ...        1.21        1.21
Canada           6.89        6.96        7.24  ...        8.22        8.35
Chile            1.42        1.45        1.44  ...        1.76        1.81

Time        2016-01-01
Country
Australia        12.98
Belgium           9.76
Brazil            1.24
Canada            8.48
Chile             1.91

[5 rows x 11 columns]
```

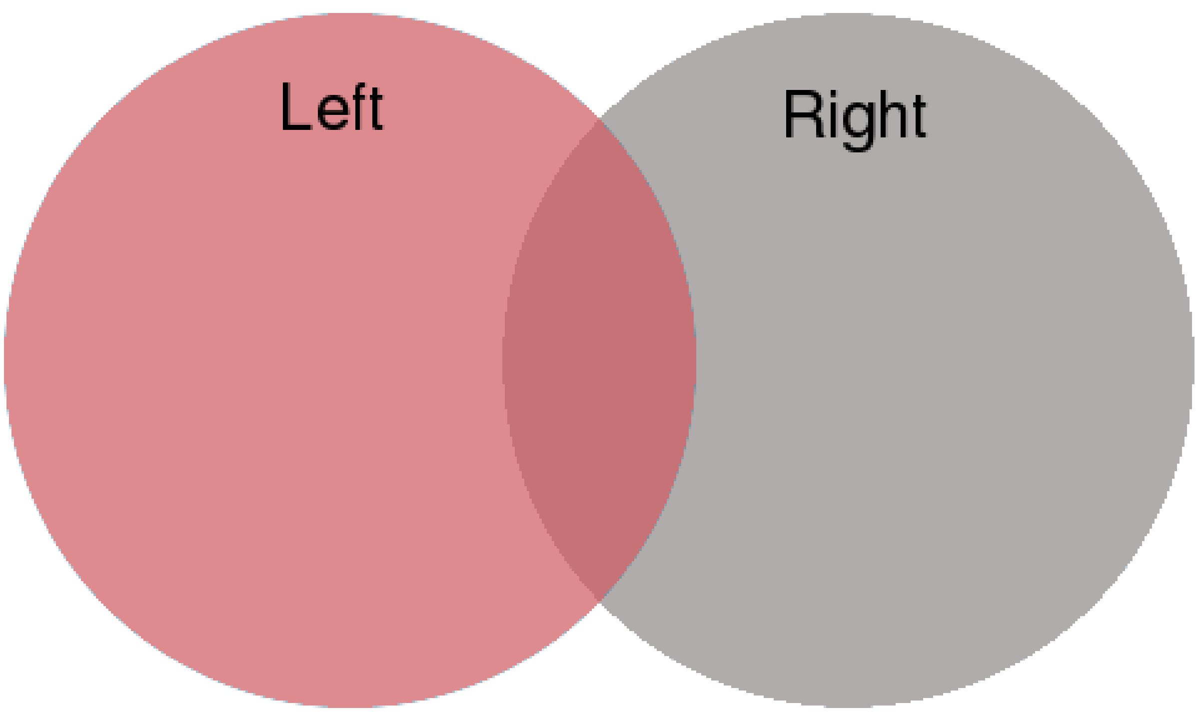
We can use either left, right, inner, or outer join to merge our datasets:

- left join includes only countries from the left dataset
- right join includes only countries from the right dataset
- outer join includes countries that are in either the left and right datasets
- inner join includes only countries common to both the left and right datasets

By default, `merge` will use an inner join.

Here we will pass `how='left'` to keep all countries in `realwage_f`, but discard countries in `worlddata` that do not have a corresponding data entry `realwage_f`.

This is illustrated by the red shading in the following diagram



We will also need to specify where the country name is located in each dataframe, which will be the `key` that is used to merge the dataframes 'on'.

Our 'left' dataframe (`realwage_f.transpose()`) contains countries in the index, so we set `left_index=True`.

Our 'right' dataframe (`worlddata`) contains countries in the 'Country' column, so we set `right_on='Country'`

```
merged = pd.merge(realwage_f.transpose(), worlddata,
                  how='left', left_index=True, right_on='Country')
merged.head()
```

```
        2006-01-01 00:00:00  2007-01-01 00:00:00  2008-01-01 00:00:00  ...  \
17.00                 12.06                12.46                12.24  ...
23.00                  9.70                 9.82                 9.87  ...
32.00                  0.87                 0.92                 0.96  ...
100.00                 6.89                 6.96                 7.24  ...
38.00                  1.42                 1.45                 1.44  ...

        2016-01-01 00:00:00    Country      Continent
17.00                 12.98  Australia      Australia
23.00                  9.76    Belgium         Europe
32.00                  1.24     Brazil  South America
100.00                 8.48     Canada  North America
38.00                  1.91      Chile  South America

[5 rows x 13 columns]
```

Countries that appeared in `realwage_f` but not in `worlddata` will have `NaN` in the Continent column.

To check whether this has occurred, we can use `.isnull()` on the continent column and filter the merged dataframe

```
merged[merged['Continent'].isnull()]
```

```
     2006-01-01 00:00:00  2007-01-01 00:00:00  2008-01-01 00:00:00  ...  \
NaN                 3.42                 3.74                 3.87  ...
NaN                 0.23                 0.45                 0.39  ...
NaN                 1.50                 1.64                 1.71  ...

     2016-01-01 00:00:00             Country  Continent
NaN                 5.28               Korea        NaN
NaN                 0.55  Russian Federation        NaN
NaN                 2.08     Slovak Republic        NaN

[3 rows x 13 columns]
```

We have three missing values!

One option to deal with NaN values is to create a dictionary containing these countries and their respective continents.

`.map()` will match countries in `merged['Country']` with their continent from the dictionary.

Notice how countries not in our dictionary are mapped with `NaN`

```
missing_continents = {'Korea': 'Asia',
                      'Russian Federation': 'Europe',
                      'Slovak Republic': 'Europe'}

merged['Country'].map(missing_continents)
```

```
17.00        NaN
23.00        NaN
32.00        NaN
100.00       NaN
38.00        NaN
108.00       NaN
41.00        NaN
225.00       NaN
53.00        NaN
58.00        NaN
45.00        NaN
68.00        NaN
233.00       NaN
86.00        NaN
88.00        NaN
91.00        NaN
NaN         Asia
117.00       NaN
122.00       NaN
123.00       NaN
138.00       NaN
153.00       NaN
151.00       NaN
174.00       NaN
175.00       NaN
NaN       Europe
NaN       Europe
198.00       NaN
200.00       NaN
227.00       NaN
241.00       NaN
240.00       NaN
Name: Country, dtype: object
```

We don't want to overwrite the entire series with this mapping.

`.fillna()` only fills in `NaN` values in `merged['Continent']` with the mapping, while leaving other values in the column unchanged

```
merged['Continent'] = merged['Continent'].fillna(merged['Country'].map(missing_
↪continents))

# Check for whether continents were correctly mapped

merged[merged['Country'] == 'Korea']
```

```
     2006-01-01 00:00:00  2007-01-01 00:00:00  2008-01-01 00:00:00  ...  \
NaN                 3.42                 3.74                 3.87  ...

     2016-01-01 00:00:00  Country  Continent
NaN                 5.28    Korea       Asia

[1 rows x 13 columns]
```

We will also combine the Americas into a single continent - this will make our visualization nicer later on.

To do this, we will use `.replace()` and loop through a list of the continent values we want to replace

```
replace = ['Central America', 'North America', 'South America']

for country in replace:
    merged.Continent = merged.Continent.replace(to_replace=country,
                                                value='America')
```

Now that we have all the data we want in a single `DataFrame`, we will reshape it back into panel form with a `MultiIndex`.

We should also ensure to sort the index using `.sort_index()` so that we can efficiently filter our dataframe later on.

By default, levels will be sorted top-down

```
merged = merged.set_index(['Continent', 'Country']).sort_index()
merged.head()
```

```
                      2006-01-01  2007-01-01  2008-01-01  ...  2014-01-01  \
Continent Country                                         ...
America   Brazil            0.87        0.92        0.96  ...        1.21
          Canada            6.89        6.96        7.24  ...        8.22
          Chile             1.42        1.45        1.44  ...        1.76
          Colombia          1.01        1.02        1.01  ...        1.13
          Costa Rica         NaN         NaN         NaN  ...        2.41

                      2015-01-01  2016-01-01
Continent Country
America   Brazil            1.21        1.24
          Canada            8.35        8.48
          Chile             1.81        1.91
          Colombia          1.13        1.12
          Costa Rica        2.56        2.63

[5 rows x 11 columns]
```

While merging, we lost our `DatetimeIndex`, as we merged columns that were not in datetime format

```
merged.columns
```

```
Index([2006-01-01 00:00:00, 2007-01-01 00:00:00, 2008-01-01 00:00:00,
       2009-01-01 00:00:00, 2010-01-01 00:00:00, 2011-01-01 00:00:00,
       2012-01-01 00:00:00, 2013-01-01 00:00:00, 2014-01-01 00:00:00,
       2015-01-01 00:00:00, 2016-01-01 00:00:00],
      dtype='object')
```

Now that we have set the merged columns as the index, we can recreate a `DatetimeIndex` using `.to_datetime()`

```
merged.columns = pd.to_datetime(merged.columns)
merged.columns = merged.columns.rename('Time')
merged.columns
```

```
DatetimeIndex(['2006-01-01', '2007-01-01', '2008-01-01', '2009-01-01',
               '2010-01-01', '2011-01-01', '2012-01-01', '2013-01-01',
               '2014-01-01', '2015-01-01', '2016-01-01'],
              dtype='datetime64[ns]', name='Time', freq=None)
```

The `DatetimeIndex` tends to work more smoothly in the row axis, so we will go ahead and transpose `merged`

```
merged = merged.transpose()
merged.head()
```

```
Continent  America                ...   Europe
Country     Brazil Canada Chile   ... Slovenia Spain United Kingdom
Time                              ...
2006-01-01    0.87   6.89  1.42   ...     3.92  3.99           9.81
2007-01-01    0.92   6.96  1.45   ...     3.88  4.10          10.07
2008-01-01    0.96   7.24  1.44   ...     3.96  4.14          10.04
2009-01-01    1.03   7.67  1.52   ...     4.08  4.32          10.15
2010-01-01    1.08   7.94  1.56   ...     4.81  4.30           9.96

[5 rows x 32 columns]
```

89.4 Grouping and Summarizing Data

Grouping and summarizing data can be particularly useful for understanding large panel datasets.

A simple way to summarize data is to call an aggregation method on the dataframe, such as `.mean()` or `.max()`.

For example, we can calculate the average real minimum wage for each country over the period 2006 to 2016 (the default is to aggregate over rows)

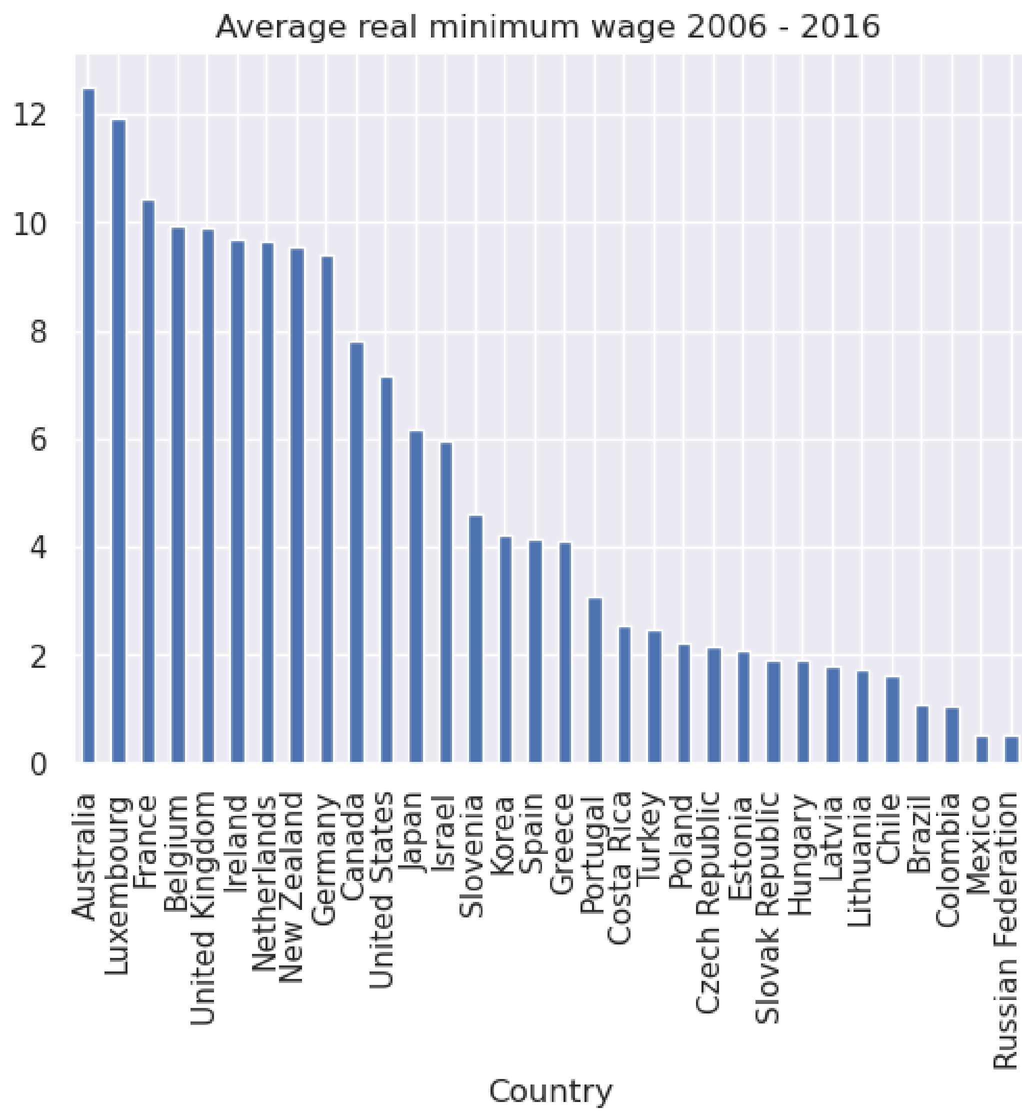
```
merged.mean().head(10)
```

```
Continent  Country
America    Brazil           1.09
           Canada           7.82
           Chile            1.62
           Colombia         1.07
           Costa Rica       2.53
           Mexico           0.53
           United States    7.15
Asia       Israel           5.95
           Japan            6.18
           Korea            4.22
dtype: float64
```

Using this series, we can plot the average real minimum wage over the past decade for each country in our data set

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme()
```

```
merged.mean().sort_values(ascending=False).plot(kind='bar',
                                                 title="Average real minimum wage 2006␣
↪- 2016")

# Set country labels
country_labels = merged.mean().sort_values(ascending=False).index.get_level_values(
↪'Country').tolist()
plt.xticks(range(0, len(country_labels)), country_labels)
plt.xlabel('Country')

plt.show()
```



Passing in `axis=1` to `.mean()` will aggregate over columns (giving the average minimum wage for all countries over time)

```
merged.mean(axis=1).head()
```

```
Time
2006-01-01    4.69
2007-01-01    4.84
2008-01-01    4.90
2009-01-01    5.08
2010-01-01    5.11
dtype: float64
```

We can plot this time series as a line graph

```
merged.mean(axis=1).plot()
plt.title('Average real minimum wage 2006 - 2016')
plt.ylabel('2015 USD')
```

(continues on next page)

(continued from previous page)

```
plt.xlabel('Year')
plt.show()
```

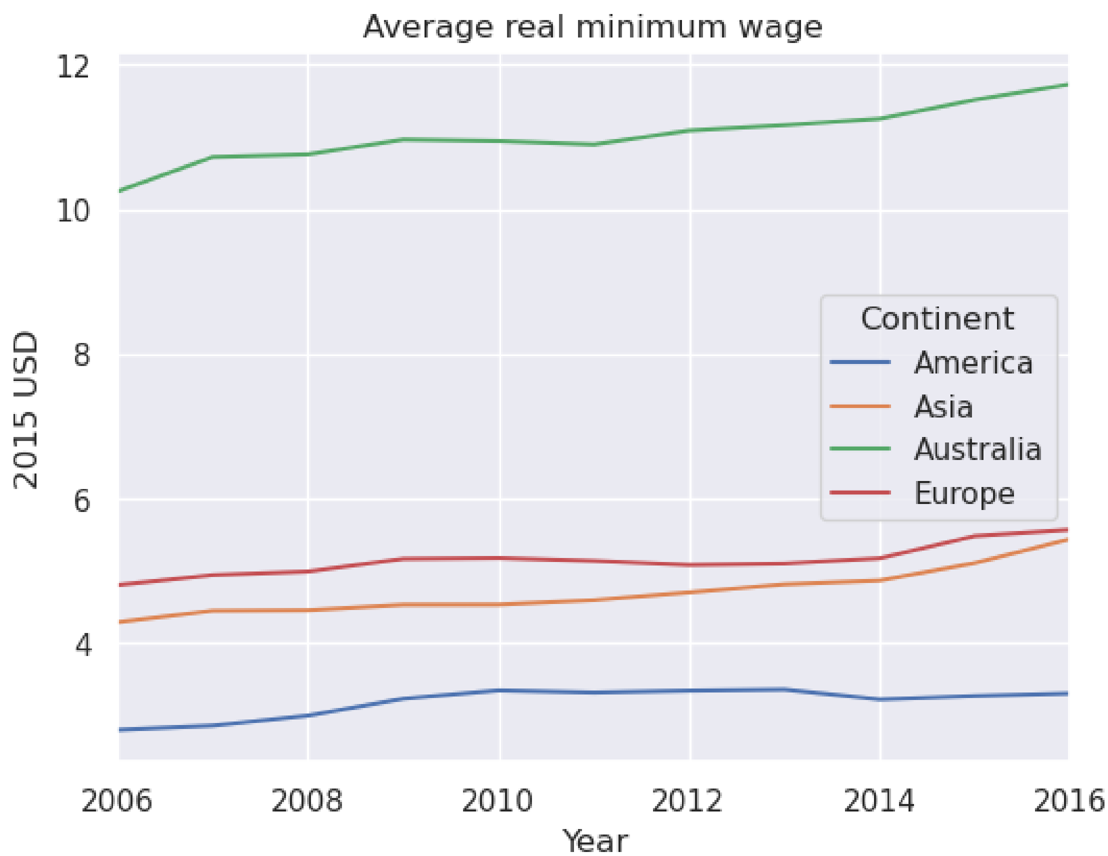
We can also specify a level of the `MultiIndex` (in the column axis) to aggregate over

```
merged.T.groupby(level='Continent').mean().T.head()
```

```
Continent   America  Asia  Australia  Europe
Time
2006-01-01     2.80  4.29      10.25    4.80
2007-01-01     2.85  4.44      10.73    4.94
2008-01-01     2.99  4.45      10.76    4.99
2009-01-01     3.23  4.53      10.97    5.16
2010-01-01     3.34  4.53      10.95    5.17
```

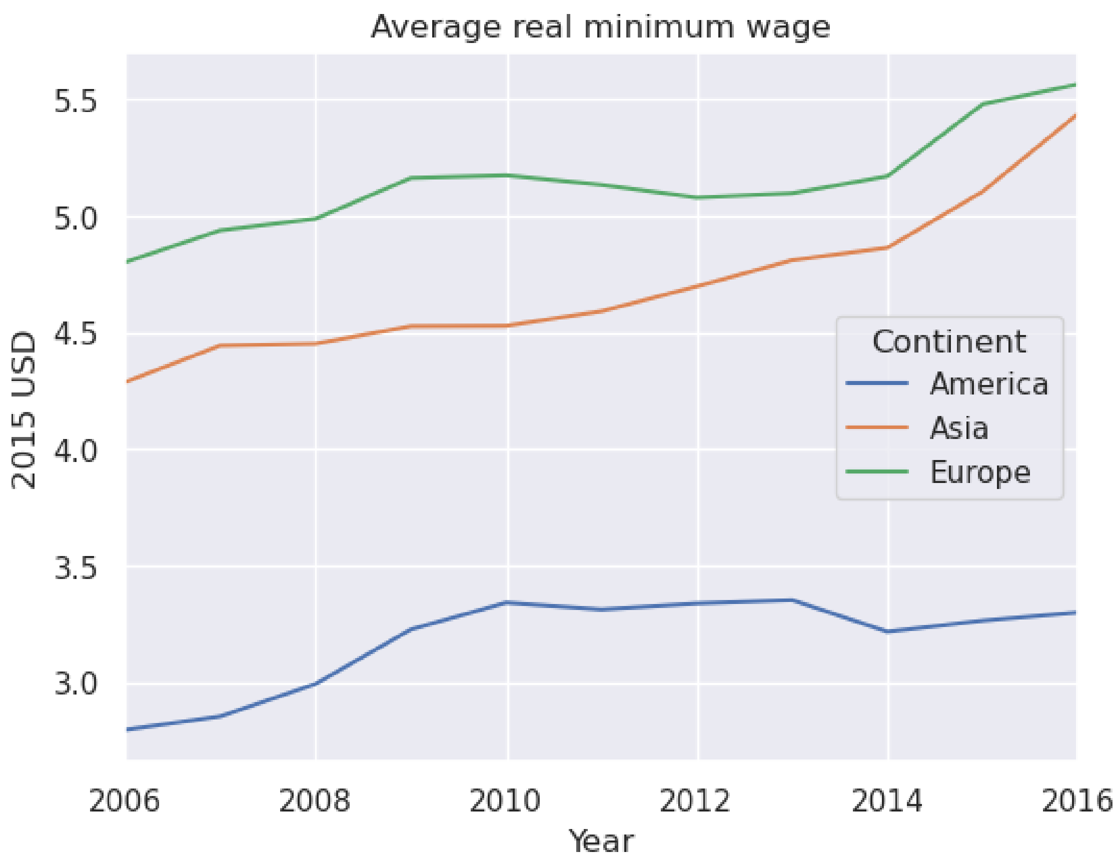
We can plot the average minimum wages in each continent as a time series

```
merged.T.groupby(level='Continent').mean().T.plot()
plt.title('Average real minimum wage')
plt.ylabel('2015 USD')
plt.xlabel('Year')
plt.show()
```



We will drop Australia as a continent for plotting purposes

```
merged = merged.drop('Australia', level='Continent', axis=1)
merged.T.groupby(level='Continent').mean().T.plot()
plt.title('Average real minimum wage')
plt.ylabel('2015 USD')
plt.xlabel('Year')
plt.show()
```



`.describe()` is useful for quickly retrieving a number of common summary statistics

```
merged.stack(future_stack=True).describe()
```

```
Continent  America  Asia  Europe
count        69.00 44.00  200.00
mean          3.19  4.70    5.15
std           3.02  1.56    3.82
min           0.52  2.22    0.23
25%           1.03  3.37    2.02
50%           1.44  5.48    3.54
75%           6.96  5.95    9.70
max           8.48  6.65   12.39
```

This is a simplified way to use `groupby`.

Using `groupby` generally follows a 'split-apply-combine' process:

- split: data is grouped based on one or more keys
- apply: a function is called on each group independently
- combine: the results of the function calls are combined into a new data structure

The `groupby` method achieves the first step of this process, creating a new `DataFrameGroupBy` object with data split into groups.

Let's split `merged` by continent again, this time using the `groupby` function, and name the resulting object `grouped`

```
grouped = merged.T.groupby(level='Continent')
grouped
```

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7e457cbde8b0>
```

Calling an aggregation method on the object applies the function to each group, the results of which are combined in a new data structure.

For example, we can return the number of countries in our dataset for each continent using `.size()`.

In this case, our new data structure is a `Series`

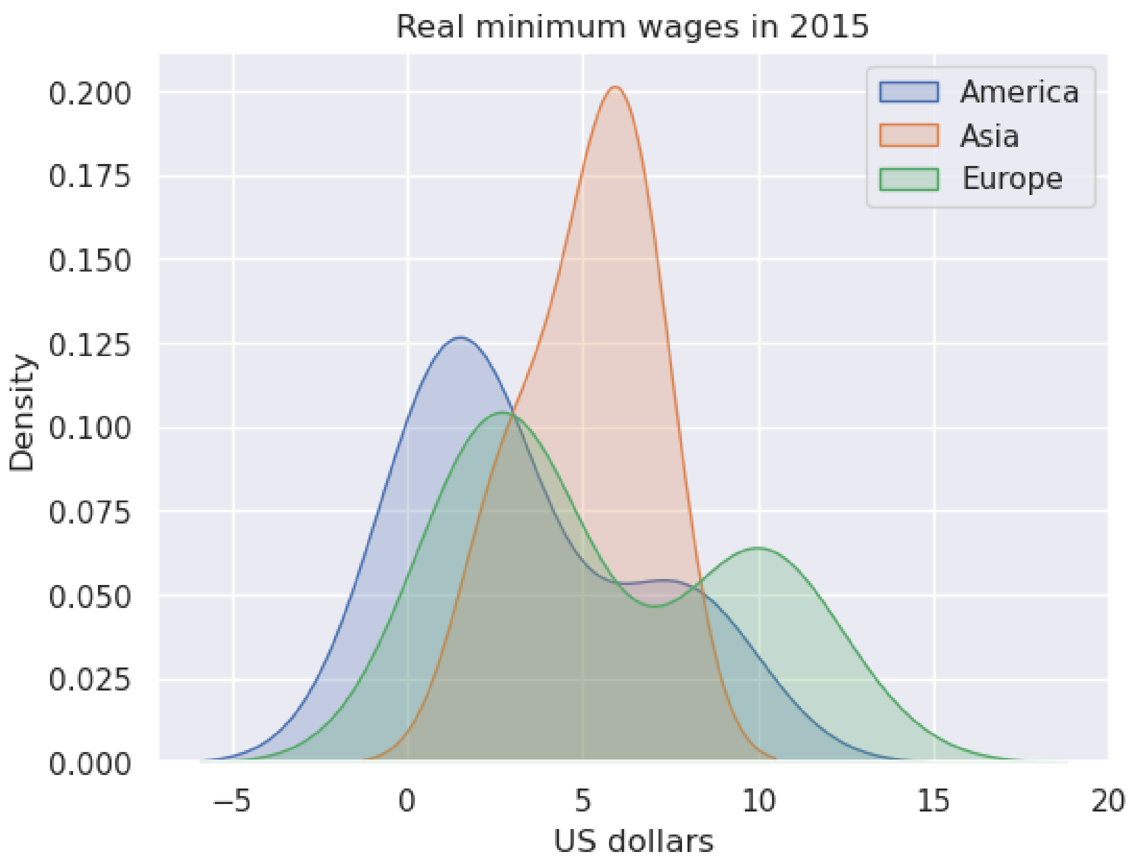
```
grouped.size()
```

```
Continent
America     7
Asia        4
Europe     19
dtype: int64
```

Calling `.get_group()` to return just the countries in a single group, we can create a kernel density estimate of the distribution of real minimum wages in 2016 for each continent.

`grouped.groups.keys()` will return the keys from the `groupby` object

```
continents = grouped.groups.keys()

for continent in continents:
    sns.kdeplot(grouped.get_group(continent).T.loc['2015'].unstack(), label=continent,
↪ fill=True)

plt.title('Real minimum wages in 2015')
plt.xlabel('US dollars')
plt.legend()
plt.show()
```



89.5 Final Remarks

This lecture has provided an introduction to some of pandas' more advanced features, including multiindices, merging, grouping and plotting.

Other tools that may be useful in panel data analysis include xarray, a python package that extends pandas to N-dimensional data structures.

89.6 Exercises

Exercise 89.6.1

In these exercises, you'll work with a dataset of employment rates in Europe by age and sex from Eurostat.

The dataset can be accessed with the following link:

```
url3 = 'https://github.com/QuantEcon/lecture-python.myst/raw/refs/heads/main/
↪lectures/_static/lecture_specific/pandas_panel/employ.csv'
```

Reading in the CSV file returns a panel dataset in long format. Use `.pivot_table()` to construct a wide format dataframe with a `MultiIndex` in the columns.

Start off by exploring the dataframe and the variables available in the `MultiIndex` levels.

Write a program that quickly returns all values in the `MultiIndex`.

Solution

```
employ = pd.read_csv(url3)
employ = employ.pivot_table(values='Value',
                            index=['DATE'],
                            columns=['UNIT','AGE', 'SEX', 'INDIC_EM', 'GEO'])
employ.index = pd.to_datetime(employ.index) # ensure that dates are datetime format
employ.head()
```

```
UNIT       Percentage of total population                        ...   \
AGE                    From 15 to 24 years                       ...
SEX                                Females                       ...
INDIC_EM                 Active population                       ...
GEO                                Austria Belgium Bulgaria      ...
DATE                                                             ...
2007-01-01                           56.00   31.60    26.00      ...
2008-01-01                           56.20   30.80    26.10      ...
2009-01-01                           56.20   29.90    24.80      ...
2010-01-01                           54.00   29.80    26.60      ...
2011-01-01                           54.80   29.80    24.80      ...

UNIT                                            Thousand persons           \
AGE                                          From 55 to 64 years
SEX                                                        Total
INDIC_EM   Total employment (resident population concept - LFS)
GEO                                                  Switzerland   Turkey
DATE
2007-01-01                                                   NaN  1,282.00
2008-01-01                                                   NaN  1,354.00
2009-01-01                                                   NaN  1,449.00
2010-01-01                                                640.00  1,583.00
2011-01-01                                                661.00  1,760.00

UNIT
AGE
SEX
INDIC_EM
GEO        United Kingdom
DATE
2007-01-01       4,131.00
2008-01-01       4,204.00
2009-01-01       4,193.00
2010-01-01       4,186.00
2011-01-01       4,164.00

[5 rows x 1440 columns]
```

This is a large dataset so it is useful to explore the levels and variables available

```
employ.columns.names
```

```
FrozenList(['UNIT', 'AGE', 'SEX', 'INDIC_EM', 'GEO'])
```

Variables within levels can be quickly retrieved with a loop

```
for name in employ.columns.names:
    print(name, employ.columns.get_level_values(name).unique())
```

```
UNIT Index(['Percentage of total population', 'Thousand persons'], dtype='object
↪', name='UNIT')
AGE Index(['From 15 to 24 years', 'From 25 to 54 years', 'From 55 to 64 years'],
↪ dtype='object', name='AGE')
SEX Index(['Females', 'Males', 'Total'], dtype='object', name='SEX')
INDIC_EM Index(['Active population', 'Total employment (resident population␣
↪concept - LFS)'], dtype='object', name='INDIC_EM')
GEO Index(['Austria', 'Belgium', 'Bulgaria', 'Croatia', 'Cyprus', 'Czech␣
↪Republic',
       'Denmark', 'Estonia', 'Euro area (17 countries)',
       'Euro area (18 countries)', 'Euro area (19 countries)',
       'European Union (15 countries)', 'European Union (27 countries)',
       'European Union (28 countries)', 'Finland',
       'Former Yugoslav Republic of Macedonia, the', 'France',
       'France (metropolitan)',
       'Germany (until 1990 former territory of the FRG)', 'Greece', 'Hungary',
       'Iceland', 'Ireland', 'Italy', 'Latvia', 'Lithuania', 'Luxembourg',
       'Malta', 'Netherlands', 'Norway', 'Poland', 'Portugal', 'Romania',
       'Slovakia', 'Slovenia', 'Spain', 'Sweden', 'Switzerland', 'Turkey',
       'United Kingdom'],
      dtype='object', name='GEO')
```

Exercise 89.6.2

Filter the above dataframe to only include employment as a percentage of 'active population'.

Create a grouped boxplot using `seaborn` of employment rates in 2015 by age group and sex.

Hint

`GEO` includes both areas and countries.

Solution

To easily filter by country, swap `GEO` to the top level and sort the `MultiIndex`

```
employ.columns = employ.columns.swaplevel(0,-1)
employ = employ.sort_index(axis=1)
```

We need to get rid of a few items in `GEO` which are not countries.

A fast way to get rid of the EU areas is to use a list comprehension to find the level values in `GEO` that begin with 'Euro'

```
geo_list = employ.columns.get_level_values('GEO').unique().tolist()
countries = [x for x in geo_list if not x.startswith('Euro')]
employ = employ[countries]
employ.columns.get_level_values('GEO').unique()
```

```
Index(['Austria', 'Belgium', 'Bulgaria', 'Croatia', 'Cyprus', 'Czech Republic',
       'Denmark', 'Estonia', 'Finland',
       'Former Yugoslav Republic of Macedonia, the', 'France',
       'France (metropolitan)',
       'Germany (until 1990 former territory of the FRG)', 'Greece', 'Hungary',
       'Iceland', 'Ireland', 'Italy', 'Latvia', 'Lithuania', 'Luxembourg',
       'Malta', 'Netherlands', 'Norway', 'Poland', 'Portugal', 'Romania',
       'Slovakia', 'Slovenia', 'Spain', 'Sweden', 'Switzerland', 'Turkey',
       'United Kingdom'],
      dtype='object', name='GEO')
```

Select only percentage employed in the active population from the dataframe

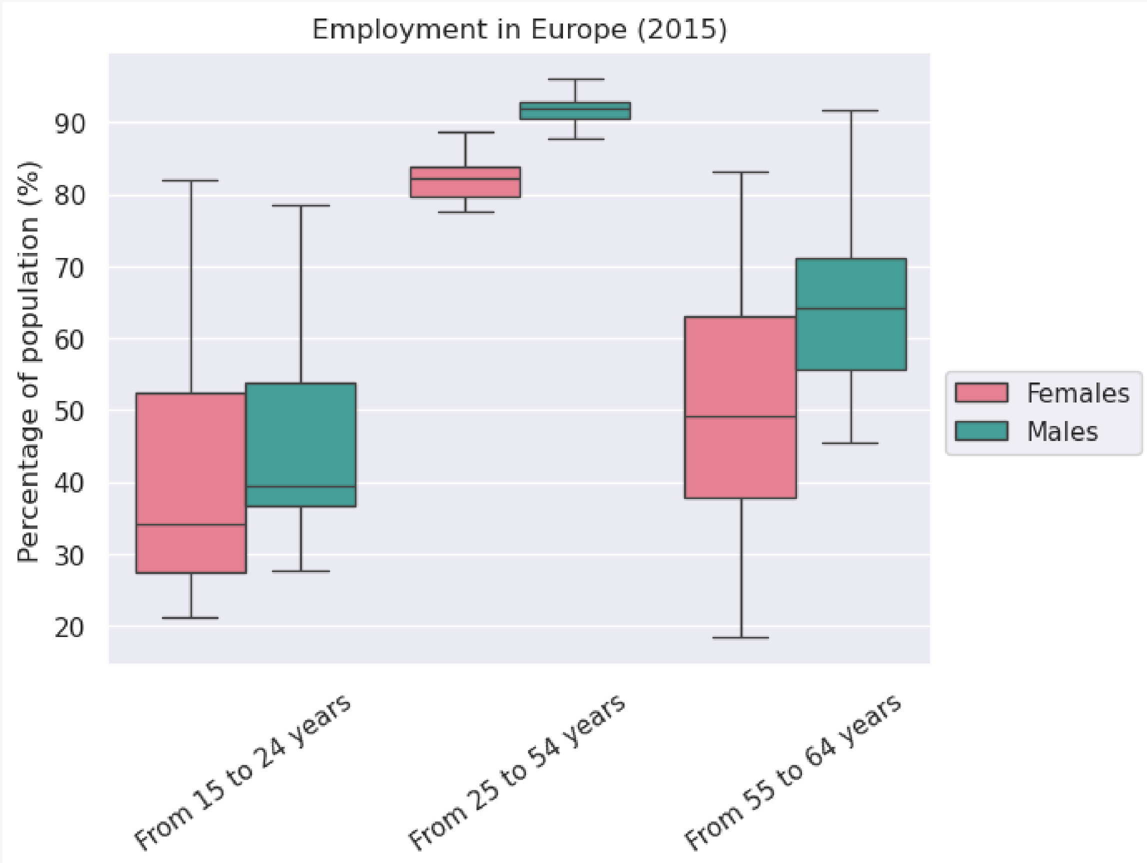
```
employ_f = employ.xs(('Percentage of total population', 'Active population'),
                     level=('UNIT', 'INDIC_EM'),
                     axis=1)
employ_f.head()
```

```
GEO                    Austria               ...      United Kingdom        \
AGE         From 15 to 24 years              ... From 55 to 64 years
SEX                    Females Males Total  ...             Females Males
DATE                                         ...
2007-01-01               56.00 62.90 59.40  ...               49.90 68.90
2008-01-01               56.20 62.90 59.50  ...               50.20 69.80
2009-01-01               56.20 62.90 59.50  ...               50.60 70.30
2010-01-01               54.00 62.60 58.30  ...               51.10 69.20
2011-01-01               54.80 63.60 59.20  ...               51.30 68.40

GEO
AGE
SEX         Total
DATE
2007-01-01 59.30
2008-01-01 59.80
2009-01-01 60.30
2010-01-01 60.00
2011-01-01 59.70

[5 rows x 306 columns]
```

Drop the 'Total' value before creating the grouped boxplot

```
employ_f = employ_f.drop('Total', level='SEX', axis=1)
```

```
box = employ_f.loc['2015'].unstack().reset_index()
sns.boxplot(x="AGE", y=0, hue="SEX", data=box, palette=("husl"), showfliers=False)
plt.xlabel('')
plt.xticks(rotation=35)
plt.ylabel('Percentage of population (%)')
plt.title('Employment in Europe (2015)')
plt.legend(bbox_to_anchor=(1,0.5))
plt.show()
```


Employment in Europe (2015)
Percentage of population (%)
90
80
70
60
50
40
30
20
From 15 to 24 years
From 25 to 54 years
From 55 to 64 years
Females
Males

CHAPTER

NINETY

LINEAR REGRESSION IN PYTHON

Contents

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install linearmodels
```

90.1 Overview

Linear regression is a standard tool for analyzing the relationship between two or more variables.

In this lecture, we'll use the Python package `statsmodels` to estimate, interpret, and visualize linear regression models.

Along the way, we'll discuss a variety of topics, including

- simple and multivariate linear regression
- visualization
- endogeneity and omitted variable bias
- two-stage least squares

As an example, we will replicate results from Acemoglu, Johnson and Robinson's seminal paper [Acemoglu *et al.*, 2001].

- You can download a copy here.

In the paper, the authors emphasize the importance of institutions in economic development.

The main contribution is the use of settler mortality rates as a source of *exogenous* variation in institutional differences.

Such variation is needed to determine whether it is institutions that give rise to greater economic growth, rather than the other way around.

Let's start with some imports:

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import statsmodels.api as sm
from statsmodels.iolib.summary2 import summary_col
from linearmodels.iv import IV2SLS
import seaborn as sns
sns.set_theme()
```

90.1.1 Prerequisites

This lecture assumes you are familiar with basic econometrics.

For an introductory text covering these topics, see, for example, [Wooldridge, 2015].

90.2 Simple Linear Regression

[Acemoglu *et al.*, 2001] wish to determine whether or not differences in institutions can help to explain observed economic outcomes.

How do we measure *institutional differences* and *economic outcomes*?

In this paper,

- economic outcomes are proxied by log GDP per capita in 1995, adjusted for exchange rates.
- institutional differences are proxied by an index of protection against expropriation on average over 1985-95, constructed by the Political Risk Services Group.

These variables and other data used in the paper are available for download on Daron Acemoglu's webpage.

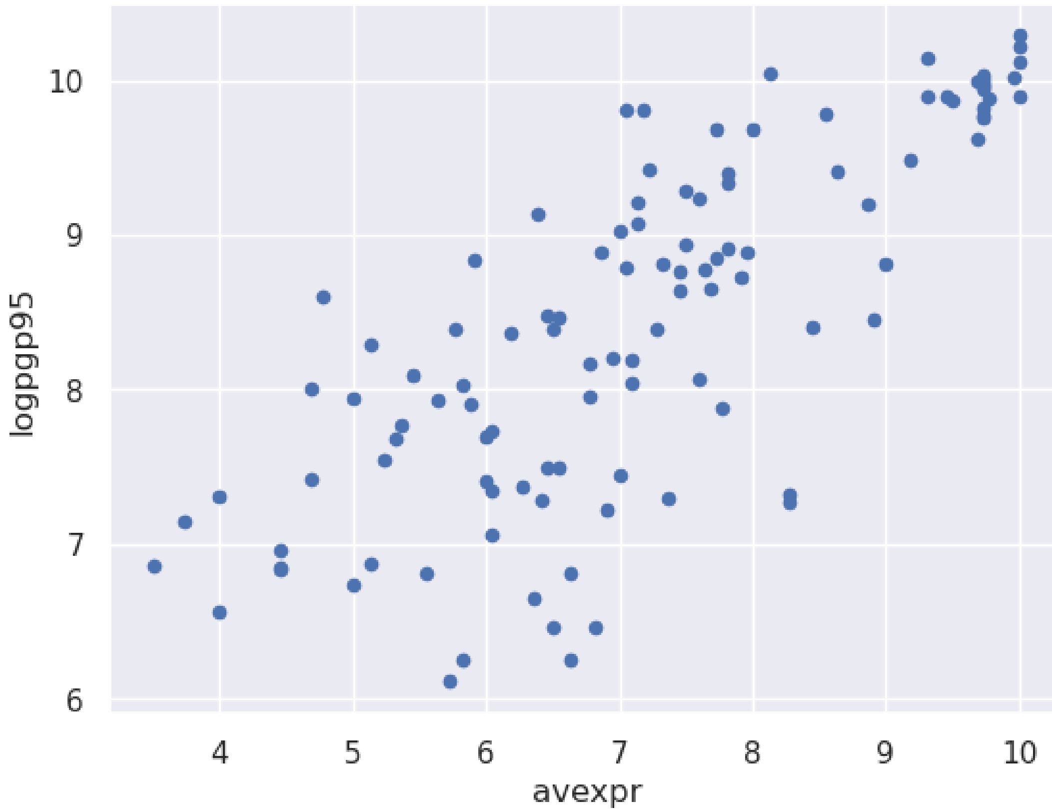
We will use pandas' `.read_stata()` function to read in data contained in the `.dta` files to dataframes

```
df1 = pd.read_stata('https://github.com/QuantEcon/lecture-python/blob/master/source/_
↪static/lecture_specific/ols/maketable1.dta?raw=true')
df1.head()
```

```
  shortnam   euro1900  excolony    avexpr  logpgp95  cons1  cons90  democ00a  \
0      AFG   0.000000       1.0       NaN       NaN    1.0     2.0       1.0
1      AGO   8.000000       1.0  5.363636  7.770645    3.0     3.0       0.0
2      ARE   0.000000       1.0  7.181818  9.804219    NaN     NaN       NaN
3      ARG  60.000004       1.0  6.386364  9.133459    1.0     6.0       3.0
4      ARM   0.000000       0.0       NaN  7.682482    NaN     NaN       NaN

   cons00a    extmort4    logem4  loghjypl  baseco
0      1.0   93.699997  4.540098       NaN     NaN
1      1.0  280.000000  5.634789 -3.411248     1.0
2      NaN         NaN       NaN       NaN     NaN
3      3.0   68.900002  4.232656 -0.872274     1.0
4      NaN         NaN       NaN       NaN     NaN
```

Let's use a scatterplot to see whether any obvious relationship exists between GDP per capita and the protection against expropriation index

```
df1.plot(x='avexpr', y='logpgp95', kind='scatter')
plt.show()
```



The plot shows a fairly strong positive relationship between protection against expropriation and log GDP per capita.

Specifically, if higher protection against expropriation is a measure of institutional quality, then better institutions appear to be positively correlated with better economic outcomes (higher GDP per capita).

Given the plot, choosing a linear model to describe this relationship seems like a reasonable assumption.

We can write our model as

$$logpgp95_i = \beta_0 + \beta_1 avexpr_i + u_i$$

where:

- β_0 is the intercept of the linear trend line on the y-axis
- β_1 is the slope of the linear trend line, representing the *marginal effect* of protection against risk on log GDP per capita
- u_i is a random error term (deviations of observations from the linear trend due to factors not included in the model)

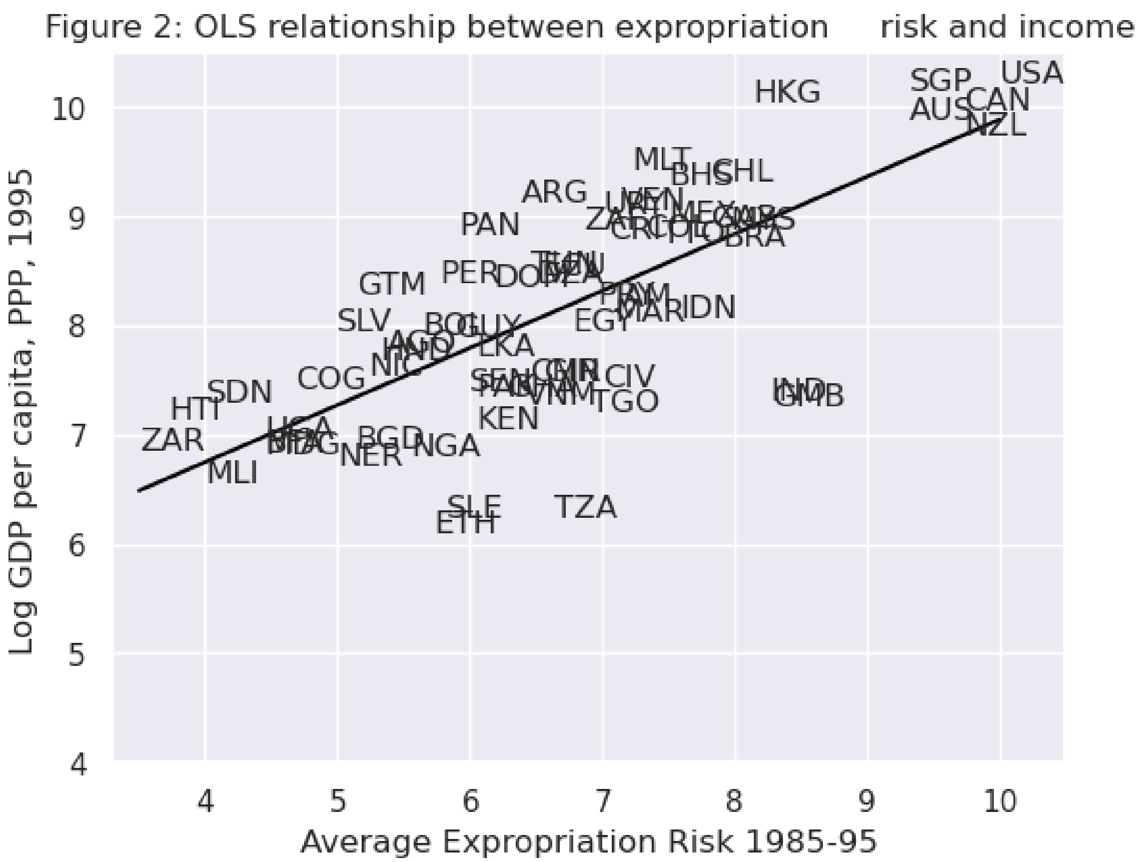
Visually, this linear model involves choosing a straight line that best fits the data, as in the following plot (Figure 2 in [Acemoglu *et al.*, 2001])

```
# Dropping NA's is required to use numpy's polyfit
df1_subset = df1.dropna(subset=['logpgp95', 'avexpr'])
```

(continues on next page)

(continued from previous page)

```
# Use only 'base sample' for plotting purposes
df1_subset = df1_subset[df1_subset['baseco'] == 1]

X = df1_subset['avexpr']
y = df1_subset['logpgp95']
labels = df1_subset['shortnam']

# Replace markers with country labels
fig, ax = plt.subplots()
ax.scatter(X, y, marker='')

for i, label in enumerate(labels):
    ax.annotate(label, (X.iloc[i], y.iloc[i]))

# Fit a linear trend line
ax.plot(np.unique(X),
         np.poly1d(np.polyfit(X, y, 1))(np.unique(X)),
         color='black')

ax.set_xlim([3.3,10.5])
ax.set_ylim([4,10.5])
ax.set_xlabel('Average Expropriation Risk 1985-95')
ax.set_ylabel('Log GDP per capita, PPP, 1995')
ax.set_title('Figure 2: OLS relationship between expropriation \
    risk and income')
plt.show()
```



The most common technique to estimate the parameters (β's) of the linear model is Ordinary Least Squares (OLS).

As the name implies, an OLS model is solved by finding the parameters that minimize **the sum of squared residuals**, i.e.

$$\min_{\hat{\beta}} \sum_{i=1}^{N} \hat{u}_i^2$$

where $\hat{u}_i$ is the difference between the observation and the predicted value of the dependent variable.

To estimate the constant term β_0, we need to add a column of 1's to our dataset (consider the equation if β_0 was replaced with $\beta_0 x_i$ and $x_i = 1$)

```
df1['const'] = 1
```

Now we can construct our model in `statsmodels` using the OLS function.

We will use `pandas` dataframes with `statsmodels`, however standard arrays can also be used as arguments

```
reg1 = sm.OLS(endog=df1['logpgp95'], exog=df1[['const', 'avexpr']], \
    missing='drop')
type(reg1)
```

```
statsmodels.regression.linear_model.OLS
```

So far we have simply constructed our model.

We need to use `.fit()` to obtain parameter estimates $\hat{\beta}_0$ and $\hat{\beta}_1$

```python
results = reg1.fit()
type(results)
```

```
statsmodels.regression.linear_model.RegressionResultsWrapper
```

We now have the fitted regression model stored in `results`.

To view the OLS regression results, we can call the `.summary()` method.

Note that an observation was mistakenly dropped from the results in the original paper (see the note located in `maketable2.do` from Acemoglu's webpage), and thus the coefficients differ slightly.

```python
print(results.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:               logpgp95   R-squared:                       0.611
Model:                            OLS   Adj. R-squared:                  0.608
Method:                 Least Squares   F-statistic:                     171.4
Date:                Mon, 16 Feb 2026   Prob (F-statistic):           4.16e-24
Time:                        04:54:56   Log-Likelihood:                -119.71
No. Observations:                 111   AIC:                             243.4
Df Residuals:                     109   BIC:                             248.8
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          4.6261      0.301     15.391      0.000       4.030       5.222
avexpr         0.5319      0.041     13.093      0.000       0.451       0.612
==============================================================================
Omnibus:                        9.251   Durbin-Watson:                   1.689
Prob(Omnibus):                  0.010   Jarque-Bera (JB):                9.170
Skew:                          -0.680   Prob(JB):                       0.0102
Kurtosis:                       3.362   Cond. No.                         33.2
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
 ↪specified.
```

From our results, we see that

- The intercept $\hat{\beta}_0 = 4.63$.
- The slope $\hat{\beta}_1 = 0.53$.
- The positive $\hat{\beta}_1$ parameter estimate implies that. institutional quality has a positive effect on economic outcomes, as we saw in the figure.
- The p-value of 0.000 for $\hat{\beta}_1$ implies that the effect of institutions on GDP is statistically significant (using $p < 0.05$ as a rejection rule).
- The R-squared value of 0.611 indicates that around 61% of variation in log GDP per capita is explained by protection against expropriation.

Using our parameter estimates, we can now write our estimated relationship as

$$\widehat{logpgp95}_i = 4.63 + 0.53 \ avexpr_i$$

This equation describes the line that best fits our data, as shown in Figure 2.

We can use this equation to predict the level of log GDP per capita for a value of the index of expropriation protection.

For example, for a country with an index value of 7.07 (the average for the dataset), we find that their predicted level of log GDP per capita in 1995 is 8.38.

```
mean_expr = np.mean(df1_subset['avexpr'])
mean_expr
```

```
np.float32(6.515625)
```

```
predicted_logpdp95 = 4.63 + 0.53 * 7.07
predicted_logpdp95
```

```
8.3771
```

An easier (and more accurate) way to obtain this result is to use `.predict()` and set $constant = 1$ and $avexpr_i = mean\_expr$

```
results.predict(exog=[1, mean_expr])
```

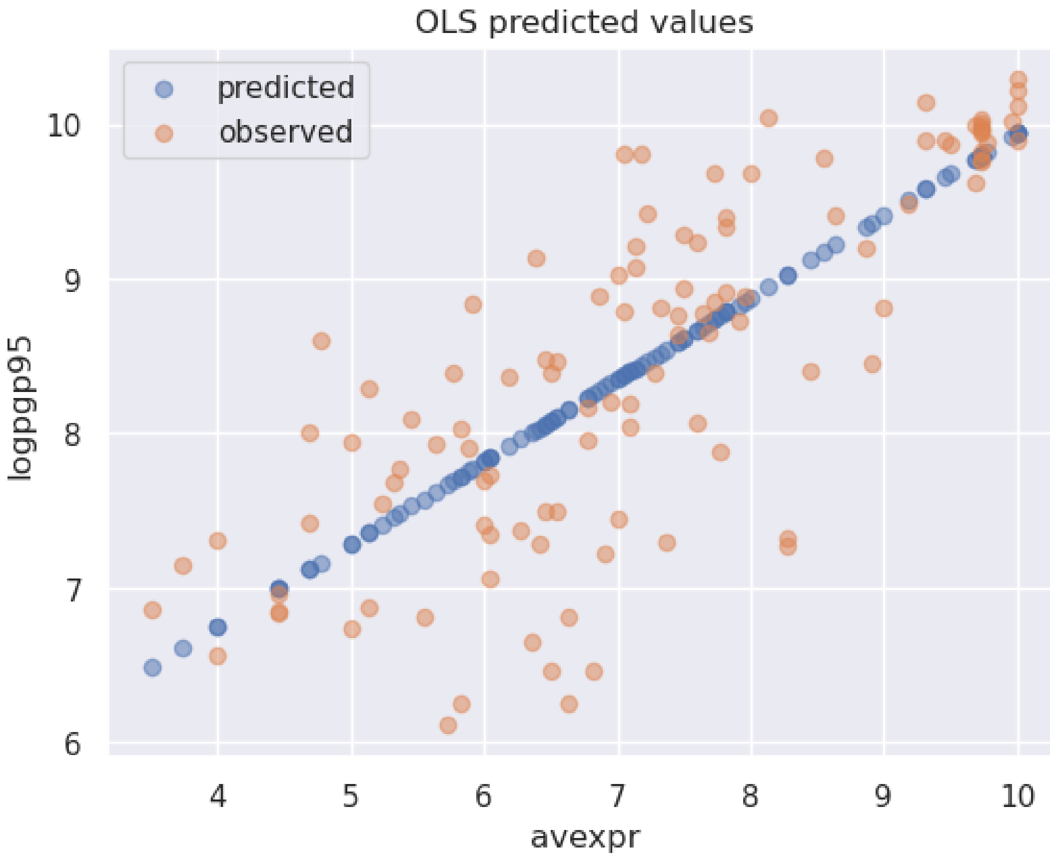
```
array([8.09156367])
```

We can obtain an array of predicted $logpgp95_i$ for every value of $avexpr_i$ in our dataset by calling `.predict()` on our results.

Plotting the predicted values against $avexpr_i$ shows that the predicted values lie along the linear line that we fitted above.

The observed values of $logpgp95_i$ are also plotted for comparison purposes

```
# Drop missing observations from whole sample

df1_plot = df1.dropna(subset=['logpgp95', 'avexpr'])

# Plot predicted values

fix, ax = plt.subplots()
ax.scatter(df1_plot['avexpr'], results.predict(), alpha=0.5,
        label='predicted')

# Plot observed values

ax.scatter(df1_plot['avexpr'], df1_plot['logpgp95'], alpha=0.5,
        label='observed')

ax.legend()
ax.set_title('OLS predicted values')
ax.set_xlabel('avexpr')
ax.set_ylabel('logpgp95')
plt.show()
```



90.3 Extending the Linear Regression Model

So far we have only accounted for institutions affecting economic performance - almost certainly there are numerous other factors affecting GDP that are not included in our model.

Leaving out variables that affect $logpgp95_i$ will result in **omitted variable bias**, yielding biased and inconsistent parameter estimates.

We can extend our bivariate regression model to a **multivariate regression model** by adding in other factors that may affect $logpgp95_i$.

[Acemoglu *et al.*, 2001] consider other factors such as:

- the effect of climate on economic outcomes; latitude is used to proxy this
- differences that affect both economic performance and institutions, eg. cultural, historical, etc.; controlled for with the use of continent dummies

Let's estimate some of the extended models considered in the paper (Table 2) using data from `maketable2.dta`

```
df2 = pd.read_stata('https://github.com/QuantEcon/lecture-python/blob/master/source/_
↪static/lecture_specific/ols/maketable2.dta?raw=true')

# Add constant term to dataset
df2['const'] = 1
```

(continues on next page)

(continued from previous page)

```
# Create lists of variables to be used in each regression
X1 = ['const', 'avexpr']
X2 = ['const', 'avexpr', 'lat_abst']
X3 = ['const', 'avexpr', 'lat_abst', 'asia', 'africa', 'other']

# Estimate an OLS regression for each set of variables
reg1 = sm.OLS(df2['logpgp95'], df2[X1], missing='drop').fit()
reg2 = sm.OLS(df2['logpgp95'], df2[X2], missing='drop').fit()
reg3 = sm.OLS(df2['logpgp95'], df2[X3], missing='drop').fit()
```

Now that we have fitted our model, we will use `summary_col` to display the results in a single table (model numbers correspond to those in the paper)

```
info_dict={'R-squared' : lambda x: f"{x.rsquared:.2f}",
           'No. observations' : lambda x: f"{int(x.nobs):d}"}

results_table = summary_col(results=[reg1,reg2,reg3],
                            float_format='%0.2f',
                            stars = True,
                            model_names=['Model 1',
                                         'Model 3',
                                         'Model 4'],
                            info_dict=info_dict,
                            regressor_order=['const',
                                             'avexpr',
                                             'lat_abst',
                                             'asia',
                                             'africa'])

results_table.add_title('Table 2 - OLS Regressions')

print(results_table)
```

```
        Table 2 - OLS Regressions
=========================================
                 Model 1 Model 3 Model 4
-----------------------------------------
const            4.63*** 4.87*** 5.85***
                 (0.30)  (0.33)  (0.34)
avexpr           0.53*** 0.46*** 0.39***
                 (0.04)  (0.06)  (0.05)
lat_abst                 0.87*   0.33
                         (0.49)  (0.45)
asia                             -0.15
                                 (0.15)
africa                           -0.92***
                                 (0.17)
other                            0.30
                                 (0.37)
R-squared        0.61    0.62    0.72
R-squared Adj.   0.61    0.62    0.70
No. observations 111     111     111
R-squared        0.61    0.62    0.72
=========================================
Standard errors in parentheses.
* p<.1, ** p<.05, ***p<.01
```

90.4 Endogeneity

As [Acemoglu *et al.*, 2001] discuss, the OLS models likely suffer from **endogeneity** issues, resulting in biased and inconsistent model estimates.

Namely, there is likely a two-way relationship between institutions and economic outcomes:

- richer countries may be able to afford or prefer better institutions
- variables that affect income may also be correlated with institutional differences
- the construction of the index may be biased; analysts may be biased towards seeing countries with higher income having better institutions

To deal with endogeneity, we can use **two-stage least squares (2SLS) regression**, which is an extension of OLS regression.

This method requires replacing the endogenous variable $avexpr_i$ with a variable that is:

1. correlated with $avexpr_i$
2. not correlated with the error term (ie. it should not directly affect the dependent variable, otherwise it would be correlated with u_i due to omitted variable bias)

The new set of regressors is called an **instrument**, which aims to remove endogeneity in our proxy of institutional differences.

The main contribution of [Acemoglu *et al.*, 2001] is the use of settler mortality rates to instrument for institutional differences.

They hypothesize that higher mortality rates of colonizers led to the establishment of institutions that were more extractive in nature (less protection against expropriation), and these institutions still persist today.

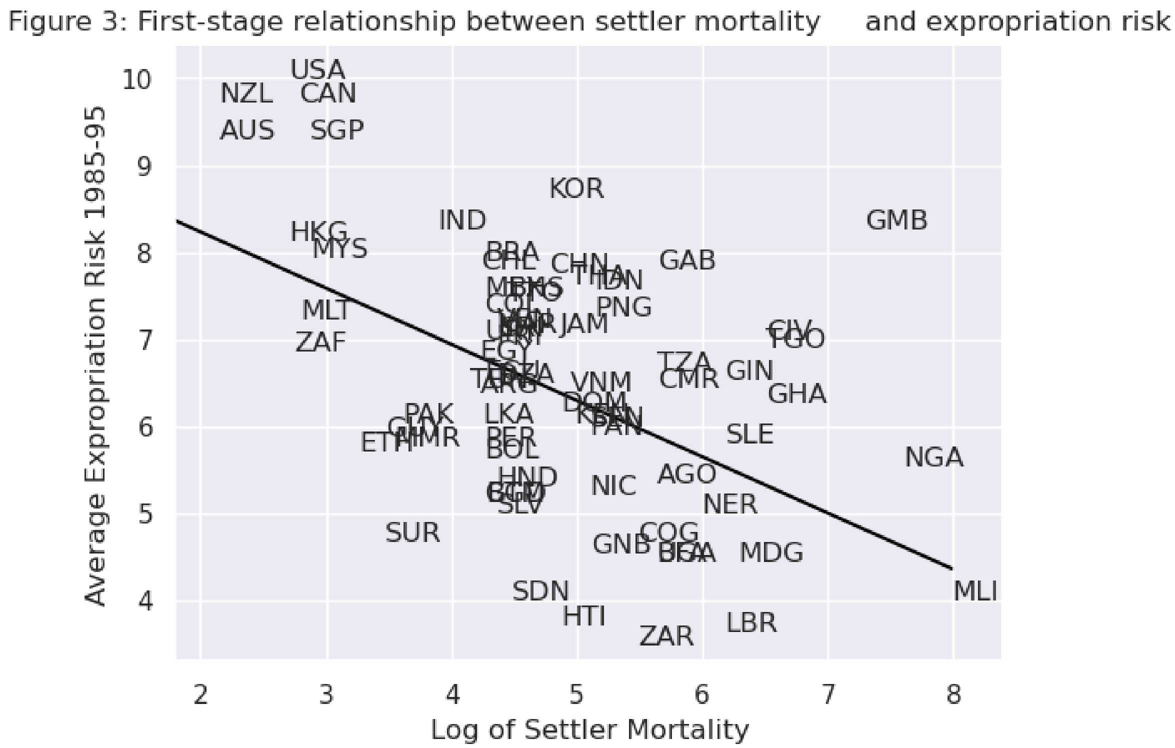
Using a scatterplot (Figure 3 in [Acemoglu *et al.*, 2001]), we can see protection against expropriation is negatively correlated with settler mortality rates, coinciding with the authors' hypothesis and satisfying the first condition of a valid instrument.

```python
# Dropping NA's is required to use numpy's polyfit
df1_subset2 = df1.dropna(subset=['logem4', 'avexpr'])

X = df1_subset2['logem4']
y = df1_subset2['avexpr']
labels = df1_subset2['shortnam']

# Replace markers with country labels
fig, ax = plt.subplots()
ax.scatter(X, y, marker='')

for i, label in enumerate(labels):
    ax.annotate(label, (X.iloc[i], y.iloc[i]))

# Fit a linear trend line
ax.plot(np.unique(X),
         np.poly1d(np.polyfit(X, y, 1))(np.unique(X)),
         color='black')

ax.set_xlim([1.8,8.4])
ax.set_ylim([3.3,10.4])
ax.set_xlabel('Log of Settler Mortality')
ax.set_ylabel('Average Expropriation Risk 1985-95')
ax.set_title('Figure 3: First-stage relationship between settler mortality \
```

(continues on next page)

(continued from previous page)

```
    and expropriation risk')
plt.show()
```



The second condition may not be satisfied if settler mortality rates in the 17th to 19th centuries have a direct effect on current GDP (in addition to their indirect effect through institutions).

For example, settler mortality rates may be related to the current disease environment in a country, which could affect current economic performance.

[Acemoglu *et al.*, 2001] argue this is unlikely because:

- The majority of settler deaths were due to malaria and yellow fever and had a limited effect on local people.
- The disease burden on local people in Africa or India, for example, did not appear to be higher than average, supported by relatively high population densities in these areas before colonization.

As we appear to have a valid instrument, we can use 2SLS regression to obtain consistent and unbiased parameter estimates.

First stage

The first stage involves regressing the endogenous variable ($avexpr_i$) on the instrument.

The instrument is the set of all exogenous variables in our model (and not just the variable we have replaced).

Using model 1 as an example, our instrument is simply a constant and settler mortality rates $logem4_i$.

Therefore, we will estimate the first-stage regression as

$$avexpr_i = \delta_0 + \delta_1 logem4_i + v_i$$

The data we need to estimate this equation is located in `maketable4.dta` (only complete data, indicated by `baseco = 1`, is used for estimation)

```
# Import and select the data
df4 = pd.read_stata('https://github.com/QuantEcon/lecture-python/blob/master/source/_
↪static/lecture_specific/ols/maketable4.dta?raw=true')
df4 = df4[df4['baseco'] == 1]

# Add a constant variable
df4['const'] = 1

# Fit the first stage regression and print summary
results_fs = sm.OLS(df4['avexpr'],
                    df4[['const', 'logem4']],
                    missing='drop').fit()
print(results_fs.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                 avexpr   R-squared:                       0.270
Model:                            OLS   Adj. R-squared:                  0.258
Method:                 Least Squares   F-statistic:                     22.95
Date:                Mon, 16 Feb 2026   Prob (F-statistic):           1.08e-05
Time:                        04:54:57   Log-Likelihood:                -104.83
No. Observations:                  64   AIC:                             213.7
Df Residuals:                      62   BIC:                             218.0
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          9.3414      0.611     15.296      0.000       8.121      10.562
logem4        -0.6068      0.127     -4.790      0.000      -0.860      -0.354
==============================================================================
Omnibus:                        0.035   Durbin-Watson:                   2.003
Prob(Omnibus):                  0.983   Jarque-Bera (JB):                0.172
Skew:                           0.045   Prob(JB):                        0.918
Kurtosis:                       2.763   Cond. No.                         19.4
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
↪specified.
```

Second stage

We need to retrieve the predicted values of $avexpr_i$ using `.predict()`.

We then replace the endogenous variable $avexpr_i$ with the predicted values $\widehat{avexpr}_i$ in the original linear model.

Our second stage regression is thus

$$logpgp95_i = \beta_0 + \beta_1 \widehat{avexpr}_i + u_i$$

```
df4['predicted_avexpr'] = results_fs.predict()

results_ss = sm.OLS(df4['logpgp95'],
                    df4[['const', 'predicted_avexpr']]).fit()
print(results_ss.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:               logpgp95   R-squared:                       0.477
Model:                            OLS   Adj. R-squared:                  0.469
Method:                 Least Squares   F-statistic:                     56.60
Date:                Mon, 16 Feb 2026   Prob (F-statistic):           2.66e-10
Time:                        04:54:57   Log-Likelihood:                -72.268
No. Observations:                  64   AIC:                             148.5
Df Residuals:                      62   BIC:                             152.9
Df Model:                           1
Covariance Type:            nonrobust
====================================================================================
                       coef    std err          t      P>|t|      [0.025      0.
↪975]
-----------------------------------------------------------------------------------
↪-
const                1.9097      0.823      2.320      0.024       0.264       3.
↪555
predicted_avexpr     0.9443      0.126      7.523      0.000       0.693       1.
↪195
==============================================================================
Omnibus:                       10.547   Durbin-Watson:                   2.137
Prob(Omnibus):                  0.005   Jarque-Bera (JB):               11.010
Skew:                          -0.790   Prob(JB):                      0.00407
Kurtosis:                       4.277   Cond. No.                         58.1
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
↪specified.
```

The second-stage regression results give us an unbiased and consistent estimate of the effect of institutions on economic outcomes.

The result suggests a stronger positive relationship than what the OLS results indicated.

Note that while our parameter estimates are correct, our standard errors are not and for this reason, computing 2SLS 'manually' (in stages with OLS) is not recommended.

We can correctly estimate a 2SLS regression in one step using the linearmodels package, an extension of `statsmodels`

Note that when using `IV2SLS`, the exogenous and instrument variables are split up in the function arguments (whereas before the instrument included exogenous variables)

```
iv = IV2SLS(dependent=df4['logpgp95'],
            exog=df4['const'],
            endog=df4['avexpr'],
            instruments=df4['logem4']).fit(cov_type='unadjusted')

print(iv.summary)
```

```
                          IV-2SLS Estimation Summary
==============================================================================
Dep. Variable:               logpgp95   R-squared:                      0.1870
Estimator:                    IV-2SLS   Adj. R-squared:                 0.1739
No. Observations:                  64   F-statistic:                    37.568
Date:                Mon, Feb 16 2026   P-value (F-stat)                0.0000
Time:                        04:54:57   Distribution:                  chi2(1)
```

(continues on next page)

(continued from previous page)

```
Cov. Estimator:            unadjusted

                             Parameter Estimates
==============================================================================
            Parameter  Std. Err.     T-stat    P-value    Lower CI    Upper CI
------------------------------------------------------------------------------
const          1.9097     1.0106     1.8897     0.0588     -0.0710      3.8903
avexpr         0.9443     0.1541     6.1293     0.0000      0.6423      1.2462
==============================================================================

Endogenous: avexpr
Instruments: logem4
Unadjusted Covariance (Homoskedastic)
Debiased: False
```

Given that we now have consistent and unbiased estimates, we can infer from the model we have estimated that institutional differences (stemming from institutions set up during colonization) can help to explain differences in income levels across countries today.

[Acemoglu *et al.*, 2001] use a marginal effect of 0.94 to calculate that the difference in the index between Chile and Nigeria (ie. institutional quality) implies up to a 7-fold difference in income, emphasizing the significance of institutions in economic development.

90.5 Summary

We have demonstrated basic OLS and 2SLS regression in `statsmodels` and `linearmodels`.

If you are familiar with R, you may want to use the formula interface to `statsmodels`, or consider using r2py to call R from within Python.

90.6 Exercises

Exercise 90.6.1

In the lecture, we think the original model suffers from endogeneity bias due to the likely effect income has on institutional development.

Although endogeneity is often best identified by thinking about the data and model, we can formally test for endogeneity using the **Hausman test**.

We want to test for correlation between the endogenous variable, $avexpr_i$, and the errors, u_i

$$H_0 : Cov(avexpr_i, u_i) = 0 \quad (no\ endogeneity)$$
$$H_1 : Cov(avexpr_i, u_i) \neq 0 \quad (endogeneity)$$

This test is running in two stages.

First, we regress $avexpr_i$ on the instrument, $logem4_i$

$$avexpr_i = \pi_0 + \pi_1 logem4_i + \upsilon_i$$

Second, we retrieve the residuals $\hat{\upsilon}_i$ and include them in the original equation

$$logpgp95_i = \beta_0 + \beta_1 avexpr_i + \alpha\hat{\upsilon}_i + u_i$$

If α is statistically significant (with a p-value < 0.05), then we reject the null hypothesis and conclude that $avexpr_i$ is endogenous.

Using the above information, estimate a Hausman test and interpret your results.

Solution

```
# Load in data
df4 = pd.read_stata('https://github.com/QuantEcon/lecture-python.myst/raw/refs/
↪heads/main/lectures/_static/lecture_specific/ols/maketable4.dta')

# Add a constant term
df4['const'] = 1

# Estimate the first stage regression
reg1 = sm.OLS(endog=df4['avexpr'],
              exog=df4[['const', 'logem4']],
              missing='drop').fit()

# Retrieve the residuals
df4['resid'] = reg1.resid

# Estimate the second stage residuals
reg2 = sm.OLS(endog=df4['logpgp95'],
              exog=df4[['const', 'avexpr', 'resid']],
              missing='drop').fit()

print(reg2.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:               logpgp95   R-squared:                       0.689
Model:                            OLS   Adj. R-squared:                  0.679
Method:                 Least Squares   F-statistic:                     74.05
Date:                Mon, 16 Feb 2026   Prob (F-statistic):           1.07e-17
Time:                        04:54:57   Log-Likelihood:                -62.031
No. Observations:                  70   AIC:                             130.1
Df Residuals:                      67   BIC:                             136.8
Df Model:                           2
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          2.4782      0.547      4.530      0.000       1.386       3.570
avexpr         0.8564      0.082     10.406      0.000       0.692       1.021
resid         -0.4951      0.099     -5.017      0.000      -0.692      -0.298
==============================================================================
Omnibus:                       17.597   Durbin-Watson:                   2.086
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               23.194
Skew:                          -1.054   Prob(JB):                     9.19e-06
Kurtosis:                       4.873   Cond. No.                         53.8
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is␣
↪correctly specified.
```

The output shows that the coefficient on the residuals is statistically significant, indicating $avexpr_i$ is endogenous.

Exercise 90.6.2

The OLS parameter β can also be estimated using matrix algebra and `numpy` (you may need to review the numpy lecture to complete this exercise).

The linear equation we want to estimate is (written in matrix form)

$$y = X\beta + u$$

To solve for the unknown parameter β, we want to minimize the sum of squared residuals

$$\min_{\hat{\beta}} \hat{u}'\hat{u}$$

Rearranging the first equation and substituting into the second equation, we can write

$$\min_{\hat{\beta}} (Y - X\hat{\beta})'(Y - X\hat{\beta})$$

Solving this optimization problem gives the solution for the $\hat{\beta}$ coefficients

$$\hat{\beta} = (X'X)^{-1}X'y$$

Using the above information, compute $\hat{\beta}$ from model 1 using `numpy` - your results should be the same as those in the `statsmodels` output from earlier in the lecture.

Solution

```
# Load in data
df1 = pd.read_stata('https://github.com/QuantEcon/lecture-python.myst/raw/refs/
↪heads/main/lectures/_static/lecture_specific/ols/maketable1.dta')
df1 = df1.dropna(subset=['logpgp95', 'avexpr'])

# Add a constant term
df1['const'] = 1

# Define the X and y variables
y = np.asarray(df1['logpgp95'])
X = np.asarray(df1[['const', 'avexpr']])

# Compute β_hat
β_hat = np.linalg.solve(X.T @ X, X.T @ y)

# Print out the results from the 2 x 1 vector β_hat
print(f'β_0 = {β_hat[0]:.2}')
print(f'β_1 = {β_hat[1]:.2}')
```

```
β_0 = 4.6
β_1 = 0.53
```

It is also possible to use `np.linalg.inv(X.T @ X) @ X.T @ y` to solve for β, however `.solve()` is preferred as it involves fewer computations.

CHAPTER

NINETYONE

MAXIMUM LIKELIHOOD ESTIMATION

GPU

This lecture was built using a machine with access to a GPU — although it will also run without one.

Google Colab has a free tier with GPUs that you can access as follows:

1. Click on the "play" icon top right
2. Select Colab
3. Set the runtime environment to include a GPU

Contents

91.1 Overview

In *Linear Regression in Python*, we estimated the relationship between dependent and explanatory variables using linear regression.

But what if a linear relationship is not an appropriate assumption for our model?

One widely used alternative is maximum likelihood estimation, which involves specifying a class of distributions, indexed by unknown parameters, and then using the data to pin down these parameter values.

The benefit relative to linear regression is that it allows more flexibility in the probabilistic relationships between variables.

Here we illustrate maximum likelihood by replicating Daniel Treisman's (2016) paper, Russia's Billionaires, which connects the number of billionaires in a country to its economic characteristics.

The paper concludes that Russia has a higher number of billionaires than economic factors such as market size and tax rate predict.

We'll require the following imports:

```
import numpy as np
import jax.numpy as jnp
import jax
import pandas as pd
from typing import NamedTuple

from jax.scipy.special import factorial, gammaln
from jax.scipy.stats import norm

from statsmodels.api import Poisson
from statsmodels.iolib.summary2 import summary_col

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

91.1.1 Prerequisites

We assume familiarity with basic probability and multivariate calculus.

91.2 Set up and assumptions

Let's consider the steps we need to go through in maximum likelihood estimation and how they pertain to this study.

91.2.1 Flow of ideas

The first step with maximum likelihood estimation is to choose the probability distribution believed to be generating the data.

More precisely, we need to make an assumption as to which *parametric class* of distributions is generating the data.

- e.g., the class of all normal distributions, or the class of all gamma distributions.

Each such class is a family of distributions indexed by a finite number of parameters.

- e.g., the class of normal distributions is a family of distributions indexed by its mean $\mu \in (-\infty, \infty)$ and standard deviation $\sigma \in (0, \infty)$.

We'll let the data pick out a particular element of the class by pinning down the parameters.

The parameter estimates so produced will be called **maximum likelihood estimates**.

91.2.2 Counting billionaires

Treisman [Treisman, 2016] is interested in estimating the number of billionaires in different countries.

The number of billionaires is integer-valued.

Hence we consider distributions that take values only in the nonnegative integers.

(This is one reason least squares regression is not the best tool for the present problem, since the dependent variable in linear regression is not restricted to integer values.)

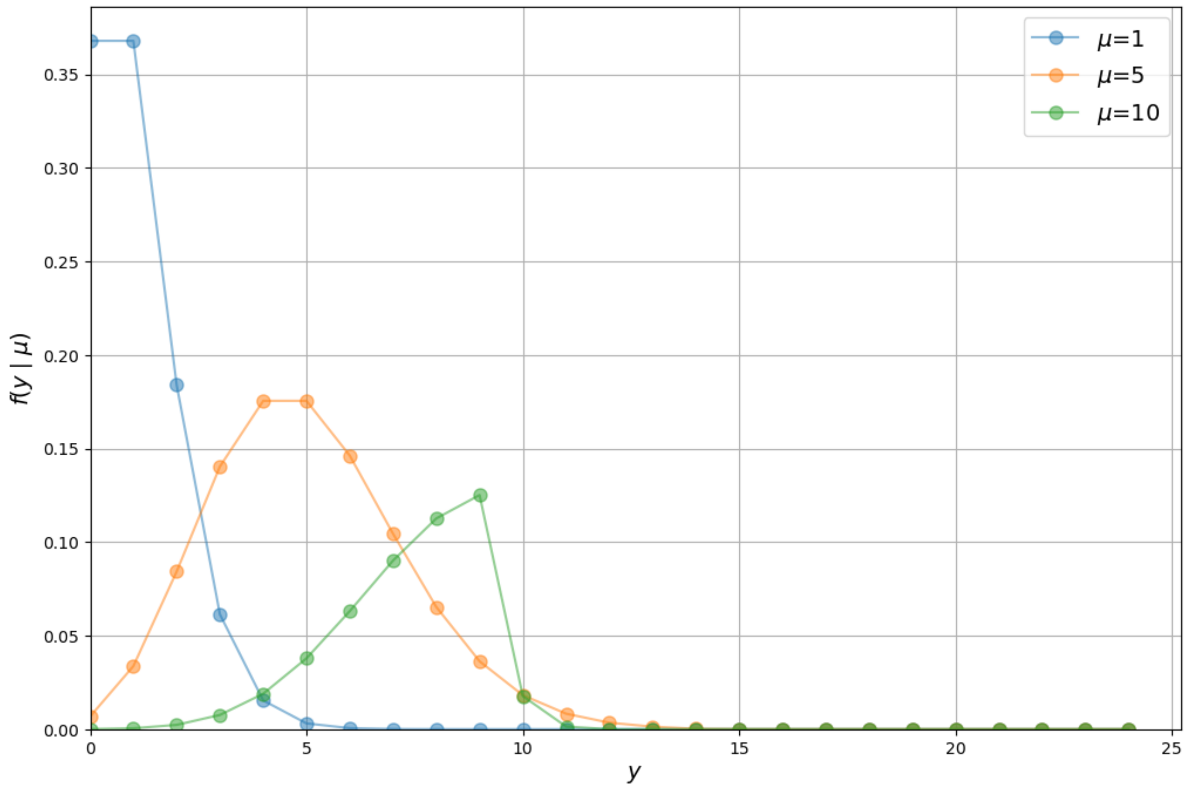
One integer distribution is the Poisson distribution, the probability mass function (pmf) of which is

$$f(y) = \frac{\mu^y}{y!} e^{-\mu}, \qquad y = 0, 1, 2, \ldots, \infty$$

We can plot the Poisson distribution over y for different values of μ as follows

```
@jax.jit
def poisson_pmf(y, μ):
    return μ**y / factorial(y) * jnp.exp(-μ)
```

```
y_values = range(0, 25)

fig, ax = plt.subplots(figsize=(12, 8))

for μ in [1, 5, 10]:
    distribution = []
    for y_i in y_values:
        distribution.append(poisson_pmf(y_i, μ))
    ax.plot(
        y_values,
        distribution,
        label=rf"$\mu$={μ}",
        alpha=0.5,
        marker="o",
        markersize=8,
    )

ax.grid()
ax.set_xlabel(r"$y$", fontsize=14)
ax.set_ylabel(r"$f(y \mid \mu)$", fontsize=14)
ax.axis(xmin=0, ymin=0)
ax.legend(fontsize=14)

plt.show()
```



Notice that the Poisson distribution begins to resemble a normal distribution as the mean of y increases.

Let's have a look at the distribution of the data we'll be working with in this lecture.

Treisman's main source of data is *Forbes'* annual rankings of billionaires and their estimated net worth.

The dataset `mle/fp.dta` can be downloaded from here or its AER page.

```
# Load in data and view
df = pd.read_stata(
    "https://github.com/QuantEcon/lecture-python.myst/raw/refs/heads/main/lectures/_
↪static/lecture_specific/mle/fp.dta"
)
df.head()
```

```
         country  ccode    year    cyear  numbil  numbil0  numbilall  netw  \
0  United States    2.0  1990.0  21990.0     NaN      NaN        NaN   NaN
1  United States    2.0  1991.0  21991.0     NaN      NaN        NaN   NaN
2  United States    2.0  1992.0  21992.0     NaN      NaN        NaN   NaN
3  United States    2.0  1993.0  21993.0     NaN      NaN        NaN   NaN
4  United States    2.0  1994.0  21994.0     NaN      NaN        NaN   NaN

   netw0  netwall  ...  gattwto08     mcapbdol    mcapbdol08  lnmcap08  \
0    NaN      NaN  ...        0.0  3060.000000  11737.599609  9.370638
1    NaN      NaN  ...        0.0  4090.000000  11737.599609  9.370638
2    NaN      NaN  ...        0.0  4490.000000  11737.599609  9.370638
3    NaN      NaN  ...        0.0  5136.198730  11737.599609  9.370638
4    NaN      NaN  ...        0.0  5067.016113  11737.599609  9.370638

   topintaxnew   topint08     rintr  noyrs  roflaw  nrrents
0    39.799999  39.799999  4.988405   20.0    1.61      NaN
```

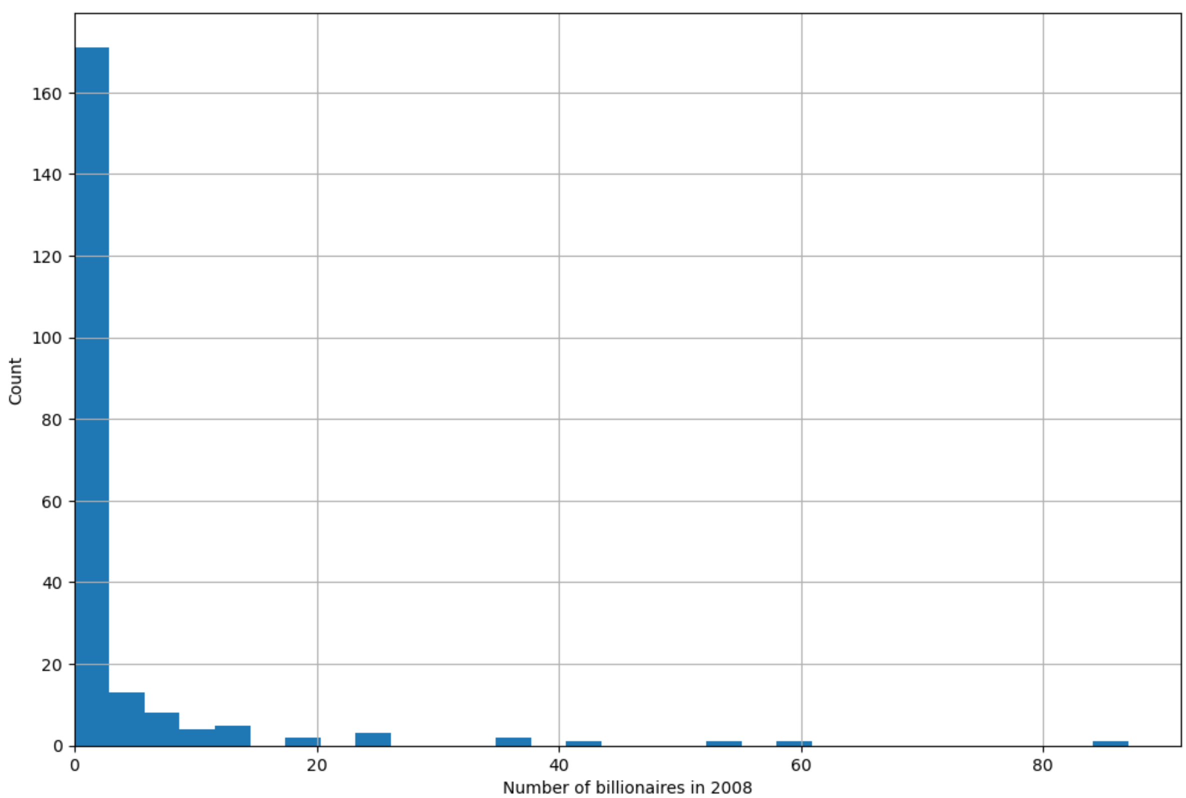
(continues on next page)

(continued from previous page)

```
1    39.799999  39.799999  4.988405   20.0     1.61       NaN
2    39.799999  39.799999  4.988405   20.0     1.61       NaN
3    39.799999  39.799999  4.988405   20.0     1.61       NaN
4    39.799999  39.799999  4.988405   20.0     1.61       NaN

[5 rows x 36 columns]
```

Using a histogram, we can view the distribution of the number of billionaires per country, `numbil0`, in 2008 (the United States is dropped for plotting purposes)

```
numbil0_2008 = df[
    (df["year"] == 2008) & (df["country"] != "United States")
].loc[:, "numbil0"]

plt.subplots(figsize=(12, 8))
plt.hist(numbil0_2008, bins=30)
plt.xlim(left=0)
plt.grid()
plt.xlabel("Number of billionaires in 2008")
plt.ylabel("Count")
plt.show()
```



From the histogram, it appears that the Poisson assumption is not unreasonable (albeit with a very low μ and some outliers).

91.3 Conditional distributions

In Treisman's paper, the dependent variable — the number of billionaires y_i in country i — is modeled as a function of GDP per capita, population size, and years membership in GATT and WTO.

Hence, the distribution of y_i needs to be conditioned on the vector of explanatory variables $\mathbf{x}_i$.

The standard formulation — the so-called **Poisson regression** model — is as follows:

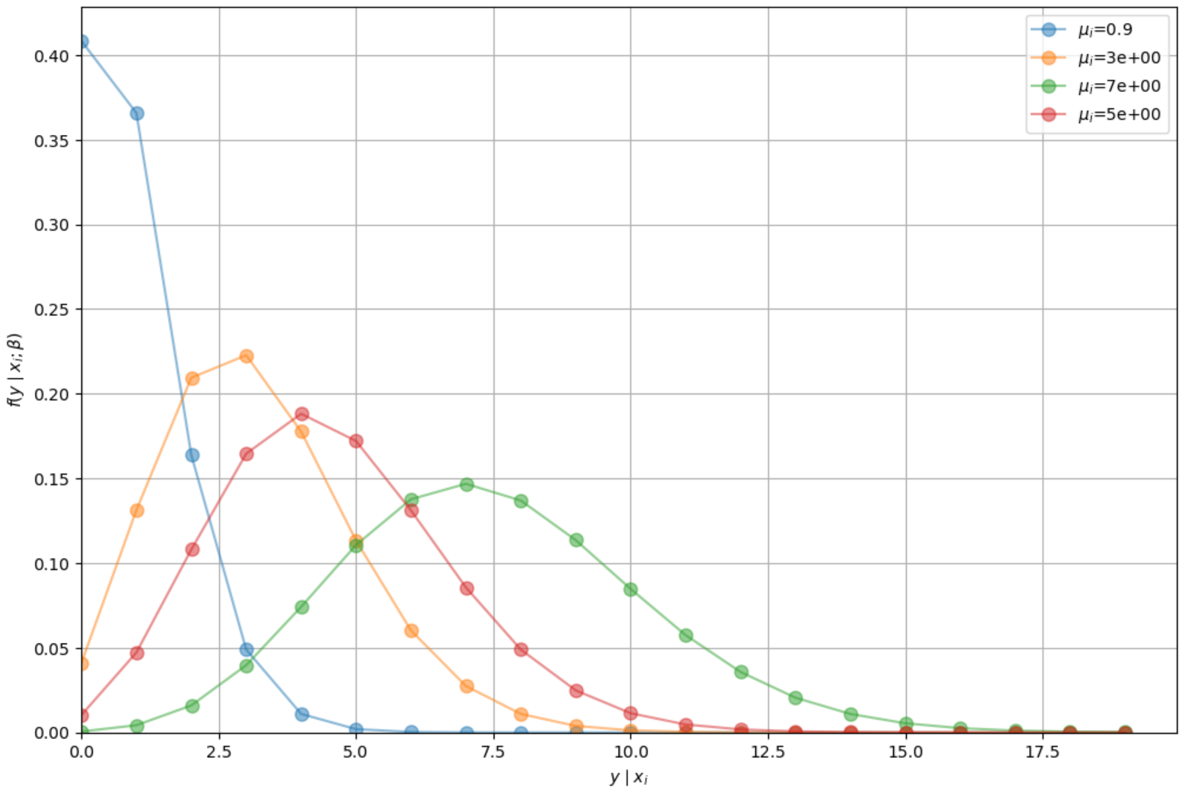
$$f(y_i \mid \mathbf{x}_i) = \frac{\mu_i^{y_i}}{y_i!} e^{-\mu_i}; \qquad y_i = 0, 1, 2, \ldots, \infty. \tag{91.1}$$

$$\text{where } \mu_i = \exp(\mathbf{x}_i' \beta) = \exp(\beta_0 + \beta_1 x_{i1} + \ldots + \beta_k x_{ik})$$

To illustrate the idea that the distribution of y_i depends on $\mathbf{x}_i$ let's run a simple simulation.

We use our `poisson_pmf` function from above and arbitrary values for β and $\mathbf{x}_i$

```
y_values = range(0, 20)

# Define a parameter vector with estimates
β = jnp.array([0.26, 0.18, 0.25, -0.1, -0.22])

# Create some observations X
datasets = [
    jnp.array([0, 1, 1, 1, 2]),
    jnp.array([2, 3, 2, 4, 0]),
    jnp.array([3, 4, 5, 3, 2]),
    jnp.array([6, 5, 4, 4, 7]),
]


fig, ax = plt.subplots(figsize=(12, 8))

for X in datasets:
    μ = jnp.exp(X @ β)
    distribution = []
    for y_i in y_values:
        distribution.append(poisson_pmf(y_i, μ))
    ax.plot(
        y_values,
        distribution,
        label=rf"$\mu_i$={μ:.1}",
        marker="o",
        markersize=8,
        alpha=0.5,
    )

ax.grid()
ax.legend()
ax.set_xlabel(r"$y \mid x_i$")
ax.set_ylabel(r"$f(y \mid x_i; \beta )$")
ax.axis(xmin=0, ymin=0)
plt.show()
```



We can see that the distribution of y_i is conditional on $\mathbf{x}_i$ (μ_i is no longer constant).

91.4 Maximum likelihood estimation

In our model for number of billionaires, the conditional distribution contains 4 ($k = 4$) parameters that we need to estimate.

We will label our entire parameter vector as β where

$$\beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix}$$

To estimate the model using MLE, we want to maximize the likelihood that our estimate $\hat{\beta}$ is the true parameter β.

Intuitively, we want to find the $\hat{\beta}$ that best fits our data.

First, we need to construct the likelihood function $\mathcal{L}(\beta)$, which is similar to a joint probability density function.

Assume we have some data $y_i = \{y_1, y_2\}$ and $y_i \sim f(y_i)$.

If y_1 and y_2 are independent, the joint pmf of these data is $f(y_1, y_2) = f(y_1) \cdot f(y_2)$.

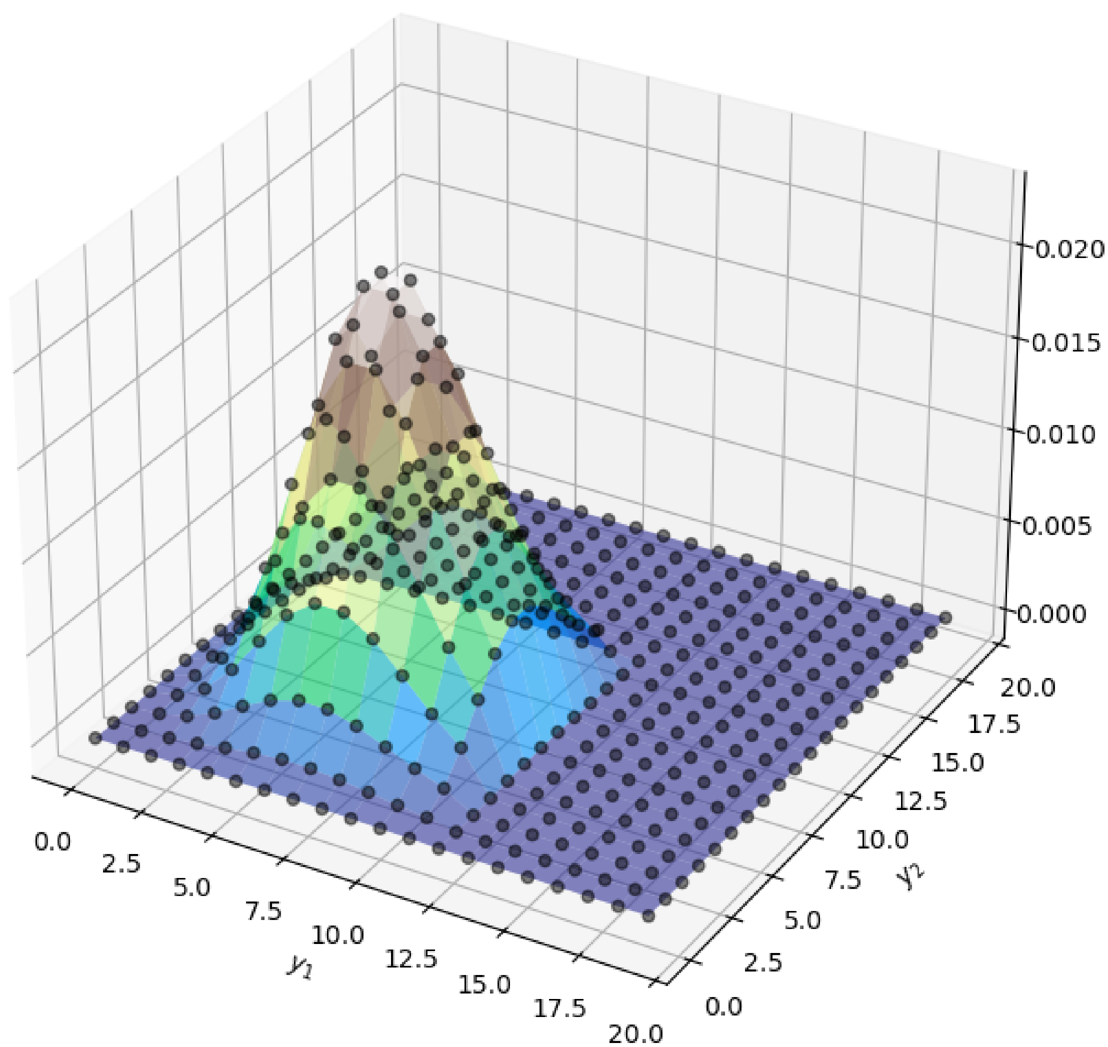
If y_i follows a Poisson distribution with $\lambda = 7$, we can visualize the joint pmf like so

```
def plot_joint_poisson(μ=7, y_n=20):
    yi_values = jnp.arange(0, y_n, 1)
```

(continues on next page)

(continued from previous page)

```
    # Create coordinate points of X and Y
    X, Y = jnp.meshgrid(yi_values, yi_values)

    # Multiply distributions together
    Z = poisson_pmf(X, μ) * poisson_pmf(Y, μ)

    fig = plt.figure(figsize=(12, 8))
    ax = fig.add_subplot(111, projection="3d")
    ax.plot_surface(X, Y, Z.T, cmap="terrain", alpha=0.6)
    ax.scatter(X, Y, Z.T, color="black", alpha=0.5, linewidths=1)
    ax.set(xlabel=r"$y_1$", ylabel=r"$y_2$")
    ax.set_zlabel(r"$f(y_1, y_2)$", labelpad=10)
    plt.show()


plot_joint_poisson(μ=7, y_n=20)
```



Similarly, the joint pmf of our data (which is distributed as a conditional Poisson distribution) can be written as

$$f(y_1, y_2, \ldots, y_n \mid \mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n; \beta) = \prod_{i=1}^{n} \frac{\mu_i^{y_i}}{y_i!} e^{-\mu_i}$$

y_i is conditional on both the values of $\mathbf{x}_i$ and the parameters β.

The likelihood function is the same as the joint pmf, but treats the parameter β as a random variable and takes the observations $(y_i, \mathbf{x}_i)$ as given

$$\begin{aligned} \mathcal{L}(\beta \mid y_1, y_2, \ldots, y_n \ ; \ \mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n) = & \prod_{i=1}^{n} \frac{\mu_i^{y_i}}{y_i!} e^{-\mu_i} \\ = & f(y_1, y_2, \ldots, y_n \mid \ \mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n; \beta) \end{aligned}$$

Now that we have our likelihood function, we want to find the $\hat{\beta}$ that yields the maximum likelihood value

$$\max_{\beta} \mathcal{L}(\beta)$$

In doing so it is generally easier to maximize the log-likelihood (consider differentiating $f(x) = x\exp(x)$ vs. $f(x) = \log(x) + x$).

Given that taking a logarithm is a monotone increasing transformation, a maximizer of the likelihood function will also be a maximizer of the log-likelihood function.

In our case the log-likelihood is

$$\begin{aligned} \log \mathcal{L}(\beta) = & \ \log \Big(f(y_1; \beta) \cdot f(y_2; \beta) \cdot \ldots \cdot f(y_n; \beta) \Big) \\ = & \sum_{i=1}^{n} \log f(y_i; \beta) \\ = & \sum_{i=1}^{n} \log \Big(\frac{\mu_i^{y_i}}{y_i!} e^{-\mu_i} \Big) \\ = & \sum_{i=1}^{n} y_i \log \mu_i - \sum_{i=1}^{n} \mu_i - \sum_{i=1}^{n} \log y_i! \end{aligned}$$

The MLE of the Poisson for $\hat{\beta}$ can be obtained by solving

$$\max_{\beta} \Big(\sum_{i=1}^{n} y_i \log \mu_i - \sum_{i=1}^{n} \mu_i - \sum_{i=1}^{n} \log y_i! \Big)$$

However, no analytical solution exists to the above problem – to find the MLE we need to use numerical methods.

91.5 MLE with numerical methods

Many distributions do not have nice, analytical solutions and therefore require numerical methods to solve for parameter estimates.

One such numerical method is the Newton-Raphson algorithm.

Our goal is to find the maximum likelihood estimate $\hat{\beta}$.

At $\hat{\beta}$, the first derivative of the log-likelihood function will be equal to 0.

Let's illustrate this by supposing

$$\log \mathcal{L}(\beta) = -(\beta - 10)^2 - 10$$

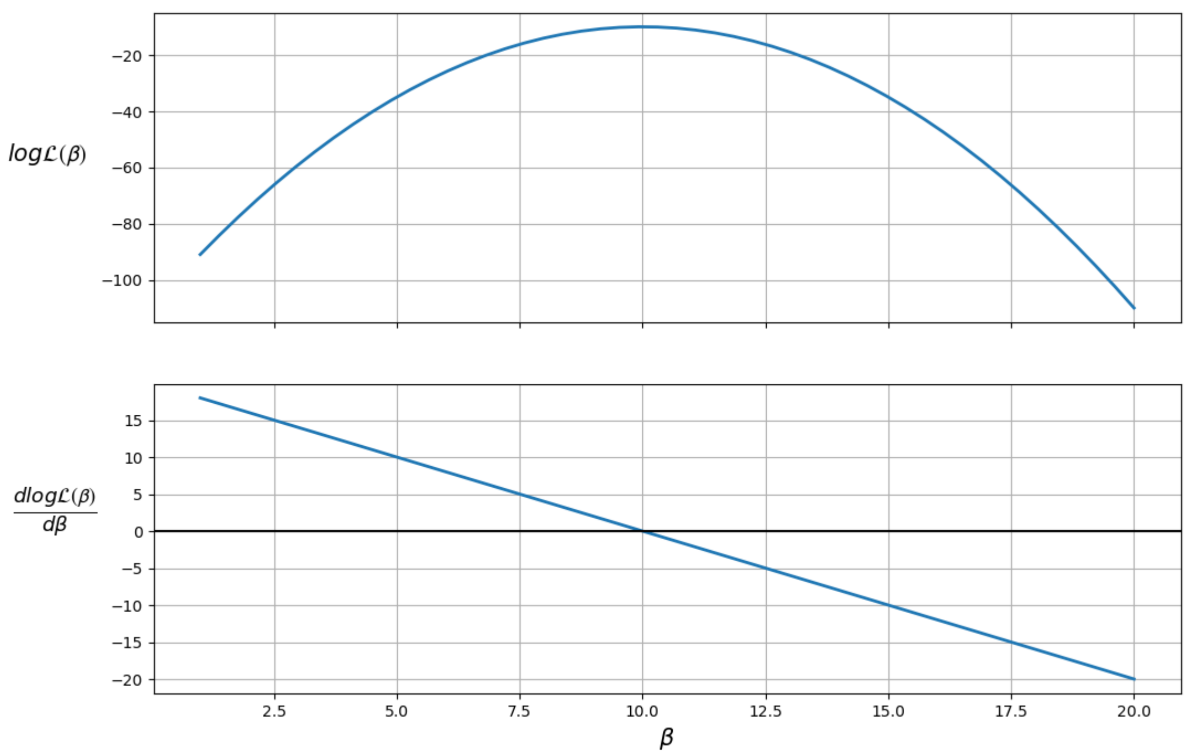
```
@jax.jit
def logL(β):
    return -((β - 10) ** 2) - 10
```

To find the value of the gradient of the above function, we can use jax.grad which auto-differentiates the given function.

We further use jax.vmap which vectorizes the given function i.e. the function acting upon scalar inputs can now be used with vector inputs.

```
dlogL = jax.vmap(jax.grad(logL))
```

```
β = jnp.linspace(1, 20)

fig, (ax1, ax2) = plt.subplots(2, sharex=True, figsize=(12, 8))

ax1.plot(β, logL(β), lw=2)
ax2.plot(β, dlogL(β), lw=2)

ax1.set_ylabel(
    r"$log \mathcal{L(\beta)}$", rotation=0, labelpad=35, fontsize=15
)
ax2.set_ylabel(
    r"$\frac{dlog \mathcal{L(\beta)}}{d \beta}$ ",
    rotation=0,
    labelpad=35,
    fontsize=19,
)
ax2.set_xlabel(r"$\beta$", fontsize=15)
ax1.grid(), ax2.grid()
plt.axhline(c="black")
plt.show()
```



The plot shows that the maximum likelihood value (the top plot) occurs when $\frac{d \log \mathcal{L}(\beta)}{d\beta} = 0$ (the bottom plot).

Therefore, the likelihood is maximized when $\beta = 10$.

We can also ensure that this value is a *maximum* (as opposed to a minimum) by checking that the second derivative (slope of the bottom plot) is negative.

The Newton-Raphson algorithm finds a point where the first derivative is 0.

To use the algorithm, we take an initial guess at the maximum value, β_0 (the OLS parameter estimates might be a reasonable guess), then

1. Use the updating rule to iterate the algorithm

$$\beta_{(k+1)} = \beta_{(k)} - H^{-1}(\beta_{(k)})G(\beta_{(k)})$$

   where:

$$G(\beta_{(k)}) = \frac{d \log \mathcal{L}(\beta_{(k)})}{d\beta_{(k)}}$$
$$H(\beta_{(k)}) = \frac{d^2 \log \mathcal{L}(\beta_{(k)})}{d\beta_{(k)} d\beta'_{(k)}}$$

2. Check whether $\beta_{(k+1)} - \beta_{(k)} < tol$

   - If true, then stop iterating and set $\hat{\beta} = \beta_{(k+1)}$
   - If false, then update $\beta_{(k+1)}$

As can be seen from the updating equation, $\beta_{(k+1)} = \beta_{(k)}$ only when $G(\beta_{(k)}) = 0$ i.e. where the first derivative is equal to 0.

(In practice, we stop iterating when the difference is below a small tolerance threshold.)

Let's have a go at implementing the Newton-Raphson algorithm.

First, we'll create a class called `PoissonRegression` so we can easily recompute the values of the log likelihood, gradient and Hessian for every iteration

```
class PoissonRegression(NamedTuple):
    X: jnp.ndarray
    y: jnp.ndarray
```

Now we can define the log likelihood function in Python

```
@jax.jit
def logL(β, model):
    y = model.y
    μ = jnp.exp(model.X @ β)
    return jnp.sum(model.y * jnp.log(μ) - μ - jnp.log(factorial(y)))
```

To find the gradient of the `poisson_logL`, we again use jax.grad.

According to the documentation,

- `jax.jacfwd` uses forward-mode automatic differentiation, which is more efficient for "tall" Jacobian matrices, while
- `jax.jacrev` uses reverse-mode, which is more efficient for "wide" Jacobian matrices.

(The documentation also states that when matrices that are near-square, `jax.jacfwd` probably has an edge over `jax.jacrev`.)

Therefore, to find the Hessian, we can directly use `jax.jacfwd`.

```
G_logL = jax.grad(logL)
H_logL = jax.jacfwd(G_logL)
```

Our function `newton_raphson` will take a `PoissonRegression` object that has an initial guess of the parameter vector β_0.

The algorithm will update the parameter vector according to the updating rule, and recalculate the gradient and Hessian matrices at the new parameter estimates.

Iteration will end when either:

- The difference between the parameter and the updated parameter is below a tolerance level.
- The maximum number of iterations has been achieved (meaning convergence is not achieved).

So we can get an idea of what's going on while the algorithm is running, an option `display=True` is added to print out values at each iteration.

```
def newton_raphson(model, β, tol=1e-3, max_iter=100, display=True):

    i = 0
    error = 100  # Initial error value

    # Print header of output
    if display:
        header = f'{"Iteration_k":<13}{"Log-likelihood":<16}{"θ":<60}'
        print(header)
        print("-" * len(header))

    # While loop runs while any value in error is greater
    # than the tolerance until max iterations are reached
    while jnp.any(error > tol) and i < max_iter:
        H, G = jnp.squeeze(H_logL(β, model)), G_logL(β, model)
        β_new = β - (jnp.dot(jnp.linalg.inv(H), G))
        error = jnp.abs(β_new - β)
        β = β_new

        if display:
            β_list = [f"{t:.3}" for t in list(β.flatten())]
            update = f"{i:<13}{logL(β, model):<16.8}{β_list}"
            print(update)

        i += 1

    print(f"Number of iterations: {i}")
    print(f"β_hat = {β.flatten()}")

    return β
```

Let's try out our algorithm with a small dataset of 5 observations and 3 variables in **X**.

```
X = jnp.array([[1, 2, 5], [1, 1, 3], [1, 4, 2], [1, 5, 2], [1, 3, 1]])

y = jnp.array([1, 0, 1, 1, 0])
```

(continues on next page)

(continued from previous page)

```
# Take a guess at initial βs
init_β = jnp.array([0.1, 0.1, 0.1])

# Create an object with Poisson model values
poi = PoissonRegression(X=X, y=y)

# Use newton_raphson to find the MLE
β_hat = newton_raphson(poi, init_β, display=True)
```

```
Iteration_k  Log-likelihood  θ
--------------------------------------------------------------------------------
↪------
0            -4.3447633      ['-1.49', '0.265', '0.244']
1            -3.5742409      ['-3.38', '0.528', '0.474']
2            -3.3999527      ['-5.06', '0.782', '0.702']
3            -3.3788645      ['-5.92', '0.909', '0.82']
4            -3.3783555      ['-6.07', '0.933', '0.843']
5            -3.3783557      ['-6.08', '0.933', '0.843']
6            -3.3783557      ['-6.08', '0.933', '0.843']
Number of iterations: 7
β_hat = [-6.078486   0.9334028  0.8432968]
```

As this was a simple model with few observations, the algorithm achieved convergence in only 7 iterations.

You can see that with each iteration, the log-likelihood value increased.

Remember, our objective was to maximize the log-likelihood function, which the algorithm has worked to achieve.

Also, note that the increase in $\log \mathcal{L}(\beta_{(k)})$ becomes smaller with each iteration.

This is because the gradient is approaching 0 as we reach the maximum, and therefore the numerator in our updating equation is becoming smaller.

The gradient vector should be close to 0 at $\hat{\beta}$

```
G_logL(β_hat, poi)
```

```
Array([ 7.4505806e-09, -2.9802322e-07,  3.7252903e-08], dtype=float32)
```

The iterative process can be visualized in the following diagram, where the maximum is found at $\beta = 10$

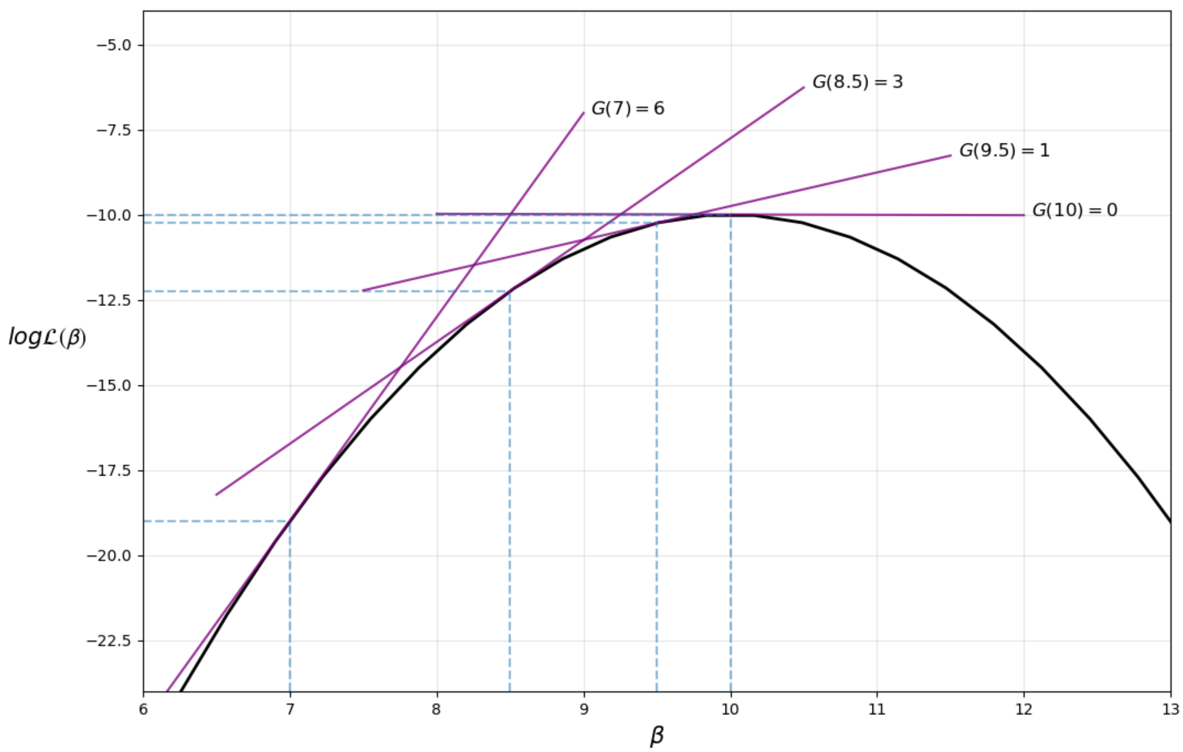
```
@jax.jit
def logL(x):
    return -((x - 10) ** 2) - 10


@jax.jit
def find_tangent(β, a=0.01):
    y1 = logL(β)
    y2 = logL(β + a)
    x = jnp.array([[β, 1], [β + a, 1]])
    m, c = jnp.linalg.lstsq(x, jnp.array([y1, y2]), rcond=None)[0]
    return m, c
```

```
β = jnp.linspace(2, 18)
fig, ax = plt.subplots(figsize=(12, 8))
ax.plot(β, logL(β), lw=2, c="black")
```

(continues on next page)

(continued from previous page)

```
for β in [7, 8.5, 9.5, 10]:
    β_line = jnp.linspace(β - 2, β + 2)
    m, c = find_tangent(β)
    y = m * β_line + c
    ax.plot(β_line, y, "-", c="purple", alpha=0.8)
    ax.text(β + 2.05, y[-1], rf"$G({β}) = {abs(m):.0f}$", fontsize=12)
    ax.vlines(β, -24, logL(β), linestyles="--", alpha=0.5)
    ax.hlines(logL(β), 6, β, linestyles="--", alpha=0.5)

ax.set(ylim=(-24, -4), xlim=(6, 13))
ax.set_xlabel(r"$\beta$", fontsize=15)
ax.set_ylabel(
    r"$log \mathcal{L(\beta)}$", rotation=0, labelpad=25, fontsize=15
)
ax.grid(alpha=0.3)
plt.show()
```



Note that our implementation of the Newton-Raphson algorithm is rather basic — for more robust implementations see, for example, scipy.optimize.

91.6 Maximum likelihood estimation with `statsmodels`

Now that we know what's going on under the hood, we can apply MLE to an interesting application.

We'll use the Poisson regression model in `statsmodels` to obtain a richer output with standard errors, test values, and more.

`statsmodels` uses the same algorithm as above to find the maximum likelihood estimates.

Before we begin, let's re-estimate our simple model with `statsmodels` to confirm we obtain the same coefficients and log-likelihood value.

Now, as `statsmodels` accepts only NumPy arrays, we can use `np.array` method to convert them to NumPy arrays.

```
X = jnp.array([[1, 2, 5], [1, 1, 3], [1, 4, 2], [1, 5, 2], [1, 3, 1]])

y = jnp.array([1, 0, 1, 1, 0])

y_numpy = np.array(y)
X_numpy = np.array(X)
stats_poisson = Poisson(y_numpy, X_numpy).fit()
print(stats_poisson.summary())
```

```
Optimization terminated successfully.
         Current function value: 0.675671
         Iterations 7
                          Poisson Regression Results
==============================================================================
Dep. Variable:                      y   No. Observations:                    5
Model:                        Poisson   Df Residuals:                        2
Method:                           MLE   Df Model:                            2
Date:                Mon, 16 Feb 2026   Pseudo R-squ.:                  0.2546
Time:                        04:51:54   Log-Likelihood:                -3.3784
converged:                       True   LL-Null:                       -4.5325
Covariance Type:            nonrobust   LLR p-value:                    0.3153
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const         -6.0785      5.279     -1.151      0.250     -16.425       4.268
x1             0.9334      0.829      1.126      0.260      -0.691       2.558
x2             0.8433      0.798      1.057      0.291      -0.720       2.407
==============================================================================
```

Now let's replicate results from Daniel Treisman's paper, Russia's Billionaires, mentioned earlier in the lecture.

Treisman starts by estimating equation (91.1), where:

- y_i is $number\ of\ billionaires_i$
- x_{i1} is $\log GDP\ per\ capita_i$
- x_{i2} is $\log population_i$
- x_{i3} is $years\ in\ GATT_i$ – years membership in GATT and WTO (to proxy access to international markets)

The paper only considers the year 2008 for estimation.

We will set up our variables for estimation like so (you should have the data assigned to `df` from earlier in the lecture)

```
# Keep only year 2008
df = df[df["year"] == 2008]

# Add a constant
df["const"] = 1

# Variable sets
reg1 = ["const", "lngdppc", "lnpop", "gattwto08"]
reg2 = [
    "const",
    "lngdppc",
    "lnpop",
    "gattwto08",
    "lnmcap08",
    "rintr",
    "topint08",
]
reg3 = [
    "const",
    "lngdppc",
    "lnpop",
    "gattwto08",
    "lnmcap08",
    "rintr",
    "topint08",
    "nrrents",
    "roflaw",
]
```

Then we can use the `Poisson` function from `statsmodels` to fit the model.

We'll use robust standard errors as in the author's paper

```
# Specify model
poisson_reg = Poisson(df[["numbil0"]], df[reg1], missing="drop").fit(
    cov_type="HC0"
)
print(poisson_reg.summary())
```

```
Optimization terminated successfully.
         Current function value: 2.226090
         Iterations 9
                          Poisson Regression Results
==============================================================================
Dep. Variable:                numbil0   No. Observations:                  197
Model:                        Poisson   Df Residuals:                      193
Method:                           MLE   Df Model:                            3
Date:                Mon, 16 Feb 2026   Pseudo R-squ.:                  0.8574
Time:                        04:51:54   Log-Likelihood:                -438.54
converged:                       True   LL-Null:                       -3074.7
Covariance Type:                  HC0   LLR p-value:                     0.000
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const        -29.0495      2.578    -11.268      0.000     -34.103     -23.997
lngdppc        1.0839      0.138      7.834      0.000       0.813       1.355
lnpop          1.1714      0.097     12.024      0.000       0.980       1.362
```

(continues on next page)

(continued from previous page)

```
gattwto08        0.0060        0.007        0.868        0.386        -0.008        0.019
==============================================================================
```

Success! The algorithm was able to achieve convergence in 9 iterations.

Our output indicates that GDP per capita, population, and years of membership in the General Agreement on Tariffs and Trade (GATT) are positively related to the number of billionaires a country has, as expected.

Let's also estimate the author's more full-featured models and display them in a single table

```
regs = [reg1, reg2, reg3]
reg_names = ["Model 1", "Model 2", "Model 3"]
info_dict = {
    "Pseudo R-squared": lambda x: f"{x.prsquared:.2f}",
    "No. observations": lambda x: f"{int(x.nobs):d}",
}
regressor_order = [
    "const",
    "lngdppc",
    "lnpop",
    "gattwto08",
    "lnmcap08",
    "rintr",
    "topint08",
    "nrrents",
    "roflaw",
]
results = []

for reg in regs:
    result = Poisson(df[["numbil0"]], df[reg], missing="drop").fit(
        cov_type="HC0", maxiter=100, disp=0
    )
    results.append(result)

results_table = summary_col(
    results=results,
    float_format="%0.3f",
    stars=True,
    model_names=reg_names,
    info_dict=info_dict,
    regressor_order=regressor_order,
)
results_table.add_title(
    "Table 1 - Explaining the Number of Billionaires \
                        in 2008"
)
print(results_table)
```

```
Table 1 - Explaining the Number of Billionaires                        in 2008
=================================================
                  Model 1    Model 2    Model 3
-------------------------------------------------
const            -29.050*** -19.444*** -20.858***
                 (2.578)    (4.820)    (4.255)
lngdppc          1.084***   0.717***   0.737***
                 (0.138)    (0.244)    (0.233)
```

(continues on next page)

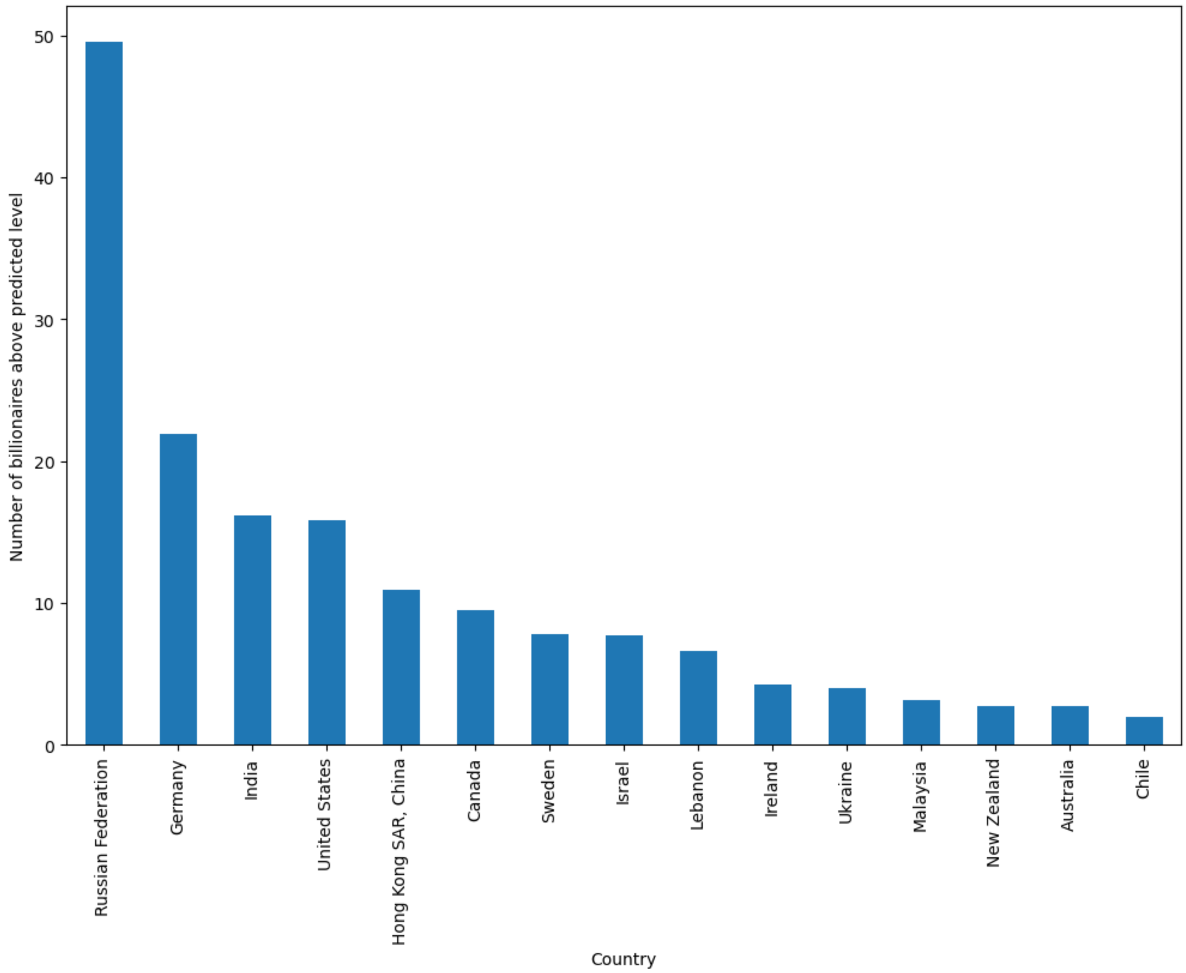
(continued from previous page)

```
lnpop              1.171***    0.806***    0.929***
                   (0.097)     (0.213)     (0.195)
gattwto08          0.006       0.007       0.004
                   (0.007)     (0.006)     (0.006)
lnmcap08                       0.399**     0.286*
                               (0.172)     (0.167)
rintr                          -0.010      -0.009
                               (0.010)     (0.010)
topint08                       -0.051***   -0.058***
                               (0.011)     (0.012)
nrrents                                    -0.005
                                           (0.010)
roflaw                                     0.203
                                           (0.372)
No. observations 197           131         131
Pseudo R-squared 0.86          0.90        0.90
==================================================
Standard errors in parentheses.
* p<.1, ** p<.05, ***p<.01
```

The output suggests that the frequency of billionaires is positively correlated with GDP per capita, population size, stock market capitalization, and negatively correlated with top marginal income tax rate.

To analyze our results by country, we can plot the difference between the predicted and actual values, then sort from highest to lowest and plot the first 15

```
data = [
    "const",
    "lngdppc",
    "lnpop",
    "gattwto08",
    "lnmcap08",
    "rintr",
    "topint08",
    "nrrents",
    "roflaw",
    "numbil0",
    "country",
]
results_df = df[data].dropna()

# Use last model (model 3)
results_df["prediction"] = results[-1].predict()

# Calculate difference
results_df["difference"] = results_df["numbil0"] - results_df["prediction"]

# Sort in descending order
results_df.sort_values("difference", ascending=False, inplace=True)

# Plot the first 15 data points
results_df[:15].plot(
    "country", "difference", kind="bar", figsize=(12, 8), legend=False
)
plt.ylabel("Number of billionaires above predicted level")
plt.xlabel("Country")
plt.show()
```



As we can see, Russia has by far the highest number of billionaires in excess of what is predicted by the model (around 50 more than expected).

Treisman uses this empirical result to discuss possible reasons for Russia's excess of billionaires, including the origination of wealth in Russia, the political climate, and the history of privatization in the years after the USSR.

91.7 Summary

In this lecture, we used Maximum Likelihood Estimation to estimate the parameters of a Poisson model.

`statsmodels` contains other built-in likelihood models such as Probit and Logit.

For further flexibility, `statsmodels` provides a way to specify the distribution manually using the `GenericLikelihoodModel` class - an example notebook can be found here.

91.8 Exercises

Exercise 91.8.1

Suppose we wanted to estimate the probability of an event y_i occurring, given some observations.

We could use a probit regression model, where the pmf of y_i is

$$f(y_i;\beta) = \mu_i^{y_i}(1-\mu_i)^{1-y_i}, \quad y_i = 0,1$$
$$\text{where} \quad \mu_i = \Phi(\mathbf{x}_i'\beta)$$

Φ represents the **cumulative normal distribution** and constrains the predicted y_i to be between 0 and 1 (as required for a probability).

β is a vector of coefficients.

Following the example in the lecture, write a class to represent the Probit model.

To begin, find the log-likelihood function and derive the gradient and Hessian.

The `jax.scipy.stats` module `norm` contains the functions needed to compute the cdf and pdf of the normal distribution.

Solution

The log-likelihood can be written as

$$\log \mathcal{L} = \sum_{i=1}^{n} \left[y_i \log \Phi(\mathbf{x}_i'\beta) + (1-y_i)\log(1-\Phi(\mathbf{x}_i'\beta)) \right]$$

Using the **fundamental theorem of calculus**, the derivative of a cumulative probability distribution is its marginal distribution

$$\frac{\partial}{\partial s}\Phi(s) = \phi(s)$$

where ϕ is the marginal normal distribution.

The gradient vector of the Probit model is

$$\frac{\partial \log \mathcal{L}}{\partial \beta} = \sum_{i=1}^{n} \left[y_i \frac{\phi(\mathbf{x}_i'\beta)}{\Phi(\mathbf{x}_i'\beta)} - (1-y_i)\frac{\phi(\mathbf{x}_i'\beta)}{1-\Phi(\mathbf{x}_i'\beta)} \right] \mathbf{x}_i$$

The Hessian of the Probit model is

$$\frac{\partial^2 \log \mathcal{L}}{\partial \beta \partial \beta'} = -\sum_{i=1}^{n} \phi(\mathbf{x}_i'\beta) \left[y_i \frac{\phi(\mathbf{x}_i'\beta) + \mathbf{x}_i'\beta\Phi(\mathbf{x}_i'\beta)}{[\Phi(\mathbf{x}_i'\beta)]^2} + (1-y_i)\frac{\phi(\mathbf{x}_i'\beta) - \mathbf{x}_i'\beta(1-\Phi(\mathbf{x}_i'\beta))}{[1-\Phi(\mathbf{x}_i'\beta)]^2} \right] \mathbf{x}_i\mathbf{x}_i'$$

Using these results, we can write a class for the Probit model as follows

```
class ProbitRegression(NamedTuple):
    X: jnp.ndarray
    y: jnp.ndarray
```

```
@jax.jit
def logL(β, model):
    y = model.y
    μ = norm.cdf(model.X @ β.T)
    return y @ jnp.log(μ) + (1 - y) @ jnp.log(1 - μ)
```

```
G_logL = jax.grad(logL)
H_logL = jax.jacfwd(G_logL)
```

Exercise 91.8.2

Use the following dataset and initial values of β to estimate the MLE with the Newton-Raphson algorithm developed earlier in the lecture

$$\mathbf{X} = \begin{bmatrix} 1 & 2 & 4 \\ 1 & 1 & 1 \\ 1 & 4 & 3 \\ 1 & 5 & 6 \\ 1 & 3 & 5 \end{bmatrix} \quad y = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \quad \beta_{(0)} = \begin{bmatrix} 0.1 \\ 0.1 \\ 0.1 \end{bmatrix}$$

Verify your results with `statsmodels` - you can import the Probit function with the following import statement

```
from statsmodels.discrete.discrete_model import Probit
```

Note that the simple Newton-Raphson algorithm developed in this lecture is very sensitive to initial values, and therefore you may fail to achieve convergence with different starting values.

Solution

Here is one solution

```
X = jnp.array([[1, 2, 4], [1, 1, 1], [1, 4, 3], [1, 5, 6], [1, 3, 5]])

y = jnp.array([1, 0, 1, 1, 0])

# Take a guess at initial βs
β = jnp.array([0.1, 0.1, 0.1])

# Create a model of Probit regression
prob = ProbitRegression(y=y, X=X)

# Run Newton-Raphson algorithm
newton_raphson(prob, β)
```

```
Iteration_k  Log-likelihood  θ
--------------------------------------------------------------------------------
↪---------
0            -2.3796887      ['-1.34', '0.775', '-0.157']
1            -2.3687525      ['-1.53', '0.775', '-0.0981']
2            -2.3687296      ['-1.55', '0.778', '-0.0971']
3            -2.3687291      ['-1.55', '0.778', '-0.0971']
Number of iterations: 4
β_hat = [-1.5462587   0.77778953 -0.09709755]
```

```
Array([-1.5462587 ,  0.77778953, -0.09709755], dtype=float32)
```

```
# Use statsmodels to verify results
y_numpy = np.array(y)
X_numpy = np.array(X)
print(Probit(y_numpy, X_numpy).fit().summary())
```

```
Optimization terminated successfully.
         Current function value: 0.473746
         Iterations 6
                          Probit Regression Results
==============================================================================
Dep. Variable:                      y   No. Observations:                    5
Model:                         Probit   Df Residuals:                        2
Method:                           MLE   Df Model:                            2
Date:                Mon, 16 Feb 2026   Pseudo R-squ.:                  0.2961
Time:                        04:51:55   Log-Likelihood:                -2.3687
converged:                       True   LL-Null:                       -3.3651
Covariance Type:            nonrobust   LLR p-value:                    0.3692
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const         -1.5463      1.866     -0.829      0.407      -5.204       2.111
x1             0.7778      0.788      0.986      0.324      -0.768       2.323
x2            -0.0971      0.590     -0.165      0.869      -1.254       1.060
==============================================================================
```

Part XV

Auctions

CHAPTER

NINETYTWO

FIRST-PRICE AND SECOND-PRICE AUCTIONS

This lecture is designed to set the stage for a subsequent lecture about Multiple Good Allocation Mechanisms

In that lecture, a planner or auctioneer simultaneously allocates several goods to set of people.

In the present lecture, a single good is allocated to one person within a set of people.

Here we'll learn about and simulate two classic auctions :

- a First-Price Sealed-Bid Auction (FPSB)
- a Second-Price Sealed-Bid Auction (SPSB) created by William Vickrey [Vickrey, 1961]

We'll also learn about and apply a

- Revenue Equivalent Theorem

We recommend watching this video about second price auctions by Anders Munk-Nielsen:

https://youtu.be/qwWk_Bqtue8

and

https://youtu.be/eYTGQCGpmXI

Anders Munk-Nielsen put his code on GitHub.

Much of our Python code below is based on his.

92.1 First-price sealed-bid auction (FPSB)

Protocols:

- A single good is auctioned.
- Prospective buyers simultaneously submit sealed bids.
- Each bidder knows only his/her own bid.
- The good is allocated to the person who submits the highest bid.
- The winning bidder pays price she has bid.

Detailed Setting:

There are $n > 2$ prospective buyers named $i = 1, 2, \ldots, n$.

Buyer i attaches value v_i to the good being sold.

Buyer i wants to maximize the expected value of her **surplus** defined as $v_i - p$, where p is the price that she pays, conditional on her winning the auction.

Evidently,

- If i bids exactly v_i, she pays what she thinks it is worth and gathers no surplus value.
- Buyer i will never want to bid more than v_i.
- If buyer i bids $b < v_i$ and wins the auction, then she gathers surplus value $v_i - b > 0$.
- If buyer i bids $b < v_i$ and someone else bids more than b, buyer i loses the auction and gets no surplus value.
- To proceed, buyer i wants to know the probability that she wins the auction as a function of her bid v_i
  - this requires that she know a probability distribution of bids v_j made by prospective buyers $j \neq i$
- Given her idea about that probability distribution, buyer i wants to set a bid that maximizes the mathematical expectation of her surplus value.

Bids are sealed, so no bidder knows bids submitted by other prospective buyers.

This means that bidders are in effect participating in a game in which players do not know **payoffs** of other players.

This is a **Bayesian game**, a Nash equilibrium of which is called a **Bayesian Nash equilibrium**.

To complete the specification of the situation, we'll assume that prospective buyers' valuations are independently and identically distributed according to a probability distribution that is known by all bidders.

Bidder optimally chooses to bid less than v_i.

92.1.1 Characterization of FPSB auction

A FPSB auction has a unique symmetric Bayesian Nash Equilibrium.

The optimal bid of buyer i is

$$\mathbf{E}[y_i | y_i < v_i] \tag{92.1}$$

where v_i is the valuation of bidder i and y_i is the maximum valuation of all other bidders:

$$y_i = \max_{j \neq i} v_j \tag{92.2}$$

A proof for this assertion is available at the Wikipedia page about Vickrey auctions

92.2 Second-price sealed-bid auction (SPSB)

Protocols: In a second-price sealed-bid (SPSB) auction, the winner pays the second-highest bid.

92.3 Characterization of SPSB auction

In a SPSB auction bidders optimally choose to bid their values.

Formally, a dominant strategy profile in a SPSB auction with a single, indivisible item has each bidder bidding its value.

A proof is provided at the Wikipedia page about Vickrey auctions

92.4 Uniform distribution of private values

We assume valuation v_i of bidder i is distributed $v_i \overset{\text{i.i.d.}}{\sim} U(0,1)$.

Under this assumption, we can analytically compute probability distributions of prices bid in both FPSB and SPSB.

We'll simulate outcomes and, by using a law of large numbers, verify that the simulated outcomes agree with analytical ones.

We can use our simulation to illustrate a **Revenue Equivalence Theorem** that asserts that on average first-price and second-price sealed bid auctions provide a seller the same revenue.

To read about the revenue equivalence theorem, see this Wikepedia page

92.5 Setup

There are n bidders.

Each bidder knows that there are $n-1$ other bidders.

92.6 First price sealed bid auction

An optimal bid for bidder i in a **FPSB** is described by equations (92.1) and (92.2).

When bids are i.i.d. draws from a uniform distribution, the CDF of y_i is

$$
\begin{aligned}
\tilde{F}_{n-1}(y) = \mathbf{P}(y_i \leq y) &= \mathbf{P}(\max_{j \neq i} v_j \leq y) \\
&= \prod_{j \neq i} \mathbf{P}(v_j \leq y) \\
&= y^{n-1}
\end{aligned}
$$

and the PDF of y_i is $\tilde{f}_{n-1}(y) = (n-1)y^{n-2}$.

Then bidder i's optimal bid in a **FPSB** auction is:

$$
\begin{aligned}
\mathbf{E}(y_i | y_i < v_i) &= \frac{\int_0^{v_i} y_i \tilde{f}_{n-1}(y_i) dy_i}{\int_0^{v_i} \tilde{f}_{n-1}(y_i) dy_i} \\
&= \frac{\int_0^{v_i} (n-1) y_i^{n-1} dy_i}{\int_0^{v_i} (n-1) y_i^{n-2} dy_i} \\
&= \frac{n-1}{n} y_i \bigg|_0^{v_i} \\
&= \frac{n-1}{n} v_i
\end{aligned}
$$

92.7 Second price sealed bid auction

In a **SPSB**, it is optimal for bidder i to bid v_i.

92.8 Python code

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import scipy.stats as stats
import scipy.interpolate as interp

# for plots
plt.rcParams.update({"text.usetex": True, 'font.size': 14})
colors = plt.rcParams['axes.prop_cycle'].by_key()['color']

# ensure the notebook generates the same randomness
np.random.seed(1337)
```

We repeat an auction with 5 bidders for 100,000 times.

The valuations of each bidder is distributed $U(0,1)$.

```
N = 5
R = 100_000

v = np.random.uniform(0, 1, (N, R))

# BNE in first-price sealed bid

b_star = lambda vi, N: ((N-1)/N) * vi
b = b_star(v,N)
```

We compute and sort bid price distributions that emerge under both FPSB and SPSB.

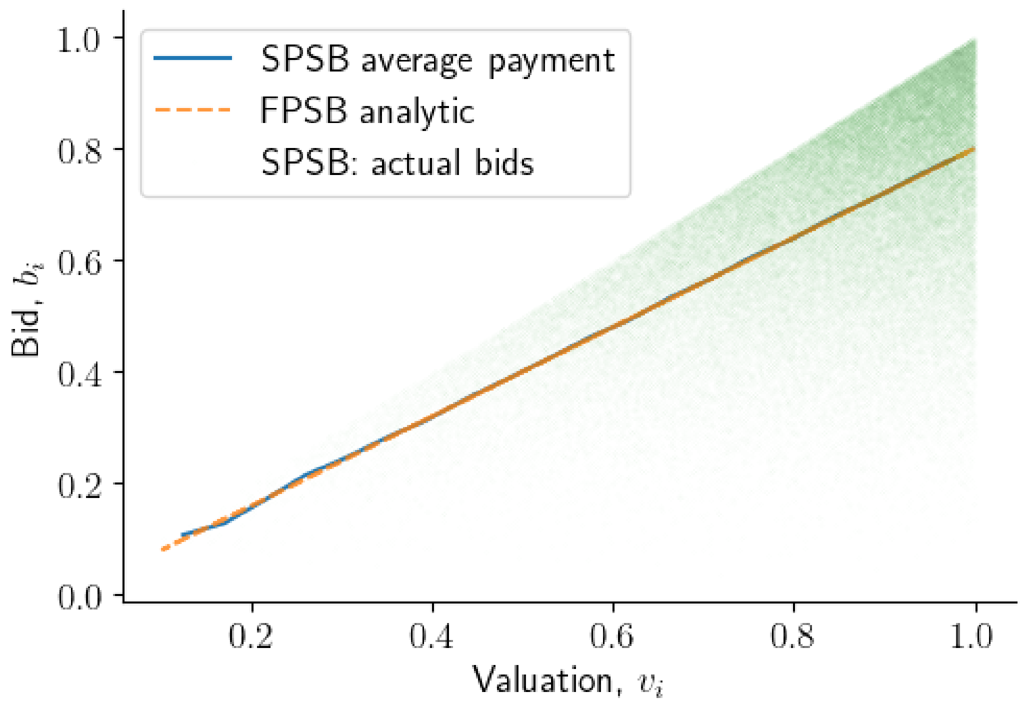
```
# Bidders' values are sorted in ascending order in each auction.
# We record the order because we want to apply it to bid price and their id.
idx = np.argsort(v, axis=0)

# same as np.sort(v, axis=0), except now we retain the idx
v = np.take_along_axis(v, idx, axis=0)
b = np.take_along_axis(b, idx, axis=0)

# the id for the bidders is created.
ii = np.repeat(np.arange(1, N+1)[:, None], R, axis=1)
# the id is sorted according to bid price as well.
ii = np.take_along_axis(ii, idx, axis=0)

# In FPSB and SPSB, winners are those with highest values.
winning_player = ii[-1, :]

# highest bid
winner_pays_fpsb = b[-1, :]
# 2nd-highest valuation
winner_pays_spsb = v[-2, :]
```

Let's now plot the *winning* bids $b_{(n)}$ (i.e. the payment) against valuations, $v_{(n)}$ for both FPSB and SPSB.

Note that

- FPSB: There is a unique bid corresponding to each valuation
- SPSB: Because it equals the valuation of a second-highest bidder, what a winner pays varies even holding fixed the winner's valuation. So here there is a frequency distribution of payments for each valuation.

```
# We intend to compute average payments of different groups of bidders
binned = stats.binned_statistic(v[-1, :], v[-2, :], statistic='mean', bins=20)
xx = binned.bin_edges
xx = [(xx[ii]+xx[ii+1])/2 for ii in range(len(xx)-1)]
yy = binned.statistic

fig, ax = plt.subplots(figsize=(6, 4))

ax.plot(xx, yy, label='SPSB average payment')
ax.plot(v[-1, :], b[-1, :], '--', alpha=0.8, label='FPSB analytic')
ax.plot(v[-1, :], v[-2, :], 'o', alpha=0.05,
                markersize=0.1, label='SPSB: actual bids')

ax.legend(loc='best')
ax.set_xlabel('Valuation, $v_i$')
ax.set_ylabel('Bid, $b_i$')
sns.despine()
```



92.9 Revenue equivalence theorem

We now compare FPSB and a SPSB auctions from the point of view of the revenues that a seller can expect to acquire.

Expected Revenue FPSB:

The winner with valuation y pays $\frac{n-1}{n} * y$, where n is the number of bidders.

Above we computed that the CDF is $F_n(y) = y^n$ and the PDF is $f_n = ny^{n-1}$.

Consequently, expected revenue is

$$\mathbf{R} = \int_0^1 \frac{n-1}{n} v_i \times n v_i^{n-1} dv_i = \frac{n-1}{n+1}$$

Expected Revenue SPSB:

The expected revenue equals n $\times$ expected payment of a bidder.

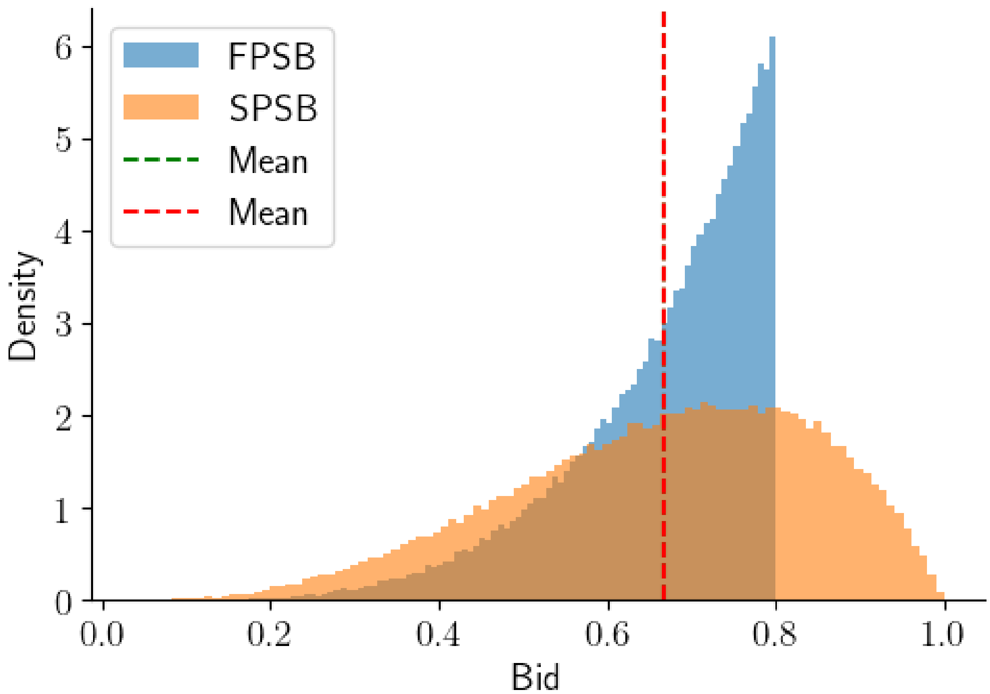
Computing this we get

$$\begin{aligned}
\mathbf{TR} &= n\mathbf{E}_{\mathbf{v_i}}\left[\mathbf{E}_{\mathbf{y_i}}[y_i|y_i < v_i]\mathbf{P}(y_i < v_i) + 0 \times \mathbf{P}(y_i > v_i)\right] \\
&= n\mathbf{E}_{\mathbf{v_i}}\left[\mathbf{E}_{\mathbf{y_i}}[y_i|y_i < v_i]\tilde{F}_{n-1}(v_i)\right] \\
&= n\mathbf{E}_{\mathbf{v_i}}[\frac{n-1}{n} \times v_i \times v_i^{n-1}] \\
&= (n-1)\mathbf{E}_{\mathbf{v_i}}[v_i^n] \\
&= \frac{n-1}{n+1}
\end{aligned}$$

Thus, while probability distributions of winning bids typically differ across the two types of auction, we deduce that expected payments are identical in FPSB and SPSB.

```python
fig, ax = plt.subplots(figsize=(6, 4))

for payment, label in zip([winner_pays_fpsb, winner_pays_spsb], ['FPSB', 'SPSB']):
    print('The average payment of %s: %.4f. Std.: %.4f. Median: %.4f' % (
        label, payment.mean(), payment.std(), np.median(payment)))
    ax.hist(payment, density=True, alpha=0.6, label=label, bins=100)

ax.axvline(winner_pays_fpsb.mean(), ls='--', c='g', label='Mean')
ax.axvline(winner_pays_spsb.mean(), ls='--', c='r', label='Mean')

ax.legend(loc='best')
ax.set_xlabel('Bid')
ax.set_ylabel('Density')
sns.despine()
```

```
The average payment of FPSB: 0.6665. Std.: 0.1129. Median: 0.6967
The average payment of SPSB: 0.6667. Std.: 0.1782. Median: 0.6862
```



Summary of FPSB and SPSB results with uniform distribution on $[0, 1]$

Auction: Sealed-Bid	First-Price	Second-Price
Winner	Agent with highest bid	Agent with highest bid
Winner pays	Winner's bid	Second-highest bid
Loser pays	0	0
Dominant strategy	No dominant strategy	Bidding truthfully is dominant strategy
Bayesian Nash equilibrium	Bidder i bids $\frac{n-1}{n}v_i$	Bidder i truthfully bids v_i
Auctioneer's revenue	$\frac{n-1}{n+1}$	$\frac{n-1}{n+1}$

Detour: Computing a Bayesian Nash Equibrium for FPSB

The Revenue Equivalence Theorem lets us find an optimal bidding strategy for a FPSB auction from outcomes of a SPSB auction.

Let $b(v_i)$ be the optimal bid in a FPSB auction.

The revenue equivalence theorem tells us that a bidder agent with value v_i on average receives the same **payment** in the two types of auction.

Consequently,

$$b(v_i)\mathbf{P}(y_i < v_i) + 0 * \mathbf{P}(y_i \geq v_i) = \mathbf{E}_{y_i}[y_i | y_i < v_i]\mathbf{P}(y_i < v_i) + 0 * \mathbf{P}(y_i \geq v_i)$$

It follows that an optimal bidding strategy in a FPSB auction is $b(v_i) = \mathbf{E}_{y_i}[y_i | y_i < v_i]$.

92.10 Calculation of bid price in FPSB

In equations (92.1) and (92.2), we displayed formulas for optimal bids in a symmetric Bayesian Nash Equilibrium of a FPSB auction.

$$\mathbf{E}[y_i | y_i < v_i]$$

where

- $v_i =$ value of bidder i
- $y_i =$: maximum value of all bidders except i, i.e., $y_i = \max_{j \neq i} v_j$

Above, we computed an optimal bid price in a FPSB auction analytically for a case in which private values are uniformly distributed.

For most probability distributions of private values, analytical solutions aren't easy to compute.

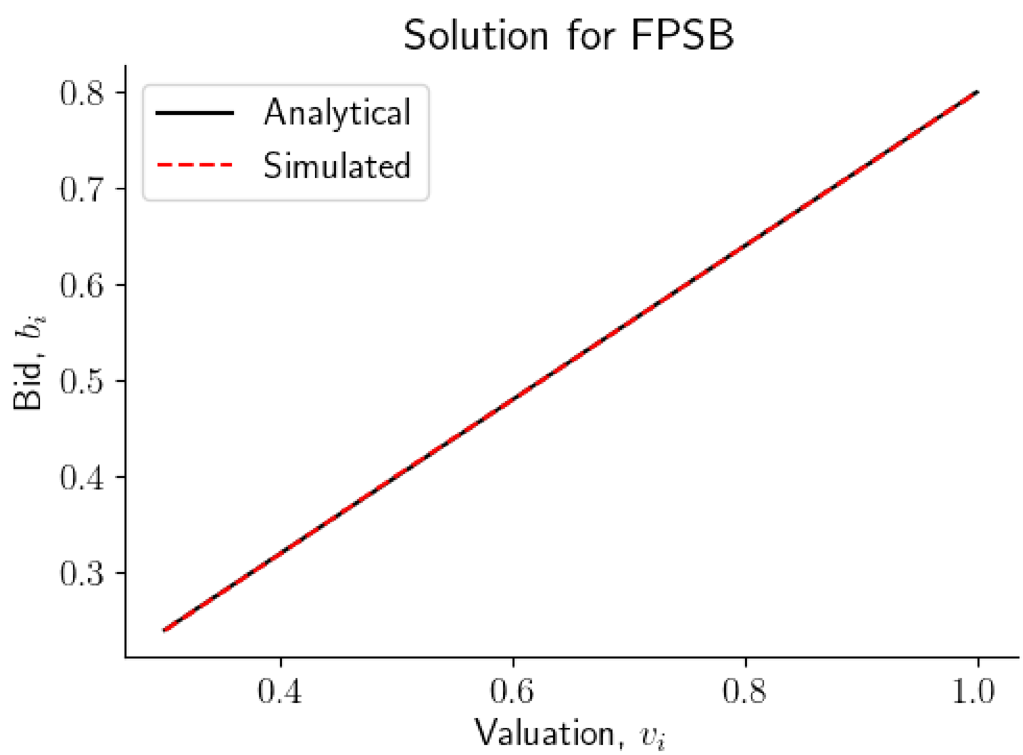
Instead, we can compute bid prices in FPSB auctions numerically as functions of the distribution of private values.

```
def evaluate_largest(v_hat, array, order=1):
    """
    A method to estimate the largest (or certain-order largest) value of the other␣
↪biders,
    conditional on player 1 wins the auction.

    Parameters:
    ----------
    v_hat : float, the value of player 1. The biggest value in the auction that␣
↪player 1 wins.

    array: 2 dimensional array of bidders' values in shape of (N,R),
           where N: number of players, R: number of auctions

    order: int. The order of largest number among bidders who lose.
                e.g. the order for largest number beside winner is 1.
                     the order for second-largest number beside winner is 2.

    """
    N, R = array.shape

    # drop the first row because we assume first row is the winner's bid
    array_residual = array[1:, :].copy()

    winning_auctions_mask = (array_residual < v_hat).all(axis=0)

    num_winning_auctions = np.sum(winning_auctions_mask)

    if num_winning_auctions == 0:
        return np.nan

    array_conditional = array_residual[:, winning_auctions_mask]

    array_conditional_sorted = np.sort(array_conditional, axis=0)

    order_largest_bids = array_conditional_sorted[-order, :]

    return np.mean(order_largest_bids)
```

We can check the accuracy of our `evaluate_largest` method by comparing it with an analytical solution.

We find that the `evaluate_largest` method functions well

```
v_grid = np.linspace(0.3, 1, 8)
bid_analytical = b_star(v_grid, N)

# Redraw valuations
v = np.random.uniform(0, 1, (N, R))
bid_simulated = [evaluate_largest(ii, v) for ii in v_grid]

fig, ax = plt.subplots(figsize=(6, 4))

ax.plot(v_grid, bid_analytical, '-', color='k', label='Analytical')
ax.plot(v_grid, bid_simulated, '--', color='r', label='Simulated')

ax.legend(loc='best')
ax.set_xlabel('Valuation, $v_i$')
ax.set_ylabel('Bid, $b_i$')
ax.set_title('Solution for FPSB')
sns.despine()
```

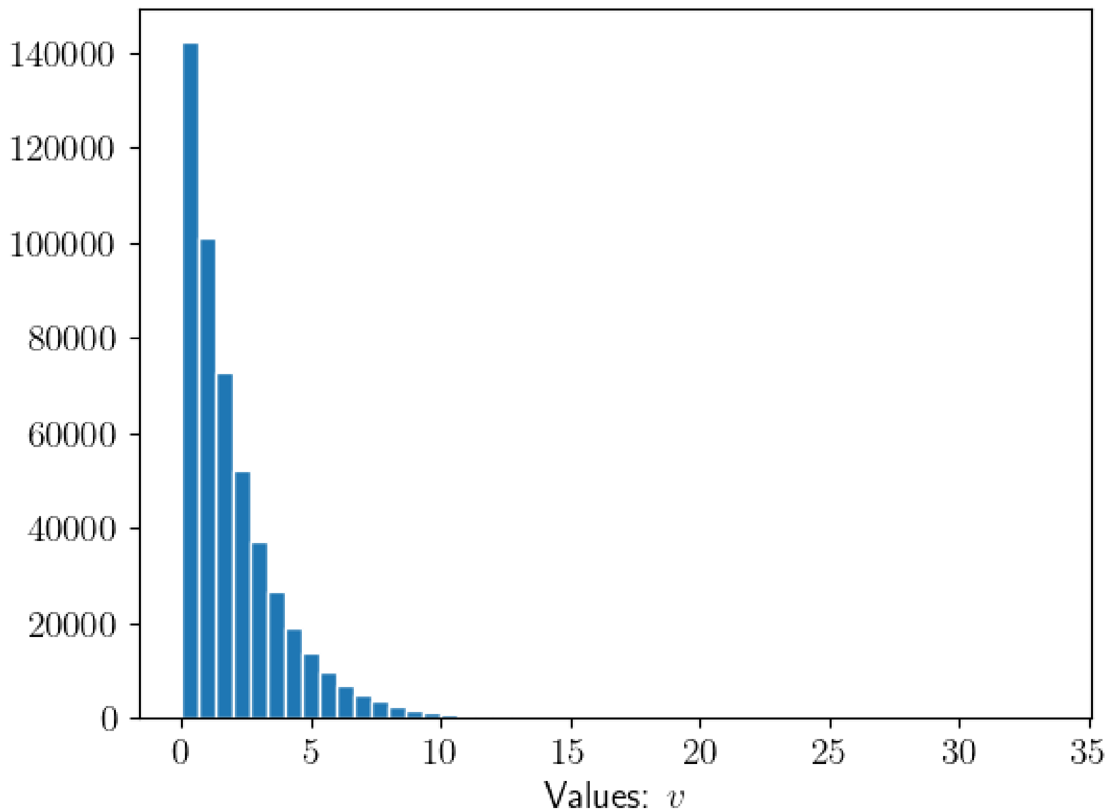


92.11 χ^2 Distribution

Let's try an example in which the distribution of private values is a χ^2 distribution.

We'll start by taking a look at a χ^2 distribution with the help of the following Python code:

```
np.random.seed(1337)
v = np.random.chisquare(df=2, size=(N * R,))

plt.hist(v, bins=50, edgecolor='w')
plt.xlabel('Values: $v$')
plt.show()
```



Now we'll get Python to construct a bid price function

```
np.random.seed(1337)
v = np.random.chisquare(df=2, size=(N, R))

# we compute the quantile of v as our grid
pct_quantile = np.linspace(0, 100, 101)[1:-1]
v_grid = np.percentile(v.flatten(), q=pct_quantile)

# nan values are returned for some low quantiles due to lack of observations
EV = [evaluate_largest(ii, v) for ii in v_grid]
```

```
# we insert 0 into our grid and bid price function as a complement
EV = np.insert(EV, 0, 0)
```

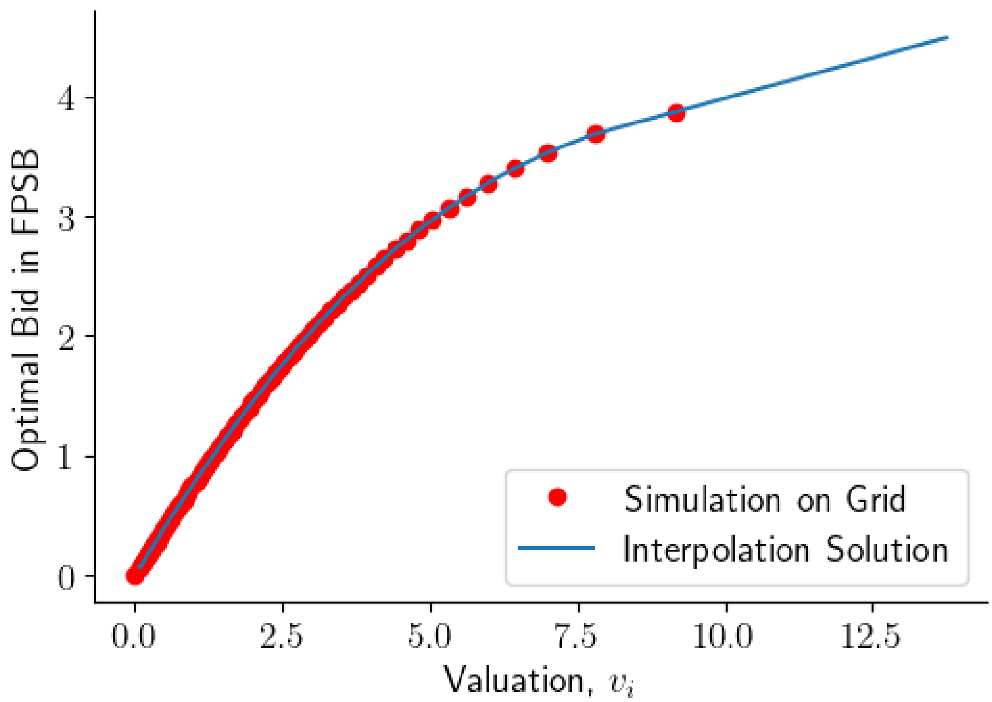
(continues on next page)

(continued from previous page)

```
v_grid = np.insert(v_grid, 0, 0)

b_star_num = interp.interp1d(v_grid, EV, fill_value="extrapolate")
```

We check our bid price function by computing and visualizing the result.

```
pct_quantile_fine = np.linspace(0, 100, 1001)[1:-1]
v_grid_fine = np.percentile(v.flatten(), q=pct_quantile_fine)

fig, ax = plt.subplots(figsize=(6, 4))

ax.plot(v_grid, EV, 'or', label='Simulation on Grid')
ax.plot(v_grid_fine, b_star_num(v_grid_fine),
                '-', label='Interpolation Solution')

ax.legend(loc='best')
ax.set_xlabel('Valuation, $v_i$')
ax.set_ylabel('Optimal Bid in FPSB')
sns.despine()
```



Now we can use Python to compute the probability distribution of the price paid by the winning bidder

```
b = b_star_num(v)

idx = np.argsort(v, axis=0)
# same as np.sort(v, axis=0), except now we retain the idx
v = np.take_along_axis(v, idx, axis=0)
b = np.take_along_axis(b, idx, axis=0)

ii = np.repeat(np.arange(1, N + 1)[:, None], R, axis=1)
ii = np.take_along_axis(ii, idx, axis=0)
```

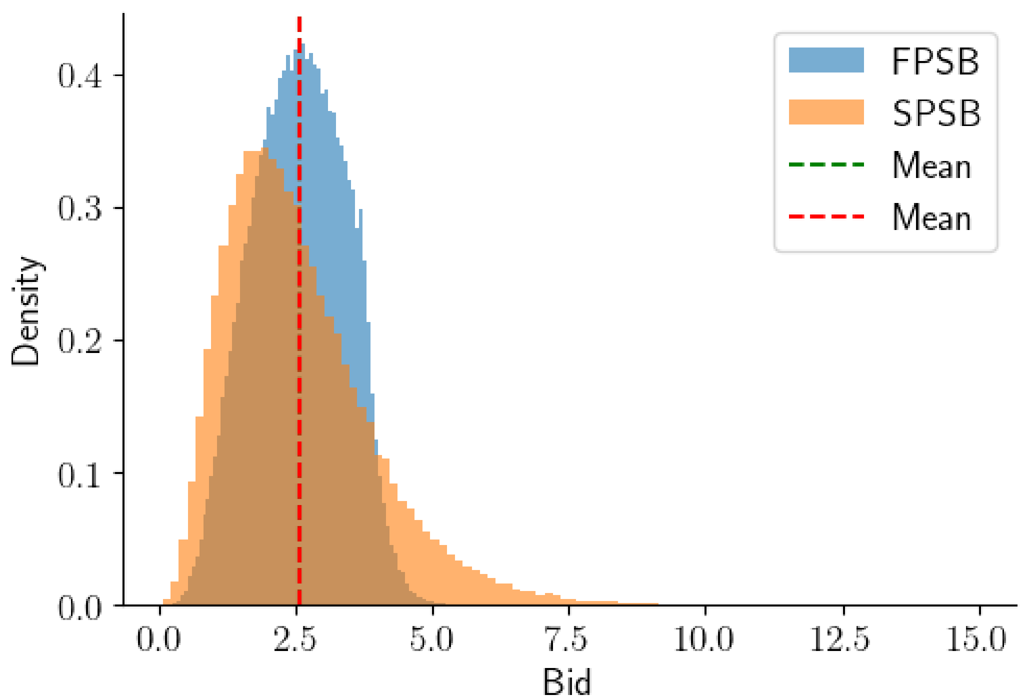
(continues on next page)

(continued from previous page)

```
winning_player = ii[-1, :]

# highest bid
winner_pays_fpsb = b[-1, :]
# 2nd-highest valuation
winner_pays_spsb = v[-2, :]
```

```
fig, ax = plt.subplots(figsize=(6, 4))

for payment, label in zip([winner_pays_fpsb, winner_pays_spsb],
                          ['FPSB', 'SPSB']):
    print('The average payment of %s: %.4f. Std.: %.4f. Median: %.4f' % (
        label, payment.mean(), payment.std(), np.median(payment)))
    ax.hist(payment, density=True, alpha=0.6, label=label, bins=100)

ax.axvline(winner_pays_fpsb.mean(), ls='--', c='g', label='Mean')
ax.axvline(winner_pays_spsb.mean(), ls='--', c='r', label='Mean')

ax.legend(loc='best')
ax.set_xlabel('Bid')
ax.set_ylabel('Density')
sns.despine()
```

```
The average payment of FPSB: 2.5693. Std.: 0.8383. Median: 2.5829
The average payment of SPSB: 2.5661. Std.: 1.3580. Median: 2.3180
```



92.12 Code summary

We assemble the functions that we have used into a Python class

```
class bid_price_solution:

    def __init__(self, array):
        """
        A class that can plot the value distribution of bidders,
        compute the optimal bid price for bidders in FPSB
        and plot the distribution of winner's payment in both FPSB and SPSB

        Parameters:
        ----------

        array: 2 dimensional array of bidders' values in shape of (N, R),
               where N: number of players, R: number of auctions

        """
        self.value_mat = array.copy()

        return None

    def plot_value_distribution(self):
        plt.hist(self.value_mat.flatten(), bins=50, edgecolor='w')
        plt.xlabel('Values: $v$')
        plt.show()

        return None

    def evaluate_largest(self, v_hat, order=1):
        N, R = self.value_mat.shape

        # drop the first row because we assume first row is the winner's bid
        array_residual = self.value_mat[1:, :].copy()

        winning_auctions_mask = (array_residual < v_hat).all(axis=0)

        num_winning_auctions = np.sum(winning_auctions_mask)

        if num_winning_auctions == 0:
            return np.nan

        array_conditional = array_residual[:, winning_auctions_mask]
        array_conditional_sorted = np.sort(array_conditional, axis=0)
        order_largest_bids = array_conditional_sorted[-order, :]

        return np.mean(order_largest_bids)

    def compute_optimal_bid_FPSB(self):
        # we compute the quantile of v as our grid
        pct_quantile = np.linspace(0, 100, 101)[1:-1]
        v_grid = np.percentile(self.value_mat.flatten(), q=pct_quantile)

        # nan values are returned for some low quantiles due to lack of observations
        EV = [self.evaluate_largest(ii) for ii in v_grid]
```

(continues on next page)

(continued from previous page)

```
        # we insert 0 into our grid and bid price function as a complement
        EV = np.insert(EV, 0, 0)
        v_grid = np.insert(v_grid, 0, 0)

        self.b_star_num = interp.interp1d(v_grid, EV,
                                          fill_value="extrapolate")

        pct_quantile_fine = np.linspace(0, 100, 1001)[1:-1]
        v_grid_fine = np.percentile(self.value_mat.flatten(),
                                    q=pct_quantile_fine)

        fig, ax = plt.subplots(figsize=(6, 4))

        ax.plot(v_grid, EV, 'or', label='Simulation on Grid')
        ax.plot(v_grid_fine, self.b_star_num(v_grid_fine),
                    '-', label='Interpolation Solution')

        ax.legend(loc='best')
        ax.set_xlabel('Valuation, $v_i$')
        ax.set_ylabel('Optimal Bid in FPSB')
        sns.despine()

        return None

    def plot_winner_payment_distribution(self):
        self.b = self.b_star_num(self.value_mat)

        idx = np.argsort(self.value_mat, axis=0)
        # same as np.sort(v, axis=0), except now we retain the idx
        self.v = np.take_along_axis(self.value_mat, idx, axis=0)
        self.b = np.take_along_axis(self.b, idx, axis=0)

        N, R = self.value_mat.shape
        self.ii = np.repeat(np.arange(1, N + 1)[:, None], R, axis=1)
        self.ii = np.take_along_axis(self.ii, idx, axis=0)

        winning_player = self.ii[-1, :]

        # highest bid
        winner_pays_fpsb = self.b[-1, :]
        # 2nd-highest valuation
        winner_pays_spsb = self.v[-2, :]

        fig, ax = plt.subplots(figsize=(6, 4))

        for payment, label in zip([winner_pays_fpsb, winner_pays_spsb],
                                  ['FPSB', 'SPSB']):
            print('The average payment of %s: %.4f. Std.: %.4f. Median: %.4f' %
                  (label, payment.mean(), payment.std(), np.median(payment)))
            ax.hist(payment, density=True, alpha=0.6, label=label, bins=100)

        ax.axvline(winner_pays_fpsb.mean(), ls='--', c='g', label='Mean')
        ax.axvline(winner_pays_spsb.mean(), ls='--', c='r', label='Mean')

        ax.legend(loc='best')
        ax.set_xlabel('Bid')
        ax.set_ylabel('Density')
```

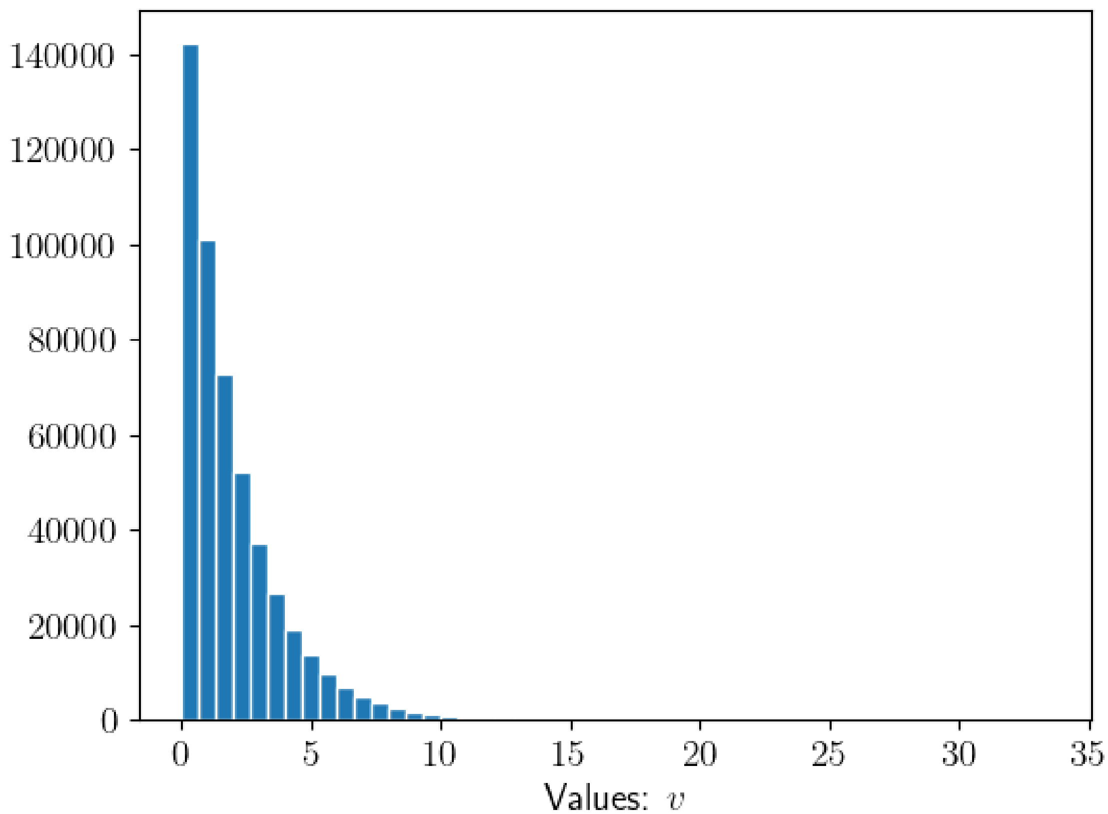
(continues on next page)

(continued from previous page)

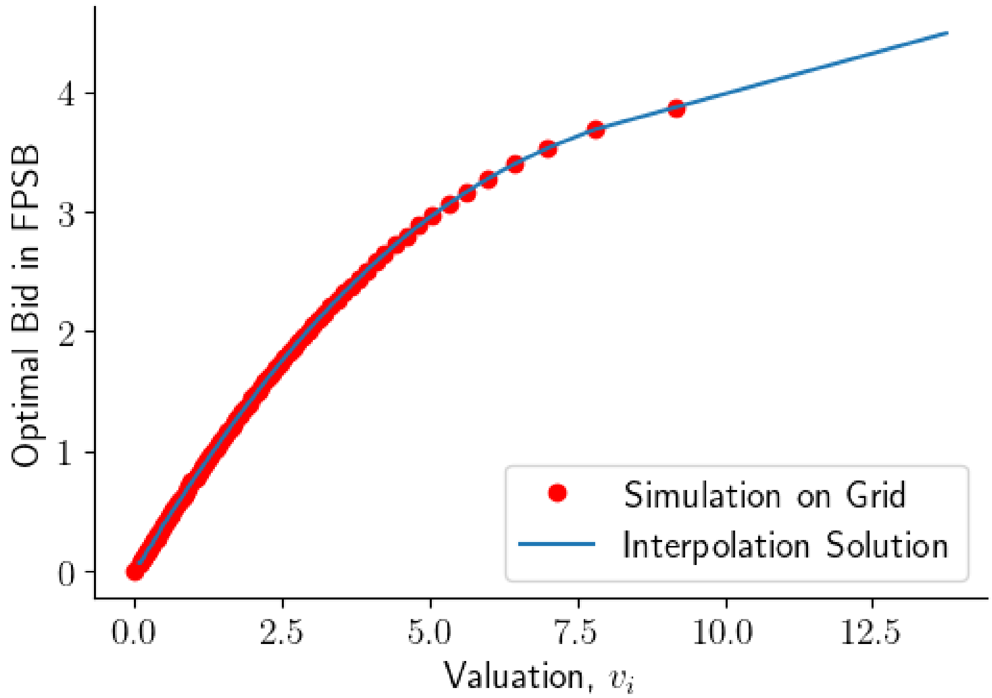
```
        sns.despine()

        return None
```

```
np.random.seed(1337)
v = np.random.chisquare(df=2, size=(N, R))

chi_squ_case = bid_price_solution(v)
```

```
chi_squ_case.plot_value_distribution()
```

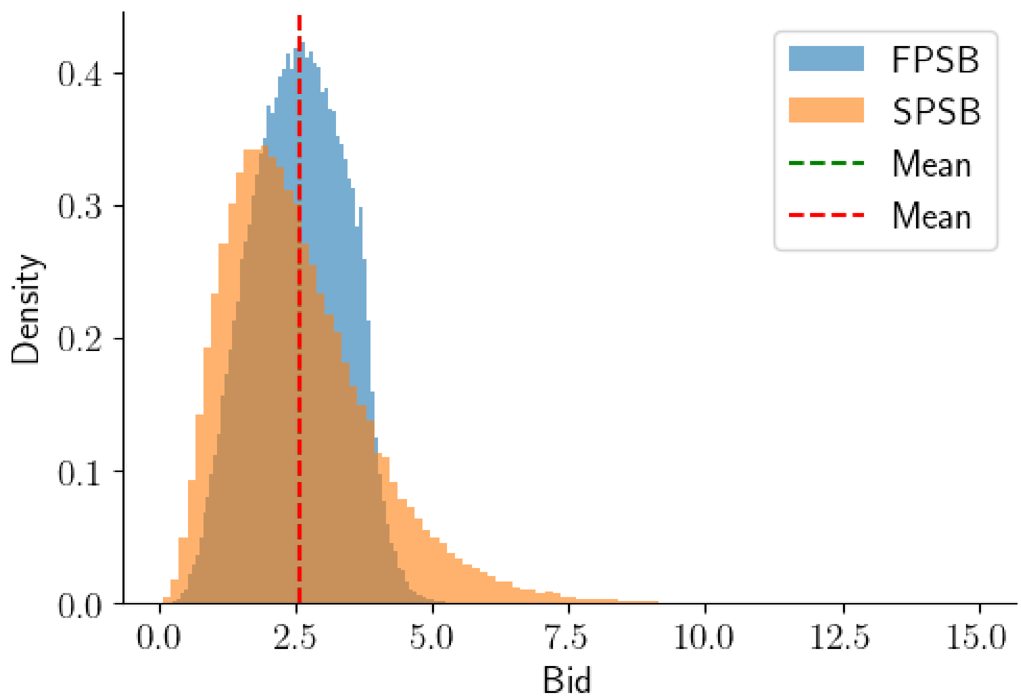



```
chi_squ_case.compute_optimal_bid_FPSB()
```




```
chi_squ_case.plot_winner_payment_distribution()
```

```
The average payment of FPSB: 2.5693. Std.: 0.8383. Median: 2.5829
The average payment of SPSB: 2.5661. Std.: 1.3580. Median: 2.3180
```



92.13 References

1. Wikipedia for FPSB: https://en.wikipedia.org/wiki/First-price_sealed-bid_auction
2. Wikipedia for SPSB: https://en.wikipedia.org/wiki/Vickrey_auction
3. Chandra Chekuri's lecture note for algorithmic game theory: https://chekuri.cs.illinois.edu/teaching/spring2008/Lectures/scribed/Notes20.pdf
4. Tim Salmon. ECO 4400 Supplemental Handout: All About Auctions: https://s2.smu.edu/tsalmon/auctions.pdf
5. Auction Theory- Revenue Equivalence Theorem: https://michaellevet.wordpress.com/2015/07/06/auction-theory-revenue-equivalence-theorem/
6. Order Statistics: https://online.stat.psu.edu/stat415/book/export/html/834

CHAPTER

NINETYTHREE

MULTIPLE GOOD ALLOCATION MECHANISMS

```
!pip install prettytable
```

93.1 Overview

This lecture describes two mechanisms for allocating n private goods ("houses") to m people ("buyers").

We assume that $m > n$ so that there are more potential buyers than there are houses.

Prospective buyers regard the houses as **substitutes**.

Buyer j attaches value v_{ij} to house i.

These values are **private**

- v_{ij} is known only to person j unless person j chooses to tell someone.

We require that a mechanism allocate **at most** one house to one prospective buyer.

We describe two distinct mechanisms

- A multiple rounds, ascending bid auction
- A special case of a Groves-Clarke [Groves, 1973], [Clarke, 1971] mechanism with a benevolent social planner

> **Note**
>
> In 1994, the multiple rounds, ascending bid auction was actually used by Stanford University to sell leases to 9 lots on the Stanford campus to eligible faculty members.

We begin with overviews of the two mechanisms.

93.2 Ascending Bids Auction for Multiple Goods

An auction is administered by an **auctioneer**

The auctioneer has an $n \times 1$ vector r of reservation prices on the n houses.

The auctioneer sells house i only if the final price bid for it exceeds r_i

The auctioneer allocates all n houses **simultaneously**

The auctioneer does not know bidders' private values v_{ij}

There are multiple **rounds**

- during each round, active participants can submit bids on any of the n houses
- each bidder can bid on only one house during one round
- a person who was high bidder on a particular house in one round is understood to submit that same bid for the same house in the next round
- between rounds, a bidder who was not a high bidder can change the house on which he/she chooses to bid
- the auction ends when the price of no house changes from one round to the next
- all n houses are allocated after the final round
- house i is retained by the auctioneer if not prospective buyer offers more that r_i for the house

In this auction, person j never tells anyone else his/her private values v_{ij}

93.3 A Benevolent Planner

This mechanism is designed so that all prospective buyers voluntarily choose to reveal their private values to a **social planner** who uses them to construct a socially optimal allocation.

Among all feasible allocations, a **socially optimal allocation** maximizes the sum of private values across all prospective buyers.

The planner tells everyone in advance how he/she will allocate houses based on the matrix of values that prospective buyers report.

The mechanism provide every prospective buyer an incentive to reveal his vector of private values to the planner.

After the planner receives everyone's vector of private values, the planner deploys a **sequential** algorithm to determine an **allocation** of houses and a set of **fees** that he charges awardees for the negative **externality** that their presence impose on other prospective buyers.

93.4 Equivalence of Allocations

Remarkably, these two mechanisms can produce virtually identical allocations.

We construct Python code for both mechanism.

We also work out some examples by hand or almost by hand.

Next, let's dive down into the details.

93.5 Ascending Bid Auction

93.5.1 Basic Setting

We start with a more detailed description of the setting.

- A seller owns n houses that he wants to sell for the maximum possible amounts to a set of m prospective eligible buyers.
- The seller wants to sell at most one house to each potential buyer.

- There are m potential eligible buyers, identified by $j = [1, 2, \ldots, m]$
  - Each potential buyer is permitted to buy at most one house.
  - Buyer j would be willing to pay at most v_{ij} for house i.
  - Buyer j knows $v_{ij}, i = 1, \ldots, n$, but no one else does.
  - If buyer j pays p_i for house i, he enjoys surplus value $v_{ij} - p_i$.
  - Each buyer j wants to choose the i that maximizes his/her surplus value $v_{ij} - p_i$.
  - The seller wants to maximize $\sum_i p_i$.

The seller conducts a **simultaneous, multiple goods, ascending bid auction**.

Outcomes of the auction are

- An $n \times 1$ vector p of sales prices $p = [p_1, \ldots, p_n]$ for the n houses.
- An $n \times m$ matrix Q of 0's and 1's, where $Q_{ij} = 1$ if and only if person j bought house i.
- An $n \times m$ matrix S of surplus values consisting of all zeros unless person j bought house i, in which case $S_{ij} = v_{ij} - p_i$

We describe rules for the auction it terms of **pseudo code**.

The pseudo code will provide a road map for writing Python code to implement the auction.

93.6 Pseudocode

Here is a quick sketch of a possible simple structure for our Python code

Inputs:

- n, m.
- an $n \times m$ non-negative matrix v of private values
- an $n \times 1$ vector r of seller-specified reservation prices
- the seller will not accept a price less than r_i for house i
- we are free to think of these reservation prices as private values of a fictitious $m + 1$ th buyer who does not actually participate in the auction
- initial bids can be thought of starting at r
- a scalar ϵ of seller-specified minimum price-bid increments

For each round of the auction, new bids on a house must be at least the prevailing highest bid so far **plus** ϵ

Auction Protocols

- the auction consists of a finite number of **rounds**
- in each round, a prospective buyer can bid on one and only one house
- after each round, a house is temporarily awarded to the person who made the highest bid for that house
  - temporarily winning bids on each house are announced
  - this sets the stage to move on to the next round
- a new round is held

  - bids for temporary winners from the previous round are again attached to the houses on which they bid; temporary winners of the last round leave their bids from the previous round unchanged
  - all other active prospective buyers must submit a new bid on some house
  - new bids on a house must be at least equal to the prevailing temporary price that won the last round **plus** ϵ
  - if a person does not submit a new bid and was also not a temporary winner from the previous round, that person must drop out of the auction permanently
  - for each house, the highest bid, whether it is a new bid or was the temporary winner from the previous round, is announced, with the person who made that new (temporarily) winning bid being (temporarily) awarded the house to start the next round
- rounds continue until no price on **any** house changes from the previous round
- houses are sold to the winning bidders from the final round at the prices that they bid

Outputs:

- an $n \times 1$ vector p of sales prices
- an $n \times m$ matrix S of surplus values consisting of all zeros unless person j bought house i, in which case $S_{ij} = v_{ij} - p_i$
- an $n \times (m+1)$ matrix Q of 0's and 1's that tells which buyer bought which house. (The last column accounts for unsold houses.)

Proposed buyer strategy:

In this pseudo code and the actual Python code below, we'll assume that all buyers choose to use the following strategy

- The strategy is optimal for each buyer

Each buyer $j = 1, \dots, m$ uses the same strategy.

The strategy has the form:

- Let $\check{p}^t$ be the $n \times 1$ vector of prevailing highest-bid prices at the beginning of round t
- Let $\epsilon > 0$ be the minimum bid increment specified by the seller
- For each prospective buyer j, compute the index of the best house to bid on during round t, namely $\hat{i}_t = \text{argmax}_i\{[v_{ij} - \check{p}^t_i - \epsilon]\}$
- If $\max_i\{[v_{ij} - \check{p}^t_i - \epsilon]\} \leq 0$, person j permanently drops out of the auction at round t
- If $v_{\hat{i}_t,j} - \check{p}^t_i - \epsilon > 0$, person j bids $\check{p}^t_i + \epsilon$ on house j

Resolving ambiguities: The protocols we have described so far leave open two possible sources of ambiguity.

(1) **The optimal bid choice for buyers in each round.** It is possible that a buyer has the same surplus value for multiple houses. The argmax function in Python always returns the first argmax element. We instead prefer to randomize among such winner. For that reason, we write our own argmax function below.

(2) **Seller's choice of winner if same price bid cast by several buyers.** To resolve this ambiguity, we use the np.random.choice function below.

Given the randomness in outcomes, it is possible that different allocations of houses could emerge from the same inputs.

However, this will happen only when the bid price increment ϵ is nonnegligible.

```
import numpy as np
import prettytable as pt

np.random.seed(100)
```

```
np.set_printoptions(precision=3, suppress=True)
```

93.7 An Example

Before building a Python class, let's step by step solve things almost "by hand" to grasp how the auction proceeds.

A step-by-step procedure also helps reduce bugs, especially when the value matrix is peculiar (e.g. the differences between values are negligible, a column containing identical values or multiple buyers have the same valuation etc.).

Fortunately, our auction behaves well and robustly with various peculiar matrices.

We provide some examples later in this lecture.

```
v = np.array([[8, 5, 9, 4],
              [4, 11, 7, 4],
              [9, 7, 6, 4]])
n, m = v.shape
r = np.array([2, 1, 0])
ϵ = 1
p = r.copy()
buyer_list = np.arange(m)
house_list = np.arange(n)
```

```
v
```

```
array([[ 8,  5,  9,  4],
       [ 4, 11,  7,  4],
       [ 9,  7,  6,  4]])
```

Remember that column indexes j indicate buyers and row indexes i indicate houses.

The above value matrix v is peculiar in the sense that Buyer 3 (indexed from 0) puts the same value 4 on every house being sold.

Maybe buyer 3 is a bureaucrat who purchases these house simply by following instructions from his superior.

```
r
```

```
array([2, 1, 0])
```

```
def find_argmax_with_randomness(v):
    """
    We build our own verion of argmax function such that the argmax index will be␣
↪returned randomly
    when there are multiple maximum values. This function is similiar to np.argmax(v,
↪axis=0)

    Parameters:
    ----------
    v: 2 dimensional np.array

    """

    n, m = v.shape
    index_array = np.arange(n)
```

(continues on next page)

(continued from previous page)

```
    result=[]

    for ii in range(m):
        max_value = v[:,ii].max()
        result.append(np.random.choice(index_array[v[:,ii] == max_value]))

    return np.array(result)
```

```
def present_dict(dt):
    """
    A function that present the information in table.

    Parameters:
    ----------
    dt: dictionary.

    """

    ymtb = pt.PrettyTable()
    ymtb.field_names = ['House Number', *dt.keys()]
    ymtb.add_row(['Buyer', *dt.values()])
    print(ymtb)
```

Check Kick Off Condition

```
def check_kick_off_condition(v, r, ϵ):
    """
    A function that checks whether the auction could be initiated given the␣
↪reservation price and value matrix.
    To avoid the situation that the reservation prices are so high that no one would␣
↪even bid in the first round.

    Parameters:
    ----------
    v : value matrix of the shape (n,m).

    r: the reservation price

    ϵ: the minimun price increment in each round

    """

    # we convert the price vector to a matrix in the same shape as value matrix to␣
↪facilitate subtraction
    p_start = (ϵ+r)[:,None] @ np.ones(m)[None,:]

    surplus_value = v - p_start
    buyer_decision = (surplus_value > 0).any(axis = 0)
    return buyer_decision.any()
```

```
check_kick_off_condition(v, r, ϵ)
```

```
np.True_
```

93.7.1 round 1

submit bid

```python
def submit_initial_bid(p_initial, ϵ, v):
    """
    A function that describes the bid information in the first round.

    Parameters:
    ----------
    p_initial: the price (or the reservation prices) at the beginning of auction.

    v: the value matrix

    ϵ: the minimun price increment in each round

    Returns:
    ----------
    p: price array after this round of bidding

    bid_info: a dictionary that contains bidding information (house number as keys␣
↪and buyer as values).

    """

    p = p_initial.copy()
    p_start_mat = (ϵ + p)[:,None] @ np.ones(m)[None,:]
    surplus_value = v - p_start_mat

    # we only care about active buyers who have positve surplus values
    active_buyer_diagnosis = (surplus_value > 0).any(axis = 0)
    active_buyer_list = buyer_list[active_buyer_diagnosis]
    active_buyer_surplus_value = surplus_value[:,active_buyer_diagnosis]
    active_buyer_choice = find_argmax_with_randomness(active_buyer_surplus_value)
    # choice means the favourite houses given the current price and ϵ

    # we only retain the unique house index because prices increase once at one round
    house_bid =  list(set(active_buyer_choice))
    p[house_bid] += ϵ

    bid_info = {}
    for house_num in house_bid:
        bid_info[house_num] = active_buyer_list[active_buyer_choice == house_num]

    return p, bid_info
```

```python
p, bid_info = submit_initial_bid(p, ϵ, v)
```

```python
p
```

```
array([3, 2, 1])
```

```python
present_dict(bid_info)
```

```
+--------------+-----+-----+-------+
| House Number |  0  |  1  |   2   |
```

(continues on next page)

(continued from previous page)

```
+---------------+-----+-----+-------+
|    Buyer      | [2] | [1] | [0 3] |
+---------------+-----+-----+-------+
```

check terminal condition

Notice that two buyers bid for house 2 (indexed from 0).

Because the auction protocol does not specify a selection rule in this case, we simply select a winner **randomly**.

This is reasonable because the seller can't distinguish these buyers and doesn't know the valuation of each buyer.

It is both convenient and practical for him to just pick a winner randomly.

There is a 50% probability that Buyer 3 is chosen as the winner for house 2, although he values it less than buyer 0.

In this case, buyer 0 has to bid one more time with a higher price, which crowds out Buyer 3.

Therefore, final price could be 3 or 4, depending on the winner in the last round.

```python
def check_terminal_condition(bid_info, p, v):
    """
    A function that checks whether the auction ends.

    Recall that the auction ends when either losers have non-positive surplus values␣
↪for each house
    or there is no loser (every buyer gets a house).

    Parameters:
    ----------
    bid_info: a dictionary that contains bidding information of house numbers (as␣
↪keys) and buyers (as values).

    p: np.array. price array of houses

    v: value matrix

    Returns:
    ----------
    allocation: a dictionary that descirbe how the houses bid are assigned.

    winner_list: a list of winners

    loser_list: a list of losers

    """

    # there may be several buyers bidding one house, we choose a winner randomly
    winner_list=[np.random.choice(bid_info[ii]) for ii in bid_info.keys()]

    allocation = {house_num:winner for house_num,winner in zip(bid_info.keys(),winner_
↪list)}

    loser_set = set(buyer_list).difference(set(winner_list))
    loser_list = list(loser_set)
    loser_num = len(loser_list)

    if loser_num == 0:
        print('The auction ends because every buyer gets one house.')
```

(continues on next page)

(continued from previous page)

```
        return allocation,winner_list,loser_list

    p_mat = (ϵ + p)[:,None] @ np.ones(loser_num)[None,:]
    loser_surplus_value = v[:,loser_list] - p_mat
    loser_decision = (loser_surplus_value > 0).any(axis = 0)

    print(~(loser_decision.any()))

    return allocation,winner_list,loser_list
```

```
allocation,winner_list,loser_list = check_terminal_condition(bid_info, p, v)
```

```
False
```

```
present_dict(allocation)
```

```
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Buyer     | 2 | 1 | 0 |
+--------------+---+---+---+
```

```
winner_list
```

```
[np.int64(2), np.int64(1), np.int64(0)]
```

```
loser_list
```

```
[np.int64(3)]
```

93.7.2 round 2

From the second round on, the auction proceeds differently from the first round.

Now only active losers (those who have positive surplus values) have an incentive to submit bids to displace temporary winners from the previous round.

```
def submit_bid(loser_list, p, ϵ, v, bid_info):
    """
    A function that executes the bid operation after the first round.
    After the first round, only active losers would cast a new bid with price as old␣
↪price + increment.
    By such bid, winners of last round are replaced by the active losers.

    Parameters:
    ----------
    loser_list: a list that includes the indexes of losers

    p: np.array. price array of houses

    ϵ: minimum increment of bid price
```

(continues on next page)

(continued from previous page)

```
    v: value matrix

    bid_info: a dictionary that contains bidding information of house numbers (as␣
↪keys) and buyers (as values).

    Returns:
    ----------
    p_end: a price array after this round of bidding

    bid_info: a dictionary that contains updated bidding information.

    """

    p_end=p.copy()

    loser_num = len(loser_list)
    p_mat = (ϵ + p_end)[:,None] @ np.ones(loser_num)[None,:]
    loser_surplus_value = v[:,loser_list] - p_mat
    loser_decision = (loser_surplus_value > 0).any(axis = 0)

    active_loser_list = np.array(loser_list)[loser_decision]
    active_loser_surplus_value = loser_surplus_value[:,loser_decision]
    active_loser_choice = find_argmax_with_randomness(active_loser_surplus_value)

    # we retain the unique house index and increasing the corresponding bid price
    house_bid = list(set(active_loser_choice))
    p_end[house_bid] += ϵ

    # we record the bidding information from active losers
    bid_info_active_loser = {}
    for house_num in house_bid:
        bid_info_active_loser[house_num] = active_loser_list[active_loser_choice ==␣
↪house_num]

    # we update the bidding information according to the bidding from actice losers
    for house_num in bid_info_active_loser.keys():
        bid_info[house_num] = bid_info_active_loser[house_num]

    return p_end,bid_info
```

```
p,bid_info = submit_bid(loser_list, p, ϵ, v, bid_info)
```

```
p
```

```
array([3, 2, 2])
```

```
present_dict(bid_info)
```

```
+--------------+-----+-----+-----+
| House Number |  0  |  1  |  2  |
+--------------+-----+-----+-----+
|    Buyer     | [2] | [1] | [3] |
+--------------+-----+-----+-----+
```

```
allocation,winner_list,loser_list = check_terminal_condition(bid_info, p, v)
```

```
False
```

```
present_dict(allocation)
```

```
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Buyer     | 2 | 1 | 3 |
+--------------+---+---+---+
```

93.7.3 round 3

```
p,bid_info = submit_bid(loser_list, p, ϵ, v, bid_info)
```

```
p
```

```
array([3, 2, 3])
```

```
present_dict(bid_info)
```

```
+--------------+-----+-----+-----+
| House Number |  0  |  1  |  2  |
+--------------+-----+-----+-----+
|    Buyer     | [2] | [1] | [0] |
+--------------+-----+-----+-----+
```

```
allocation,winner_list,loser_list = check_terminal_condition(bid_info, p, v)
```

```
False
```

```
present_dict(allocation)
```

```
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Buyer     | 2 | 1 | 0 |
+--------------+---+---+---+
```

93.7.4 round 4

```
p,bid_info = submit_bid(loser_list, p, ϵ, v, bid_info)
```

```
p
```

```
array([3, 3, 3])
```

```
present_dict(bid_info)
```

```
+--------------+-----+-----+-----+
| House Number |  0  |  1  |  2  |
+--------------+-----+-----+-----+
|    Buyer     | [2] | [3] | [0] |
+--------------+-----+-----+-----+
```

Notice that Buyer 3 now switches to bid for house 1 having recongized that house 2 is no longer his best option.

```
allocation,winner_list,loser_list = check_terminal_condition(bid_info, p, v)
```

```
False
```

```
present_dict(allocation)
```

```
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Buyer     | 2 | 3 | 0 |
+--------------+---+---+---+
```

93.7.5 round 5

```
p,bid_info = submit_bid(loser_list, p, ϵ, v, bid_info)
```

```
p
```

```
array([3, 4, 3])
```

```
present_dict(bid_info)
```

```
+--------------+-----+-----+-----+
| House Number |  0  |  1  |  2  |
+--------------+-----+-----+-----+
|    Buyer     | [2] | [1] | [0] |
+--------------+-----+-----+-----+
```

Now Buyer 1 bids for house 1 again with price at 4, which crowds out Buyer 3, marking the end of the auction.

```
allocation,winner_list,loser_list = check_terminal_condition(bid_info, p, v)
```

```
True
```

```
present_dict(allocation)
```

```
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Buyer     | 2 | 1 | 0 |
+--------------+---+---+---+
```

```
# as for the houses unsold

house_unsold_list = list(set(house_list).difference(set(allocation.keys())))
house_unsold_list
```

```
[]
```

```
total_revenue = p[list(allocation.keys())].sum()
total_revenue
```

```
np.int64(10)
```

93.8 A Python Class

Above we simulated an ascending bid auction step by step.

When defining functions, we repeatedly computed some intermediate objects because our Python function loses track of variables once the function is executed.

That of course led to redundancy in our code

It is much more efficient to collect all of the aforementioned code into a class that records information about all rounds.

```
class ascending_bid_auction:

    def __init__(self, v, r, ϵ):
        """
        A class that simulates an ascending bid auction for houses.

        Given buyers' value matrix, sellers' reservation prices and minimum increment␣
↪of bid prices,
        this class can execute an ascending bid auction and present information round␣
↪by round until the end.

        Parameters:
        ----------
        v: 2 dimensional value matrix

        r: np.array of reservation prices

        ϵ: minimum increment of bid price

        """

        self.v = v.copy()
        self.n,self.m = self.v.shape
        self.r = r
        self.ϵ = ϵ
        self.p = r.copy()
        self.buyer_list = np.arange(self.m)
        self.house_list = np.arange(self.n)
        self.bid_info_history = []
        self.allocation_history = []
        self.winner_history = []
```

(continues on next page)

(continued from previous page)

```
        self.loser_history = []


    def find_argmax_with_randomness(self, v):
        n,m = v.shape
        index_array = np.arange(n)
        result=[]

        for ii in range(m):
            max_value = v[:,ii].max()
            result.append(np.random.choice(index_array[v[:,ii] == max_value]))

        return np.array(result)


    def check_kick_off_condition(self):
        # we convert the price vector to a matrix in the same shape as value matrix␣
↪to facilitate subtraction
        p_start = (self.ϵ + self.r)[:,None] @ np.ones(self.m)[None,:]
        self.surplus_value = self.v - p_start
        buyer_decision = (self.surplus_value > 0).any(axis = 0)
        return buyer_decision.any()


    def submit_initial_bid(self):
        # we intend to find the optimal choice of each buyer
        p_start_mat = (self.ϵ + self.p)[:,None] @ np.ones(self.m)[None,:]
        self.surplus_value = self.v - p_start_mat

        # we only care about active buyers who have positve surplus values
        active_buyer_diagnosis = (self.surplus_value > 0).any(axis = 0)
        active_buyer_list = self.buyer_list[active_buyer_diagnosis]
        active_buyer_surplus_value = self.surplus_value[:,active_buyer_diagnosis]
        active_buyer_choice = self.find_argmax_with_randomness(active_buyer_surplus_
↪value)

        # we only retain the unique house index because prices increase once at one␣
↪round
        house_bid =  list(set(active_buyer_choice))
        self.p[house_bid] += self.ϵ

        bid_info = {}
        for house_num in house_bid:
            bid_info[house_num] = active_buyer_list[active_buyer_choice == house_num]
        self.bid_info_history.append(bid_info)

        print('The bid information is')
        ymtb = pt.PrettyTable()
        ymtb.field_names = ['House Number', *bid_info.keys()]
        ymtb.add_row(['Buyer', *bid_info.values()])
        print(ymtb)

        print('The bid prices for houses are')
        ymtb = pt.PrettyTable()
        ymtb.field_names = ['House Number', *self.house_list]
        ymtb.add_row(['Price', *self.p])
        print(ymtb)
```

(continues on next page)

(continued from previous page)

```
        self.winner_list=[np.random.choice(bid_info[ii]) for ii in bid_info.keys()]
        self.winner_history.append(self.winner_list)

        self.allocation = {house_num:[winner] for house_num,winner in zip(bid_info.
↪keys(),self.winner_list)}
        self.allocation_history.append(self.allocation)

        loser_set = set(self.buyer_list).difference(set(self.winner_list))
        self.loser_list = list(loser_set)
        self.loser_history.append(self.loser_list)

        print('The winners are')
        print(self.winner_list)

        print('The losers are')
        print(self.loser_list)
        print('\n')


    def check_terminal_condition(self):
        loser_num = len(self.loser_list)

        if loser_num == 0:
            print('The auction ends because every buyer gets one house.')
            print('\n')
            return True

        p_mat = (self.ϵ + self.p)[:,None] @ np.ones(loser_num)[None,:]
        self.loser_surplus_value = self.v[:,self.loser_list] - p_mat
        self.loser_decision = (self.loser_surplus_value > 0).any(axis = 0)

        return ~(self.loser_decision.any())


    def submit_bid(self):
        bid_info = self.allocation_history[-1].copy()  # we only record the bid info␣
↪of winner

        loser_num = len(self.loser_list)
        p_mat = (self.ϵ + self.p)[:,None] @ np.ones(loser_num)[None,:]
        self.loser_surplus_value = self.v[:,self.loser_list] - p_mat
        self.loser_decision = (self.loser_surplus_value > 0).any(axis = 0)

        active_loser_list = np.array(self.loser_list)[self.loser_decision]
        active_loser_surplus_value = self.loser_surplus_value[:,self.loser_decision]
        active_loser_choice = self.find_argmax_with_randomness(active_loser_surplus_
↪value)

        # we retain the unique house index and increasing the corresponding bid price
        house_bid = list(set(active_loser_choice))
        self.p[house_bid] += self.ϵ

        # we record the bidding information from active losers
        bid_info_active_loser = {}
        for house_num in house_bid:
            bid_info_active_loser[house_num] = active_loser_list[active_loser_choice␣
```

(continues on next page)

(continued from previous page)

```
↪== house_num]

        # we update the bidding information according to the bidding from actice␣
↪losers
        for house_num in bid_info_active_loser.keys():
            bid_info[house_num] = bid_info_active_loser[house_num]
        self.bid_info_history.append(bid_info)

        print('The bid information is')
        ymtb = pt.PrettyTable()
        ymtb.field_names = ['House Number', *bid_info.keys()]
        ymtb.add_row(['Buyer', *bid_info.values()])
        print(ymtb)

        print('The bid prices for houses are')
        ymtb = pt.PrettyTable()
        ymtb.field_names = ['House Number', *self.house_list]
        ymtb.add_row(['Price', *self.p])
        print(ymtb)

        self.winner_list=[np.random.choice(bid_info[ii]) for ii in bid_info.keys()]
        self.winner_history.append(self.winner_list)

        self.allocation = {house_num:[winner] for house_num,winner in zip(bid_info.
↪keys(),self.winner_list)}
        self.allocation_history.append(self.allocation)

        loser_set = set(self.buyer_list).difference(set(self.winner_list))
        self.loser_list = list(loser_set)
        self.loser_history.append(self.loser_list)

        print('The winners are')
        print(self.winner_list)

        print('The losers are')
        print(self.loser_list)
        print('\n')


    def start_auction(self):
        print('The Ascending Bid Auction for Houses')
        print('\n')

        print('Basic Information: %d houses, %d buyers'%(self.n, self.m))

        print('The valuation matrix is as follows')
        ymtb = pt.PrettyTable()
        ymtb.field_names = ['Buyer Number', *(np.arange(self.m))]
        for ii in range(self.n):
            ymtb.add_row(['House %d'%(ii), *self.v[ii,:]])
        print(ymtb)

        print('The reservation prices for houses are')
        ymtb = pt.PrettyTable()
        ymtb.field_names = ['House Number', *self.house_list]
        ymtb.add_row(['Price', *self.r])
        print(ymtb)
```

(continues on next page)

(continued from previous page)

```
        print('The minimum increment of bid price is %.2f' % self.ϵ)
        print('\n')

        ctr = 1
        if self.check_kick_off_condition():
            print('Auction starts successfully')
            print('\n')
            print('Round %d'% ctr)

            self.submit_initial_bid()

            while True:
                if self.check_terminal_condition():
                    print('Auction ends')
                    print('\n')

                    print('The final result is as follows')
                    print('\n')
                    print('The allocation plan is')
                    ymtb = pt.PrettyTable()
                    ymtb.field_names = ['House Number', *self.allocation.keys()]
                    ymtb.add_row(['Buyer', *self.allocation.values()])
                    print(ymtb)

                    print('The bid prices for houses are')
                    ymtb = pt.PrettyTable()
                    ymtb.field_names = ['House Number', *self.house_list]
                    ymtb.add_row(['Price', *self.p])
                    print(ymtb)

                    print('The winners are')
                    print(self.winner_list)

                    print('The losers are')
                    print(self.loser_list)

                    self.house_unsold_list = list(set(self.house_list).
↪difference(set(self.allocation.keys())))
                    print('The houses unsold are')
                    print(self.house_unsold_list)

                    self.total_revenue = self.p[list(self.allocation.keys())].sum()
                    print('The total revenue is %.2f' % self.total_revenue)

                    break

                ctr += 1
                print('Round %d'% ctr)
                self.submit_bid()

            # we compute the surplus matrix S and the quantity matrix X as required␣
↪in 1.1
            self.S = np.zeros((self.n, self.m))
            for ii,jj in zip(self.allocation.keys(),self.allocation.values()):
                self.S[ii,jj] = self.v[ii,jj] - self.p[ii]

            self.Q = np.zeros((self.n, self.m + 1))  # the last column records the␣
```

(continues on next page)

(continued from previous page)

```
↪houses unsold
            for ii,jj in zip(self.allocation.keys(),self.allocation.values()):
                self.Q[ii,jj] = 1
            for ii in self.house_unsold_list:
                self.Q[ii,-1] = 1

            # we sort the allocation result by the house number
            house_sold_list = list(self.allocation.keys())
            house_sold_list.sort()

            dict_temp = {}
            for ii in house_sold_list:
                dict_temp[ii] = self.allocation[ii]
            self.allocation = dict_temp

        else:
            print('The auction can not start because of high reservation prices')
```

Let's use our class to conduct the auction described in one of the above examples.

```
v = np.array([[8,5,9,4],[4,11,7,4],[9,7,6,4]])
r = np.array([2,1,0])
ϵ = 1

auction_1 = ascending_bid_auction(v, r, ϵ)

auction_1.start_auction()
```

```
The Ascending Bid Auction for Houses


Basic Information: 3 houses, 4 buyers
The valuation matrix is as follows
+--------------+---+----+---+---+
| Buyer Number | 0 | 1  | 2 | 3 |
+--------------+---+----+---+---+
|   House 0    | 8 | 5  | 9 | 4 |
|   House 1    | 4 | 11 | 7 | 4 |
|   House 2    | 9 | 7  | 6 | 4 |
+--------------+---+----+---+---+
The reservation prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 2 | 1 | 0 |
+--------------+---+---+---+
The minimum increment of bid price is 1.00


Auction starts successfully


Round 1
The bid information is
+--------------+-----+-----+-------+
| House Number |  0  |  1  |   2   |
```

(continues on next page)

(continued from previous page)

```
+--------------+-----+-----+-------+
|    Buyer     | [2] | [1] | [0 3] |
+--------------+-----+-----+-------+
The bid prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 3 | 2 | 1 |
+--------------+---+---+---+
The winners are
[np.int64(2), np.int64(1), np.int64(0)]
The losers are
[np.int64(3)]


Round 2
The bid information is
+--------------+---------------+---------------+-----+
| House Number |       0       |       1       |  2  |
+--------------+---------------+---------------+-----+
|    Buyer     | [np.int64(2)] | [np.int64(1)] | [3] |
+--------------+---------------+---------------+-----+
The bid prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 3 | 2 | 2 |
+--------------+---+---+---+
The winners are
[np.int64(2), np.int64(1), np.int64(3)]
The losers are
[np.int64(0)]


Round 3
The bid information is
+--------------+---------------+---------------+-----+
| House Number |       0       |       1       |  2  |
+--------------+---------------+---------------+-----+
|    Buyer     | [np.int64(2)] | [np.int64(1)] | [0] |
+--------------+---------------+---------------+-----+
The bid prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 3 | 2 | 3 |
+--------------+---+---+---+
The winners are
[np.int64(2), np.int64(1), np.int64(0)]
The losers are
[np.int64(3)]


Round 4
The bid information is
+--------------+---------------+-----+---------------+
| House Number |       0       |  1  |       2       |
```

(continues on next page)

(continued from previous page)

```
+--------------+---------------+-----+---------------+
|    Buyer     | [np.int64(2)] | [3] | [np.int64(0)] |
+--------------+---------------+-----+---------------+
The bid prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 3 | 3 | 3 |
+--------------+---+---+---+
The winners are
[np.int64(2), np.int64(3), np.int64(0)]
The losers are
[np.int64(1)]


Round 5
The bid information is
+--------------+---------------+-----+---------------+
| House Number |       0       |  1  |       2       |
+--------------+---------------+-----+---------------+
|    Buyer     | [np.int64(2)] | [1] | [np.int64(0)] |
+--------------+---------------+-----+---------------+
The bid prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 3 | 4 | 3 |
+--------------+---+---+---+
The winners are
[np.int64(2), np.int64(1), np.int64(0)]
The losers are
[np.int64(3)]


Auction ends


The final result is as follows


The allocation plan is
+--------------+---------------+---------------+---------------+
| House Number |       0       |       1       |       2       |
+--------------+---------------+---------------+---------------+
|    Buyer     | [np.int64(2)] | [np.int64(1)] | [np.int64(0)] |
+--------------+---------------+---------------+---------------+
The bid prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 3 | 4 | 3 |
+--------------+---+---+---+
The winners are
[np.int64(2), np.int64(1), np.int64(0)]
The losers are
[np.int64(3)]
The houses unsold are
```

(continues on next page)

(continued from previous page)

```
[]
The total revenue is 10.00
```

```
# the surplus matrix S

auction_1.S
```

```
array([[0., 0., 6., 0.],
       [0., 7., 0., 0.],
       [6., 0., 0., 0.]])
```

```
# the quantity matrix X

auction_1.Q
```

```
array([[0., 0., 1., 0., 0.],
       [0., 1., 0., 0., 0.],
       [1., 0., 0., 0., 0.]])
```

93.9 Robustness Checks

Let's do some stress testing of our code by applying it to auctions characterized by different matrices of private values.

1. number of houses = number of buyers

```
v2 = np.array([[8,5,9],[4,11,7],[9,7,6]])

auction_2 = ascending_bid_auction(v2, r, ϵ)

auction_2.start_auction()
```

```
The Ascending Bid Auction for Houses


Basic Information: 3 houses, 3 buyers
The valuation matrix is as follows
+--------------+---+----+---+
| Buyer Number | 0 | 1  | 2 |
+--------------+---+----+---+
|   House 0    | 8 | 5  | 9 |
|   House 1    | 4 | 11 | 7 |
|   House 2    | 9 | 7  | 6 |
+--------------+---+----+---+
The reservation prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 2 | 1 | 0 |
+--------------+---+---+---+
The minimum increment of bid price is 1.00
```

(continues on next page)

(continued from previous page)

```
Auction starts successfully


Round 1
The bid information is
+--------------+-----+-----+-----+
| House Number |  0  |  1  |  2  |
+--------------+-----+-----+-----+
|    Buyer     | [2] | [1] | [0] |
+--------------+-----+-----+-----+
The bid prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 3 | 2 | 1 |
+--------------+---+---+---+
The winners are
[np.int64(2), np.int64(1), np.int64(0)]
The losers are
[]


The auction ends because every buyer gets one house.


Auction ends


The final result is as follows


The allocation plan is
+--------------+---------------+---------------+---------------+
| House Number |       0       |       1       |       2       |
+--------------+---------------+---------------+---------------+
|    Buyer     | [np.int64(2)] | [np.int64(1)] | [np.int64(0)] |
+--------------+---------------+---------------+---------------+
The bid prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 3 | 2 | 1 |
+--------------+---+---+---+
The winners are
[np.int64(2), np.int64(1), np.int64(0)]
The losers are
[]
The houses unsold are
[]
The total revenue is 6.00
```

2. multilple excess buyers

```
v3 = np.array([[8,5,9,4,3],[4,11,7,4,6],[9,7,6,4,2]])

auction_3 = ascending_bid_auction(v3, r, ϵ)

```

(continues on next page)

(continued from previous page)

```
auction_3.start_auction()
```

```
The Ascending Bid Auction for Houses


Basic Information: 3 houses, 5 buyers
The valuation matrix is as follows
+--------------+---+----+---+---+---+
| Buyer Number | 0 | 1  | 2 | 3 | 4 |
+--------------+---+----+---+---+---+
|   House 0    | 8 | 5  | 9 | 4 | 3 |
|   House 1    | 4 | 11 | 7 | 4 | 6 |
|   House 2    | 9 | 7  | 6 | 4 | 2 |
+--------------+---+----+---+---+---+
The reservation prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 2 | 1 | 0 |
+--------------+---+---+---+
The minimum increment of bid price is 1.00


Auction starts successfully


Round 1
The bid information is
+--------------+-----+-------+-------+
| House Number |  0  |   1   |   2   |
+--------------+-----+-------+-------+
|    Buyer     | [2] | [1 4] | [0 3] |
+--------------+-----+-------+-------+
The bid prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 3 | 2 | 1 |
+--------------+---+---+---+
The winners are
[np.int64(2), np.int64(4), np.int64(3)]
The losers are
[np.int64(0), np.int64(1)]


Round 2
The bid information is
+--------------+---------------+-----+-----+
| House Number |       0       |  1  |  2  |
+--------------+---------------+-----+-----+
|    Buyer     | [np.int64(2)] | [1] | [0] |
+--------------+---------------+-----+-----+
The bid prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
```

(continues on next page)

(continued from previous page)

```
|    Price     | 3 | 3 | 2 |
+--------------+---+---+---+
The winners are
[np.int64(2), np.int64(1), np.int64(0)]
The losers are
[np.int64(3), np.int64(4)]


Round 3
The bid information is
+--------------+---------------+-----+-----+
| House Number |       0       |  1  |  2  |
+--------------+---------------+-----+-----+
|    Buyer     | [np.int64(2)] | [4] | [3] |
+--------------+---------------+-----+-----+
The bid prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 3 | 4 | 3 |
+--------------+---+---+---+
The winners are
[np.int64(2), np.int64(4), np.int64(3)]
The losers are
[np.int64(0), np.int64(1)]


Round 4
The bid information is
+--------------+---------------+-----+-----+
| House Number |       0       |  1  |  2  |
+--------------+---------------+-----+-----+
|    Buyer     | [np.int64(2)] | [1] | [0] |
+--------------+---------------+-----+-----+
The bid prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 3 | 5 | 4 |
+--------------+---+---+---+
The winners are
[np.int64(2), np.int64(1), np.int64(0)]
The losers are
[np.int64(3), np.int64(4)]


Auction ends


The final result is as follows


The allocation plan is
+--------------+---------------+---------------+---------------+
| House Number |       0       |       1       |       2       |
+--------------+---------------+---------------+---------------+
|    Buyer     | [np.int64(2)] | [np.int64(1)] | [np.int64(0)] |
```

(continues on next page)

(continued from previous page)

```
+--------------+--------------+--------------+---------------+
The bid prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 3 | 5 | 4 |
+--------------+---+---+---+
The winners are
[np.int64(2), np.int64(1), np.int64(0)]
The losers are
[np.int64(3), np.int64(4)]
The houses unsold are
[]
The total revenue is 12.00
```

3. more houses than buyers

```
v4 = np.array([[8,5,4],[4,11,7],[9,7,9],[6,4,5],[2,2,2]])
r2 = np.array([2,1,0,1,1])

auction_4 = ascending_bid_auction(v4, r2, ϵ)

auction_4.start_auction()
```

```
The Ascending Bid Auction for Houses


Basic Information: 5 houses, 3 buyers
The valuation matrix is as follows
+--------------+---+----+---+
| Buyer Number | 0 | 1  | 2 |
+--------------+---+----+---+
|   House 0    | 8 | 5  | 4 |
|   House 1    | 4 | 11 | 7 |
|   House 2    | 9 | 7  | 9 |
|   House 3    | 6 | 4  | 5 |
|   House 4    | 2 | 2  | 2 |
+--------------+---+----+---+
The reservation prices for houses are
+--------------+---+---+---+---+---+
| House Number | 0 | 1 | 2 | 3 | 4 |
+--------------+---+---+---+---+---+
|    Price     | 2 | 1 | 0 | 1 | 1 |
+--------------+---+---+---+---+---+
The minimum increment of bid price is 1.00


Auction starts successfully


Round 1
The bid information is
+--------------+-----+-------+
| House Number |  1  |   2   |
+--------------+-----+-------+
|    Buyer     | [1] | [0 2] |
```

(continues on next page)

(continued from previous page)

```
+--------------+-----+-------+
The bid prices for houses are
+--------------+---+---+---+---+---+
| House Number | 0 | 1 | 2 | 3 | 4 |
+--------------+---+---+---+---+---+
|    Price     | 2 | 2 | 1 | 1 | 1 |
+--------------+---+---+---+---+---+
The winners are
[np.int64(1), np.int64(2)]
The losers are
[np.int64(0)]


Round 2
The bid information is
+--------------+---------------+-----+
| House Number |       1       |  2  |
+--------------+---------------+-----+
|    Buyer     | [np.int64(1)] | [0] |
+--------------+---------------+-----+
The bid prices for houses are
+--------------+---+---+---+---+---+
| House Number | 0 | 1 | 2 | 3 | 4 |
+--------------+---+---+---+---+---+
|    Price     | 2 | 2 | 2 | 1 | 1 |
+--------------+---+---+---+---+---+
The winners are
[np.int64(1), np.int64(0)]
The losers are
[np.int64(2)]


Round 3
The bid information is
+--------------+---------------+-----+
| House Number |       1       |  2  |
+--------------+---------------+-----+
|    Buyer     | [np.int64(1)] | [2] |
+--------------+---------------+-----+
The bid prices for houses are
+--------------+---+---+---+---+---+
| House Number | 0 | 1 | 2 | 3 | 4 |
+--------------+---+---+---+---+---+
|    Price     | 2 | 2 | 3 | 1 | 1 |
+--------------+---+---+---+---+---+
The winners are
[np.int64(1), np.int64(2)]
The losers are
[np.int64(0)]


Round 4
The bid information is
+--------------+---------------+-----+
| House Number |       1       |  2  |
+--------------+---------------+-----+
|    Buyer     | [np.int64(1)] | [0] |
```

(continues on next page)

(continued from previous page)

```
+--------------+--------------+-----+
The bid prices for houses are
+--------------+---+---+---+---+---+
| House Number | 0 | 1 | 2 | 3 | 4 |
+--------------+---+---+---+---+---+
|    Price     | 2 | 2 | 4 | 1 | 1 |
+--------------+---+---+---+---+---+
The winners are
[np.int64(1), np.int64(0)]
The losers are
[np.int64(2)]


Round 5
The bid information is
+--------------+-----+--------------+
| House Number |  1  |      2       |
+--------------+-----+--------------+
|    Buyer     | [2] | [np.int64(0)] |
+--------------+-----+--------------+
The bid prices for houses are
+--------------+---+---+---+---+---+
| House Number | 0 | 1 | 2 | 3 | 4 |
+--------------+---+---+---+---+---+
|    Price     | 2 | 3 | 4 | 1 | 1 |
+--------------+---+---+---+---+---+
The winners are
[np.int64(2), np.int64(0)]
The losers are
[np.int64(1)]


Round 6
The bid information is
+--------------+-----+--------------+
| House Number |  1  |      2       |
+--------------+-----+--------------+
|    Buyer     | [1] | [np.int64(0)] |
+--------------+-----+--------------+
The bid prices for houses are
+--------------+---+---+---+---+---+
| House Number | 0 | 1 | 2 | 3 | 4 |
+--------------+---+---+---+---+---+
|    Price     | 2 | 4 | 4 | 1 | 1 |
+--------------+---+---+---+---+---+
The winners are
[np.int64(1), np.int64(0)]
The losers are
[np.int64(2)]


Round 7
The bid information is
+--------------+--------------+-----+
| House Number |      1       |  2  |
+--------------+--------------+-----+
|    Buyer     | [np.int64(1)] | [2] |
```

(continues on next page)

(continued from previous page)

```
+--------------+--------------+-----+
The bid prices for houses are
+--------------+---+---+---+---+---+
| House Number | 0 | 1 | 2 | 3 | 4 |
+--------------+---+---+---+---+---+
|    Price     | 2 | 4 | 5 | 1 | 1 |
+--------------+---+---+---+---+---+
The winners are
[np.int64(1), np.int64(2)]
The losers are
[np.int64(0)]


Round 8
The bid information is
+--------------+--------------+--------------+-----+
| House Number |      1       |      2       |  0  |
+--------------+--------------+--------------+-----+
|    Buyer     | [np.int64(1)] | [np.int64(2)] | [0] |
+--------------+--------------+--------------+-----+
The bid prices for houses are
+--------------+---+---+---+---+---+
| House Number | 0 | 1 | 2 | 3 | 4 |
+--------------+---+---+---+---+---+
|    Price     | 3 | 4 | 5 | 1 | 1 |
+--------------+---+---+---+---+---+
The winners are
[np.int64(1), np.int64(2), np.int64(0)]
The losers are
[]


The auction ends because every buyer gets one house.


Auction ends


The final result is as follows


The allocation plan is
+--------------+--------------+--------------+--------------+
| House Number |      1       |      2       |      0       |
+--------------+--------------+--------------+--------------+
|    Buyer     | [np.int64(1)] | [np.int64(2)] | [np.int64(0)] |
+--------------+--------------+--------------+--------------+
The bid prices for houses are
+--------------+---+---+---+---+---+
| House Number | 0 | 1 | 2 | 3 | 4 |
+--------------+---+---+---+---+---+
|    Price     | 3 | 4 | 5 | 1 | 1 |
+--------------+---+---+---+---+---+
The winners are
[np.int64(1), np.int64(2), np.int64(0)]
The losers are
[]
```

(continues on next page)

(continued from previous page)

```
The houses unsold are
[np.int64(3), np.int64(4)]
The total revenue is 12.00
```

4. some houses have extremely high reservation prices

```
v5 = np.array([[8,5,4],[4,11,7],[9,7,9],[6,4,5],[2,2,2]])
r3 = np.array([10,1,0,1,1])

auction_5 = ascending_bid_auction(v5, r3, ϵ)

auction_5.start_auction()
```

```
The Ascending Bid Auction for Houses


Basic Information: 5 houses, 3 buyers
The valuation matrix is as follows
+--------------+---+----+---+
| Buyer Number | 0 | 1  | 2 |
+--------------+---+----+---+
|   House 0    | 8 | 5  | 4 |
|   House 1    | 4 | 11 | 7 |
|   House 2    | 9 | 7  | 9 |
|   House 3    | 6 | 4  | 5 |
|   House 4    | 2 | 2  | 2 |
+--------------+---+----+---+
The reservation prices for houses are
+--------------+----+---+---+---+---+
| House Number | 0  | 1 | 2 | 3 | 4 |
+--------------+----+---+---+---+---+
|    Price     | 10 | 1 | 0 | 1 | 1 |
+--------------+----+---+---+---+---+
The minimum increment of bid price is 1.00


Auction starts successfully


Round 1
The bid information is
+--------------+-----+-------+
| House Number |  1  |   2   |
+--------------+-----+-------+
|    Buyer     | [1] | [0 2] |
+--------------+-----+-------+
The bid prices for houses are
+--------------+----+---+---+---+---+
| House Number | 0  | 1 | 2 | 3 | 4 |
+--------------+----+---+---+---+---+
|    Price     | 10 | 2 | 1 | 1 | 1 |
+--------------+----+---+---+---+---+
The winners are
[np.int64(1), np.int64(0)]
The losers are
[np.int64(2)]
```

(continues on next page)

(continued from previous page)

```
Round 2
The bid information is
+--------------+---------------+-----+
| House Number |       1       |  2  |
+--------------+---------------+-----+
|    Buyer     | [np.int64(1)] | [2] |
+--------------+---------------+-----+
The bid prices for houses are
+--------------+----+---+---+---+---+
| House Number | 0  | 1 | 2 | 3 | 4 |
+--------------+----+---+---+---+---+
|    Price     | 10 | 2 | 2 | 1 | 1 |
+--------------+----+---+---+---+---+
The winners are
[np.int64(1), np.int64(2)]
The losers are
[np.int64(0)]


Round 3
The bid information is
+--------------+---------------+-----+
| House Number |       1       |  2  |
+--------------+---------------+-----+
|    Buyer     | [np.int64(1)] | [0] |
+--------------+---------------+-----+
The bid prices for houses are
+--------------+----+---+---+---+---+
| House Number | 0  | 1 | 2 | 3 | 4 |
+--------------+----+---+---+---+---+
|    Price     | 10 | 2 | 3 | 1 | 1 |
+--------------+----+---+---+---+---+
The winners are
[np.int64(1), np.int64(0)]
The losers are
[np.int64(2)]


Round 4
The bid information is
+--------------+---------------+-----+
| House Number |       1       |  2  |
+--------------+---------------+-----+
|    Buyer     | [np.int64(1)] | [2] |
+--------------+---------------+-----+
The bid prices for houses are
+--------------+----+---+---+---+---+
| House Number | 0  | 1 | 2 | 3 | 4 |
+--------------+----+---+---+---+---+
|    Price     | 10 | 2 | 4 | 1 | 1 |
+--------------+----+---+---+---+---+
The winners are
[np.int64(1), np.int64(2)]
The losers are
[np.int64(0)]
```

(continues on next page)

(continued from previous page)

```
Round 5
The bid information is
+--------------+---------------+-----+
| House Number |       1       |  2  |
+--------------+---------------+-----+
|    Buyer     | [np.int64(1)] | [0] |
+--------------+---------------+-----+
The bid prices for houses are
+--------------+----+---+---+---+---+
| House Number | 0  | 1 | 2 | 3 | 4 |
+--------------+----+---+---+---+---+
|    Price     | 10 | 2 | 5 | 1 | 1 |
+--------------+----+---+---+---+---+
The winners are
[np.int64(1), np.int64(0)]
The losers are
[np.int64(2)]


Round 6
The bid information is
+--------------+-----+---------------+
| House Number |  1  |       2       |
+--------------+-----+---------------+
|    Buyer     | [2] | [np.int64(0)] |
+--------------+-----+---------------+
The bid prices for houses are
+--------------+----+---+---+---+---+
| House Number | 0  | 1 | 2 | 3 | 4 |
+--------------+----+---+---+---+---+
|    Price     | 10 | 3 | 5 | 1 | 1 |
+--------------+----+---+---+---+---+
The winners are
[np.int64(2), np.int64(0)]
The losers are
[np.int64(1)]


Round 7
The bid information is
+--------------+-----+---------------+
| House Number |  1  |       2       |
+--------------+-----+---------------+
|    Buyer     | [1] | [np.int64(0)] |
+--------------+-----+---------------+
The bid prices for houses are
+--------------+----+---+---+---+---+
| House Number | 0  | 1 | 2 | 3 | 4 |
+--------------+----+---+---+---+---+
|    Price     | 10 | 4 | 5 | 1 | 1 |
+--------------+----+---+---+---+---+
The winners are
[np.int64(1), np.int64(0)]
The losers are
[np.int64(2)]
```

(continues on next page)

(continued from previous page)

```
Round 8
The bid information is
+--------------+---------------+-----+
| House Number |       1       |  2  |
+--------------+---------------+-----+
|    Buyer     | [np.int64(1)] | [2] |
+--------------+---------------+-----+
The bid prices for houses are
+--------------+----+---+---+---+---+
| House Number | 0  | 1 | 2 | 3 | 4 |
+--------------+----+---+---+---+---+
|    Price     | 10 | 4 | 6 | 1 | 1 |
+--------------+----+---+---+---+---+
The winners are
[np.int64(1), np.int64(2)]
The losers are
[np.int64(0)]


Round 9
The bid information is
+--------------+---------------+---------------+-----+
| House Number |       1       |       2       |  3  |
+--------------+---------------+---------------+-----+
|    Buyer     | [np.int64(1)] | [np.int64(2)] | [0] |
+--------------+---------------+---------------+-----+
The bid prices for houses are
+--------------+----+---+---+---+---+
| House Number | 0  | 1 | 2 | 3 | 4 |
+--------------+----+---+---+---+---+
|    Price     | 10 | 4 | 6 | 2 | 1 |
+--------------+----+---+---+---+---+
The winners are
[np.int64(1), np.int64(2), np.int64(0)]
The losers are
[]


The auction ends because every buyer gets one house.


Auction ends


The final result is as follows


The allocation plan is
+--------------+---------------+---------------+---------------+
| House Number |       1       |       2       |       3       |
+--------------+---------------+---------------+---------------+
|    Buyer     | [np.int64(1)] | [np.int64(2)] | [np.int64(0)] |
+--------------+---------------+---------------+---------------+
The bid prices for houses are
+--------------+----+---+---+---+---+
```

(continues on next page)

(continued from previous page)

```
| House Number | 0  | 1 | 2 | 3 | 4 |
+--------------+----+---+---+---+---+
|    Price     | 10 | 4 | 6 | 2 | 1 |
+--------------+----+---+---+---+---+
The winners are
[np.int64(1), np.int64(2), np.int64(0)]
The losers are
[]
The houses unsold are
[np.int64(0), np.int64(4)]
The total revenue is 12.00
```

5. reservation prices are so high that the auction can't start

```
r4 = np.array([15,15,15])

auction_6 = ascending_bid_auction(v, r4, ϵ)

auction_6.start_auction()
```

```
The Ascending Bid Auction for Houses


Basic Information: 3 houses, 4 buyers
The valuation matrix is as follows
+--------------+---+----+---+---+
| Buyer Number | 0 | 1  | 2 | 3 |
+--------------+---+----+---+---+
|   House 0    | 8 | 5  | 9 | 4 |
|   House 1    | 4 | 11 | 7 | 4 |
|   House 2    | 9 | 7  | 6 | 4 |
+--------------+---+----+---+---+
The reservation prices for houses are
+--------------+----+----+----+
| House Number | 0  | 1  | 2  |
+--------------+----+----+----+
|    Price     | 15 | 15 | 15 |
+--------------+----+----+----+
The minimum increment of bid price is 1.00


The auction can not start because of high reservation prices
```

93.10 A Groves-Clarke Mechanism

We now decribe an alternative way for society to allocate n houses to m possible buyers in a way that maximizes total value across all potential buyers.

We continue to assume that each buyer can purchase at most one house.

The mechanism is a very special case of a Groves-Clarke mechanism [Groves, 1973], [Clarke, 1971].

Its special structure substantially simplifies writing Python code to find an optimal allocation.

Our mechanims works like this.

- The values V_{ij} are private information to person j
- The mechanism makes each person j willing to tell a social planner his private values $V_{i,j}$ for all $i = 1, \ldots, n$.
- The social planner asks all potential bidders to tell the planner their private values V_{ij}
- The social planner tells no one these, but uses them to allocate houses and set prices
- The mechanism is designed in a way that makes all prospective buyers want to tell the planner their private values
  - truth telling is a dominant strategy for each potential buyer
- The planner finds a house, bidder pair with highest private value by computing $(\tilde{i}, \tilde{j}) = \text{argmax}(V_{ij})$
- The planner assigns house $\tilde{i}$ to buyer $\tilde{j}$
- The planner charges buyer $\tilde{j}$ the price $\max_{-\tilde{j}} V_{\tilde{i},j}$, where $-\tilde{j}$ means all j's except $\tilde{j}$.
- The planner creates a matrix of private values for the remaining houses $-\tilde{i}$ by deleting row (i.e., house) $\tilde{i}$ and column (i.e., buyer) $\tilde{j}$ from V.
  - (But in doing this, the planner keeps track of the real names of the bidders and the houses).
- The planner returns to the original step and repeat it.
- The planner iterates until all n houses are allocated and the charges for all n houses are set.

93.11 An Example Solved by Hand

Let's see how our Groves-Clarke algorithm would work for the following simple matrix V matrix of private values

$$V = \begin{bmatrix} 10 & 9 & 8 & 7 & 6 \\ 9 & 9 & 7 & 6 & 6 \\ 8 & 6 & 6 & 9 & 4 \\ 7 & 5 & 6 & 4 & 9 \end{bmatrix}$$

Remark: In the first step, when the highest private value corresponds to more than one house, bidder pairs, we choose the pair with the highest sale price. If a highest sale price corresponds to two or more pairs with highest private values, we randomly choose one.

```
np.random.seed(666)

V_orig = np.array([[10, 9, 8, 7, 6],  # record the origianl values
                   [9, 9, 7, 6, 6],
                   [8, 6, 6, 9, 4],
                   [7, 5, 6, 4, 9]])
V = np.copy(V_orig)  # used iteratively
n, m = V.shape
p = np.zeros(n) # prices of houses
Q = np.zeros((n, m)) # keep record the status of houses and buyers
```

First assignment

First, we find house, bidder pair with highest private value.

```
i, j = np.where(V==np.max(V))
i, j
```

```
(array([0]), array([0]))
```

So, house 0 will be sold to buyer 0 at a price of 9. We then update the sale price of house 0 and the status matrix Q.

```
p[i] = np.max(np.delete(V[i, :], j))
Q[i, j] = 1
p, Q
```

```
(array([9., 0., 0., 0.]),
 array([[1., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.]]))
```

Then we remove row 0 and column 0 from V. To keep the real number of houses and buyers, we set this row and this column to -1, which will have the same result as removing them since $V \geq 0$.

```
V[i, :] = -1
V[:, j] = -1
V
```

```
array([[-1, -1, -1, -1, -1],
       [-1,  9,  7,  6,  6],
       [-1,  6,  6,  9,  4],
       [-1,  5,  6,  4,  9]])
```

Second assignment

We find house, bidder pair with the highest private value again.

```
i, j = np.where(V==np.max(V))
i, j
```

```
(array([1, 2, 3]), array([1, 3, 4]))
```

In this special example, there are three pairs (1, 1), (2, 3) and (3, 4) with the highest private value. To solve this problem, we choose the one with highest sale price.

```
p_candidate = np.zeros(len(i))
for k in range(len(i)):
    p_candidate[k] = np.max(np.delete(V[i[k], :], j[k]))
k, = np.where(p_candidate==np.max(p_candidate))
i, j = i[k], j[k]
i, j
```

```
(array([1]), array([1]))
```

So, house 1 will be sold to buyer 1 at a price of 7. We update matrices.

```
p[i] = np.max(np.delete(V[i, :], j))
Q[i, j] = 1
V[i, :] = -1
V[:, j] = -1
p, Q, V
```

```
(array([9., 7., 0., 0.]),
 array([[1., 0., 0., 0., 0.],
        [0., 1., 0., 0., 0.],
```

(continues on next page)

(continued from previous page)

```
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.]]),
 array([[-1, -1, -1, -1, -1],
        [-1, -1, -1, -1, -1],
        [-1, -1,  6,  9,  4],
        [-1, -1,  6,  4,  9]]))
```

Third assignment

```
i, j = np.where(V==np.max(V))
i, j
```

```
(array([2, 3]), array([3, 4]))
```

In this special example, there are two pairs (2, 3) and (3, 4) with the highest private value.

To resolve the assignment, we choose the one with highest sale price.

```
p_candidate = np.zeros(len(i))
for k in range(len(i)):
    p_candidate[k] = np.max(np.delete(V[i[k], :], j[k]))
k, = np.where(p_candidate==np.max(p_candidate))
i, j = i[k], j[k]
i, j
```

```
(array([2, 3]), array([3, 4]))
```

The two pairs even have the same sale price.

We randomly choose one pair.

```
k = np.random.choice(len(i))
i, j = i[k], j[k]
i, j
```

```
(np.int64(2), np.int64(3))
```

Finally, house 2 will be sold to buyer 3.

We update matrices accordingly.

```
p[i] = np.max(np.delete(V[i, :], j))
Q[i, j] = 1
V[i, :] = -1
V[:, j] = -1
p, Q, V
```

```
(array([9., 7., 6., 0.]),
 array([[1., 0., 0., 0., 0.],
        [0., 1., 0., 0., 0.],
        [0., 0., 0., 1., 0.],
        [0., 0., 0., 0., 0.]]),
 array([[-1, -1, -1, -1, -1],
        [-1, -1, -1, -1, -1],
        [-1, -1, -1, -1, -1],
        [-1, -1,  6, -1,  9]]))
```

Fourth assignment

```
i, j = np.where(V==np.max(V))
i, j
```

```
(array([3]), array([4]))
```

House 3 will be sold to buyer 4.

The final outcome follows.

```
p[i] = np.max(np.delete(V[i, :], j))
Q[i, j] = 1
V[i, :] = -1
V[:, j] = -1
S = V_orig*Q - np.diag(p)@Q
p, Q, V, S
```

```
(array([9., 7., 6., 6.]),
 array([[1., 0., 0., 0., 0.],
        [0., 1., 0., 0., 0.],
        [0., 0., 0., 1., 0.],
        [0., 0., 0., 0., 1.]]),
 array([[-1, -1, -1, -1, -1],
        [-1, -1, -1, -1, -1],
        [-1, -1, -1, -1, -1],
        [-1, -1, -1, -1, -1]]),
 array([[1., 0., 0., 0., 0.],
        [0., 2., 0., 0., 0.],
        [0., 0., 0., 3., 0.],
        [0., 0., 0., 0., 3.]]))
```

93.12 Another Python Class

It is efficient to assemble our calculations in a single Python Class.

```
class GC_Mechanism:

    def __init__(self, V):
        """
        Implementation of the special Groves Clarke Mechanism for house auction.

        Parameters:
        ----------
        V: 2 dimensional private value matrix

        """

        self.V_orig = V.copy()
        self.V = V.copy()
        self.n, self.m = self.V.shape
        self.p = np.zeros(self.n)
        self.Q = np.zeros((self.n, self.m))
        self.S = np.copy(self.Q)
```

(continues on next page)

(continued from previous page)

```
    def find_argmax(self):
        """
        Find the house-buyer pair with the highest value.
        When the highest private value corresponds to more than one house, bidder␣
↪pairs,
        we choose the pair with the highest sale price.
        Moreoever, if the highest sale price corresponds to two or more pairs with␣
↪highest private value,
        We randomly choose one.

        Parameters:
        ----------
        V: 2 dimensional private value matrix with -1 indicating revomed rows and␣
↪columns

        Returns:
        ----------
        i: the index of the sold house

        j: the index of the buyer

        """
        i, j = np.where(self.V==np.max(self.V))

        if (len(i)>1):
            p_candidate = np.zeros(len(i))
            for k in range(len(i)):
                p_candidate[k] = np.max(np.delete(self.V[i[k], :], j[k]))
            k, = np.where(p_candidate==np.max(p_candidate))
            i, j = i[k], j[k]

            if (len(i)>1):
                k = np.random.choice(len(i))
                k = np.array([k])
                i, j = i[k], j[k]
        return i, j

    def update_status(self, i, j):
        self.p[i] = np.max(np.delete(self.V[i, :], j))
        self.Q[i, j] = 1
        self.V[i, :] = -1
        self.V[:, j] = -1

    def calculate_surplus(self):
        self.S = self.V_orig*self.Q - np.diag(self.p)@self.Q

    def start(self):
        while (np.max(self.V)>=0):
            i, j = self.find_argmax()
            self.update_status(i, j)
            print("House %i is sold to buyer %i at price %i"%(i[0], j[0], self.
↪p[i[0]]))
            print("\n")
        self.calculate_surplus()
        print("Prices of house:\n", self.p)
        print("\n")
        print("The status matrix:\n", self.Q)
```

(continues on next page)

(continued from previous page)

```
        print("\n")
        print("The surplus matrix:\n", self.S)
```

```
np.random.seed(666)

V_orig = np.array([[10, 9, 8, 7, 6],
                   [9, 9, 7, 6, 6],
                   [8, 6, 6, 9, 4],
                   [7, 5, 6, 4, 9]])
gc_mechanism = GC_Mechanism(V_orig)
gc_mechanism.start()
```

```
House 0 is sold to buyer 0 at price 9


House 1 is sold to buyer 1 at price 7


House 2 is sold to buyer 3 at price 6


House 3 is sold to buyer 4 at price 6


Prices of house:
 [9. 7. 6. 6.]


The status matrix:
 [[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]


The surplus matrix:
 [[1. 0. 0. 0. 0.]
 [0. 2. 0. 0. 0.]
 [0. 0. 0. 3. 0.]
 [0. 0. 0. 0. 3.]]
```

93.12.1 Elaborations

Here we use some additional notation designed to conform with standard notation in parts of the VCG literature.

We want to verify that our pseudo code is indeed a **pivot mechanism**, also called a **VCG** (Vickrey-Clarke-Groves) mechanism.

- The mechanism is named after [Groves, 1973], [Clarke, 1971], and [Vickrey, 1961].

To prepare for verifying this, we add some notation.

Let X be the set of feasible allocations of houses under the protocols above (i.e., at most one house to each person).

Let $X(v)$ be the allocation that the mechanism chooses for matrix v of private values.

The mechanism maps a matrix v of private values into an $x \in X$.

Let $v_j(x)$ be the value that person j attaches to allocation $x \in X$.

Let $\check{t}_j(v)$ the payment that the mechanism charges person j.

The VCG mechanism chooses the allocation

$$X(v) = \text{argmax}_{x \in X} \sum_{j=1}^{m} v_j(x) \tag{93.1}$$

and charges person j a "social cost"

$$\check{t}_j(v) = \max_{x \in X} \sum_{k \neq j} v_k(x) - \sum_{k \neq j} v_k(X(v)) \tag{93.2}$$

In our setting, equation (93.1) says that the VCG allocation allocates houses to maximize the total value of the successful prospective buyers.

In our setting, equation (93.2) says that the mechanism charges people for the externality that their presence in society imposes on other prospective buyers.

Thus, notice that according to equation (93.2):

- unsuccessful prospective buyers pay 0 because removing them from "society" would not affect the allocation chosen by the mechanim
- successful prospective buyers pay the difference between the total value society could achieve without them present and the total value that others present in society do achieve under the mechanism.

The generalized second-price auction described in our pseudo code above does indeed satisfy (1). We want to compute $\check{t}_j$ for $j = 1, \dots, m$ and compare with p_j from the second price auction.

93.12.2 Social Cost

Using the GC_Mechanism class, we can calculate the social cost of each buyer.

Let's see a simpler example with private value matrix

$$V = \begin{bmatrix} 10 & 9 & 8 & 7 & 6 \\ 9 & 8 & 7 & 6 & 6 \\ 8 & 7 & 6 & 5 & 4 \end{bmatrix}$$

To begin with, we implement the GC mechanism and see the outcome.

```
np.random.seed(666)

V_orig = np.array([[10, 9, 8, 7, 6],
                   [9, 8, 7, 6, 6],
                   [8, 7, 6, 5, 4]])
gc_mechanism = GC_Mechanism(V_orig)
gc_mechanism.start()
```

```
House 0 is sold to buyer 0 at price 9


House 1 is sold to buyer 1 at price 7


House 2 is sold to buyer 2 at price 5
```

(continues on next page)

(continued from previous page)

```
Prices of house:
 [9. 7. 5.]


The status matrix:
 [[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]]


The surplus matrix:
 [[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]]
```

We exclude buyer 0 and calculate the allocation.

```
V_exc_0 = np.copy(V_orig)
V_exc_0[:, 0] = -1
V_exc_0
gc_mechanism_exc_0 = GC_Mechanism(V_exc_0)
gc_mechanism_exc_0.start()
```

```
House 0 is sold to buyer 1 at price 8


House 1 is sold to buyer 2 at price 6


House 2 is sold to buyer 3 at price 4


Prices of house:
 [8. 6. 4.]


The status matrix:
 [[0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]]


The surplus matrix:
 [[-0.  1.  0.  0.  0.]
 [-0.  0.  1.  0.  0.]
 [-0.  0.  0.  1.  0.]]
```

Calculate the social cost of buyer 0.

```
print("The social cost of buyer 0:",
     np.sum(gc_mechanism_exc_0.Q*gc_mechanism_exc_0.V_orig)-np.sum(np.delete(gc_
↪mechanism.Q*gc_mechanism.V_orig, 0, axis=1)))
```

```
The social cost of buyer 0: 7.0
```

Repeat this process for buyer 1 and buyer 2

```
V_exc_1 = np.copy(V_orig)
V_exc_1[:, 1] = -1
V_exc_1
gc_mechanism_exc_1 = GC_Mechanism(V_exc_1)
gc_mechanism_exc_1.start()

print("\nThe social cost of buyer 1:",
     np.sum(gc_mechanism_exc_1.Q*gc_mechanism_exc_1.V_orig)-np.sum(np.delete(gc_
↪mechanism.Q*gc_mechanism.V_orig, 1, axis=1)))
```

```
House 0 is sold to buyer 0 at price 8


House 1 is sold to buyer 2 at price 6


House 2 is sold to buyer 3 at price 4


Prices of house:
 [8. 6. 4.]


The status matrix:
 [[1. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]]


The surplus matrix:
 [[ 2. -0.  0.  0.  0.]
 [ 0. -0.  1.  0.  0.]
 [ 0. -0.  0.  1.  0.]]

The social cost of buyer 1: 6.0
```

```
V_exc_2 = np.copy(V_orig)
V_exc_2[:, 2] = -1
V_exc_2
gc_mechanism_exc_2 = GC_Mechanism(V_exc_2)
gc_mechanism_exc_2.start()

print("\nThe social cost of buyer 2:",
     np.sum(gc_mechanism_exc_2.Q*gc_mechanism_exc_2.V_orig)-np.sum(np.delete(gc_
↪mechanism.Q*gc_mechanism.V_orig, 2, axis=1)))
```

```
House 0 is sold to buyer 0 at price 9


House 1 is sold to buyer 1 at price 6


```

(continues on next page)

(continued from previous page)

```
House 2 is sold to buyer 3 at price 4


Prices of house:
 [9. 6. 4.]


The status matrix:
 [[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 0. 1. 0.]]


The surplus matrix:
 [[ 1.  0. -0.  0.  0.]
 [ 0.  2. -0.  0.  0.]
 [ 0.  0. -0.  1.  0.]]

The social cost of buyer 2: 5.0
```

Part XVI

Other

CHAPTER

NINETYFOUR

TROUBLESHOOTING

Contents

This page is for readers experiencing errors when running the code from the lectures.

94.1 Fixing Your Local Environment

The basic assumption of the lectures is that code in a lecture should execute whenever

1. it is executed in a Jupyter notebook and
2. the notebook is running on a machine with the latest version of Anaconda Python.

You have installed Anaconda, haven't you, following the instructions in this lecture?

Assuming that you have, the most common source of problems for our readers is that their Anaconda distribution is not up to date.

Here's a useful article on how to update Anaconda.

Another option is to simply remove Anaconda and reinstall.

You also need to keep the external code libraries, such as QuantEcon.py up to date.

For this task you can either

- use `pip install --upgrade quantecon` on the command line, or
- execute `!pip install --upgrade quantecon` within a Jupyter notebook.

If your local environment is still not working you can do two things.

First, you can use a remote machine instead, by clicking on the Launch Notebook icon available for each lecture

Second, you can report an issue, so we can try to fix your local set up.

We like getting feedback on the lectures so please don't hesitate to get in touch.

94.2 Reporting an Issue

One way to give feedback is to raise an issue through our issue tracker.

Please be as specific as possible. Tell us where the problem is and as much detail about your local set up as you can provide.

Finally, you can provide direct feedback to contact@quantecon.org

CHAPTER

NINETYFIVE

REFERENCES

CHAPTER

NINETYSIX

EXECUTION STATISTICS

This table contains the latest execution statistics.

Document	Modified	Method	Run Time (s)	Status
aiyagari	2026-02-16 04:00	cache	24.73	✅
ak2	2026-02-16 04:00	cache	12.42	✅
ak_aiyagari	2026-02-16 04:00	cache	31.35	✅
ar1_bayes	2026-02-16 04:05	cache	258.27	✅
ar1_turningpts	2026-02-16 04:05	cache	26.36	✅
back_prop	2026-02-16 04:07	cache	99.06	✅
bayes_nonconj	2026-02-16 04:26	cache	1176.66	✅
career	2026-02-16 04:27	cache	11.56	✅
cass_fiscal	2026-02-16 04:27	cache	18.14	✅
cass_fiscal_2	2026-02-16 04:27	cache	5.71	✅
cass_koopmans_1	2026-02-16 04:27	cache	14.24	✅
cass_koopmans_2	2026-02-16 04:27	cache	6.81	✅
chow_business_cycles	2026-02-19 10:09	cache	14.53	✅
cross_product_trick	2026-02-16 04:27	cache	1.14	✅
divergence_measures	2026-02-16 04:28	cache	13.17	✅
doubts_or_variability	2026-02-19 10:09	cache	7.96	✅
eig_circulant	2026-02-16 04:28	cache	5.9	✅
endogenous_lake	2026-02-16 04:36	cache	488.99	✅
exchangeable	2026-02-16 04:36	cache	8.09	✅
finite_markov	2026-02-16 04:36	cache	6.23	✅
ge_arrow	2026-02-16 04:36	cache	1.99	✅
harrison_kreps	2026-02-16 04:36	cache	3.55	✅
hoist_failure	2026-02-16 04:37	cache	52.74	✅
house_auction	2026-02-16 04:37	cache	2.98	✅
ifp_advanced	2026-02-16 04:38	cache	26.84	✅
ifp_discrete	2026-02-16 04:38	cache	13.44	✅
ifp_egm	2026-02-16 04:38	cache	10.62	✅
ifp_egm_transient_shocks	2026-02-16 04:38	cache	18.15	✅
ifp_opi	2026-02-16 04:38	cache	9.25	✅
imp_sample	2026-02-16 04:43	cache	254.96	✅
intro	2026-02-16 04:43	cache	0.93	✅
inventory_dynamics	2026-02-16 04:43	cache	9.96	✅
jv	2026-02-16 04:43	cache	15.52	✅
kalman	2026-02-16 04:43	cache	7.92	✅
kalman_2	2026-02-16 04:44	cache	43.9	✅
kesten_processes	2026-02-16 04:44	cache	17.69	✅

continues on next page

Table 96.1 – continued from previous page

Document	Modified	Method	Run Time (s)	Status
lagrangian_lqdp	2026-02-16 04:45	cache	27.96	✅
lake_model	2026-02-16 04:45	cache	11.04	✅
likelihood_bayes	2026-02-16 04:46	cache	45.07	✅
likelihood_ratio_process	2026-02-16 04:46	cache	22.88	✅
likelihood_ratio_process_2	2026-02-16 04:47	cache	30.35	✅
likelihood_var	2026-02-16 04:47	cache	21.07	✅
linear_algebra	2026-02-16 04:47	cache	2.61	✅
linear_models	2026-02-16 04:47	cache	7.97	✅
lln_clt	2026-02-16 04:47	cache	10.58	✅
lq_inventories	2026-02-16 04:48	cache	12.23	✅
lqcontrol	2026-02-16 04:48	cache	4.8	✅
markov_asset	2026-02-16 04:48	cache	5.0	✅
markov_perf	2026-02-16 04:48	cache	4.48	✅
mccall_fitted_vfi	2026-02-16 04:48	cache	43.71	✅
mccall_model	2026-02-16 04:49	cache	15.11	✅
mccall_model_with_sep_markov	2026-02-16 04:49	cache	24.6	✅
mccall_model_with_separation	2026-02-16 04:49	cache	14.74	✅
mccall_persist_trans	2026-02-16 04:50	cache	11.31	✅
mccall_q	2026-02-16 04:50	cache	15.79	✅
measurement_models	2026-02-19 10:09	cache	3.29	✅
mix_model	2026-02-16 04:51	cache	83.83	✅
mle	2026-02-16 04:51	cache	10.83	✅
morris_learn	2026-02-16 04:52	cache	15.81	✅
multi_hyper	2026-02-16 04:52	cache	18.39	✅
multivariate_normal	2026-02-16 04:52	cache	4.56	✅
navy_captain	2026-02-16 04:53	cache	27.98	✅
newton_method	2026-02-16 04:54	cache	57.18	✅
odu	2026-02-16 04:54	cache	48.75	✅
ols	2026-02-16 04:54	cache	8.28	✅
opt_transport	2026-02-16 04:55	cache	12.74	✅
os	2026-02-16 04:55	cache	1.86	✅
os_egm	2026-02-16 04:55	cache	2.41	✅
os_egm_jax	2026-02-16 04:55	cache	5.43	✅
os_numerical	2026-02-16 04:55	cache	19.55	✅
os_stochastic	2026-02-16 04:56	cache	62.84	✅
os_time_iter	2026-02-16 04:56	cache	4.82	✅
pandas_panel	2026-02-16 04:56	cache	4.7	✅
perm_income	2026-02-16 04:56	cache	3.56	✅
perm_income_cons	2026-02-16 04:57	cache	7.32	✅
prob_matrix	2026-02-16 04:57	cache	9.31	✅
prob_meaning	2026-02-16 04:58	cache	59.4	✅
qr_decomp	2026-02-16 04:58	cache	1.36	✅
rand_resp	2026-02-16 04:58	cache	2.65	✅
rational_expectations	2026-02-16 04:58	cache	3.79	✅
re_with_feedback	2026-02-16 04:58	cache	10.33	✅
samuelson	2026-02-16 04:58	cache	10.84	✅
sir_model	2026-02-16 04:58	cache	2.91	✅
stats_examples	2026-02-16 04:58	cache	4.44	✅
status	2026-02-16 04:58	cache	6.96	✅
svd_intro	2026-02-16 04:59	cache	1.42	✅

continues on next page

Table 96.1 – continued from previous page

Document	Modified	Method	Run Time (s)	Status
troubleshooting	2026-02-16 04:43	cache	0.93	✅
two_auctions	2026-02-16 04:59	cache	24.07	✅
two_computation	2026-02-19 10:10	cache	63.53	✅
uncertainty_traps	2026-02-16 04:59	cache	2.72	✅
util_rand_resp	2026-02-16 04:59	cache	2.77	✅
var_dmd	2026-02-16 04:43	cache	0.93	✅
von_neumann_model	2026-02-16 04:59	cache	2.81	✅
wald_friedman	2026-02-16 04:59	cache	19.71	✅
wald_friedman_2	2026-02-16 05:00	cache	12.9	✅
wealth_dynamics	2026-02-16 05:00	cache	32.86	✅
zreferences	2026-02-16 04:43	cache	0.93	✅

These lectures are built on `linux` instances through `github actions`.

These lectures are using the following python version

```
!python --version
```

```
Python 3.13.9
```

and the following package versions

```
!conda list
```

This lecture series has access to the following GPU

```
!nvidia-smi
```

```
Mon Feb 16 04:58:57 2026
+-----------------------------------------------------------------------------------
↪-------+
| NVIDIA-SMI 580.105.08             Driver Version: 580.105.08     CUDA Version:␣
↪13.0     |
+-----------------------------------------+------------------------+---------------
↪-------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile␣
↪Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util ␣
↪Compute M. |
|                                         |                        |               ␣
↪MIG M. |
|=========================================+========================+======================|
|   0  Tesla T4                       On  |   00000000:00:1E.0 Off |               ␣
↪     0 |
| N/A   24C    P8              9W /   70W |       0MiB /  15360MiB |      0%     ␣
↪Default |
|                                         |                        |               ␣
↪   N/A |
+-----------------------------------------+------------------------+---------------
↪-------+

+-----------------------------------------------------------------------------------
↪-------+
```

(continues on next page)

(continued from previous page)

```
| Processes:                                                                                ␣
↪       |
|  GPU   GI   CI              PID   Type   Process name                                 GPU␣
↪Memory |
|        ID   ID                                                                       ␣
↪Usage      |
|=========================================================================================|
|  No running processes found                                                               ␣
↪       |
+-------------------------------------------------------------------------------------------
↪-------+
```

You can check the backend used by JAX using:

```
import jax
# Check if JAX is using GPU
print(f"JAX backend: {jax.devices()[0].platform}")
```

```
JAX backend: gpu
```

BIBLIOGRAPHY

[AJR01] Daron Acemoglu, Simon Johnson, and James A Robinson. The colonial origins of comparative development: an empirical investigation. *The American Economic Review*, 91(5):1369–1401, 2001.

[Aiy94] S Rao Aiyagari. Uninsured Idiosyncratic Risk and Aggregate Saving. *The Quarterly Journal of Economics*, 109(3):659–684, 1994.

[Alc50] Armen A Alchian. Uncertainty, evolution, and economic theory. *Journal of political economy*, 58(3):211–221, 1950.

[AM05] D. B. O. Anderson and J. B. Moore. *Optimal Filtering*. Dover Publications, 2005.

[AHMS96] E. W. Anderson, L. P. Hansen, E. R. McGrattan, and T. J. Sargent. Mechanics of Forming and Estimating Dynamic Linear Economies. In *Handbook of Computational Economics*. Elsevier, vol 1 edition, 1996.

[And76] Harald Anderson. Estimation of a proportion through randomized response. *International Statistical Review/Revue Internationale de Statistique*, pages 213–217, 1976.

[Apo90] George Apostolakis. The concept of probability in safety assessments of technological systems. *Science*, 250(4986):1359–1364, 1990.

[AK87] Alan J Auerbach and Laurence J Kotlikoff. *Dynamic fiscal policy*. Cambridge University Press, Cambridge, 1987.

[Axt01] Robert L Axtell. Zipf distribution of us firm sizes. *science*, 293(5536):1818–1820, 2001.

[BHS09] Francisco Barillas, Lars Peter Hansen, and Thomas J. Sargent. Doubts or variability? *Journal of Economic Theory*, 144(6):2388–2418, November 2009. URL: https://ideas.repec.org/a/eee/jetheo/v144y2009i6p2388-2418.html, doi:.

[Bar79] Robert J Barro. On the Determination of the Public Debt. *Journal of Political Economy*, 87(5):940–971, 1979.

[BB18] Jess Benhabib and Alberto Bisin. Skewed wealth distributions: theory and empirics. *Journal of Economic Literature*, 56(4):1261–91, 2018.

[BBZ15] Jess Benhabib, Alberto Bisin, and Shenghao Zhu. The wealth distribution in bewley economies with capital income risk. *Journal of Economic Theory*, 159:489–515, 2015.

[BS79] L M Benveniste and J A Scheinkman. On the Differentiability of the Value Function in Dynamic Models of Economics. *Econometrica*, 47(3):727–732, 1979.

[Ber75] Dmitri Bertsekas. *Dynamic Programming and Stochastic Control*. Academic Press, New York, 1975.

[Bew77] Truman Bewley. The permanent income hypothesis: a theoretical formulation. *Journal of Economic Theory*, 16(2):252–292, 1977.

[Bew86] Truman F Bewley. Stationary monetary equilibrium with a continuum of independently fluctuating consumers. In Werner Hildenbran and Andreu Mas-Colell, editors, *Contributions to Mathematical Economics in Honor of Gerard Debreu*, pages 27–102. North-Holland, Amsterdam, 1986.

[Bis06] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[BK80] Olivier Jean Blanchard and Charles M Kahn. The Solution of Linear Difference Models under Rational Expectations. *Econometrica*, 48(5):1305–1311, July 1980.

[BE06] Lawrence Blume and David Easley. If you're so smart, why aren't you rich? belief selection in complete and incomplete markets. *Econometrica*, 74(4):929–966, 2006.

[BCE+18] Lawrence E Blume, Timothy Cogley, David A Easley, Thomas J Sargent, and Viktor Tsyrennikov. A case for incomplete markets. *Journal of Economic Theory*, 178:191–221, 2018.

[BK19] Steven L. Brunton and J. Nathan Kutz. *Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control*. Cambridge University Press, 2019.

[BK22] Steven L. Brunton and J. Nathan Kutz. *Data-Driven Science and Engineering, Second Edition*. Cambridge University Press, New York, 2022.

[BDM+16] Dariusz Buraczewski, Ewa Damek, Thomas Mikosch, and others. *Stochastic models with power-law tails*. Springer, 2016.

[Bur23] Jennifer Burns. *Milton Friedman: The Last Conservative*. Farrar, Straus, and Giroux, New York, 2023.

[Cag56] Philip Cagan. The monetary dynamics of hyperinflation. In Milton Friedman, editor, *Studies in the Quantity Theory of Money*, pages 25–117. University of Chicago Press, Chicago, 1956.

[Cap85] Andrew S Caplin. The variability of aggregate demand with (s, s) inventory policies. *Econometrica*, pages 1395–1409, 1985.

[Car01] Christopher D Carroll. A Theory of the Consumption Function, with and without Liquidity Constraints. *Journal of Economic Perspectives*, 15(3):23–45, 2001.

[Car06] Christopher D Carroll. The method of endogenous gridpoints for solving dynamic stochastic optimization problems. *Economics Letters*, 91(3):312–320, 2006.

[Cas65] David Cass. Optimum growth in an aggregative model of capital accumulation. *Review of Economic Studies*, 32(3):233–240, 1965.

[CM88] A Chadhuri and R Mukerjee. *Randomized Response: Theory and Technique*. Marcel Dekker, New York, 1988.

[Cho68] Gregory C. Chow. The acceleration principle and the nature of business cycles. *The Quarterly Journal of Economics*, 82(3):403–418, aug 1968.

[CL69] Gregory C. Chow and Richard E. Levitan. Nature of business cycles implicit in a linear economic model. *The Quarterly Journal of Economics*, 83(3):504–517, aug 1969.

[Cla71] E. Clarke. Multipart pricing of public goods. *Public Choice*, 8:19–33, 1971.

[Col90] Wilbur John Coleman. Solving the Stochastic Growth Model by Policy-Function Iteration. *Journal of Business & Economic Statistics*, 8(1):27–29, 1990.

[DFH06] Steven J Davis, R Jason Faberman, and John Haltiwanger. The flow approach to labor markets: new data sources, micro-macro links and the recent downturn. *Journal of Economic Perspectives*, 2006.

[dF37] Bruno de Finetti. La prevision: ses lois logiques, ses sources subjectives. *Annales de l'Institute Henri Poincare'*, 7:1 – 68, 1937. English translation in Kyburg and Smokler (eds.), \it Studies in Subjective Probability, Wiley, New York, 1964.

[Dea91] Angus Deaton. Saving and Liquidity Constraints. *Econometrica*, 59(5):1221–1248, 1991.

[DP94] Angus Deaton and Christina Paxson. Intertemporal Choice and Inequality. *Journal of Political Economy*, 102(3):437–467, 1994.

[DH10] Wouter J Den Haan. Comparison of solutions to the incomplete markets model with aggregate uncertainty. *Journal of Economic Dynamics and Control*, 34(1):4–27, 2010.

[Dia65] Peter A Diamond. National debt in a neoclassical growth model. *The American Economic Review*, 55(5):1126–1150, 1965.

[DS10] Ulrich Doraszelski and Mark Satterthwaite. Computable markov-perfect industry dynamics. *The RAND Journal of Economics*, 41(2):215–243, 2010.

[DSS58] Robert Dorfman, Paul A. Samuelson, and Robert M. Solow. *Linear Programming and Economic Analysis: Revised Edition*. McGraw Hill, New York, 1958.

[DLP13] Y E Du, Ehud Lehrer, and A D Y Pauzner. Competitive economy as a ranking device over networks. submitted, 2013.

[Dud02] R M Dudley. *Real Analysis and Probability*. Cambridge Studies in Advanced Mathematics. Cambridge University Press, 2002.

[DRS89] Timothy Dunne, Mark J Roberts, and Larry Samuelson. The growth and failure of us manufacturing plants. *The Quarterly Journal of Economics*, 104(4):671–698, 1989.

[ESAW18] Ashraf Ben El-Shanawany, Keith H Ardron, and Simon P Walker. Lognormal approximations of fault tree uncertainty distributions. *Risk Analysis*, 38(8):1576–1584, 2018.

[EG87] Robert F Engle and Clive W J Granger. Co-integration and Error Correction: Representation, Estimation, and Testing. *Econometrica*, 55(2):251–276, 1987.

[EP95] Richard Ericson and Ariel Pakes. Markov-perfect industry dynamics: a framework for empirical work. *The Review of Economic Studies*, 62(1):53–82, 1995.

[Eva87] David S Evans. The relationship between firm growth, size, and age: estimates for 100 manufacturing industries. *The Journal of Industrial Economics*, pages 567–581, 1987.

[EH01] G W Evans and S Honkapohja. *Learning and Expectations in Macroeconomics*. Frontiers of Economic Research. Princeton University Press, 2001.

[Fab82] Joseph F Faber. Life tables for the United States: 1900–2050. Actuarial Study Actuarial Study No. 87, Social Security Administration, Office of the Actuary, 1982.

[FSTD15] Pablo Fajgelbaum, Edouard Schaal, and Mathieu Taschereau-Dumouchel. Uncertainty traps. Technical Report, National Bureau of Economic Research, 2015.

[FR24] Lasse Fischer and Aaditya Ramdas. Improving the (approximate) sequential probability ratio test by avoiding overshoot. *arXiv preprint arXiv:2410.16076*, 2024.

[FPS77] Michael A Fligner, George E Policello, and Jagbir Singh. A comparison of two randomized response survey methods with consideration for the level of respondent protection. *Communications in Statistics-Theory and Methods*, 6(15):1511–1524, 1977.

[Fri56] M. Friedman. *A Theory of the Consumption Function*. Princeton University Press, 1956.

[Fri53] Milton Friedman. *Essays in positive economics*. University of Chicago press, 1953.

[FF98] Milton Friedman and Rose D Friedman. *Two Lucky People*. University of Chicago Press, 1998.

[Fri33] Ragar Frisch. Propagation problems and impulse problems in dynamic economics. In *Economic Essays in Honour of Gustav Cassel*, pages 171–205. Allen and Unwin, 1933.

[Gab16] Xavier Gabaix. Power laws in economics: an introduction. *Journal of Economic Perspectives*, 30(1):185–206, 2016.

[Gal89] David Gale. *The theory of linear economic models*. University of Chicago press, 1989.

[Gal16] Alfred Galichon. *Optimal Transport Methods in Economics*. Princeton University Press, Princeton, New Jersey, 2016.

[Gib31] Robert Gibrat. *Les inégalités économiques: Applications d'une loi nouvelle, la loi de l'effet proportionnel*. PhD thesis, Recueil Sirey, 1931.

[Gor95] Geoffrey J Gordon. Stable function approximation in dynamic programming. In *Machine Learning Proceedings 1995*, pages 261–268. Elsevier, 1995.

[GAESH69] Bernard G Greenberg, Abdel-Latif A Abul-Ela, Walt R Simmons, and Daniel G Horvitz. The unrelated question randomized response model: theoretical framework. *Journal of the American Statistical Association*, 64(326):520–539, 1969.

[GKAH77] Bernard G Greenberg, Roy R Kuebler, James R Abernathy, and Daniel G Horvitz. Respondent hazards in the unrelated question randomized response model. *Journal of Statistical Planning and Inference*, 1(1):53–60, 1977.

[GS93] Moses A Greenfield and Thomas J Sargent. A probabilistic analysis of a catastrophic transuranic waste hoise accident at the wipp. Environmental Evaluation Group, Albuquerque, New Mexico, June 1993. URL: http://www.tomsargent.com/research/EEG-53.pdf.

[Gro73] T. Groves. Incentives in teams. *Econometrica*, 41:617–631, 1973.

[Hal87] Bronwyn H Hall. The relationship between firm size and firm growth in the us manufacturing sector. *The Journal of Industrial Economics*, pages 583–606, 1987.

[Hal71] Robert E Hall. The dynamic effects of fiscal policy in an economy with foresight. *The Review of Economic Studies*, 38(2):229–244, 1971.

[Hal78] Robert E Hall. Stochastic Implications of the Life Cycle-Permanent Income Hypothesis: Theory and Evidence. *Journal of Political Economy*, 86(6):971–987, 1978.

[HM82] Robert E Hall and Frederic S Mishkin. The Sensitivity of Consumption to Transitory Income: Estimates from Panel Data on Households. *National Bureau of Economic Research Working Paper Series*, 1982.

[HTW67] Michael J Hamburger, Gerald L Thompson, and Roman L Weil. Computation of expansion rates for the generalized von neumann model of an expanding economy. *Econometrica, Journal of the Econometric Society*, pages 542–547, 1967.

[Ham05] James D Hamilton. What's real about the business cycle? *Federal Reserve Bank of St. Louis Review*, pages 435–452, 2005.

[Han93] Gary D Hansen. The cyclical and secular behaviour of the labour input: comparing efficiency units and hours worked. *Journal of Applied Econometrics*, 8(1):71–80, 1993.

[HS08] L P Hansen and T J Sargent. *Robustness*. Princeton University Press, 2008.

[HS13] L P Hansen and T J Sargent. *Recursive Models of Dynamic Linear Economies*. The Gorman Lectures in Economics. Princeton University Press, 2013.

[HJ91] Lars Peter Hansen and Ravi Jagannathan. Implications of security market data for models of dynamic economies. *Journal of Political Economy*, 99(2):225–262, 1991. doi:10.1086/261750.

[HR87] Lars Peter Hansen and Scott F Richard. The role of conditioning information in deducing testable restrictions implied by dynamc asset pricing models. *Econometrica*, 55(3):587–613, May 1987.

[HS95] Lars Peter Hansen and Thomas J Sargent. Discounted linear exponential quadratic gaussian control. *IEEE Transactions on Automatic control*, 40(5):968–971, 1995.

[HS01] Lars Peter Hansen and Thomas J. Sargent. Robust control and model uncertainty. *American Economic Review*, 91(2):60–66, 2001.

[HS82] Lars Peter Hansen and Kenneth J Singleton. Generalized instrumental variables estimation of nonlinear rational expectations models. *Econometrica: Journal of the Econometric Society*, pages 1269–1286, 1982.

[HS83] Lars Peter Hansen and Kenneth J Singleton. Stochastic consumption, risk aversion, and the temporal behavior of asset returns. *Journal of political economy*, 91(2):249–265, 1983.

[HK78] J. Michael Harrison and David M. Kreps. Speculative investor behavior in a stock market with heterogeneous expectations. *The Quarterly Journal of Economics*, 92(2):323–336, 1978.

[HK79] J. Michael Harrison and David M. Kreps. Martingales and arbitrage in multiperiod securities markets. *Journal of Economic Theory*, 20(3):381–408, June 1979.

[Har67] John C. Harsanyi. Games with incomplete information played by “Bayesian” players, I–III part I. The basic model. *Management Science*, 14(3):159–182, 1967.

[Har68a] John C. Harsanyi. Games with incomplete information played by “Bayesian” players, I–III part III. The basic probability distribution of the game. *Management Science*, 14(7):486–502, 1968.

[Har68b] John C. Harsanyi. Games with incomplete information played by “Bayesian” players, I–III part II. Bayesian equilibrium points. *Management Science*, 14(5):320–334, 1968.

[HL96] John Heaton and Deborah J Lucas. Evaluating the effects of incomplete markets on risk sharing and asset pricing. *Journal of Political Economy*, pages 443–487, 1996.

[HLL96] O Hernandez-Lerma and J B Lasserre. *Discrete-Time Markov Control Processes: Basic Optimality Criteria*. Number Vol 1 in Applications of Mathematics Stochastic Modelling and Applied Probability. Springer, 1996.

[Hic37] John R Hicks. Mr. keynes and the" classics"; a suggested interpretation. *Econometrica*, pages 147–159, 1937.

[HMMS60] Charles Holt, Franco Modigliani, John F. Muth, and Herbert Simon. *Planning Production, Inventories, and Work Force*. Prentice-Hall International Series in Management, New Jersey, 1960.

[Hop92] Hugo A Hopenhayn. Entry, exit, and firm dynamics in long run equilibrium. *Econometrica: Journal of the Econometric Society*, pages 1127–1150, 1992.

[HIS97] He Huang, Selahattin Imrohoroglu, and Thomas J Sargent. Two computations to fund social security. *Macroeconomic Dynamics*, 1(1):7–44, 1997.

[Hug93] Mark Huggett. The risk-free rate in heterogeneous-agent incomplete-insurance economies. *Journal of Economic Dynamics and Control*, 17(5-6):953–969, 1993.

[Hur50] Leonid Hurwicz. Least squares bias in time series. *Statistical inference in dynamic economic models*, 10:365–383, 1950.

[Haggstrom02] Olle Häggström. *Finite Markov chains and algorithmic applications*. Volume 52. Cambridge University Press, 2002.

[Int02] Michael D Intriligator. *Mathematical optimization and economic theory*. SIAM, 2002.

[Janich94] K Jänich. *Linear Algebra*. Springer Undergraduate Texts in Mathematics and Technology. Springer, 1994.

[Jac73] David Jacobson. Optimal stochastic linear systems with exponential performance criteria and their relation to deterministic differential games. *IEEE Transactions on Automatic control*, 18(2):124–131, 1973.

[Jef46] Harold Jeffreys. An invariant form for the prior probability in estimation problems. *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences*, 186(1007):453–461, 1946.

[JYC88] Robert J. Shiller John Y. Campbell. The Dividend-Price Ratio and Expectations of Future Dividends and Discount Factors. *Review of Financial Studies*, 1(3):195–228, 1988.

[Jov79] Boyan Jovanovic. Firm-specific capital and turnover. *Journal of Political Economy*, 87(6):1246–1260, 1979.

[Jud90] K L Judd. Cournot versus bertrand: a dynamic resolution. Technical Report, Hoover Institution, Stanford University, 1990.

[KMT56] John G Kemeny, Oskar Morgenstern, and Gerald L Thompson. A generalization of the von neumann model of an expanding economy. *Econometrica, Journal of the Econometric Society*, pages 115–135, 1956.

[Koo65] Tjalling C. Koopmans. On the concept of optimal economic growth. In Tjalling C. Koopmans, editor, *The Economic Approach to Development Planning*, pages 225–287. Chicago, 1965.

[Kre88] David M. Kreps. *Notes on the Theory of Choice*. Westview Press, Boulder, Colorado, 1988.

[Kuh13] Moritz Kuhn. Recursive Equilibria In An Aiyagari-Style Economy With Permanent Income Shocks. *International Economic Review*, 54:807–835, 2013.

[KL51] Solomon Kullback and Richard A Leibler. On information and sufficiency. *The Annals of Mathematical Statistics*, 22(1):79–86, 1951.

[KBBWP16] J. N. Kutz, S. L. Brunton, Brunton B. W, and J. L. Proctor. *Dynamic mode decomposition: data-driven modeling of complex systems*. SIAM, 2016.

[Lan75] Jan Lanke. On the choice of the unrelated question in simmons' version of randomized response. *Journal of the American Statistical Association*, 70(349):80–83, 1975.

[Lan76] Jan Lanke. On the degree of protection in randomized interviews. *International Statistical Review/Revue Internationale de Statistique*, pages 197–203, 1976.

[LL01] Martin Lettau and Sydney Ludvigson. Consumption, Aggregate Wealth, and Expected Stock Returns. *Journal of Finance*, 56(3):815–849, 06 2001.

[LL04] Martin Lettau and Sydney C. Ludvigson. Understanding Trend and Cycle in Asset Values: Reevaluating the Wealth Effect on Consumption. *American Economic Review*, 94(1):276–299, March 2004.

[LM80] David Levhari and Leonard J Mirman. The great fish war: an example using a dynamic cournot-nash solution. *The Bell Journal of Economics*, pages 322–334, 1980.

[LW76] Frederick W Leysieffer and Stanley L Warner. Respondent jeopardy and optimal designs in randomized response models. *Journal of the American Statistical Association*, 71(355):649–656, 1976.

[LS18] L Ljungqvist and T J Sargent. *Recursive Macroeconomic Theory*. MIT Press, 4 edition, 2018.

[Lju93] Lars Ljungqvist. A unified approach to measures of privacy in randomized response models: a utilitarian perspective. *Journal of the American Statistical Association*, 88(421):97–103, 1993.

[Luc78] Robert E Lucas, Jr. Asset prices in an exchange economy. *Econometrica*, 46(6):1429–1445, 1978.

[Luc03] Robert E Lucas, Jr. Macroeconomic Priorities. *American Economic Review*, 93(1):1–14, March 2003. URL: https://www.aeaweb.org/articles?id=10.1257/000282803321455133.

[LP71] Robert E Lucas, Jr. and Edward C Prescott. Investment under uncertainty. *Econometrica: Journal of the Econometric Society*, pages 659–681, 1971.

[MST20] Qingyin Ma, John Stachurski, and Alexis Akira Toda. The income fluctuation problem and the evolution of wealth. *Journal of Economic Theory*, 187:105003, 2020.

[MaC82] Thomas E. MaCurdy. The use of time series processes to model the error structure of earnings in a longitudinal data analysis. *Journal of Econometrics*, 18(1):83–114, 1982.

[MS89] Albert Marcet and Thomas J Sargent. Convergence of Least-Squares Learning in Environments with Hidden State Variables and Private Information. *Journal of Political Economy*, 97(6):1306–1322, 1989.

[MCWG95] A Mas-Colell, M D Whinston, and J R Green. *Microeconomic Theory*. Volume 1. Oxford University Press, 1995.

[McC70] J J McCall. Economics of Information and Job Search. *The Quarterly Journal of Economics*, 84(1):113–126, 1970.

[MP04] Costas Meghir and Luigi Pistaferri. Income variance dynamics and heterogeneity. *Econometrica*, 72(1):1–32, 2004.

[MP85] Rajnish Mehra and Edward C Prescott. The equity premium: A puzzle. *Journal of Monetary Economics*, 15(2):145–161, 1985.

[MT98] Enrique G Mendoza and Linda L Tesar. The international ramifications of tax reforms: supply-side economics in a global economy. *American Economic Review*, pages 226–245, 1998.

[Mil77] Edward M Miller. Risk, uncertainty, and divergence of opinion. *The Journal of finance*, 32(4):1151–1168, 1977.

[MB54] F. Modigliani and R. Brumberg. Utility analysis and the consumption function: An interpretation of cross-section data. In K.K Kurihara, editor, *Post-Keynesian Economics*. 1954.

[Mor96] Stephen Morris. Speculative investor behavior and learning. *The Quarterly Journal of Economics*, 111(4):1111–1133, 1996.

[Mut60] John F Muth. Optimal properties of exponentially weighted forecasts. *Journal of the american statistical association*, 55(290):299–306, 1960.

[Nea99] Derek Neal. The Complexity of Job Mobility among Young Men. *Journal of Labor Economics*, 17(2):237–261, 1999.

[NP33] J. Neyman and E. S Pearson. On the problem of the most efficient tests of statistical hypotheses. *Phil. Trans. R. Soc. Lond. A. 231 (694–706)*, pages 289–337, 1933.

[OW69] Guy H. Orcutt and Herbert S. Winokur. First order autoregression: inference, estimation, and prediction. *Econometrica*, 37(1):1–14, 1969.

[Par99] Jonathan A Parker. The Reaction of Household Consumption to Predictable Changes in Social Security Taxes. *American Economic Review*, 89(4):959–973, 1999.

[PalS13] Jenő Pál and John Stachurski. Fitted value function iteration with probability one contractions. *Journal of Economic Dynamics and Control*, 37(1):251–264, 2013.

[Rab02] Guillaume Rabault. When do borrowing constraints bind? Some new results on the income fluctuation problem. *Journal of Economic Dynamics and Control*, 26(2):217–245, 2002.

[Ref96] Kevin L Reffett. Production-based asset pricing in monetary economies with transactions costs. *Economica*, pages 427–443, 1996.

[Rei09] Michael Reiter. Solving heterogeneous-agent models by projection and perturbation. *Journal of Economic Dynamics and Control*, 33(3):649–665, 2009.

[Rya12] Stephen P Ryan. The costs of environmental regulation in a concentrated industry. *Econometrica*, 80(3):1019–1061, 2012.

[Sam39] Paul A. Samuelson. Interactions between the multiplier analysis and the principle of acceleration. *Review of Economic Studies*, 21(2):75–78, 1939.

[Sar77] Thomas J Sargent. The Demand for Money During Hyperinflations under Rational Expectations: I. *International Economic Review*, 18(1):59–82, February 1977.

[Sar87] Thomas J Sargent. *Macroeconomic Theory*. Academic Press, New York, 2nd edition, 1987.

[Sar89] Thomas J Sargent. Two models of measurements and the investment accelerator. *Journal of Political Economy*, 97(2):251–287, 1989.

[SS25] Thomas J Sargent and John Stachurski. *Dynamic Programming: Finite States*. Cambridge University Press, 2025.

[SE77] Jack Schechtman and Vera L S Escudero. Some results on an income fluctuation problem. *Journal of Economic Theory*, 16(2):151–166, 1977.

[Sch14] Jose A. Scheinkman. *Speculation, Trading, and Bubbles*. Columbia University Press, New York, 2014.

[Sch10] Peter J Schmid. Dynamic mode decomposition of numerical and experimental data. *Journal of fluid mechanics*, 656:5–28, 2010.

[Sha48] Claude E Shannon. A mathematical theory of communication. *The Bell system technical journal*, 27(3):379–423, 1948.

[Slu27] Eugen Slutsky. The summation of random causes as the source of cyclic processes. In *Problems of Economic Conditions*, volume 3. The Conjuncture Institute, Moscow, 1927.

[Sta08] John Stachurski. Continuous state dynamic programming via nonexpansive approximation. *Computational Economics*, 31(2):141–160, 2008.

[ST19] John Stachurski and Alexis Akira Toda. An impossibility theorem for wealth in heterogeneous-agent models with limited heterogeneity. *Journal of Economic Theory*, 182:1–24, 2019.

[STY04] Kjetil Storesletten, Christopher I Telmer, and Amir Yaron. Consumption and risk sharing over the life cycle. *Journal of Monetary Economics*, 51(3):609–633, 2004.

[Sun96] R K Sundaram. *A First Course in Optimization Theory*. Cambridge University Press, 1996.

[SB18] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[Tal00] Thomas D Tallarini. Risk-sensitive real business cycles. *Journal of Monetary Economics*, 45(3):507–532, June 2000.

[Tau86] George Tauchen. Finite state markov-chain approximations to univariate and vector autoregressions. *Economics Letters*, 20(2):177–181, 1986.

[Tob92] James Tobin. An old keynesian counterattacks. *Eastern Economic Journal*, 18(4):387–400, 1992.

[Tre16] Daniel Treisman. Russia's billionaires. *The American Economic Review*, 106(5):236–241, 2016.

[TRL+14] J. H. Tu, C. W. Rowley, D. M. Luchtenburg, S. L. Brunton, and J. N. Kutz. On dynamic mode decomposition: theory and applications. *Journal of Computational Dynamics*, 1(2):391–421, 2014.

[VL11] Ngo Van Long. Dynamic games in the economics of natural resources: a survey. *Dynamic Games and Applications*, 1(1):115–148, 2011.

[Vic61] W. Vickrey. Counterspeculation, auctions, and competitive sealed tenders. *Journal of Finance*, 16:8–37, 1961.

[vN28] John von Neumann. Zur theorie der gesellschaftsspiele. *Mathematische annalen*, 100(1):295–320, 1928.

[vN37] John von Neumann. Uber ein okonomsiches gleichungssystem und eine verallgemeinering des browerschen fixpunktsatzes. In *Erge. Math. Kolloq.*, volume 8, 73–83. 1937.

[Wal47] Abraham Wald. *Sequential Analysis*. John Wiley and Sons, New York, 1947.

[Wal80] W Allen Wallis. The statistical research group, 1942–1945. *Journal of the American Statistical Association*, 75(370):320–330, 1980.

[War65] Stanley L Warner. Randomized response: a survey technique for eliminating evasive answer bias. *Journal of the American Statistical Association*, 60(309):63–69, 1965.

[Wec79] William E Wecker. Predicting the turning points of a time series. *Journal of business*, pages 35–50, 1979.

[Wei89] Philippe Weil. The equity premium puzzle and the risk-free rate puzzle. *Journal of Monetary Economics*, 24(3):401–421, 1989. doi:10.1016/0304-3932(89)90028-7.

[Whi83] Charles Whiteman. *Linear Rational Expectations Models: A User's Guide*. University of Minnesota Press, Minneapolis, Minnesota, 1983.

[Whi81] Peter Whittle. Risk-sensitive linear/quadratic/gaussian control. *Advances in Applied Probability*, 13(4):764–777, 1981.

[Whi90] Peter Whittle. *Risk-Sensitive Optimal Control*. Wiley, New York, 1990.

[Woo15] Jeffrey M Wooldridge. *Introductory econometrics: A modern approach*. Nelson Education, 2015.

[YS05] G Alastair Young and Richard L Smith. *Essentials of statistical inference*. Cambridge University Press, 2005.

PROOF INDEX

INDEX

V