

Quantitative Economics with Python

THOMAS J. SARGENT AND JOHN STACHURSKI

March 9, 2021

Contents

I Tools and Techniques	1
1 Geometric Series for Elementary Economics	3
2 Multivariate Hypergeometric Distribution	23
3 Modeling COVID 19	33
4 Linear Algebra	45
5 Complex Numbers and Trigonometry	69
6 LLN and CLT	79
7 Heavy-Tailed Distributions	97
8 Multivariate Normal Distribution	115
9 Univariate Time Series with Matrix Algebra	151
II Introduction to Dynamics	163
10 Dynamics in One Dimension	165
11 AR1 Processes	185
12 Finite Markov Chains	199
13 Inventory Dynamics	225
14 Linear State Space Models	235
15 Application: The Samuelson Multiplier-Accelerator	259
16 Kesten Processes and Firm Dynamics	293

17 Wealth Distribution Dynamics	307
18 A First Look at the Kalman Filter	323
19 Shortest Paths	341
20 Cass-Koopmans Planning Problem	351
21 Cass-Koopmans Competitive Equilibrium	367
III Search	383
22 Job Search I: The McCall Search Model	385
23 Job Search II: Search and Separation	403
24 Job Search III: Fitted Value Function Iteration	417
25 Job Search IV: Correlated Wage Offers	427
26 Job Search V: Modeling Career Choice	437
27 Job Search VI: On-the-Job Search	451
IV Consumption, Savings and Growth	463
28 Cake Eating I: Introduction to Optimal Saving	465
29 Cake Eating II: Numerical Methods	477
30 Optimal Growth I: The Stochastic Optimal Growth Model	493
31 Optimal Growth II: Accelerating the Code with Numba	511
32 Optimal Growth III: Time Iteration	523
33 Optimal Growth IV: The Endogenous Grid Method	535
34 The Income Fluctuation Problem I: Basic Model	543
35 The Income Fluctuation Problem II: Stochastic Returns on Assets	559

CONTENTS	5
V Information	573
36 Job Search VII: Search with Learning	575
37 Likelihood Ratio Processes	603
38 A Problem that Stumped Milton Friedman	621
39 Exchangeability and Bayesian Updating	639
40 Likelihood Ratio Processes and Bayesian Learning	655
41 Bayesian versus Frequentist Decision Rules	663
VI LQ Control	691
42 LQ Control: Foundations	693
43 The Permanent Income Model	723
44 Permanent Income II: LQ Techniques	741
45 Production Smoothing via Inventories	759
VII Multiple Agent Models	779
46 Schelling's Segregation Model	781
47 A Lake Model of Employment and Unemployment	793
48 Rational Expectations Equilibrium	821
49 Stability in Linear Rational Expectations Models	837
50 Markov Perfect Equilibrium	859
51 Uncertainty Traps	877
52 The Aiyagari Model	891
VIII Asset Pricing and Finance	901
53 Asset Pricing: Finite State Models	903

54 Heterogeneous Beliefs and Bubbles	925
IX Data and Empirics	937
55 Pandas for Panel Data	939
56 Linear Regression in Python	961
57 Maximum Likelihood Estimation	979

Part I

Tools and Techniques

Chapter 1

Geometric Series for Elementary Economics

1.1 Contents

- Overview 1.2
- Key Formulas 1.3
- Example: The Money Multiplier in Fractional Reserve Banking 1.4
- Example: The Keynesian Multiplier 1.5
- Example: Interest Rates and Present Values 1.6
- Back to the Keynesian Multiplier 1.7

1.2 Overview

The lecture describes important ideas in economics that use the mathematics of geometric series.

Among these are

- the Keynesian **multiplier**
- the money **multiplier** that prevails in fractional reserve banking systems
- interest rates and present values of streams of payouts from assets

(As we shall see below, the term **multiplier** comes down to meaning **sum of a convergent geometric series**)

These and other applications prove the truth of the wise crack that

“in economics, a little knowledge of geometric series goes a long way “

Below we'll use the following imports:

```
In [1]: import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np
import sympy as sym
from sympy import init_printing
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D
```

1.3 Key Formulas

To start, let c be a real number that lies strictly between -1 and 1 .

- We often write this as $c \in (-1, 1)$.
- Here $(-1, 1)$ denotes the collection of all real numbers that are strictly less than 1 and strictly greater than -1 .
- The symbol \in means *in* or *belongs to the set after the symbol*.

We want to evaluate geometric series of two types – infinite and finite.

1.3.1 Infinite Geometric Series

The first type of geometric that interests us is the infinite series

$$1 + c + c^2 + c^3 + \dots$$

Where \dots means that the series continues without end.

The key formula is

$$1 + c + c^2 + c^3 + \dots = \frac{1}{1 - c} \quad (1)$$

To prove key formula (1), multiply both sides by $(1 - c)$ and verify that if $c \in (-1, 1)$, then the outcome is the equation $1 = 1$.

1.3.2 Finite Geometric Series

The second series that interests us is the finite geometric series

$$1 + c + c^2 + c^3 + \dots + c^T$$

where T is a positive integer.

The key formula here is

$$1 + c + c^2 + c^3 + \dots + c^T = \frac{1 - c^{T+1}}{1 - c}$$

Remark: The above formula works for any value of the scalar c . We don't have to restrict c to be in the set $(-1, 1)$.

We now move on to describe some famous economic applications of geometric series.

1.4 Example: The Money Multiplier in Fractional Reserve Banking

In a fractional reserve banking system, banks hold only a fraction $r \in (0, 1)$ of cash behind each **deposit receipt** that they issue

- In recent times
 - cash consists of pieces of paper issued by the government and called dollars or pounds or ...
 - a *deposit* is a balance in a checking or savings account that entitles the owner to ask the bank for immediate payment in cash
- When the UK and France and the US were on either a gold or silver standard (before 1914, for example)
 - cash was a gold or silver coin
 - a *deposit receipt* was a *bank note* that the bank promised to convert into gold or silver on demand; (sometimes it was also a checking or savings account balance)

Economists and financiers often define the **supply of money** as an economy-wide sum of **cash plus deposits**.

In a **fractional reserve banking system** (one in which the reserve ratio r satisfies $0 < r < 1$), **banks create money** by issuing deposits *backed* by fractional reserves plus loans that they make to their customers.

A geometric series is a key tool for understanding how banks create money (i.e., deposits) in a fractional reserve system.

The geometric series formula (1) is at the heart of the classic model of the money creation process – one that leads us to the celebrated **money multiplier**.

1.4.1 A Simple Model

There is a set of banks named $i = 0, 1, 2, \dots$

Bank i 's loans L_i , deposits D_i , and reserves R_i must satisfy the balance sheet equation (because **balance sheets balance**):

$$L_i + R_i = D_i \quad (2)$$

The left side of the above equation is the sum of the bank's **assets**, namely, the loans L_i it has outstanding plus its reserves of cash R_i .

The right side records bank i 's liabilities, namely, the deposits D_i held by its depositors; these are IOU's from the bank to its depositors in the form of either checking accounts or savings accounts (or before 1914, bank notes issued by a bank stating promises to redeem note for gold or silver on demand).

Each bank i sets its reserves to satisfy the equation

$$R_i = rD_i \quad (3)$$

where $r \in (0, 1)$ is its **reserve-deposit ratio** or **reserve ratio** for short

- the reserve ratio is either set by a government or chosen by banks for precautionary reasons

Next we add a theory stating that bank $i + 1$'s deposits depend entirely on loans made by bank i , namely

$$D_{i+1} = L_i \quad (4)$$

Thus, we can think of the banks as being arranged along a line with loans from bank i being immediately deposited in $i + 1$

- in this way, the debtors to bank i become creditors of bank $i + 1$

Finally, we add an *initial condition* about an exogenous level of bank 0's deposits

D_0 is given exogenously

We can think of D_0 as being the amount of cash that a first depositor put into the first bank in the system, bank number $i = 0$.

Now we do a little algebra.

Combining equations (2) and (3) tells us that

$$L_i = (1 - r)D_i \quad (5)$$

This states that bank i loans a fraction $(1 - r)$ of its deposits and keeps a fraction r as cash reserves.

Combining equation (5) with equation (4) tells us that

$$D_{i+1} = (1 - r)D_i \text{ for } i \geq 0$$

which implies that

$$D_i = (1 - r)^i D_0 \text{ for } i \geq 0 \quad (6)$$

Equation (6) expresses D_i as the i th term in the product of D_0 and the geometric series

$$1, (1 - r), (1 - r)^2, \dots$$

Therefore, the sum of all deposits in our banking system $i = 0, 1, 2, \dots$ is

$$\sum_{i=0}^{\infty} (1 - r)^i D_0 = \frac{D_0}{1 - (1 - r)} = \frac{D_0}{r} \quad (7)$$

1.4.2 Money Multiplier

The **money multiplier** is a number that tells the multiplicative factor by which an exogenous injection of cash into bank 0 leads to an increase in the total deposits in the banking system.

Equation (7) asserts that the **money multiplier** is $\frac{1}{r}$

- An initial deposit of cash of D_0 in bank 0 leads the banking system to create total deposits of $\frac{D_0}{r}$.
- The initial deposit D_0 is held as reserves, distributed throughout the banking system according to $D_0 = \sum_{i=0}^{\infty} R_i$.

1.5 Example: The Keynesian Multiplier

The famous economist John Maynard Keynes and his followers created a simple model intended to determine national income y in circumstances in which

- there are substantial unemployed resources, in particular **excess supply** of labor and capital
- prices and interest rates fail to adjust to make aggregate **supply equal demand** (e.g., prices and interest rates are frozen)
- national income is entirely determined by aggregate demand

1.5.1 Static Version

An elementary Keynesian model of national income determination consists of three equations that describe aggregate demand for y and its components.

The first equation is a national income identity asserting that consumption c plus investment i equals national income y :

$$c + i = y$$

The second equation is a Keynesian consumption function asserting that people consume a fraction $b \in (0, 1)$ of their income:

$$c = by$$

The fraction $b \in (0, 1)$ is called the **marginal propensity to consume**.

The fraction $1 - b \in (0, 1)$ is called the **marginal propensity to save**.

The third equation simply states that investment is exogenous at level i .

- *exogenous* means *determined outside this model*.

Substituting the second equation into the first gives $(1 - b)y = i$.

Solving this equation for y gives

$$y = \frac{1}{1-b}i$$

The quantity $\frac{1}{1-b}$ is called the **investment multiplier** or simply the **multiplier**.

Applying the formula for the sum of an infinite geometric series, we can write the above equation as

$$y = i \sum_{t=0}^{\infty} b^t$$

where t is a nonnegative integer.

So we arrive at the following equivalent expressions for the multiplier:

$$\frac{1}{1-b} = \sum_{t=0}^{\infty} b^t$$

The expression $\sum_{t=0}^{\infty} b^t$ motivates an interpretation of the multiplier as the outcome of a dynamic process that we describe next.

1.5.2 Dynamic Version

We arrive at a dynamic version by interpreting the nonnegative integer t as indexing time and changing our specification of the consumption function to take time into account

- we add a one-period lag in how income affects consumption

We let c_t be consumption at time t and i_t be investment at time t .

We modify our consumption function to assume the form

$$c_t = b y_{t-1}$$

so that b is the marginal propensity to consume (now) out of last period's income.

We begin with an initial condition stating that

$$y_{-1} = 0$$

We also assume that

$$i_t = i \text{ for all } t \geq 0$$

so that investment is constant over time.

It follows that

$$y_0 = i + c_0 = i + b y_{-1} = i$$

and

$$y_1 = c_1 + i = b y_0 + i = (1 + b)i$$

and

$$y_2 = c_2 + i = b y_1 + i = (1 + b + b^2)i$$

and more generally

$$y_t = b y_{t-1} + i = (1 + b + b^2 + \dots + b^t)i$$

or

$$y_t = \frac{1 - b^{t+1}}{1 - b} i$$

Evidently, as $t \rightarrow +\infty$,

$$y_t \rightarrow \frac{1}{1 - b} i$$

Remark 1: The above formula is often applied to assert that an exogenous increase in investment of Δi at time 0 ignites a dynamic process of increases in national income by successive amounts

$$\Delta i, (1 + b)\Delta i, (1 + b + b^2)\Delta i, \dots$$

at times 0, 1, 2,

Remark 2 Let g_t be an exogenous sequence of government expenditures.

If we generalize the model so that the national income identity becomes

$$c_t + i_t + g_t = y_t$$

then a version of the preceding argument shows that the **government expenditures multiplier** is also $\frac{1}{1-b}$, so that a permanent increase in government expenditures ultimately leads to an increase in national income equal to the multiplier times the increase in government expenditures.

1.6 Example: Interest Rates and Present Values

We can apply our formula for geometric series to study how interest rates affect values of streams of dollar payments that extend over time.

We work in discrete time and assume that $t = 0, 1, 2, \dots$ indexes time.

We let $r \in (0, 1)$ be a one-period **net nominal interest rate**

- if the nominal interest rate is 5 percent, then $r = .05$

A one-period **gross nominal interest rate** R is defined as

$$R = 1 + r \in (1, 2)$$

- if $r = .05$, then $R = 1.05$

Remark: The gross nominal interest rate R is an **exchange rate or relative price** of dollars at between times t and $t + 1$. The units of R are dollars at time $t + 1$ per dollar at time t .

When people borrow and lend, they trade dollars now for dollars later or dollars later for dollars now.

The price at which these exchanges occur is the gross nominal interest rate.

- If I sell x dollars to you today, you pay me Rx dollars tomorrow.

- This means that you borrowed x dollars for me at a gross interest rate R and a net interest rate r .

We assume that the net nominal interest rate r is fixed over time, so that R is the gross nominal interest rate at times $t = 0, 1, 2, \dots$

Two important geometric sequences are

$$1, R, R^2, \dots \quad (8)$$

and

$$1, R^{-1}, R^{-2}, \dots \quad (9)$$

Sequence (8) tells us how dollar values of an investment **accumulate** through time.

Sequence (9) tells us how to **discount** future dollars to get their values in terms of today's dollars.

1.6.1 Accumulation

Geometric sequence (8) tells us how one dollar invested and re-invested in a project with gross one period nominal rate of return accumulates

- here we assume that net interest payments are reinvested in the project
- thus, 1 dollar invested at time 0 pays interest r dollars after one period, so we have $r + 1 = R$ dollars at time 1
- at time 1 we reinvest $1 + r = R$ dollars and receive interest of rR dollars at time 2 plus the *principal* R dollars, so we receive $rR + R = (1 + r)R = R^2$ dollars at the end of period 2
- and so on

Evidently, if we invest x dollars at time 0 and reinvest the proceeds, then the sequence

$$x, xR, xR^2, \dots$$

tells how our account accumulates at dates $t = 0, 1, 2, \dots$

1.6.2 Discounting

Geometric sequence (9) tells us how much future dollars are worth in terms of today's dollars.

Remember that the units of R are dollars at $t + 1$ per dollar at t .

It follows that

- the units of R^{-1} are dollars at t per dollar at $t + 1$
- the units of R^{-2} are dollars at t per dollar at $t + 2$
- and so on; the units of R^{-j} are dollars at t per dollar at $t + j$

So if someone has a claim on x dollars at time $t + j$, it is worth xR^{-j} dollars at time t (e.g., today).

1.6.3 Application to Asset Pricing

A **lease** requires a payments stream of x_t dollars at times $t = 0, 1, 2, \dots$ where

$$x_t = G^t x_0$$

where $G = (1 + g)$ and $g \in (0, 1)$.

Thus, lease payments increase at g percent per period.

For a reason soon to be revealed, we assume that $G < R$.

The **present value** of the lease is

$$\begin{aligned} p_0 &= x_0 + x_1/R + x_2/(R^2) + \dots \\ &= x_0(1 + GR^{-1} + G^2 R^{-2} + \dots) \\ &= x_0 \frac{1}{1 - GR^{-1}} \end{aligned}$$

where the last line uses the formula for an infinite geometric series.

Recall that $R = 1 + r$ and $G = 1 + g$ and that $R > G$ and $r > g$ and that r and g are typically small numbers, e.g., .05 or .03.

Use the Taylor series of $\frac{1}{1+r}$ about $r = 0$, namely,

$$\frac{1}{1+r} = 1 - r + r^2 - r^3 + \dots$$

and the fact that r is small to approximate $\frac{1}{1+r} \approx 1 - r$.

Use this approximation to write p_0 as

$$\begin{aligned} p_0 &= x_0 \frac{1}{1 - GR^{-1}} \\ &= x_0 \frac{1}{1 - (1 + g)(1 - r)} \\ &= x_0 \frac{1}{1 - (1 + g - r - rg)} \\ &\approx x_0 \frac{1}{r - g} \end{aligned}$$

where the last step uses the approximation $rg \approx 0$.

The approximation

$$p_0 = \frac{x_0}{r - g}$$

is known as the **Gordon formula** for the present value or current price of an infinite payment stream $x_0 G^t$ when the nominal one-period interest rate is r and when $r > g$.

We can also extend the asset pricing formula so that it applies to finite leases.

Let the payment stream on the lease now be x_t for $t = 1, 2, \dots, T$, where again

$$x_t = G^t x_0$$

The present value of this lease is:

$$\begin{aligned} p_0 &= x_0 + x_1/R + \cdots + x_T/R^T \\ &= x_0(1 + GR^{-1} + \cdots + G^T R^{-T}) \\ &= \frac{x_0(1 - G^{T+1}R^{-(T+1)})}{1 - GR^{-1}} \end{aligned}$$

Applying the Taylor series to $R^{-(T+1)}$ about $r = 0$ we get:

$$\frac{1}{(1+r)^{T+1}} = 1 - r(T+1) + \frac{1}{2}r^2(T+1)(T+2) + \cdots \approx 1 - r(T+1)$$

Similarly, applying the Taylor series to G^{T+1} about $g = 0$:

$$(1+g)^{T+1} = 1 + (T+1)g(1+g)^T + (T+1)Tg^2(1+g)^{T-1} + \cdots \approx 1 + (T+1)g$$

Thus, we get the following approximation:

$$p_0 = \frac{x_0(1 - (1 + (T+1)g)(1 - r(T+1)))}{1 - (1 - r)(1 + g)}$$

Expanding:

$$\begin{aligned} p_0 &= \frac{x_0(1 - 1 + (T+1)^2rg - r(T+1) + g(T+1))}{1 - 1 + r - g + rg} \\ &= \frac{x_0(T+1)((T+1)rg + r - g)}{r - g + rg} \\ &\approx \frac{x_0(T+1)(r - g)}{r - g} + \frac{x_0rg(T+1)}{r - g} \\ &= x_0(T+1) + \frac{x_0rg(T+1)}{r - g} \end{aligned}$$

We could have also approximated by removing the second term $rgx_0(T+1)$ when T is relatively small compared to $1/(rg)$ to get $x_0(T+1)$ as in the finite stream approximation.

We will plot the true finite stream present-value and the two approximations, under different values of T , and g and r in Python.

First we plot the true finite stream present-value after computing it below

```
In [2]: # True present value of a finite lease
def finite_lease_pv_true(T, g, r, x_0):
    G = (1 + g)
    R = (1 + r)
    return (x_0 * (1 - G**(T + 1) * R**(-T - 1))) / (1 - G * R**(-1))
# First approximation for our finite lease

def finite_lease_pv_approx_1(T, g, r, x_0):
    p = x_0 * (T + 1) + x_0 * r * g * (T + 1) / (r - g)
```

```

    return p

# Second approximation for our finite lease
def finite_lease_pv_approx_2(T, g, r, x_0):
    return (x_0 * (T + 1))

# Infinite lease
def infinite_lease(g, r, x_0):
    G = (1 + g)
    R = (1 + r)
    return x_0 / (1 - G * R**(-1))

```

Now that we have defined our functions, we can plot some outcomes.

First we study the quality of our approximations

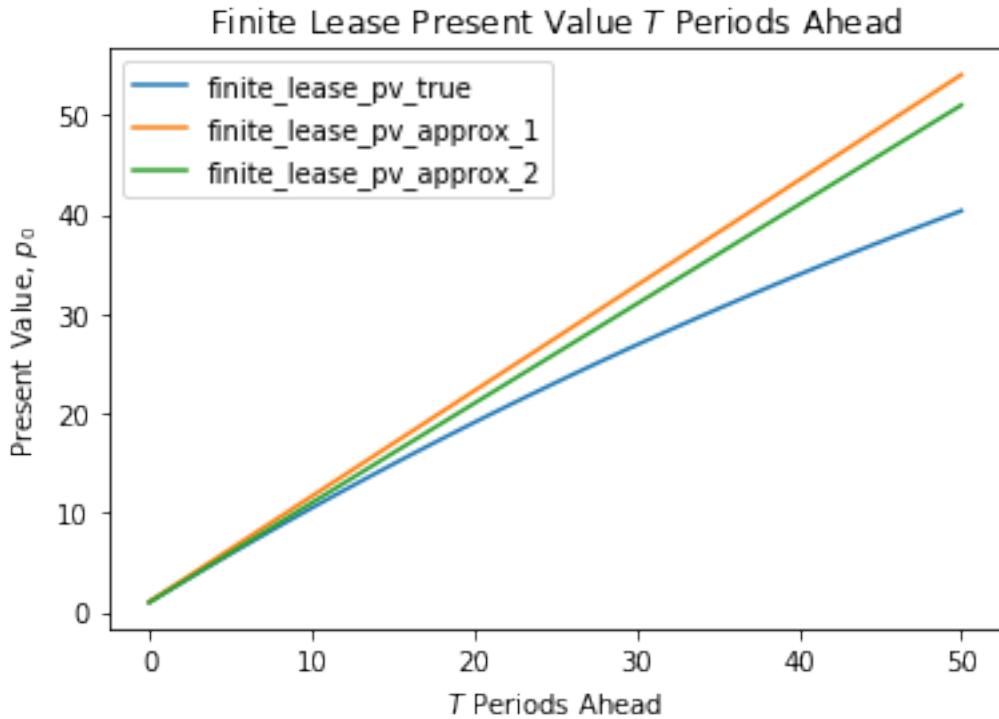
```
In [3]: def plot_function(axes, x_vals, func, args):
    axes.plot(x_vals, func(*args), label=func.__name__)

T_max = 50

T = np.arange(0, T_max+1)
g = 0.02
r = 0.03
x_0 = 1

our_args = (T, g, r, x_0)
funcs = [finite_lease_pv_true,
         finite_lease_pv_approx_1,
         finite_lease_pv_approx_2]
## the three functions we want to compare

fig, ax = plt.subplots()
ax.set_title('Finite Lease Present Value $T$ Periods Ahead')
for f in funcs:
    plot_function(ax, T, f, our_args)
ax.legend()
ax.set_xlabel('$T$ Periods Ahead')
ax.set_ylabel('Present Value, $p_0$')
plt.show()
```

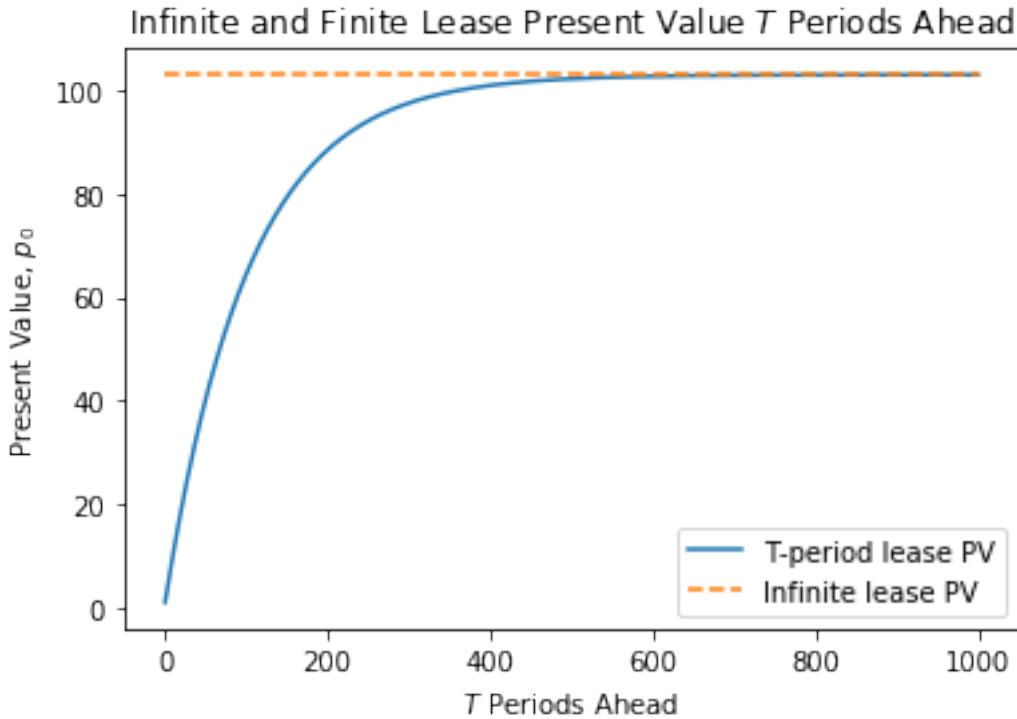


Evidently our approximations perform well for small values of T .

However, holding g and r fixed, our approximations deteriorate as T increases.

Next we compare the infinite and finite duration lease present values over different lease lengths T .

```
In [4]: # Convergence of infinite and finite
T_max = 1000
T = np.arange(0, T_max+1)
fig, ax = plt.subplots()
ax.set_title('Infinite and Finite Lease Present Value $T$ Periods Ahead')
f_1 = finite_lease_pv_true(T, g, r, x_0)
f_2 = np.ones(T_max+1)*infinite_lease(g, r, x_0)
ax.plot(T, f_1, label='T-period lease PV')
ax.plot(T, f_2, '--', label='Infinite lease PV')
ax.set_xlabel('$T$ Periods Ahead')
ax.set_ylabel('Present Value, $p_0$')
ax.legend()
plt.show()
```

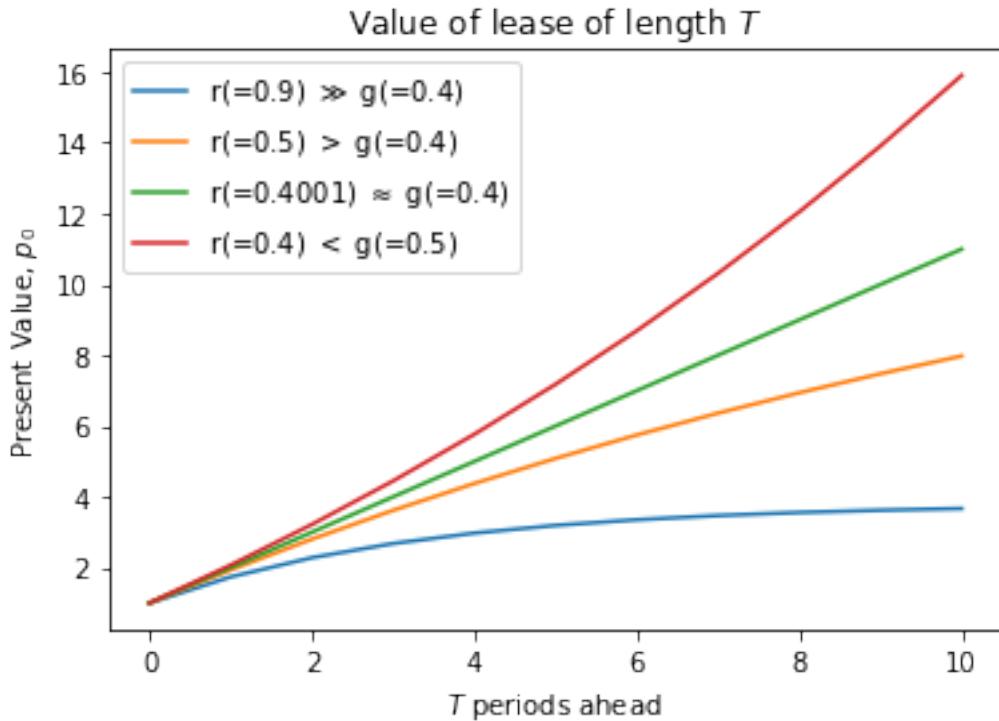


The graph above shows how as duration $T \rightarrow +\infty$, the value of a lease of duration T approaches the value of a perpetual lease.

Now we consider two different views of what happens as r and g covary

```
In [5]: # First view
# Changing r and g
fig, ax = plt.subplots()
ax.set_title('Value of lease of length $T$')
ax.set_ylabel('Present Value, $p_0$')
ax.set_xlabel('$T$ periods ahead')
T_max = 10
T=np.arange(0, T_max+1)

rs, gs = (0.9, 0.5, 0.4001, 0.4), (0.4, 0.4, 0.4, 0.5),
comparisons = ('$\gg$', '$>$', r'$\approx$', '$<$')
for r, g, comp in zip(rs, gs, comparisons):
    ax.plot(finite_lease_pv_true(T, g, r, x_0), label=f'r={r} {comp}\
g={g}' )
ax.legend()
plt.show()
```



This graph gives a big hint for why the condition $r > g$ is necessary if a lease of length $T = +\infty$ is to have finite value.

For fans of 3-d graphs the same point comes through in the following graph.

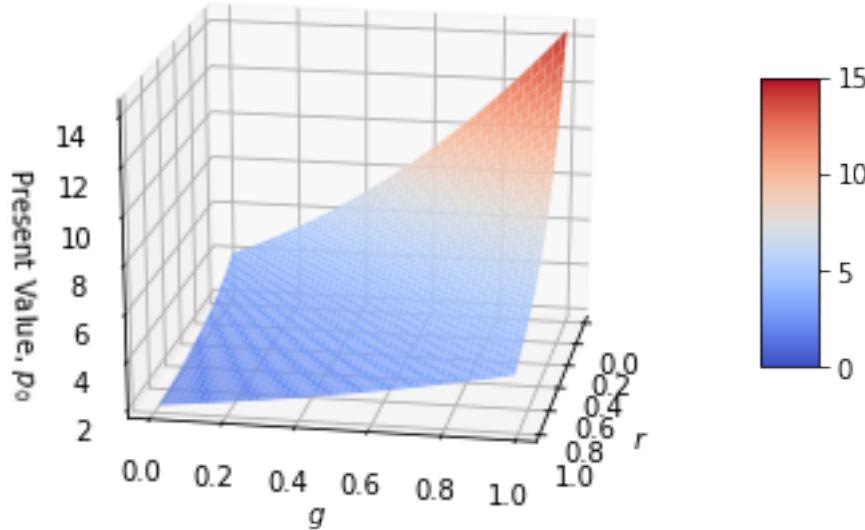
If you aren't enamored of 3-d graphs, feel free to skip the next visualization!

```
In [6]: # Second view
fig = plt.figure()
T = 3
ax = fig.gca(projection='3d')
r = np.arange(0.01, 0.99, 0.005)
g = np.arange(0.011, 0.991, 0.005)

rr, gg = np.meshgrid(r, g)
z = finite_lease_pv_true(T, gg, rr, x_0)

# Removes points where undefined
same = (rr == gg)
z[same] = np.nan
surf = ax.plot_surface(rr, gg, z, cmap=cm.coolwarm,
    antialiased=True, clim=(0, 15))
fig.colorbar(surf, shrink=0.5, aspect=5)
ax.set_xlabel('$r$')
ax.set_ylabel('$g$')
ax.set_zlabel('Present Value, $p_0$')
ax.view_init(20, 10)
ax.set_title('Three Period Lease PV with Varying $g$ and $r$')
plt.show()
```

Three Period Lease PV with Varying g and r



We can use a little calculus to study how the present value p_0 of a lease varies with r and g .

We will use a library called [SymPy](#).

SymPy enables us to do symbolic math calculations including computing derivatives of algebraic equations.

We will illustrate how it works by creating a symbolic expression that represents our present value formula for an infinite lease.

After that, we'll use SymPy to compute derivatives

```
In [7]: # Creates algebraic symbols that can be used in an algebraic expression
g, r, x0 = sym.symbols('g, r, x0')
G = (1 + g)
R = (1 + r)
p0 = x0 / (1 - G * R**(-1))
init_printing()
print('Our formula is:')
p0
```

Our formula is:

```
Out[7]:  $\frac{x_0}{\frac{-g+1}{r+1} + 1}$ 
```

```
In [8]: print('dp0 / dg is:')
dp_dg = sym.diff(p0, g)
dp_dg
```

$dp0 / dg$ is:

Out[8]: $\frac{x_0}{(r+1) \left(-\frac{g+1}{r+1} + 1\right)^2}$

In [9]: `print('dp0 / dr is:')`
`dp_dr = sym.diff(p0, r)`
`dp_dr`

`dp0 / dr is:`

Out[9]: $-\frac{x_0(g+1)}{(r+1)^2 \left(-\frac{g+1}{r+1} + 1\right)^2}$

We can see that for $\frac{\partial p_0}{\partial r} < 0$ as long as $r > g$, $r > 0$ and $g > 0$ and x_0 is positive, so $\frac{\partial p_0}{\partial r}$ will always be negative.

Similarly, $\frac{\partial p_0}{\partial g} > 0$ as long as $r > g$, $r > 0$ and $g > 0$ and x_0 is positive, so $\frac{\partial p_0}{\partial g}$ will always be positive.

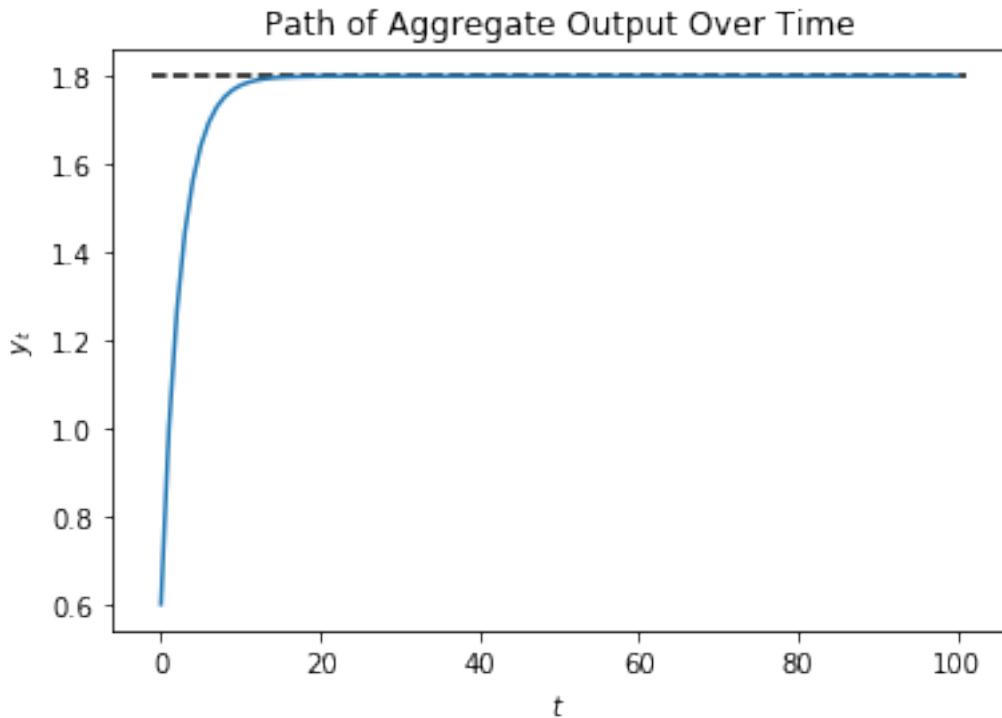
1.7 Back to the Keynesian Multiplier

We will now go back to the case of the Keynesian multiplier and plot the time path of y_t , given that consumption is a constant fraction of national income, and investment is fixed.

```
In [10]: # Function that calculates a path of y
def calculate_y(i, b, g, T, y_init):
    y = np.zeros(T+1)
    y[0] = i + b * y_init + g
    for t in range(1, T+1):
        y[t] = b * y[t-1] + i + g
    return y

# Initial values
i_0 = 0.3
g_0 = 0.3
# 2/3 of income goes towards consumption
b = 2/3
y_init = 0
T = 100

fig, ax = plt.subplots()
ax.set_title('Path of Aggregate Output Over Time')
ax.set_xlabel('$t$')
ax.set_ylabel('$y_t$')
ax.plot(np.arange(0, T+1), calculate_y(i_0, b, g_0, T, y_init))
# Output predicted by geometric series
ax.hlines(i_0 / (1 - b) + g_0 / (1 - b), xmin=-1, xmax=101, linestyles='--')
plt.show()
```



In this model, income grows over time, until it gradually converges to the infinite geometric series sum of income.

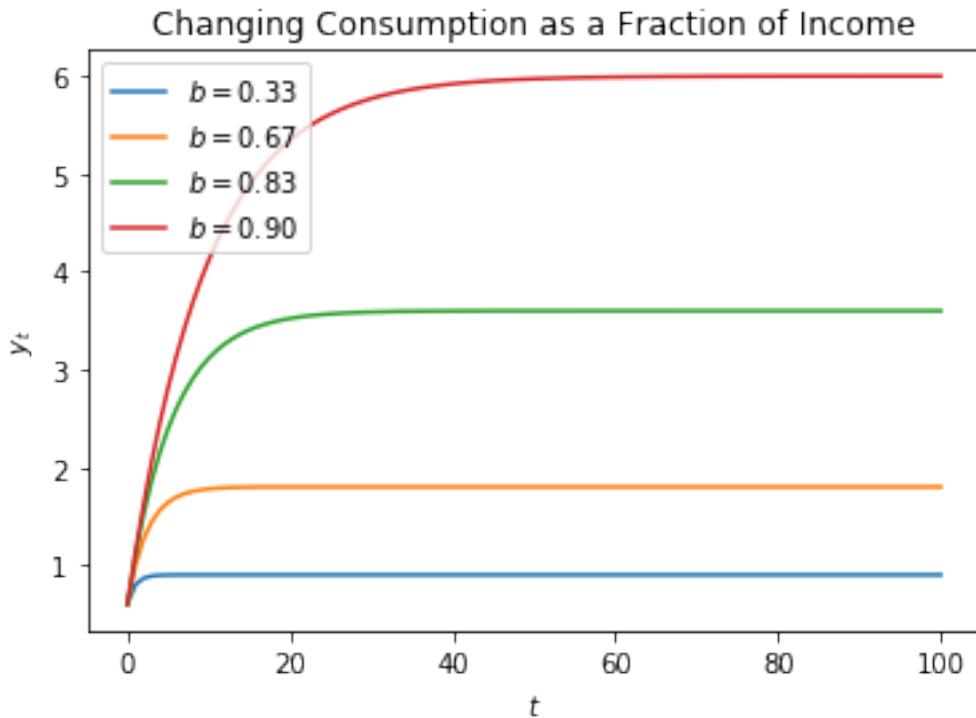
We now examine what will happen if we vary the so-called **marginal propensity to consume**, i.e., the fraction of income that is consumed

In [11]: `bs = (1/3, 2/3, 5/6, 0.9)`

```

fig,ax = plt.subplots()
ax.set_title('Changing Consumption as a Fraction of Income')
ax.set_ylabel('$y_t$')
ax.set_xlabel('$t$')
x = np.arange(0, T+1)
for b in bs:
    y = calculate_y(i_0, b, g_0, T, y_init)
    ax.plot(x, y, label=r'$b=' + f'{b:.2f}' + '$')
ax.legend()
plt.show()

```



Increasing the marginal propensity to consume b increases the path of output over time.

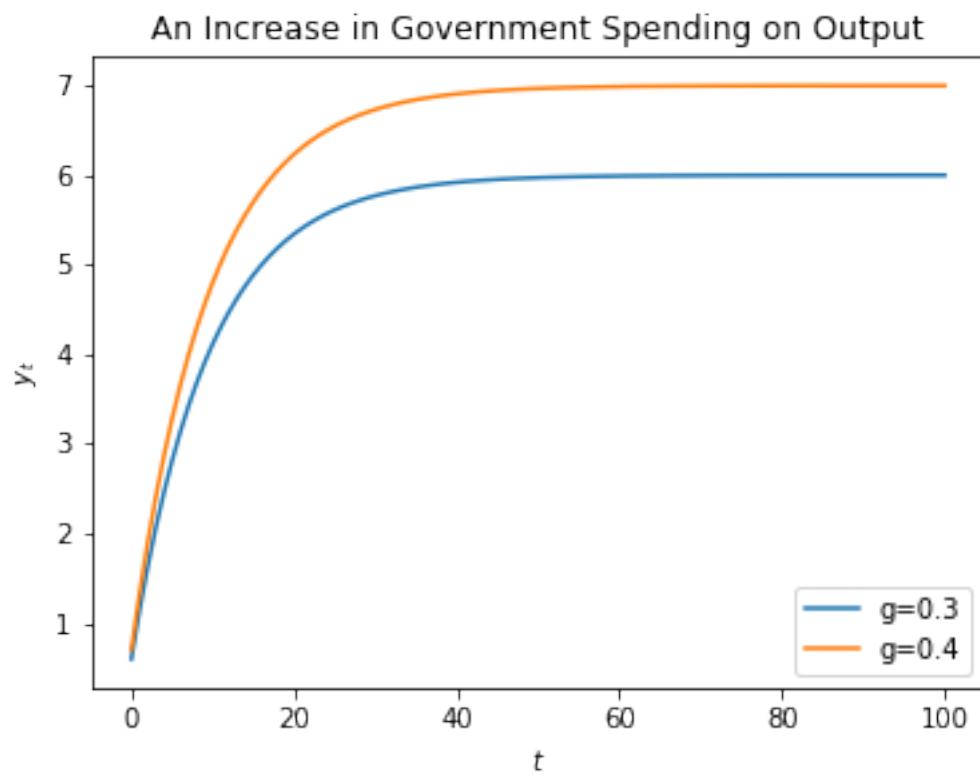
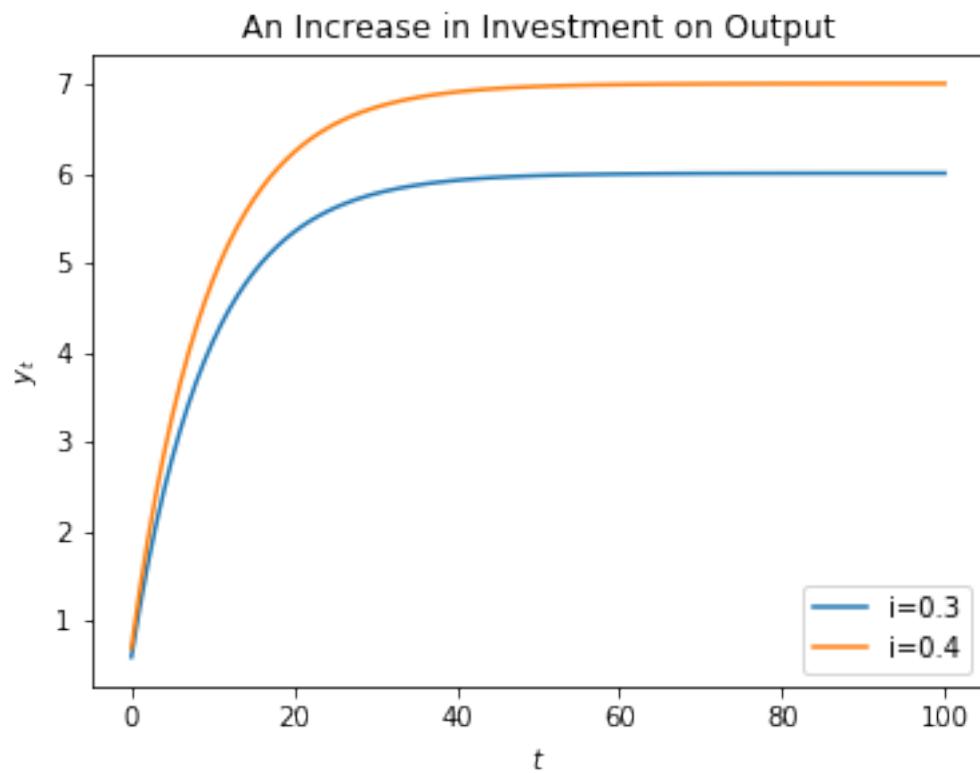
Now we will compare the effects on output of increases in investment and government spending.

```
In [12]: fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(6, 10))
fig.subplots_adjust(hspace=0.3)

x = np.arange(0, T+1)
values = [0.3, 0.4]

for i in values:
    y = calculate_y(i, b, g_0, T, y_init)
    ax1.plot(x, y, label=f"i={i}")
for g in values:
    y = calculate_y(i_0, b, g, T, y_init)
    ax2.plot(x, y, label=f"g={g}")

axes = ax1, ax2
param_labels = "Investment", "Government Spending"
for ax, param in zip(axes, param_labels):
    ax.set_title(f'An Increase in {param} on Output')
    ax.legend(loc = "lower right")
    ax.set_ylabel('$y_t$')
    ax.set_xlabel('$t$')
plt.show()
```



Notice here, whether government spending increases from 0.3 to 0.4 or investment increases from 0.3 to 0.4, the shifts in the graphs are identical.

Chapter 2

Multivariate Hypergeometric Distribution

2.1 Contents

- Overview 2.2
- The Administrator's Problem 2.3
- Usage 2.4

2.2 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

2.3 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, $i = 1, 2, \dots, c$

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.

2.3.1 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

$$X = \begin{pmatrix} k_1 \\ k_2 \\ \vdots \\ k_c \end{pmatrix}.$$

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

$$X = \begin{pmatrix} k_1 \\ k_2 \\ \vdots \\ k_4 \end{pmatrix}.$$

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**.

2.3.2 Multivariate Hypergeometric Distribution

Let's start with some imports.

```
In [1]: import numpy as np
from scipy.special import comb
from scipy.stats import normaltest
from numba import njit, prange
import matplotlib.pyplot as plt
%matplotlib inline
import matplotlib.cm as cm
```

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:

$$\mathbb{E}(X_i) = n \frac{K_i}{N}$$

Variances and covariances:

$$\text{Var}(X_i) = n \frac{N-n}{N-1} \frac{K_i}{N} \left(1 - \frac{K_i}{N}\right)$$

$$\text{Cov}(X_i, X_j) = -n \frac{N-n}{N-1} \frac{K_i}{N} \frac{K_j}{N}$$

To do our work for us, we'll write an `Urn` class.

```
In [2]: 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):
    """
    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
    mu = n * K_arr / N

    # variance-covariance matrix
    Sigma = np.ones((c, c)) * n * (N - n) / (N - 1) / N ** 2
    for i in range(c-1):
        Sigma[i, i] *= K_arr[i] * (N - K_arr[i])
        for j in range(i+1, c):
            Sigma[i, j] *= - K_arr[i] * K_arr[j]
            Sigma[j, i] = Sigma[i, j]

    Sigma[-1, -1] *= K_arr[-1] * (N - K_arr[-1])

```

```

    return mu, Sigma

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
        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

```

2.4 Usage

2.4.1 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

$$P(2 \text{ black}, 2 \text{ white}, 2 \text{ red}) = \frac{\binom{5}{2} \binom{10}{2} \binom{15}{2}}{\binom{30}{6}} = 0.079575596816976$$

```
In [3]: # 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 $X = (2 \ 2 \ 2)$

```
In [4]: k_arr = [2, 2, 2] # array of number of observed successes
urn.pmf(k_arr)
```

```
Out[4]: 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.

```
In [5]: k_arr = [[2, 2, 2], [1, 3, 2]]
urn.pmf(k_arr)
```

```
Out[5]: array([0.0795756, 0.1061008])
```

Now let's compute the mean vector and variance-covariance matrix.

```
In [6]: n = 6
μ, Σ = urn.moments(n)
```

```
In [7]: μ
```

```
Out[7]: array([1., 2., 3.])
```

```
In [8]: Σ
```

```
Out[8]: array([[ 0.68965517, -0.27586207, -0.4137931 ],
   [-0.27586207,  1.10344828, -0.82758621],
   [-0.4137931 , -0.82758621,  1.24137931]])
```

2.4.2 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 $(157, 11, 46, 24)$.

```
In [9]: K_arr = [157, 11, 46, 24]
urn = Urn(K_arr)
```

Let's compute the probability of the outcome $(10, 1, 4, 0)$.

```
In [10]: k_arr = [10, 1, 4, 0]
urn.pmf(k_arr)
```

```
Out[10]: 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.

```
In [11]: k_arr = [[5, 5, 4, 1], [10, 1, 2, 2], [13, 0, 2, 0]]
urn.pmf(k_arr)
```

```
Out[11]: array([6.21412534e-06, 2.70935969e-02, 1.61839976e-02])
```

Now let's compute the mean and variance-covariance matrix of X when $n = 6$.

```
In [12]: n = 6 # number of draws
μ, Σ = urn.moments(n)
```

```
In [13]: # mean
```

 μ

```
Out[13]: array([3.95798319, 0.27731092, 1.15966387, 0.60504202])
```

```
In [14]: # variance-covariance matrix
```

 Σ

```
Out[14]: 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.

```
In [15]: size = 10_000_000
sample = urn.simulate(n, size=size)
```

```
In [16]: # mean
np.mean(sample, 0)
```

```
Out[16]: array([3.9589964, 0.2772218, 1.1591317, 0.6046501])
```

```
In [17]: # variance covariance matrix
np.cov(sample.T)
```

```
Out[17]: array([[ 1.31869504, -0.17902413, -0.7490397 , -0.39063121],
 [-0.17902413,  0.2587115 , -0.05227908, -0.02740829],
 [-0.7490397 , -0.05227908,  0.91559029, -0.11427151],
 [-0.39063121, -0.02740829, -0.11427151,  0.53231101]])
```

Evidently, the sample means and covariances approximate their population counterparts well.

2.4.3 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.

```
In [18]: sample_normal = np.random.multivariate_normal(mu, Sigma, size=size)
```

```
In [19]: def bivariate_normal(x, y, mu, Sigma, i, j):
```

```
    mu_x, mu_y = mu[i], mu[j]
    sigma_x, sigma_y = np.sqrt(Sigma[i, i]), np.sqrt(Sigma[j, j])
    sigma_xy = Sigma[i, j]
```

```

x_mu = x - mu_x
y_mu = y - mu_y

rho = sigma_xy / (sigma_x * sigma_y)
z = x_mu**2 / sigma_x**2 + y_mu**2 / sigma_y**2 - 2 * rho * x_mu * y_mu / (sigma_x * sigma_y)
denom = 2 * np.pi * sigma_x * sigma_y * np.sqrt(1 - rho**2)

return np.exp(-z / (2 * (1 - rho**2))) / denom

```

In [20]:

```

@njit
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

```

In [21]:

```

c = urn.c
fig, axs = plt.subplots(c, c, figsize=(14, 14))

# grids for plotting 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)

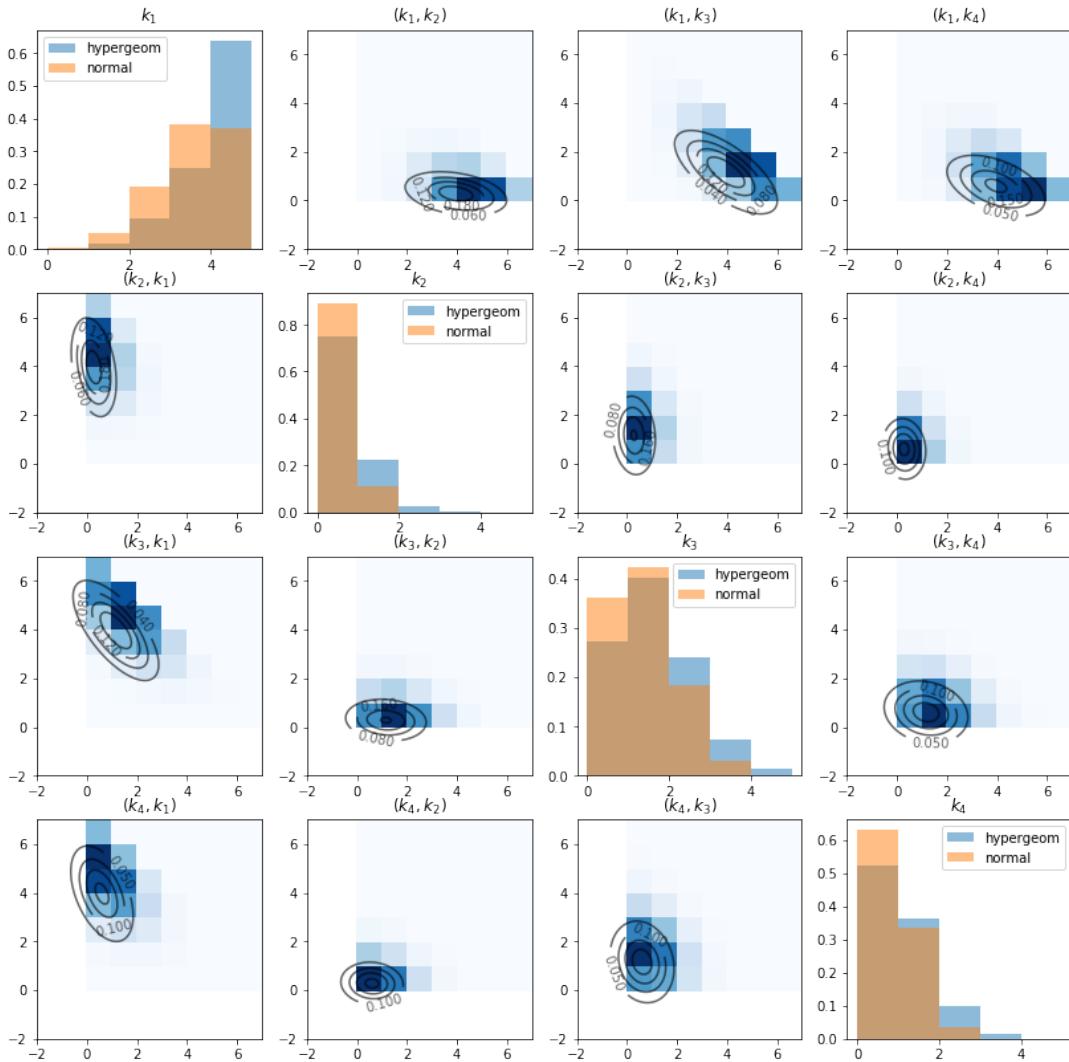
for i in range(c):
    axs[i, i].hist(sample[:, i], bins=np.arange(0, n, 1), alpha=0.5, density=True, label='hypergeom')
    axs[i, i].hist(sample_normal[:, i], bins=np.arange(0, n, 1), alpha=0.5, density=True, label='normal')
    axs[i, i].legend()
    axs[i, i].set_title('$k_{\{ ' + str(i+1) + '\}}$')
    for j in range(c):
        if i == j:
            continue

        # bivariate Gaussian density function
        Z = bivariate_normal(X, Y, mu, Sigma, 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 distribution
        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.

```
In [22]: test_multihyper = normaltest(sample)
```

```
test_multihyper.pvalue
```

```
Out[22]: 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.

```
In [23]: test_normal = normaltest(sample_normal)
test_normal.pvalue
```

```
Out[23]: array([0.26897505, 0.51279047, 0.38868022, 0.33874123])
```

The lesson to take away from this is that the normal approximation is imperfect.

Chapter 3

Modeling COVID 19

3.1 Contents

- Overview 3.2
- The SIR Model 3.3
- Implementation 3.4
- Experiments 3.5
- Ending Lockdown 3.6

3.2 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:

```
In [1]: import numpy as np
         from numpy import exp

         import matplotlib.pyplot as plt
```

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

```
In [2]: from scipy.integrate import odeint
```

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

3.3 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 (R).

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.

3.3.1 Time Path

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

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

$$\begin{aligned}\dot{s}(t) &= -\beta(t) s(t) i(t) \\ \dot{e}(t) &= \beta(t) s(t) i(t) - \sigma e(t) \\ \dot{i}(t) &= \sigma e(t) - \gamma i(t)\end{aligned}\tag{1}$$

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) can be written in vector form as

$$\dot{x} = F(x, t), \quad x := (s, e, i) \quad (2)$$

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

3.3.2 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.)

3.4 Implementation

First we set the population size to match the US.

In [3]: `pop_size = 3.3e8`

Next we fix parameters as described above.

In [4]: `γ = 1 / 18`
`σ = 1 / 5.2`

Now we construct a function that represents F in (2)

```
In [5]: 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 R_0 can be either constant or a given function of time.

The initial conditions are set to

```
In [6]: # initial conditions of s, e, i
i_0 = 1e-7
e_0 = 4 * i_0
s_0 = 1 - i_0 - e_0
```

In vector form the initial condition is

```
In [7]: x_0 = s_0, e_0, i_0
```

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

```
In [8]: 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
```

3.5 Experiments

Let's run some experiments using this code.

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

```
In [9]: t_length = 550
grid_size = 1000
t_vec = np.linspace(0, t_length, grid_size)
```

3.5.1 Experiment 1: Constant R_0 Case

Let's start with the case where R_0 is constant.

We calculate the time path of infected people under different assumptions for R_0 :

```
In [10]: R0_vals = np.linspace(1.6, 3.0, 6)
labels = [f'R0 = {r:.2f}' for r in R0_vals]
i_paths, c_paths = [], []

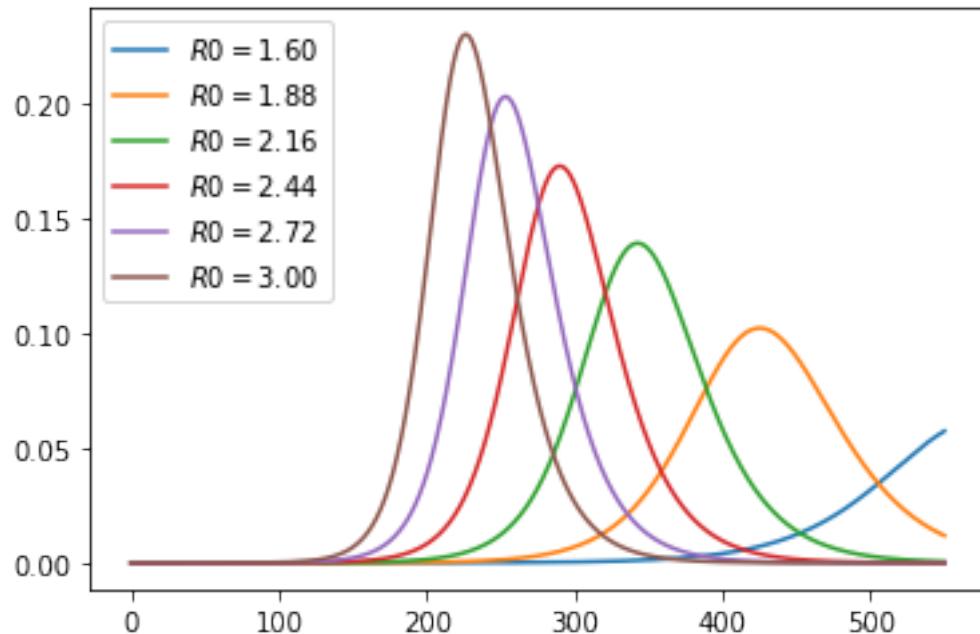
for r in R0_vals:
    i_path, c_path = solve_path(r, t_vec)
    i_paths.append(i_path)
    c_paths.append(c_path)
```

Here's some code to plot the time paths.

```
In [11]: def plot_paths(paths, labels, times=t_vec):
    fig, ax = plt.subplots()
    for path, label in zip(paths, labels):
        ax.plot(times, path, label=label)
    ax.legend(loc='upper left')
    plt.show()
```

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

```
In [12]: plot_paths(i_paths, labels)
```

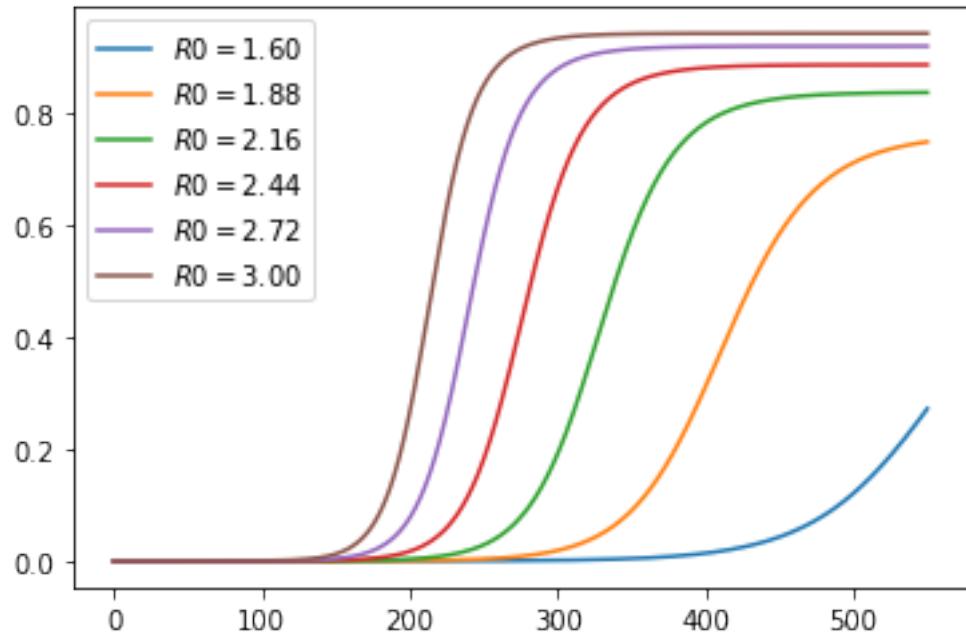


As expected, lower effective transmission rates defer the peak of infections.

They also lead to a lower peak in current cases.

Here are cumulative cases, as a fraction of population:

```
In [13]: plot_paths(c_paths, labels)
```



3.5.2 Experiment 2: Changing Mitigation

Let's look at a scenario where mitigation (e.g., social distancing) is successively imposed. Here's a specification for R_0 as a function of time.

```
In [14]: def R0_mitigating(t, r0=3, eta=1, r_bar=1.6):
    R0 = r0 * exp(-eta * t) + (1 - exp(-eta * t)) * r_bar
    return R0
```

The idea is that R_0 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:

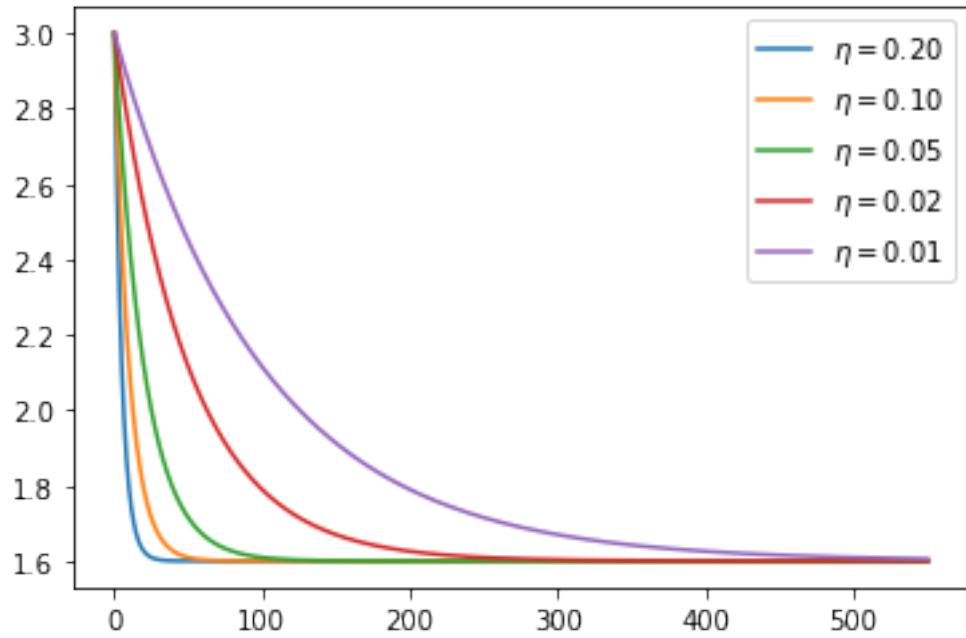
```
In [15]: eta_vals = 1/5, 1/10, 1/20, 1/50, 1/100
labels = [f'r$\eta = {eta:.2f}$' for eta in eta_vals]
```

This is what the time path of R_0 looks like at these alternative rates:

```
In [16]: fig, ax = plt.subplots()

for eta, label in zip(eta_vals, labels):
    ax.plot(t_vec, R0_mitigating(t_vec, eta=eta), label=label)

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

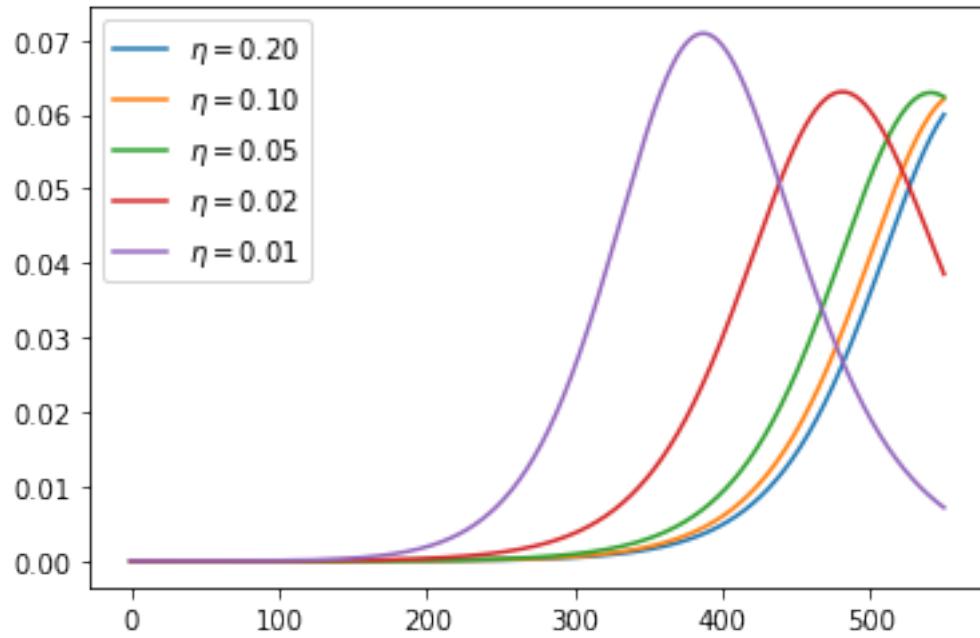


Let's calculate the time path of infected people:

```
In [17]: 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)
```

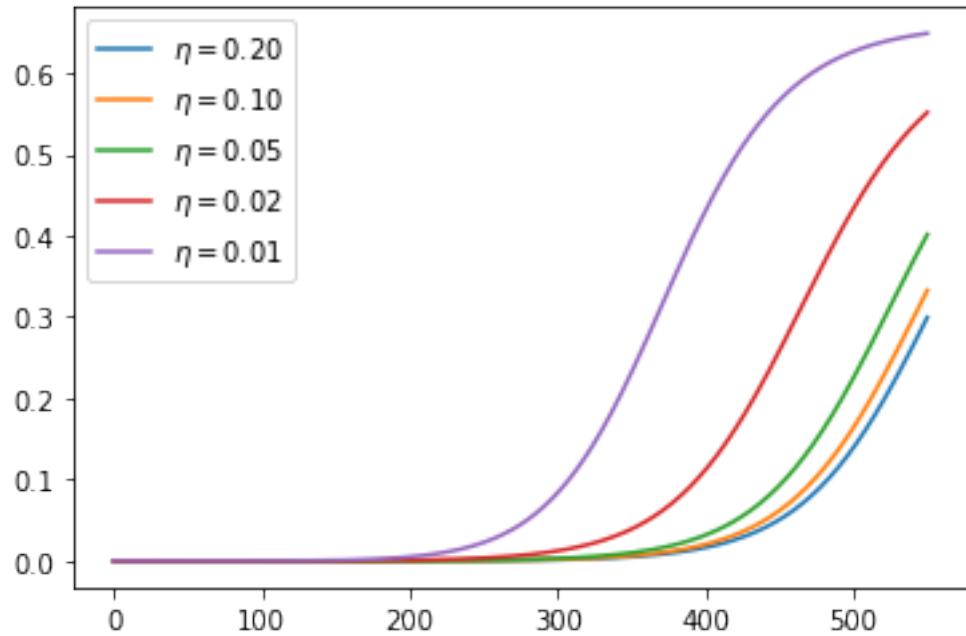
These are current cases under the different scenarios:

```
In [18]: plot_paths(i_paths, labels)
```



Here are cumulative cases, as a fraction of population:

```
In [19]: plot_paths(c_paths, labels)
```



3.6 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.

```
In [20]: # 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:

```
In [21]: R0_paths = (lambda t: 0.5 if t < 30 else 2,
                    lambda t: 0.5 if t < 120 else 2)

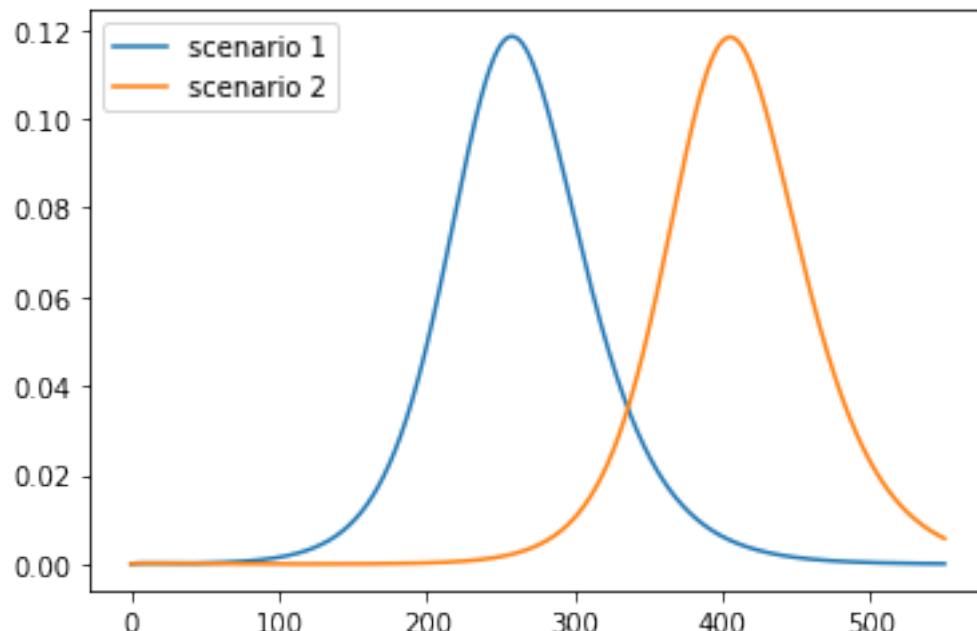
labels = [f'scenario {i}' for i in (1, 2)]

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:

```
In [22]: plot_paths(i_paths, labels)
```



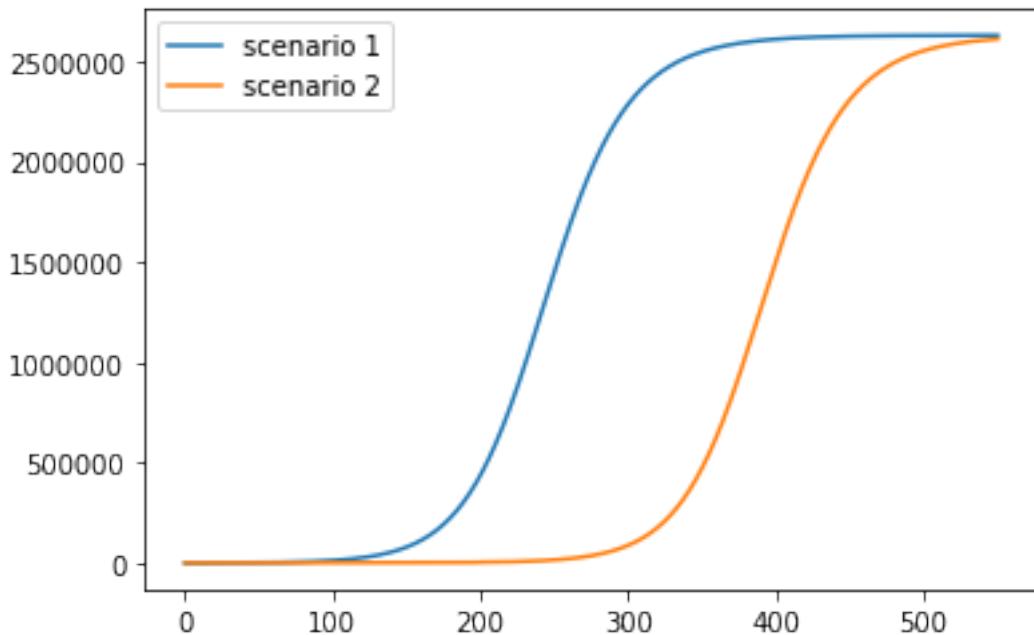
What kind of mortality can we expect under these scenarios?

Suppose that 1% of cases result in death

```
In [23]: v = 0.01
```

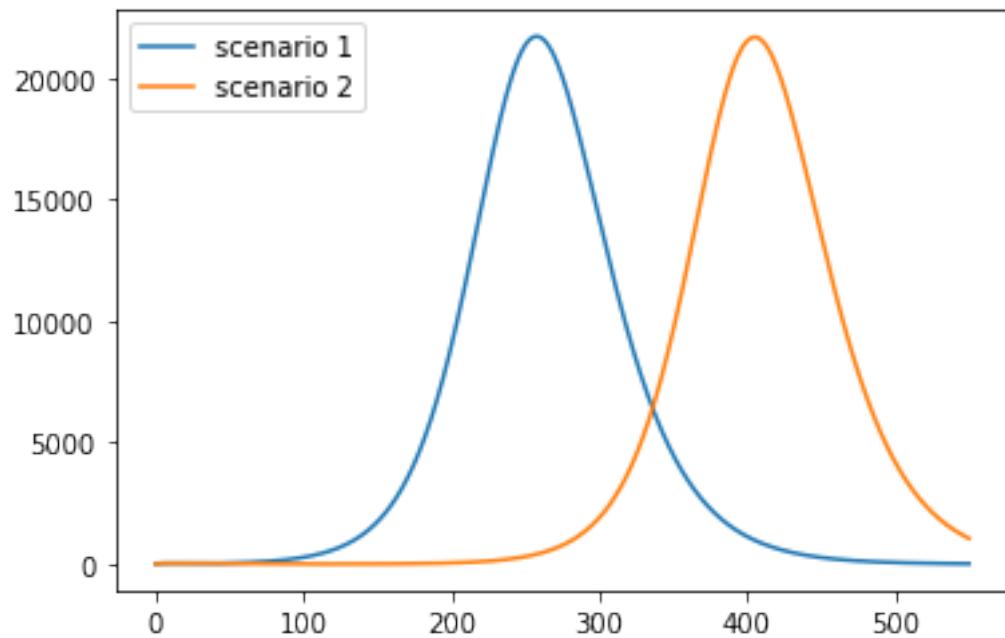
This is the cumulative number of deaths:

```
In [24]: paths = [path * v * pop_size for path in c_paths]
plot_paths(paths, labels)
```



This is the daily death rate:

```
In [25]: paths = [path * v * gamma * 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 4

Linear Algebra

4.1 Contents

- Overview 4.2
- Vectors 4.3
- Matrices 4.4
- Solving Systems of Equations 4.5
- Eigenvalues and Eigenvectors 4.6
- Further Topics 4.7
- Exercises 4.8
- Solutions 4.9

4.2 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

$$\begin{aligned}y_1 &= ax_1 + bx_2 \\y_2 &= cx_1 + dx_2\end{aligned}$$

or, more generally,

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

The objective here is to solve for the “unknowns” x_1, \dots, x_k given a_{11}, \dots, a_{nk} and y_1, \dots, 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:

```
In [1]: import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D
from scipy.interpolate import interp2d
from scipy.linalg import inv, solve, det, eig
```

4.3 Vectors

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

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 .

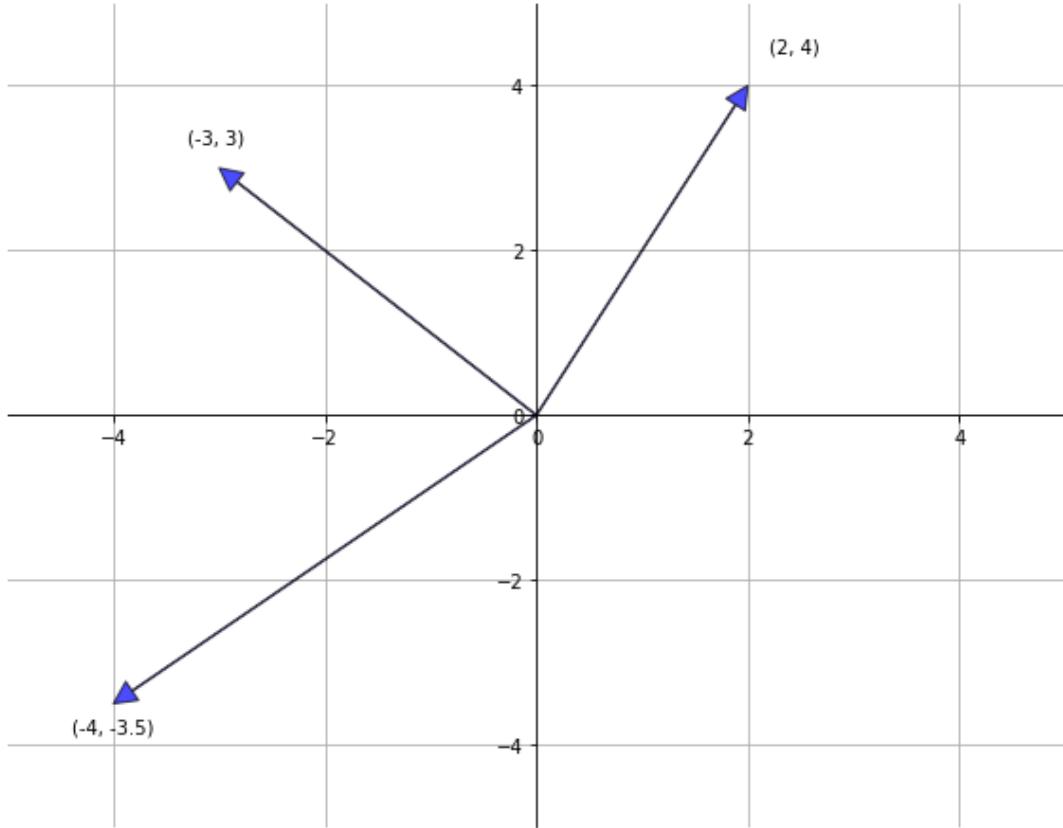
For example, \mathbb{R}^2 is the plane, and a vector in \mathbb{R}^2 is just a point in the plane.

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

The following figure represents three vectors in this manner

```
In [2]: 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()
```



4.3.1 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

```
In [3]: fig, ax = plt.subplots(figsize=(10, 8))
# Set the axes through the origin
for spine in ['left', 'bottom']:
```

```

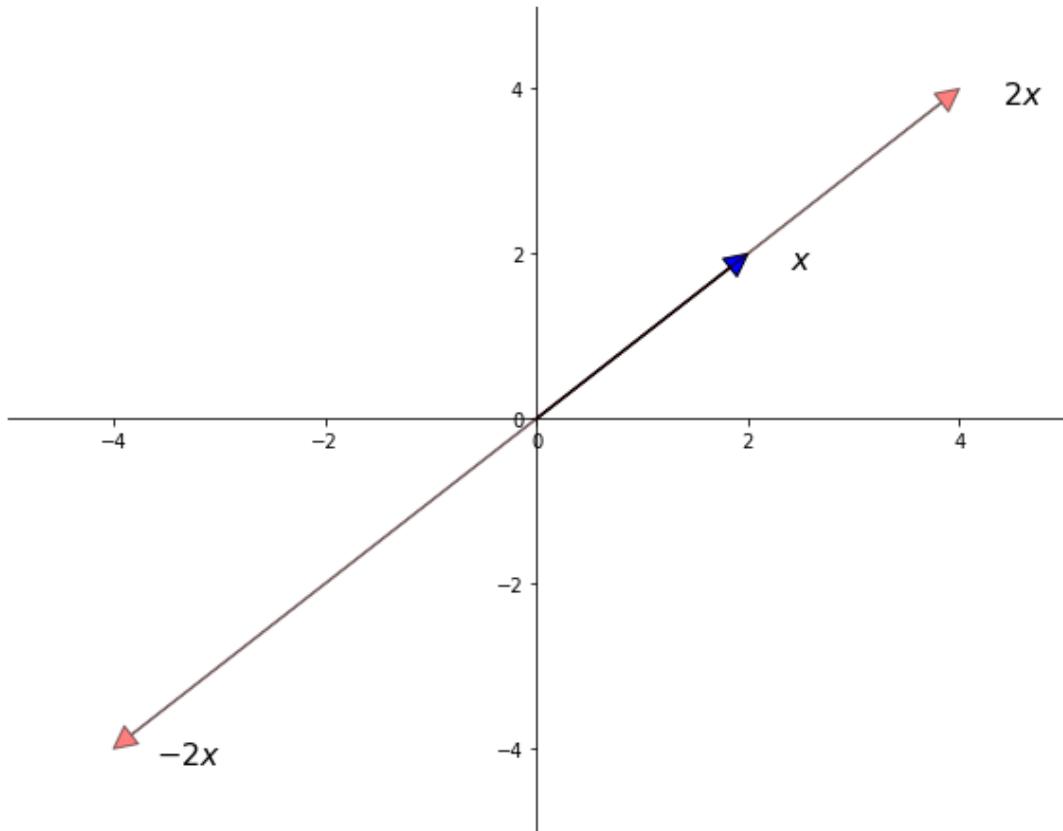
    ax.spines[spine].set_position('zero')
for spine in ['right', 'top']:
    ax.spines[spine].set_color('none')

ax.set(xlim=(-5, 5), ylim=(-5, 5))
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

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

```
Out[4]: array([3., 5., 7.])
```

```
In [5]: 4 * x
```

```
Out[5]: array([4., 4., 4.])
```

4.3.2 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

```
In [6]: np.sum(x * y)          # Inner product of x and y
```

```
Out[6]: 12.0
```

```
In [7]: np.sqrt(np.sum(x**2)) # Norm of x, take one
```

```
Out[7]: 1.7320508075688772
```

```
In [8]: np.linalg.norm(x)      # Norm of x, take two
```

```
Out[8]: 1.7320508075688772
```

4.3.3 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 β_1, \dots, β_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.

```
In [9]: fig = plt.figure(figsize=(10, 8))
ax = fig.gca(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,))

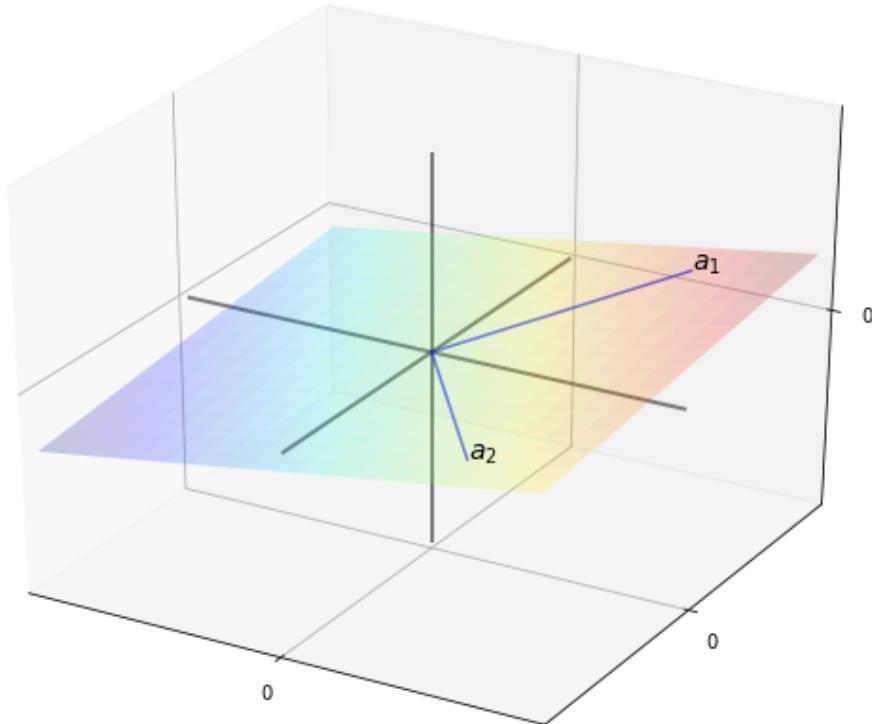
gs = 3
z = np.linspace(x_min, x_max, gs)
x = np.zeros(gs)
y = np.zeros(gs)
ax.plot(x, y, z, 'k-', lw=2, alpha=0.5)
ax.plot(z, x, y, 'k-', lw=2, alpha=0.5)
ax.plot(y, z, x, 'k-', lw=2, alpha=0.5)

# 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 .

4.3.4 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 $A := \{a_1, \dots, a_k\}$ 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 $A := \{a_1, \dots, a_k\} \subset \mathbb{R}^n$

1. No vector in A can be formed as a linear combination of the other elements.
2. If $\beta_1 a_1 + \dots + \beta_k a_k = 0$ for scalars β_1, \dots, β_k , then $\beta_1 = \dots = \beta_k = 0$.

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

4.3.5 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 $A := \{a_1, \dots, a_k\} \subset \mathbb{R}^n$ is linearly independent and

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

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

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

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

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

4.4 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 $i = 1, \dots, n$ 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 .

4.4.1 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

$$Ax = \begin{bmatrix} a_{11} & \cdots & a_{1k} \\ \vdots & \vdots & \vdots \\ a_{n1} & \cdots & a_{nk} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_k \end{bmatrix} := \begin{bmatrix} a_{11}x_1 + \cdots + a_{1k}x_k \\ \vdots \\ a_{n1}x_1 + \cdots + a_{nk}x_k \end{bmatrix} \quad (2)$$

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 $IA = A$.

4.4.2 Matrices in NumPy

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

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

```
In [10]: A = ((1, 2),
             (3, 4))
```

```
type(A)
```

```
Out[10]: tuple
```

```
In [11]: A = np.array(A)
```

```
type(A)
```

```
Out[11]: numpy.ndarray
```

```
In [12]: A.shape
```

```
Out[12]: (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

```
In [13]: A = np.identity(3)
B = np.ones((3, 3))
2 * A
```

```
Out[13]: array([[2., 0., 0.],
 [0., 2., 0.],
 [0., 0., 2.]])
```

```
In [14]: A + B
```

```
Out[14]: 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.

4.4.3 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 $f: \mathbb{R}^k \rightarrow \mathbb{R}^n$ 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 .

4.5 Solving Systems of Equations

Recall again the system of equations (1).

If we compare (1) and (2), we see that (1) can now be written more conveniently as

$$y = Ax \quad (3)$$

The problem we face is to determine a vector $x \in \mathbb{R}^k$ that solves (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

```
In [15]: 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)

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)

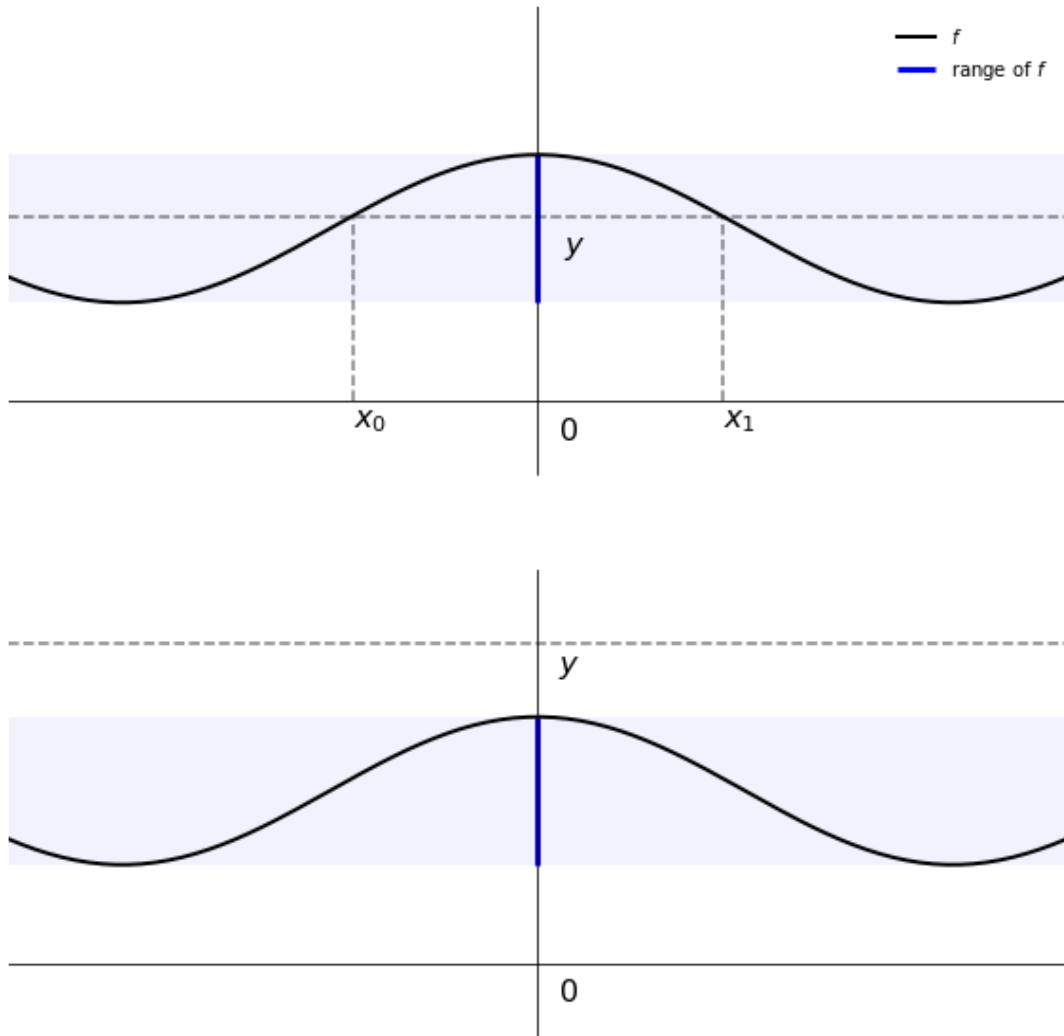
ax = axes[0]

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 (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, \dots, 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 $\{a_1, \dots, a_k\}$ are linearly independent and $y = Ax = x_1 a_1 + \cdots + x_k a_k$, then no $z \neq x$ satisfies $y = Az$.

4.5.1 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.

4.5.2 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.

4.5.3 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.

4.5.4 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

```
In [16]: 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
```

Out[16]: -2.0

```
In [17]: A_inv = inv(A) # Compute the inverse
A_inv
```

Out[17]: array([[-2., 1.],
 [1.5, -0.5]])

```
In [18]: x = A_inv @ y # Solution
A @ x # Should equal y
```

Out[18]: array([[1.],
 [1.]])

```
In [19]: solve(A, y) # Produces the same solution
```

Out[19]: 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)`.

4.6 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

```
In [20]: 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)

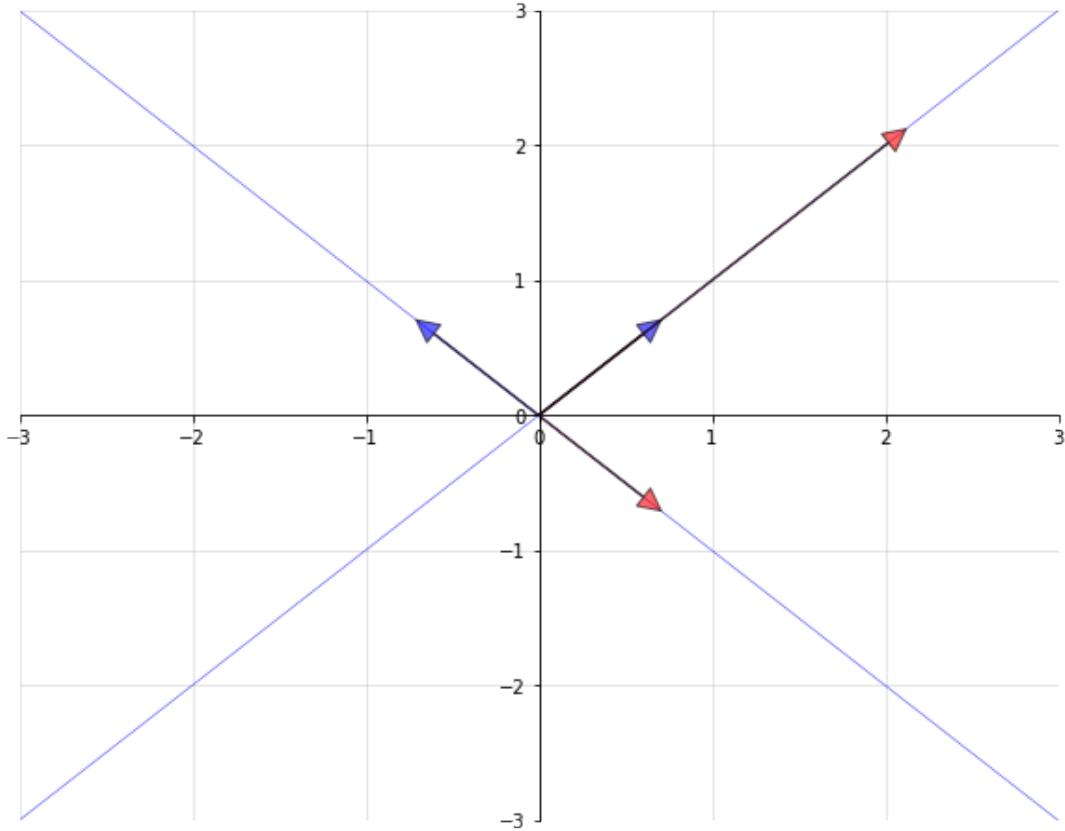
xmin, xmax = -3, 3
ymin, ymax = -3, 3
ax.set(xlim=(xmin, xmax), ylim=(ymin, ymax))

# 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 $\lambda_1, \dots, \lambda_n$ are its eigenvalues, then the eigenvalues of A^{-1} are $1/\lambda_1, \dots, 1/\lambda_n$.

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

```
In [21]: A = ((1, 2),
             (2, 1))

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

Out[21]: array([ 3.+0.j, -1.+0.j])

In [22]: evecs

Out[22]: 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.

4.6.1 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.

4.7 Further Topics

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

4.7.1 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 Section ??.

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

$$(I - A)^{-1} = \sum_{k=0}^{\infty} A^k \quad (4)$$

Spectral Radius

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

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

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 (4) is valid.

4.7.2 Positive Definite Matrices

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

We say that A is

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

4.7.3 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. $\frac{\partial Ax}{\partial x} = A'$
3. $\frac{\partial x'Ax}{\partial x} = (A + A')x$
4. $\frac{\partial y'Bz}{\partial y} = Bz$
5. $\frac{\partial y'Bz}{\partial B} = yz'$

Exercise 1 below asks you to apply these formulas.

4.7.4 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 [\[60\]](#).

4.8 Exercises

4.8.1 Exercise 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.

4.9 Solutions

4.9.1 Solution to Exercise 1

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

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

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 $\tilde{P} := A'PA - A'PB(Q + B'PB)^{-1}B'PA$

Footnotes

[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](#).

[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.

Chapter 5

Complex Numbers and Trigonometry

5.1 Contents

- Overview 5.2
- De Moivre's Theorem 5.3
- Applications of de Moivre's Theorem 5.4

5.2 Overview

This lecture introduces some elementary mathematics and trigonometry.

Useful and interesting in its own right, these concepts reap substantial rewards when studying dynamics generated by linear difference equations or linear differential equations.

For example, these tools are keys to understanding outcomes attained by Paul Samuelson (1939) [93] in his classic paper on interactions between the investment accelerator and the Keynesian consumption function, our topic in the lecture [Samuelson Multiplier Accelerator](#).

In addition to providing foundations for Samuelson's work and extensions of it, this lecture can be read as a stand-alone quick reminder of key results from elementary high school trigonometry.

So let's dive in.

5.2.1 Complex Numbers

A complex number has a **real part** x and a purely **imaginary part** y .

The Euclidean, polar, and trigonometric forms of a complex number z are:

$$z = x + iy = re^{i\theta} = r(\cos \theta + i \sin \theta)$$

The second equality above is known as **Euler's formula**

- Euler contributed many other formulas too!

The complex conjugate \bar{z} of z is defined as

$$\bar{z} = x - iy = re^{-i\theta} = r(\cos \theta - i \sin \theta)$$

The value x is the **real** part of z and y is the **imaginary** part of z .

The symbol $|z| = \sqrt{\bar{z} \cdot z} = r$ represents the **modulus** of z .

The value r is the Euclidean distance of vector (x, y) from the origin:

$$r = |z| = \sqrt{x^2 + y^2}$$

The value θ is the angle of (x, y) with respect to the real axis.

Evidently, the tangent of θ is $(\frac{y}{x})$.

Therefore,

$$\theta = \tan^{-1}\left(\frac{y}{x}\right)$$

Three elementary trigonometric functions are

$$\cos \theta = \frac{x}{r} = \frac{e^{i\theta} + e^{-i\theta}}{2}, \quad \sin \theta = \frac{y}{r} = \frac{e^{i\theta} - e^{-i\theta}}{2i}, \quad \tan \theta = \frac{y}{x}$$

We'll need the following imports:

```
In [1]: import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from sympy import *
```

5.2.2 An Example

Consider the complex number $z = 1 + \sqrt{3}i$.

For $z = 1 + \sqrt{3}i$, $x = 1$, $y = \sqrt{3}$.

It follows that $r = 2$ and $\theta = \tan^{-1}(\sqrt{3}) = \frac{\pi}{3} = 60^\circ$.

Let's use Python to plot the trigonometric form of the complex number $z = 1 + \sqrt{3}i$.

```
In [2]: # Abbreviate useful values and functions
```

```
π = np.pi
zeros = np.zeros
ones = np.ones

# Set parameters
r = 2
θ = π/3
x = r * np.cos(θ)
x_range = np.linspace(0, x, 1000)
θ_range = np.linspace(0, θ, 1000)
```

```
# Plot
fig = plt.figure(figsize=(8, 8))
ax = plt.subplot(111, projection='polar')

ax.plot((0, 0), (0, r), marker='o', color='b')          # Plot r
ax.plot(zeros(x_range.shape), x_range, color='b')       # Plot x
ax.plot(theta_range, x / np.cos(theta_range), color='b') # Plot y
ax.plot(theta_range, ones(theta_range.shape) * 0.1, color='r') # Plot theta

ax.margins(0) # Let the plot starts at origin

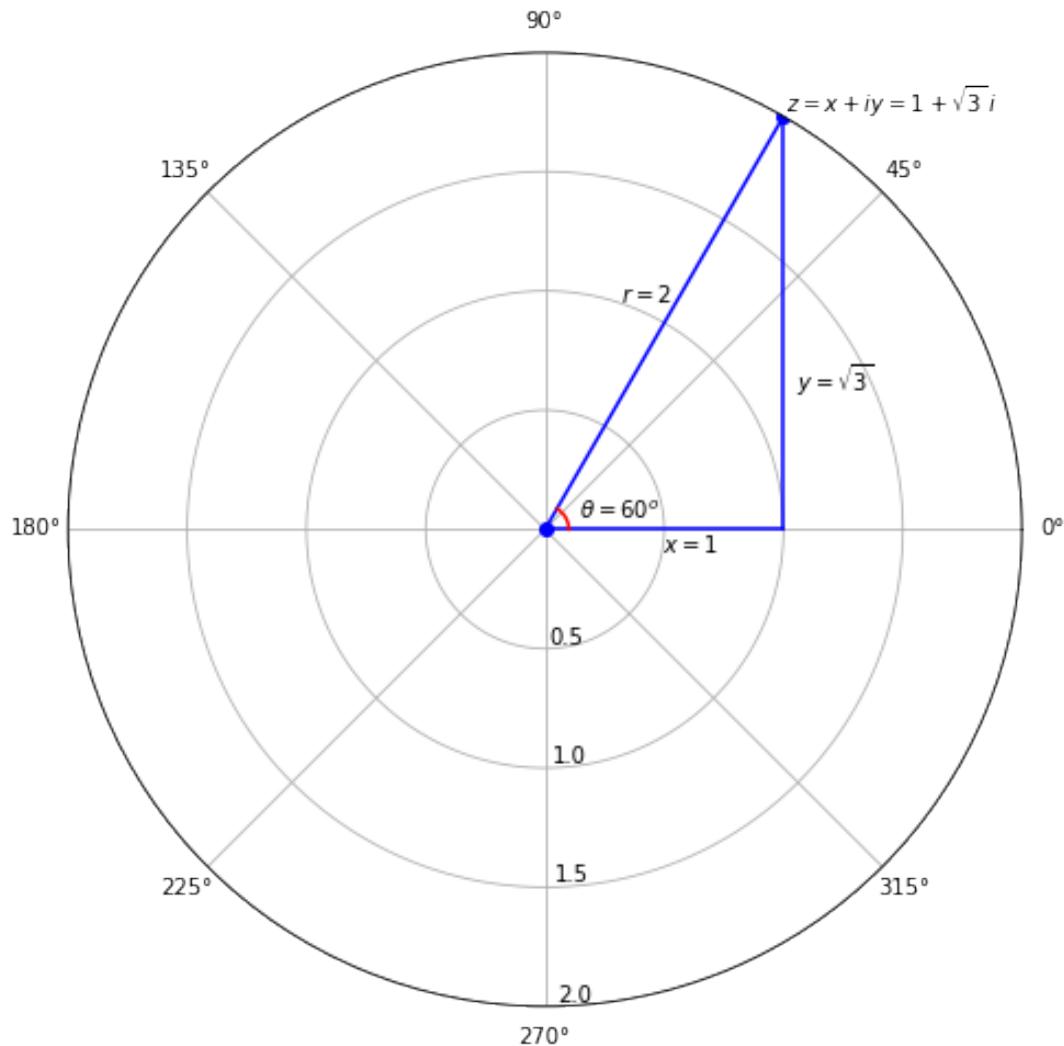
ax.set_title("Trigonometry of complex numbers", va='bottom',
             fontsize='x-large')

ax.set_rmax(2)
ax.set_rticks((0.5, 1, 1.5, 2)) # Less radial ticks
ax.set_rlabel_position(-88.5)    # Get radial labels away from plotted line

ax.text(0, r+0.01, r'$z = x + iy = 1 + \sqrt{3}i$')    # Label z
ax.text(theta+0.2, 1, '$r = 2$')                         # Label r
ax.text(theta-0.2, 0.5, '$x = 1$')                      # Label x
ax.text(theta+0.5, 1.2, r'$y = \sqrt{3}$')              # Label y
ax.text(theta+0.25, 0.15, r'$\theta = 60^\circ$')        # Label theta

ax.grid(True)
plt.show()
```

Trigonometry of complex numbers



5.3 De Moivre's Theorem

de Moivre's theorem states that:

$$(r(\cos \theta + i \sin \theta))^n = r^n e^{in\theta} = r^n (\cos n\theta + i \sin n\theta)$$

To prove de Moivre's theorem, note that

$$(r(\cos \theta + i \sin \theta))^n = (re^{i\theta})^n$$

and compute.

5.4 Applications of de Moivre's Theorem

5.4.1 Example 1

We can use de Moivre's theorem to show that $r = \sqrt{x^2 + y^2}$.

We have

$$\begin{aligned} 1 &= e^{i\theta}e^{-i\theta} \\ &= (\cos \theta + i \sin \theta)(\cos(-\theta) + i \sin(-\theta)) \\ &= (\cos \theta + i \sin \theta)(\cos \theta - i \sin \theta) \\ &= \cos^2 \theta + \sin^2 \theta \\ &= \frac{x^2}{r^2} + \frac{y^2}{r^2} \end{aligned}$$

and thus

$$x^2 + y^2 = r^2$$

We recognize this as a theorem of **Pythagoras**.

5.4.2 Example 2

Let $z = re^{i\theta}$ and $\bar{z} = re^{-i\theta}$ so that \bar{z} is the **complex conjugate** of z .

(z, \bar{z}) form a **complex conjugate pair** of complex numbers.

Let $a = pe^{i\omega}$ and $\bar{a} = pe^{-i\omega}$ be another complex conjugate pair.

For each element of a sequence of integers $n = 0, 1, 2, \dots,$

To do so, we can apply de Moivre's formula.

Thus,

$$\begin{aligned} x_n &= az^n + \bar{a}\bar{z}^n \\ &= pe^{i\omega}(re^{i\theta})^n + pe^{-i\omega}(re^{-i\theta})^n \\ &= pr^n e^{i(\omega+n\theta)} + pr^n e^{-i(\omega+n\theta)} \\ &= pr^n [\cos(\omega + n\theta) + i \sin(\omega + n\theta) + \cos(\omega + n\theta) - i \sin(\omega + n\theta)] \\ &= 2pr^n \cos(\omega + n\theta) \end{aligned}$$

5.4.3 Example 3

This example provides machinery that is at the heart of Samuelson's analysis of his multiplier-accelerator model [93].

Thus, consider a **second-order linear difference equation**

$$x_{n+2} = c_1 x_{n+1} + c_2 x_n$$

whose **characteristic polynomial** is

$$z^2 - c_1 z - c_2 = 0$$

or

$$(z^2 - c_1 z - c_2) = (z - z_1)(z - z_2) = 0$$

has roots z_1, z_2 .

A **solution** is a sequence $\{x_n\}_{n=0}^{\infty}$ that satisfies the difference equation.

Under the following circumstances, we can apply our example 2 formula to solve the difference equation

- the roots z_1, z_2 of the characteristic polynomial of the difference equation form a complex conjugate pair
- the values x_0, x_1 are given initial conditions

To solve the difference equation, recall from example 2 that

$$x_n = 2pr^n \cos(\omega + n\theta)$$

where ω, p are coefficients to be determined from information encoded in the initial conditions x_1, x_0 .

Since $x_0 = 2p \cos \omega$ and $x_1 = 2pr \cos(\omega + \theta)$ the ratio of x_1 to x_0 is

$$\frac{x_1}{x_0} = \frac{r \cos(\omega + \theta)}{\cos \omega}$$

We can solve this equation for ω then solve for p using $x_0 = 2pr^0 \cos(\omega + n\theta)$.

With the **sympy** package in Python, we are able to solve and plot the dynamics of x_n given different values of n .

In this example, we set the initial values: - $r = 0.9$ - $\theta = \frac{1}{4}\pi$ - $x_0 = 4$ - $x_1 = r \cdot 2\sqrt{2} = 1.8\sqrt{2}$.

We first numerically solve for ω and p using **nsolve** in the **sympy** package based on the above initial condition:

```
In [3]: # Set parameters
r = 0.9
θ = π/4
x0 = 4
x1 = 2 * r * sqrt(2)

# Define symbols to be calculated
ω, p = symbols('ω p', real=True)

# Solve for ω
## Note: we choose the solution near θ
eq1 = Eq(x1/x0 - r * cos(ω+θ) / cos(ω), 0)
ω = nsolve(eq1, ω, θ)
ω = np.float(ω)
print(f'ω = {ω:1.3f}')
```

```
# Solve for p
eq2 = Eq(x0 - 2 * p * cos(omega), 0)
p = nsolve(eq2, p, 0)
p = np.float(p)
print(f'p = {p:.3f}')
```

```
omega = 0.000
p = 2.000
```

Using the code above, we compute that $\omega = 0$ and $p = 2$.

Then we plug in the values we solve for ω and p and plot the dynamic.

```
In [4]: # Define range of n
max_n = 30
n = np.arange(0, max_n+1, 0.01)

# Define x_n
x = lambda n: 2 * p * r**n * np.cos(omega + n * theta)

# Plot
fig, ax = plt.subplots(figsize=(12, 8))

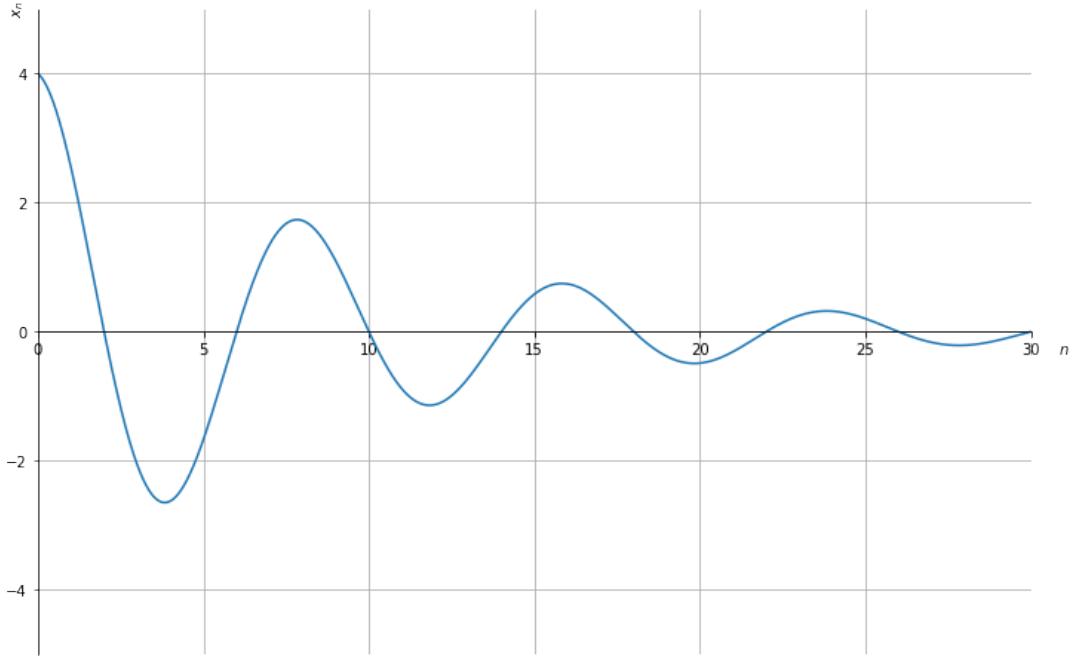
ax.plot(n, x(n))
ax.set(xlim=(0, max_n), ylim=(-5, 5), xlabel='$n$', ylabel='$x_n$')

# Set x-axis in the middle of the plot
ax.spines['bottom'].set_position('center')
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')

ticklab = ax.xaxis.get_ticklabels()[0] # Set x-label position
trans = ticklab.get_transform()
ax.xaxis.set_label_coords(31, 0, transform=trans)

ticklab = ax.yaxis.get_ticklabels()[0] # Set y-label position
trans = ticklab.get_transform()
ax.yaxis.set_label_coords(0, 5, transform=trans)

ax.grid()
plt.show()
```



5.4.4 Trigonometric Identities

We can obtain a complete suite of trigonometric identities by appropriately manipulating polar forms of complex numbers.

We'll get many of them by deducing implications of the equality

$$e^{i(\omega+\theta)} = e^{i\omega}e^{i\theta}$$

For example, we'll calculate identities for

$\cos(\omega + \theta)$ and $\sin(\omega + \theta)$.

Using the sine and cosine formulas presented at the beginning of this lecture, we have:

$$\begin{aligned}\cos(\omega + \theta) &= \frac{e^{i(\omega+\theta)} + e^{-i(\omega+\theta)}}{2} \\ \sin(\omega + \theta) &= \frac{e^{i(\omega+\theta)} - e^{-i(\omega+\theta)}}{2i}\end{aligned}$$

We can also obtain the trigonometric identities as follows:

$$\begin{aligned}\cos(\omega + \theta) + i \sin(\omega + \theta) &= e^{i(\omega+\theta)} \\ &= e^{i\omega}e^{i\theta} \\ &= (\cos \omega + i \sin \omega)(\cos \theta + i \sin \theta) \\ &= (\cos \omega \cos \theta - \sin \omega \sin \theta) + i(\cos \omega \sin \theta + \sin \omega \cos \theta)\end{aligned}$$

Since both real and imaginary parts of the above formula should be equal, we get:

$$\begin{aligned}\cos(\omega + \theta) &= \cos \omega \cos \theta - \sin \omega \sin \theta \\ \sin(\omega + \theta) &= \cos \omega \sin \theta + \sin \omega \cos \theta\end{aligned}$$

The equations above are also known as the **angle sum identities**. We can verify the equations using the `simplify` function in the `sympy` package:

```
In [5]: # Define symbols
ω, θ = symbols('ω θ', real=True)

# Verify
print("cos(ω)cos(θ) - sin(ω)sin(θ) =", 
      simplify(cos(ω)*cos(θ) - sin(ω) * sin(θ)))
print("cos(ω)sin(θ) + sin(ω)cos(θ) =", 
      simplify(cos(ω)*sin(θ) + sin(ω) * cos(θ)))

cos(ω)cos(θ) - sin(ω)sin(θ) = cos(θ + ω)
cos(ω)sin(θ) + sin(ω)cos(θ) = sin(θ + ω)
```

5.4.5 Trigonometric Integrals

We can also compute the trigonometric integrals using polar forms of complex numbers.

For example, we want to solve the following integral:

$$\int_{-\pi}^{\pi} \cos(\omega) \sin(\omega) d\omega$$

Using Euler's formula, we have:

$$\begin{aligned}\int \cos(\omega) \sin(\omega) d\omega &= \int \frac{(e^{i\omega} + e^{-i\omega})}{2} \frac{(e^{i\omega} - e^{-i\omega})}{2i} d\omega \\ &= \frac{1}{4i} \int e^{2i\omega} - e^{-2i\omega} d\omega \\ &= \frac{1}{4i} \left(\frac{-i}{2} e^{2i\omega} - \frac{i}{2} e^{-2i\omega} + C_1 \right) \\ &= -\frac{1}{8} \left[\left(e^{i\omega} \right)^2 + \left(e^{-i\omega} \right)^2 - 2 \right] + C_2 \\ &= -\frac{1}{8} (e^{i\omega} - e^{-i\omega})^2 + C_2 \\ &= \frac{1}{2} \left(\frac{e^{i\omega} - e^{-i\omega}}{2i} \right)^2 + C_2 \\ &= \frac{1}{2} \sin^2(\omega) + C_2\end{aligned}$$

and thus:

$$\int_{-\pi}^{\pi} \cos(\omega) \sin(\omega) d\omega = \frac{1}{2} \sin^2(\pi) - \frac{1}{2} \sin^2(-\pi) = 0$$

We can verify the analytical as well as numerical results using `integrate` in the `sympy` package:

```
In [6]: # Set initial printing
init_printing()

ω = Symbol('ω')
print('The analytical solution for integral of cos(ω)sin(ω) is:')
integrate(cos(ω) * sin(ω), ω)
```

The analytical solution for integral of $\cos(\omega)\sin(\omega)$ is:

```
Out[6]:  $\frac{\sin^2(\omega)}{2}$ 
```

```
In [7]: print('The numerical solution for the integral of cos(ω)sin(ω) \
from -π to π is:')
integrate(cos(ω) * sin(ω), (ω, -π, π))
```

The numerical solution for the integral of $\cos(\omega)\sin(\omega)$ from $-\pi$ to π is:

```
Out[7]: 0
```

5.4.6 Exercises

We invite the reader to verify analytically and with the `sympy` package the following two equalities:

$$\int_{-\pi}^{\pi} \cos(\omega)^2 d\omega = \frac{\pi}{2}$$

$$\int_{-\pi}^{\pi} \sin(\omega)^2 d\omega = \frac{\pi}{2}$$

Chapter 6

LLN and CLT

6.1 Contents

- Overview 6.2
- Relationships 6.3
- LLN 6.4
- CLT 6.5
- Exercises 6.6
- Solutions 6.7

6.2 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.
- The multivariate case.

Some of these extensions are presented as exercises.

We'll need the following imports:

```
In [1]: import random
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from scipy.stats import t, beta, lognorm, expon, gamma, uniform, cauchy
from scipy.stats import gaussian_kde, poisson, binom, norm, chi2
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.collections import PolyCollection
from scipy.linalg import inv, sqrtm
```

6.3 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.

6.4 LLN

We begin with the law of large numbers, which tells us when sample averages will converge to their population means.

6.4.1 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, \dots, 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

$$\mathbb{P}\{\bar{X}_n \rightarrow \mu \text{ as } n \rightarrow \infty\} = 1 \quad (1)$$

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 $\bar{X}_n \rightarrow \mu$ can be evaluated.

In this setting, (1) should be interpreted as meaning that the probability of the computer producing a sequence where $\bar{X}_n \rightarrow \mu$ fails to occur is zero.

6.4.2 Proof

The proof of Kolmogorov's strong law is nontrivial – see, for example, theorem 8.3.5 of [31].

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, \dots, X_n is IID with $\mathbb{E}X_i^2 < \infty$, then, for any $\epsilon > 0$, we have

$$\mathbb{P}\left\{|\bar{X}_n - \mu| \geq \epsilon\right\} \rightarrow 0 \quad \text{as } n \rightarrow \infty \quad (2)$$

(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

$$\mathbb{P}\left\{|\bar{X}_n - \mu| \geq \epsilon\right\} \leq \frac{\mathbb{E}[(\bar{X}_n - \mu)^2]}{\epsilon^2} \quad (3)$$

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 (3), we come to the estimate

$$\mathbb{P}\left\{|\bar{X}_n - \mu| \geq \epsilon\right\} \leq \frac{\sigma^2}{n\epsilon^2} \quad (4)$$

The claim in (2) is now clear.

Of course, if the sequence X_1, \dots, 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, \dots, 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.

6.4.3 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 $i = 1, \dots, 100$.

In each of the three cases, convergence of \bar{X}_n to μ occurs as predicted

In [2]: `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)

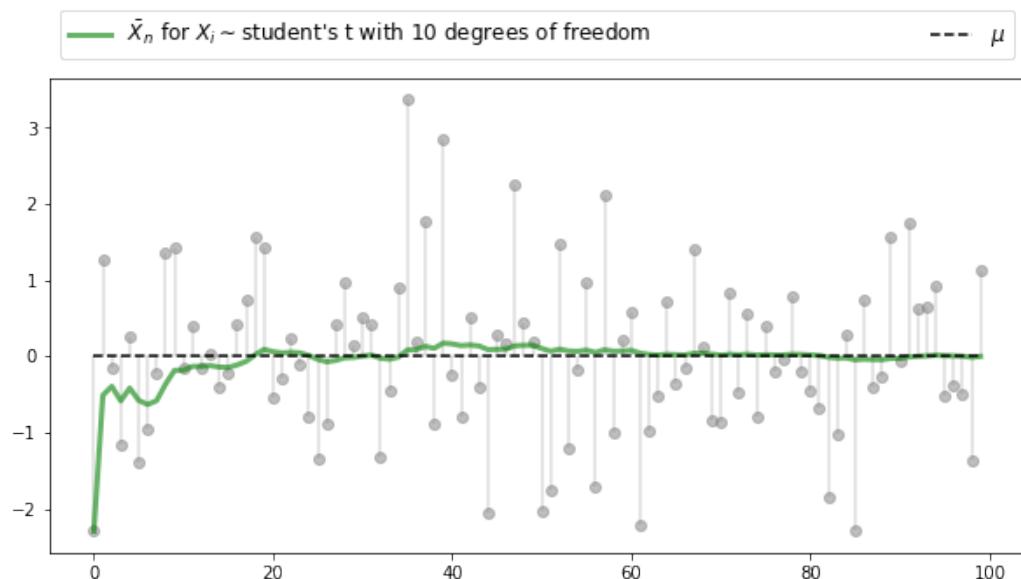
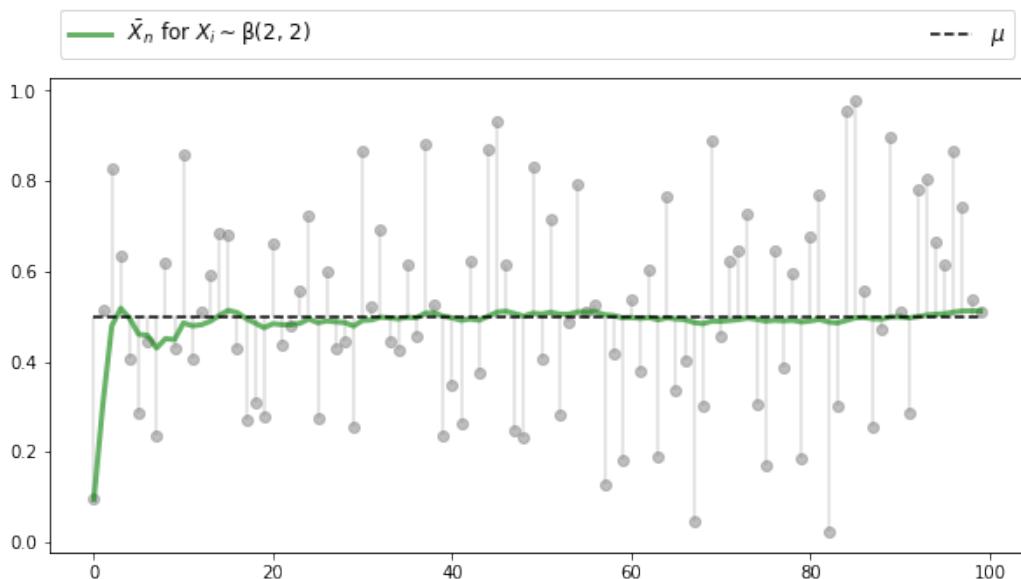
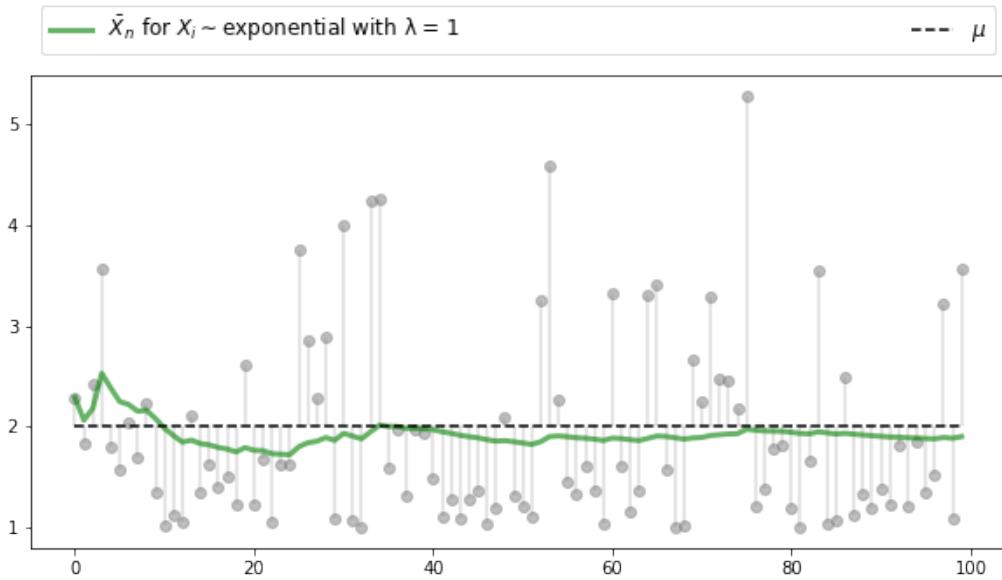
    # Generate n draws from the distribution
    data = distribution.rvs(n)

    # 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)
    xlabel = '$\bar{X}_n$ for $X_i \sim' + name
    ax.set_xlabel(xlabel)
```

```
    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='$\mu$')
    ax.vlines(list(range(n)), m, data, lw=0.2)
    ax.legend(**legend_args, fontsize=12)

plt.show()
```



The three distributions are chosen at random from a selection stored in the dictionary `distributions`.

6.5 CLT

Next, we turn to the central limit theorem, which tells us about the distribution of the deviation between sample averages and population means.

6.5.1 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, \dots, X_n is IID, with common mean μ and common variance $\sigma^2 \in (0, \infty)$, then

$$\sqrt{n}(\bar{X}_n - \mu) \xrightarrow{d} N(0, \sigma^2) \quad \text{as } n \rightarrow \infty \quad (5)$$

Here $\xrightarrow{d} N(0, \sigma^2)$ indicates convergence in distribution to a centered (i.e, zero mean) normal with standard deviation σ .

6.5.2 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 [31]).

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, \dots, 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 $n = 1, 2, 4, 8$

```
In [3]: 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))

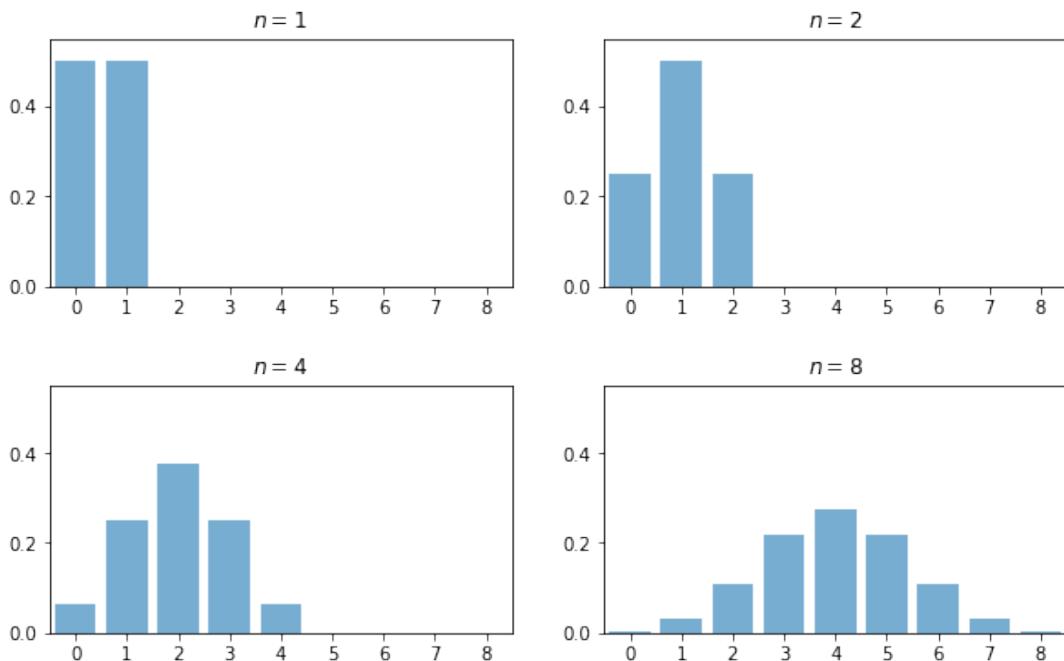
for ax, n in zip(axes, ns):
```

```

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](#).

6.5.3 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 $N(0, \sigma^2)$.

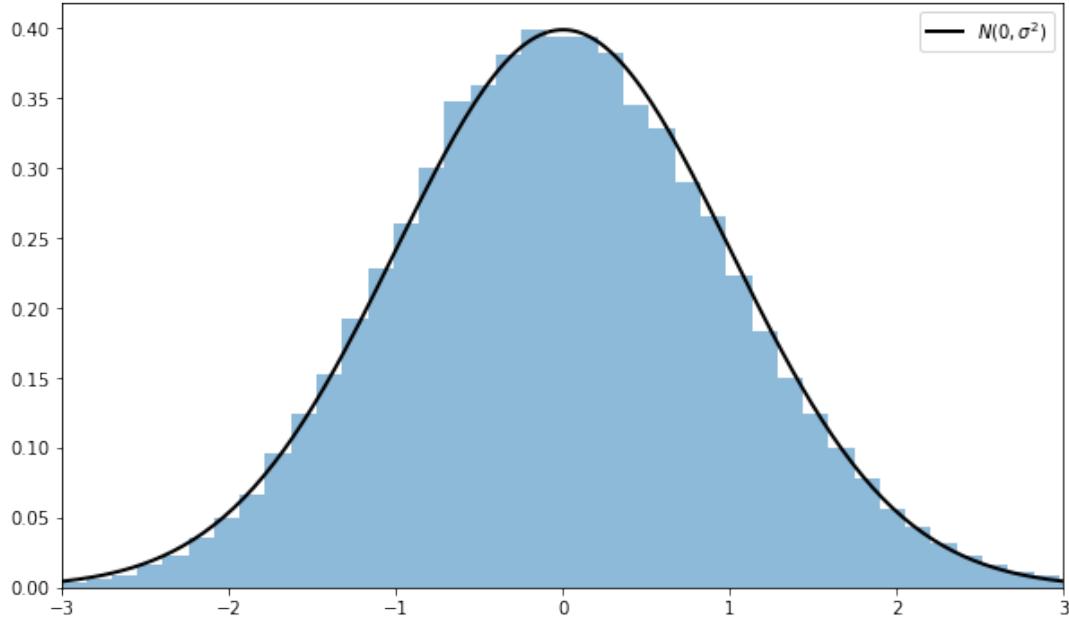
Here's some code that does exactly this for the exponential distribution $F(x) = 1 - e^{-\lambda x}$.

(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.)

```
In [4]: # 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='N(0, σ²)')
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 .

6.5.4 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

In [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 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
fig = plt.figure(figsize = (10, 6))
ax = fig.gca(projection='3d')

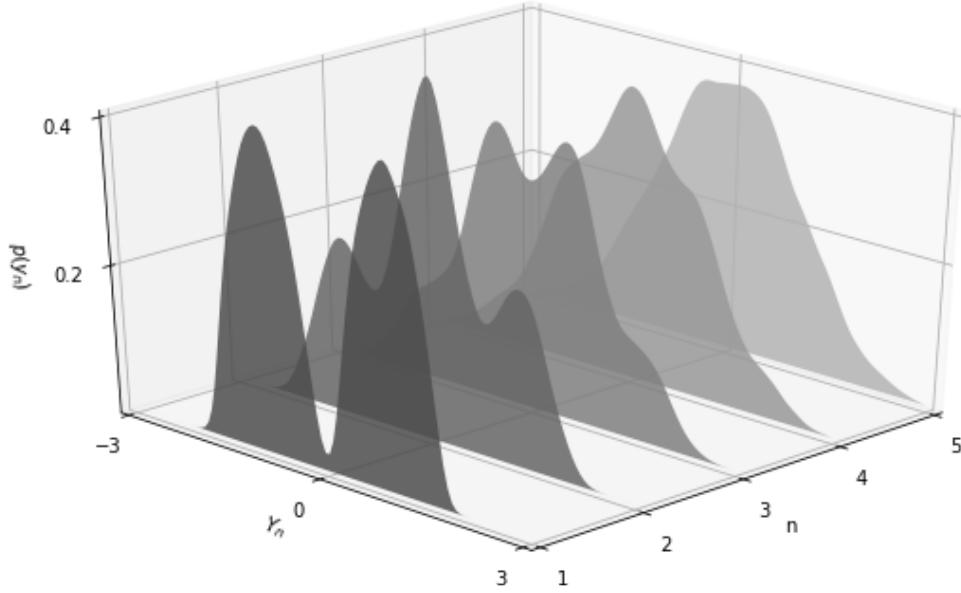
a, b = -3, 3
gs = 100
xs = np.linspace(a, b, gs)

# 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, xlabel='$Y_n$', ylabel='$p(y_n)$',
       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.

6.5.5 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, \dots, X_k) .

Each realization of \mathbf{X} is an element of \mathbb{R}^k .

A collection of random vectors $\mathbf{X}_1, \dots, \mathbf{X}_n$ is called independent if, given any n vectors $\mathbf{x}_1, \dots, \mathbf{x}_n$ in \mathbb{R}^k , we have

$$\mathbb{P}\{\mathbf{X}_1 \leq \mathbf{x}_1, \dots, \mathbf{X}_n \leq \mathbf{x}_n\} = \mathbb{P}\{\mathbf{X}_1 \leq \mathbf{x}_1\} \times \dots \times \mathbb{P}\{\mathbf{X}_n \leq \mathbf{x}_n\}$$

(The vector inequality $\mathbf{X} \leq \mathbf{x}$ means that $X_j \leq x_j$ for $j = 1, \dots, k$)

Let $\mu_j := \mathbb{E}[X_j]$ for all $j = 1, \dots, k$.

The expectation $\mathbb{E}[\mathbf{X}]$ of \mathbf{X} is defined to be the vector of expectations:

$$\mathbb{E}[\mathbf{X}] := \begin{pmatrix} \mathbb{E}[X_1] \\ \mathbb{E}[X_2] \\ \vdots \\ \mathbb{E}[X_k] \end{pmatrix} = \begin{pmatrix} \mu_1 \\ \mu_2 \\ \vdots \\ \mu_k \end{pmatrix} =: \boldsymbol{\mu}$$

The *variance-covariance matrix* of random vector \mathbf{X} is defined as

$$\text{Var}[\mathbf{X}] := \mathbb{E}[(\mathbf{X} - \mu)(\mathbf{X} - \mu)']$$

Expanding this out, we get

$$\text{Var}[\mathbf{X}] = \begin{pmatrix} \mathbb{E}[(X_1 - \mu_1)(X_1 - \mu_1)] & \cdots & \mathbb{E}[(X_1 - \mu_1)(X_k - \mu_k)] \\ \mathbb{E}[(X_2 - \mu_2)(X_1 - \mu_1)] & \cdots & \mathbb{E}[(X_2 - \mu_2)(X_k - \mu_k)] \\ \vdots & \vdots & \vdots \\ \mathbb{E}[(X_k - \mu_k)(X_1 - \mu_1)] & \cdots & \mathbb{E}[(X_k - \mu_k)(X_k - \mu_k)] \end{pmatrix}$$

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 $\mathbf{X}_1, \dots, \mathbf{X}_n$ 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

$$\mathbb{P}\{\bar{\mathbf{X}}_n \rightarrow \mu \text{ as } n \rightarrow \infty\} = 1 \quad (6)$$

Here $\bar{\mathbf{X}}_n \rightarrow \mu$ means that $\|\bar{\mathbf{X}}_n - \mu\| \rightarrow 0$, where $\|\cdot\|$ is the standard Euclidean norm.

The CLT tells us that, provided Σ is finite,

$$\sqrt{n}(\bar{\mathbf{X}}_n - \mu) \xrightarrow{d} N(\mathbf{0}, \Sigma) \quad \text{as } n \rightarrow \infty \quad (7)$$

6.6 Exercises

6.6.1 Exercise 1

One very useful consequence of the central limit theorem is as follows.

Assume the conditions of the CLT as [stated above](#).

If $g: \mathbb{R} \rightarrow \mathbb{R}$ is differentiable at μ and $g'(\mu) \neq 0$, then

$$\sqrt{n}\{g(\bar{X}_n) - g(\mu)\} \xrightarrow{d} N(0, g'(\mu)^2 \sigma^2) \quad \text{as } n \rightarrow \infty \quad (8)$$

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?

6.6.2 Exercise 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. $\mathbf{X}_1, \dots, \mathbf{X}_n$ 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

$$\sqrt{n}(\bar{\mathbf{X}}_n - \mu) \xrightarrow{d} N(\mathbf{0}, \Sigma) \quad (9)$$

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

$$\text{Var}[\mathbf{AX}] = \mathbf{A} \text{Var}[\mathbf{X}] \mathbf{A}'$$

Second, by the [continuous mapping theorem](#), if $\mathbf{Z}_n \xrightarrow{d} \mathbf{Z}$ in \mathbb{R}^k and \mathbf{A} is constant and $k \times k$, then

$$\mathbf{AZ}_n \xrightarrow{d} \mathbf{AZ}$$

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

$$\mathbf{QSQ}' = \mathbf{I}$$

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 \mathbf{S} , then

$$\mathbf{Z}_n := \sqrt{n} \mathbf{Q}(\bar{\mathbf{X}}_n - \mu) \xrightarrow{d} \mathbf{Z} \sim N(\mathbf{0}, \mathbf{I})$$

Applying the continuous mapping theorem one more time tells us that

$$\|\mathbf{Z}_n\|^2 \xrightarrow{d} \|\mathbf{Z}\|^2$$

Given the distribution of \mathbf{Z} , we conclude that

$$n \|\mathbf{Q}(\bar{\mathbf{X}}_n - \mu)\|^2 \xrightarrow{d} \chi^2(k) \quad (10)$$

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) 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.

Hints:

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.

6.7 Solutions

6.7.1 Exercise 1

Here is one solution

```
In [6]: """
    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()

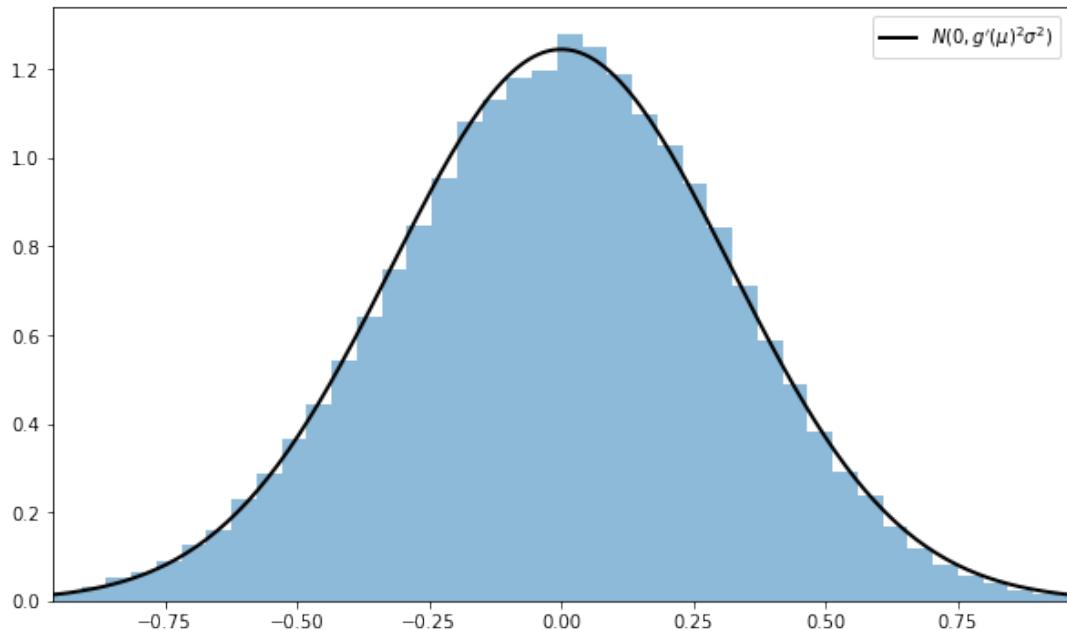
g = np.sin
g_prime = np.cos
```

```

# 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 = "$N(0, g'(\mu)^2 \sigma^2)$"
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.

6.7.2 Exercise 2

First we want to verify the claim that

$$\sqrt{n}\mathbf{Q}(\bar{\mathbf{X}}_n - \mu) \xrightarrow{d} N(\mathbf{0}, \mathbf{I})$$

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

$$\mathbf{Q}\mathbf{Y}_n \xrightarrow{d} \mathbf{Q}\mathbf{Y}$$

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

$$\text{Var}[\mathbf{Q}\mathbf{Y}] = \mathbf{Q}\text{Var}[\mathbf{Y}]\mathbf{Q}' = \mathbf{Q}\Sigma\mathbf{Q}' = \mathbf{I}$$

In conclusion, $\mathbf{Q}\mathbf{Y}_n \xrightarrow{d} \mathbf{Q}\mathbf{Y} \sim N(\mathbf{0}, \mathbf{I})$, which is what we aimed to show.

Now we turn to the simulation exercise.

Our solution is as follows

```
In [7]: # 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)

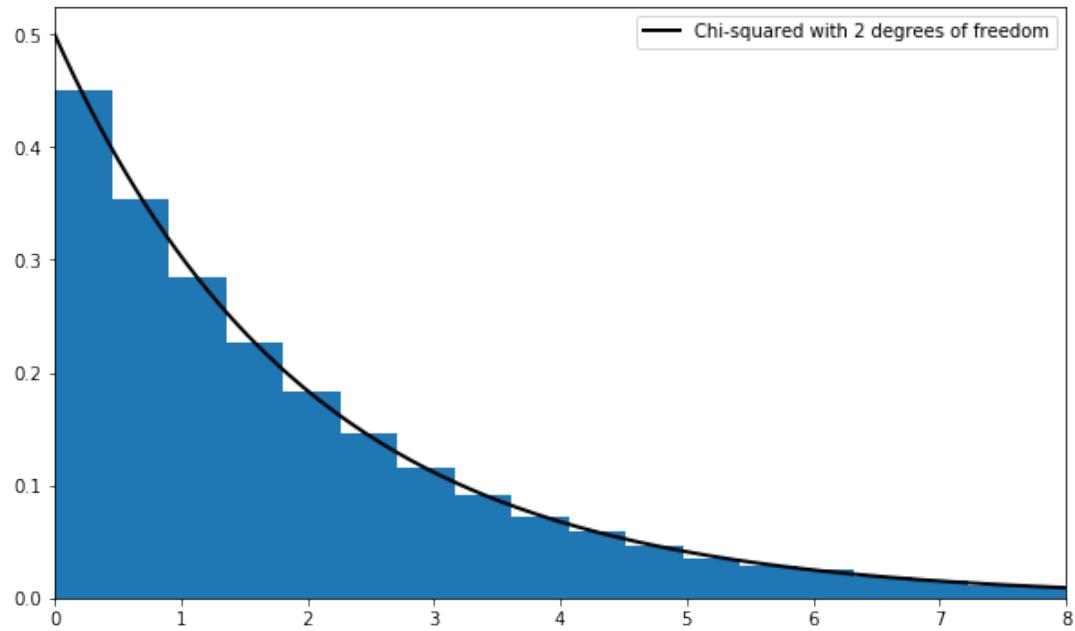
# Compute Σ^{-1/2}
Q = inv(sqrtm(Σ))

# 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()
```



Chapter 7

Heavy-Tailed Distributions

7.1 Contents

- Overview 7.2
- Visual Comparisons 7.3
- Failure of the LLN 7.4
- Classifying Tail Properties 7.5
- Exercises 7.6
- Solutions 7.7

In addition to what's in Anaconda, this lecture will need the following libraries:

```
In [1]: !pip install quantecon  
!pip install --upgrade yfinance
```

7.2 Overview

Most commonly used probability distributions in classical statistics and the natural sciences have either bounded support or light tails.

When a distribution is light-tailed, extreme observations are rare and draws tend not to deviate too much from the mean.

Having internalized these kinds of distributions, many researchers and practitioners use rules of thumb such as “outcomes more than four or five standard deviations from the mean can safely be ignored.”

However, some distributions encountered in economics have far more probability mass in the tails than distributions like the normal distribution.

With such **heavy-tailed** distributions, what would be regarded as extreme outcomes for someone accustomed to thin tailed distributions occur relatively frequently.

Examples of heavy-tailed distributions observed in economic and financial settings include

- the income distributions and the wealth distribution (see, e.g., [108], [9]),
- the firm size distribution ([7], [41]),
- the distribution of returns on holding assets over short time horizons ([76], [88]), and
- the distribution of city sizes ([91], [41]).

These heavy tails turn out to be important for our understanding of economic outcomes.

As one example, the heaviness of the tail in the wealth distribution is one natural measure of inequality.

It matters for taxation and redistribution policies, as well as for flow-on effects for productivity growth, business cycles, and political economy

- see, e.g., [2], [43], [15] or [3].

This lecture formalizes some of the concepts introduced above and reviews the key ideas.

Let's start with some imports:

```
In [2]: import numpy as np
import quantecon as qe
import matplotlib.pyplot as plt
%matplotlib inline

/home/ubuntu/anaconda3/lib/python3.7/site-packages/numba/np/ufunc/parallel.py:
 355:
NumbaWarning: The TBB threading layer requires TBB version 2019.5 or later i.e.,
TBB_INTERFACE_VERSION >= 11005. Found TBB_INTERFACE_VERSION = 11004. The TBB
  threading
layer is disabled.
warnings.warn(problem)
```

The following two lines can be added to avoid an annoying FutureWarning, and prevent a specific compatibility issue between pandas and matplotlib from causing problems down the line:

```
In [3]: from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters()
```

7.3 Visual Comparisons

One way to build intuition on the difference between light and heavy tails is to plot independent draws and compare them side-by-side.

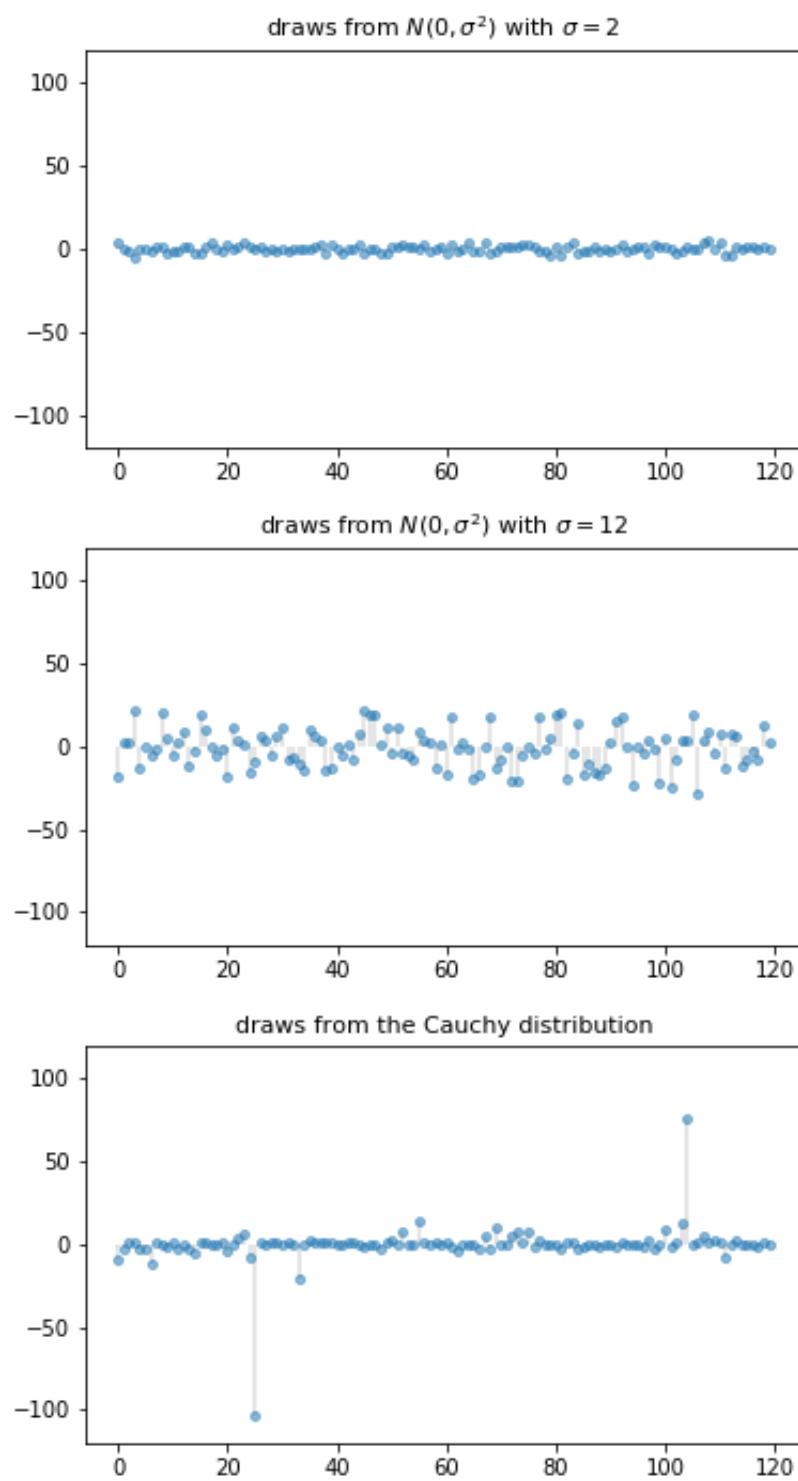
7.3.1 A Simulation

The figure below shows a simulation. (You will be asked to replicate it in the exercises.)

The top two subfigures each show 120 independent draws from the normal distribution, which is light-tailed.

The bottom subfigure shows 120 independent draws from the Cauchy distribution, which is

heavy-tailed.



In the top subfigure, the standard deviation of the normal distribution is 2, and the draws are clustered around the mean.

In the middle subfigure, the standard deviation is increased to 12 and, as expected, the amount of dispersion rises.

The bottom subfigure, with the Cauchy draws, shows a different pattern: tight clustering around the mean for the great majority of observations, combined with a few sudden large deviations from the mean.

This is typical of a heavy-tailed distribution.

7.3.2 Heavy Tails in Asset Returns

Next let's look at some financial data.

Our aim is to plot the daily change in the price of Amazon (AMZN) stock for the period from 1st January 2015 to 1st November 2019.

This equates to daily returns if we set dividends aside.

The code below produces the desired plot using Yahoo financial data via the `yfinance` library.

```
In [4]: import yfinance as yf
import pandas as pd

s = yf.download('AMZN', '2015-1-1', '2019-11-1')['Adj Close']

r = s.pct_change()

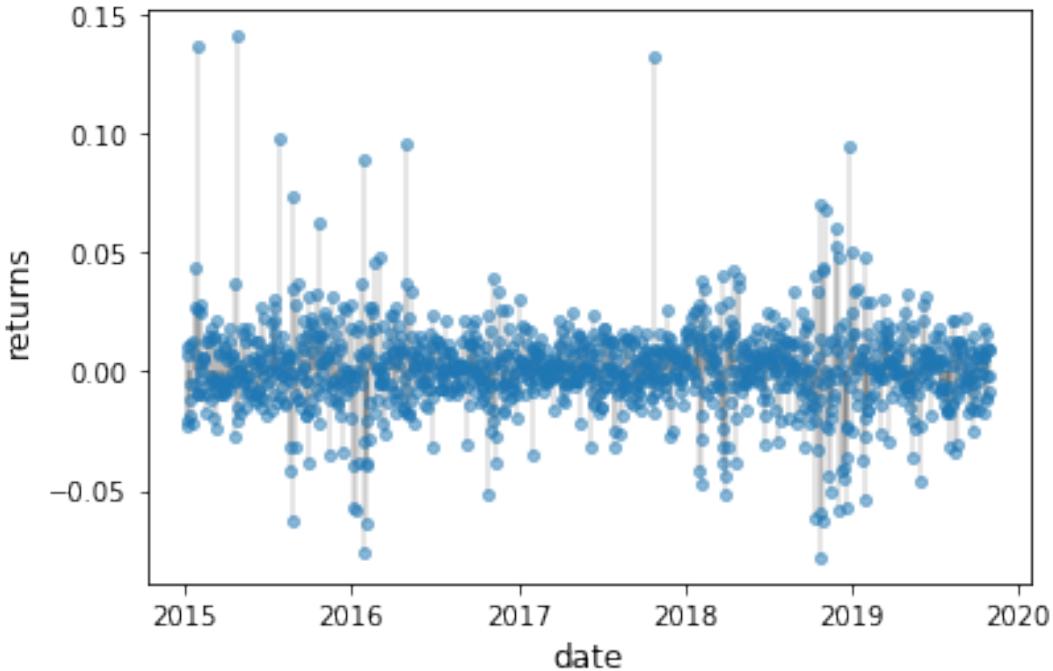
fig, ax = plt.subplots()

ax.plot(r, linestyle='', marker='o', alpha=0.5, ms=4)
ax.vlines(r.index, 0, r.values, lw=0.2)

ax.set_ylabel('returns', fontsize=12)
ax.set_xlabel('date', fontsize=12)

plt.show()
```

[*****100%*****] 1 of 1 completed



Five of the 1217 observations are more than 5 standard deviations from the mean.

Overall, the figure is suggestive of heavy tails, although not to the same degree as the Cauchy distribution the figure above.

If, however, one takes tick-by-tick data rather daily data, the heavy-tailedness of the distribution increases further.

7.4 Failure of the LLN

One impact of heavy tails is that sample averages can be poor estimators of the underlying mean of the distribution.

To understand this point better, recall [our earlier discussion](#) of the Law of Large Numbers, which considered IID X_1, \dots, X_n with common distribution F

If $\mathbb{E}|X_i|$ is finite, then the sample mean $\bar{X}_n := \frac{1}{n} \sum_{i=1}^n X_i$ satisfies

$$\mathbb{P}\{\bar{X}_n \rightarrow \mu \text{ as } n \rightarrow \infty\} = 1 \quad (1)$$

where $\mu := \mathbb{E}X_i = \int xF(x)$ is the common mean of the sample.

The condition $\mathbb{E}|X_i| = \int |x|F(x) < \infty$ holds in most cases but can fail if the distribution F is very heavy tailed.

For example, it fails for the Cauchy distribution.

Let's have a look at the behavior of the sample mean in this case, and see whether or not the LLN is still valid.

```
In [5]: from scipy.stats import cauchy
```

```

np.random.seed(1234)
N = 1_000

distribution = cauchy()

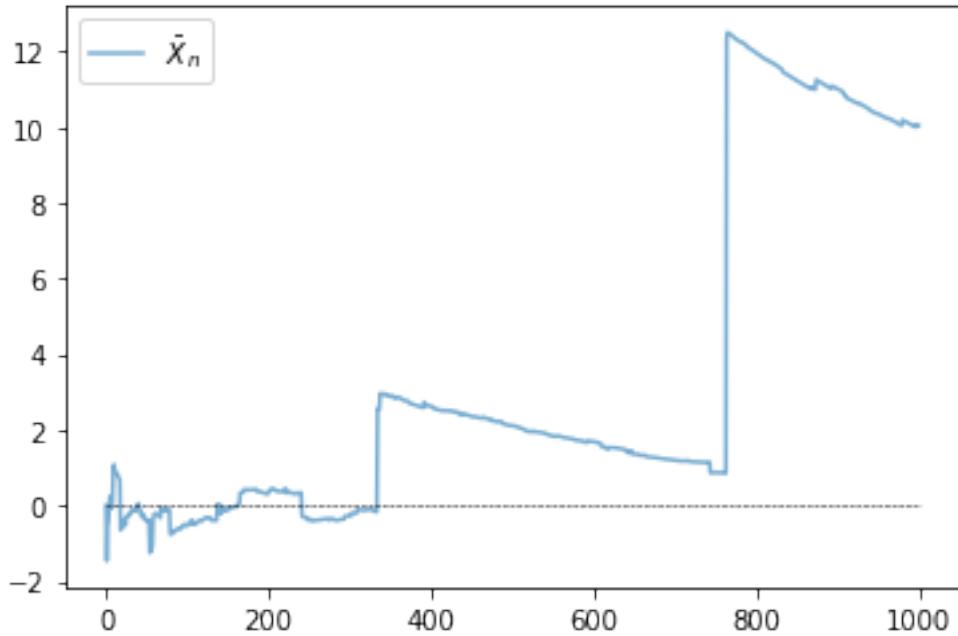
fig, ax = plt.subplots()
data = distribution.rvs(N)

# Compute sample mean at each n
sample_mean = np.empty(N)
for n in range(1, N):
    sample_mean[n] = np.mean(data[:n])

# Plot
ax.plot(range(N), sample_mean, alpha=0.6, label='$\bar{X}_n$')
ax.plot(range(N), np.zeros(N), 'k--', lw=0.5)
ax.legend()

plt.show()

```



The sequence shows no sign of converging.

Will convergence occur if we take n even larger?

The answer is no.

To see this, recall that the [characteristic function](#) of the Cauchy distribution is

$$\phi(t) = \mathbb{E}e^{itX} = \int e^{itx} f(x)dx = e^{-|t|} \quad (2)$$

Using independence, the characteristic function of the sample mean becomes

$$\begin{aligned}
\mathbb{E} e^{it\bar{X}_n} &= \mathbb{E} \exp \left\{ i \frac{t}{n} \sum_{j=1}^n X_j \right\} \\
&= \mathbb{E} \prod_{j=1}^n \exp \left\{ i \frac{t}{n} X_j \right\} \\
&= \prod_{j=1}^n \mathbb{E} \exp \left\{ i \frac{t}{n} X_j \right\} = [\phi(t/n)]^n
\end{aligned}$$

In view of (2), this is just $e^{-|t|}$.

Thus, in the case of the Cauchy distribution, the sample mean itself has the very same Cauchy distribution, regardless of n !

In particular, the sequence \bar{X}_n does not converge to any point.

7.5 Classifying Tail Properties

To keep our discussion precise, we need some definitions concerning tail properties.

We will focus our attention on the right hand tails of nonnegative random variables and their distributions.

The definitions for left hand tails are very similar and we omit them to simplify the exposition.

7.5.1 Light and Heavy Tails

A distribution F on \mathbb{R}_+ is called **heavy-tailed** if

$$\int_0^\infty \exp(tx)F(dx) = \infty \text{ for all } t > 0. \quad (3)$$

We say that a nonnegative random variable X is **heavy-tailed** if its distribution $F(x) := \mathbb{P}\{X \leq x\}$ is heavy-tailed.

This is equivalent to stating that its **moment generating function** $m(t) := \mathbb{E} \exp(tX)$ is infinite for all $t > 0$.

- For example, the lognormal distribution is heavy-tailed because its moment generating function is infinite everywhere on $(0, \infty)$.

A distribution F on \mathbb{R}_+ is called **light-tailed** if it is not heavy-tailed.

A nonnegative random variable X is **light-tailed** if its distribution F is light-tailed.

- Example: Every random variable with bounded support is light-tailed. (Why?)
- Example: If X has the exponential distribution, with cdf $F(x) = 1 - \exp(-\lambda x)$ for some $\lambda > 0$, then its moment generating function is finite whenever $t < \lambda$. Hence X is light-tailed.

One can show that if X is light-tailed, then all of its moments are finite.

The contrapositive is that if some moment is infinite, then X is heavy-tailed.

The latter condition is not necessary, however.

- Example: the lognormal distribution is heavy-tailed but every moment is finite.

7.5.2 Pareto Tails

One specific class of heavy-tailed distributions has been found repeatedly in economic and social phenomena: the class of so-called power laws.

Specifically, given $\alpha > 0$, a nonnegative random variable X is said to have a **Pareto tail** with **tail index** α if

$$\lim_{x \rightarrow \infty} x^\alpha \mathbb{P}\{X > x\} = c. \quad (4)$$

Evidently (4) implies the existence of positive constants b and \bar{x} such that $\mathbb{P}\{X > x\} \geq bx^{-\alpha}$ whenever $x \geq \bar{x}$.

The implication is that $\mathbb{P}\{X > x\}$ converges to zero no faster than $x^{-\alpha}$.

In some sources, a random variable obeying (4) is said to have a **power law tail**.

The primary example is the **Pareto distribution**, which has distribution

$$F(x) = \begin{cases} 1 - (\bar{x}/x)^\alpha & \text{if } x \geq \bar{x} \\ 0 & \text{if } x < \bar{x} \end{cases} \quad (5)$$

for some positive constants \bar{x} and α .

It is easy to see that if $X \sim F$, then $\mathbb{P}\{X > x\}$ satisfies (4).

Thus, in line with the terminology, Pareto distributed random variables have a Pareto tail.

7.5.3 Rank-Size Plots

One graphical technique for investigating Pareto tails and power laws is the so-called **rank-size plot**.

This kind of figure plots log size against log rank of the population (i.e., location in the population when sorted from smallest to largest).

Often just the largest 5 or 10% of observations are plotted.

For a sufficiently large number of draws from a Pareto distribution, the plot generates a straight line. For distributions with thinner tails, the data points are concave.

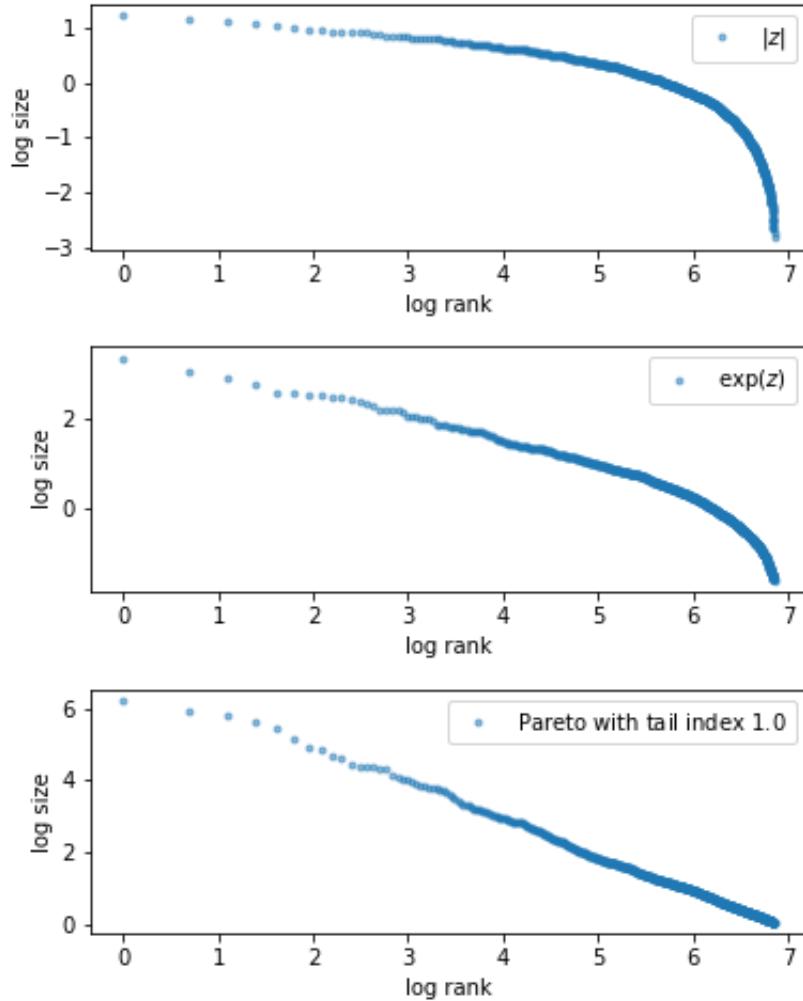
A discussion of why this occurs can be found in [84].

The figure below provides one example, using simulated data.

The rank-size plots shows draws from three different distributions: folded normal, chi-squared with 1 degree of freedom and Pareto.

The Pareto sample produces a straight line, while the lines produced by the other samples are concave.

You are asked to reproduce this figure in the exercises.



7.6 Exercises

7.6.1 Exercise 1

Replicate the figure presented above that compares normal and Cauchy draws.

Use `np.random.seed(11)` to set the seed.

7.6.2 Exercise 2

Prove: If X has a Pareto tail with tail index α , then $\mathbb{E}[X^r] = \infty$ for all $r \geq \alpha$.

7.6.3 Exercise 3

Repeat exercise 1, but replace the three distributions (two normal, one Cauchy) with three Pareto distributions using different choices of α .

For α , try 1.15, 1.5 and 1.75.

Use `np.random.seed(11)` to set the seed.

7.6.4 Exercise 4

Replicate the rank-size plot figure [presented above](#).

If you like you can use the function `qe.rank_size` from the `quantecon` library to generate the plots.

Use `np.random.seed(13)` to set the seed.

7.6.5 Exercise 5

There is an ongoing argument about whether the firm size distribution should be modeled as a Pareto distribution or a lognormal distribution (see, e.g., [40], [65] or [99]).

This sounds esoteric but has real implications for a variety of economic phenomena.

To illustrate this fact in a simple way, let us consider an economy with 100,000 firms, an interest rate of $r = 0.05$ and a corporate tax rate of 15%.

Your task is to estimate the present discounted value of projected corporate tax revenue over the next 10 years.

Because we are forecasting, we need a model.

We will suppose that

1. the number of firms and the firm size distribution (measured in profits) remain fixed and
2. the firm size distribution is either lognormal or Pareto.

Present discounted value of tax revenue will be estimated by

1. generating 100,000 draws of firm profit from the firm size distribution,
2. multiplying by the tax rate, and
3. summing the results with discounting to obtain present value.

The Pareto distribution is assumed to take the form (5) with $\bar{x} = 1$ and $\alpha = 1.05$.

(The value the tail index α is plausible given the data [41].)

To make the lognormal option as similar as possible to the Pareto option, choose its parameters such that the mean and median of both distributions are the same.

Note that, for each distribution, your estimate of tax revenue will be random because it is based on a finite number of draws.

To take this into account, generate 100 replications (evaluations of tax revenue) for each of the two distributions and compare the two samples by

- producing a `violin plot` visualizing the two samples side-by-side and
- printing the mean and standard deviation of both samples.

For the seed use `np.random.seed(1234)`.

What differences do you observe?

(Note: a better approach to this problem would be to model firm dynamics and try to track individual firms given the current distribution. We will discuss firm dynamics in later lectures.)

7.7 Solutions

7.7.1 Exercise 1

```
In [6]: n = 120
np.random.seed(11)

fig, axes = plt.subplots(3, 1, figsize=(6, 12))

for ax in axes:
    ax.set_ylim((-120, 120))

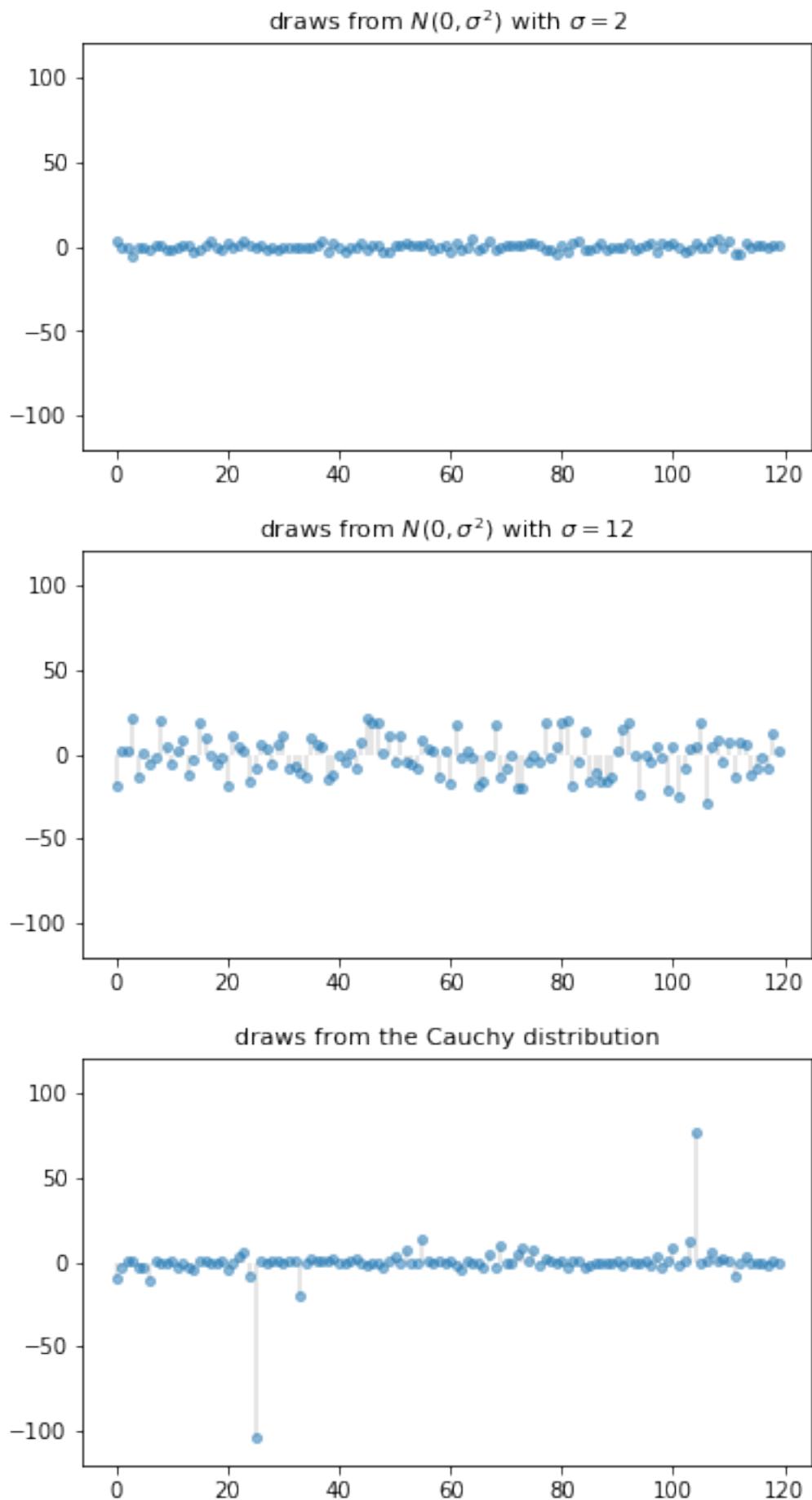
s_vals = 2, 12

for ax, s in zip(axes[:2], s_vals):
    data = np.random.randn(n) * s
    ax.plot(list(range(n)), data, linestyle=' ', marker='o', alpha=0.5, ms=4)
    ax.vlines(list(range(n)), 0, data, lw=0.2)
    ax.set_title(f"draws from $N(0, \sigma^2)$ with $\sigma = {s}$", fontsize=11)

ax = axes[2]
distribution = cauchy()
data = distribution.rvs(n)
ax.plot(list(range(n)), data, linestyle=' ', marker='o', alpha=0.5, ms=4)
ax.vlines(list(range(n)), 0, data, lw=0.2)
ax.set_title("draws from the Cauchy distribution", fontsize=11)

plt.subplots_adjust(hspace=0.25)

plt.show()
```



7.7.2 Exercise 2

Let X have a Pareto tail with tail index α and let F be its cdf.

Fix $r \geq \alpha$.

As discussed after (4), we can take positive constants b and \bar{x} such that

$$\mathbb{P}\{X > x\} \geq bx^{-\alpha} \text{ whenever } x \geq \bar{x}$$

But then

$$\mathbb{E}X^r = r \int_0^\infty x^{r-1} \mathbb{P}\{X > x\} x \geq r \int_0^{\bar{x}} x^{r-1} \mathbb{P}\{X > x\} x + r \int_{\bar{x}}^\infty x^{r-1} bx^{-\alpha} x.$$

We know that $\int_{\bar{x}}^\infty x^{r-\alpha-1} x = \infty$ whenever $r - \alpha - 1 \geq -1$.

Since $r \geq \alpha$, we have $\mathbb{E}X^r = \infty$.

7.7.3 Exercise 3

```
In [7]: from scipy.stats import pareto

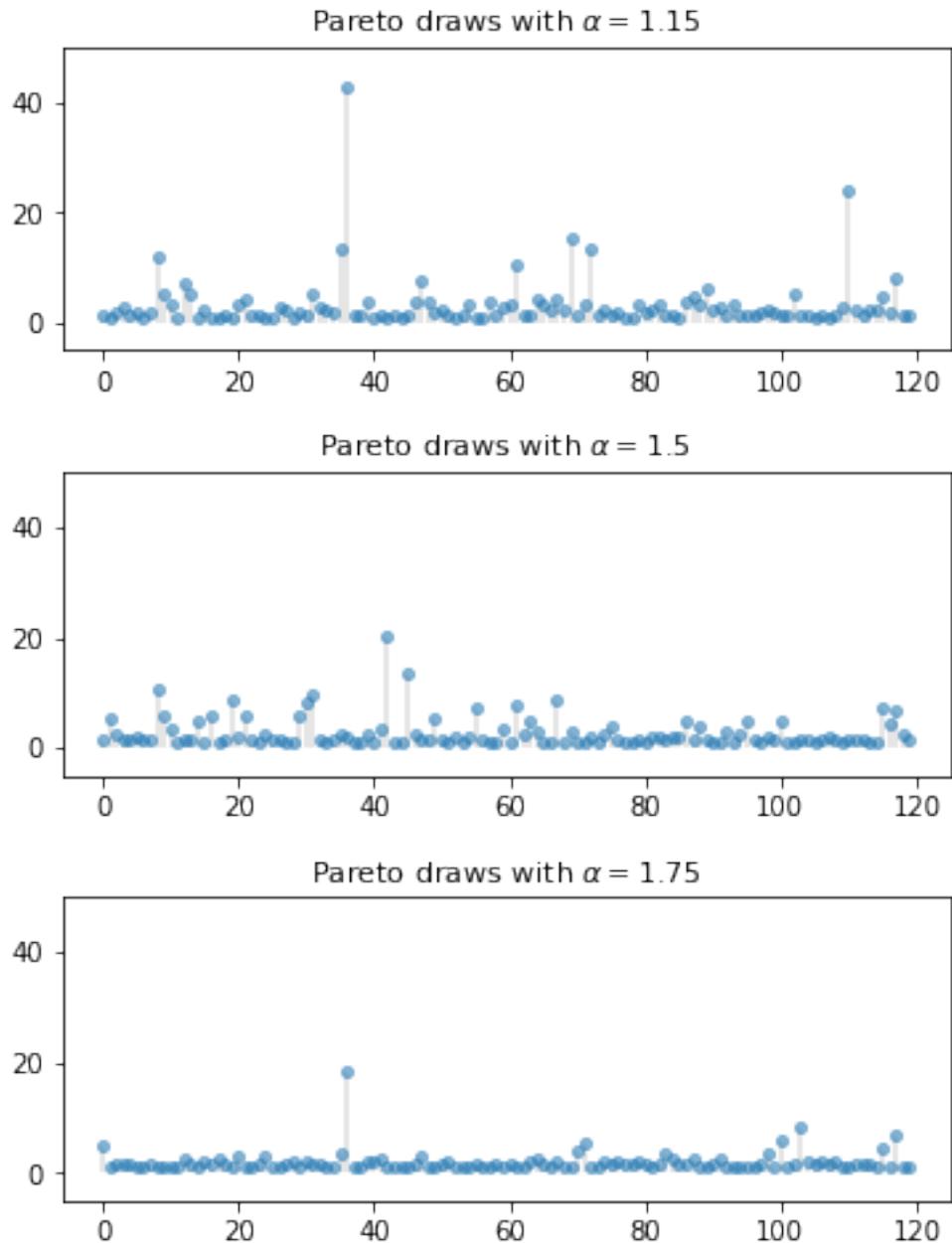
np.random.seed(11)

n = 120
alphas = [1.15, 1.50, 1.75]

fig, axes = plt.subplots(3, 1, figsize=(6, 8))

for (a, ax) in zip(alphas, axes):
    ax.set_xlim((-5, 50))
    data = pareto.rvs(size=n, scale=1, b=a)
    ax.plot(list(range(n)), data, linestyle='-', marker='o', alpha=0.5, ms=4)
    ax.vlines(list(range(n)), 0, data, lw=0.2)
    ax.set_title(f"Pareto draws with $\alpha = {a}$", fontsize=11)

plt.subplots_adjust(hspace=0.4)
plt.show()
```



7.7.4 Exercise 4

First let's generate the data for the plots:

```
In [8]: sample_size = 1000
np.random.seed(13)
z = np.random.randn(sample_size)

data_1 = np.abs(z)
data_2 = np.exp(z)
data_3 = np.exp(np.random.exponential(scale=1.0, size=sample_size))

data_list = [data_1, data_2, data_3]
```

Now we plot the data:

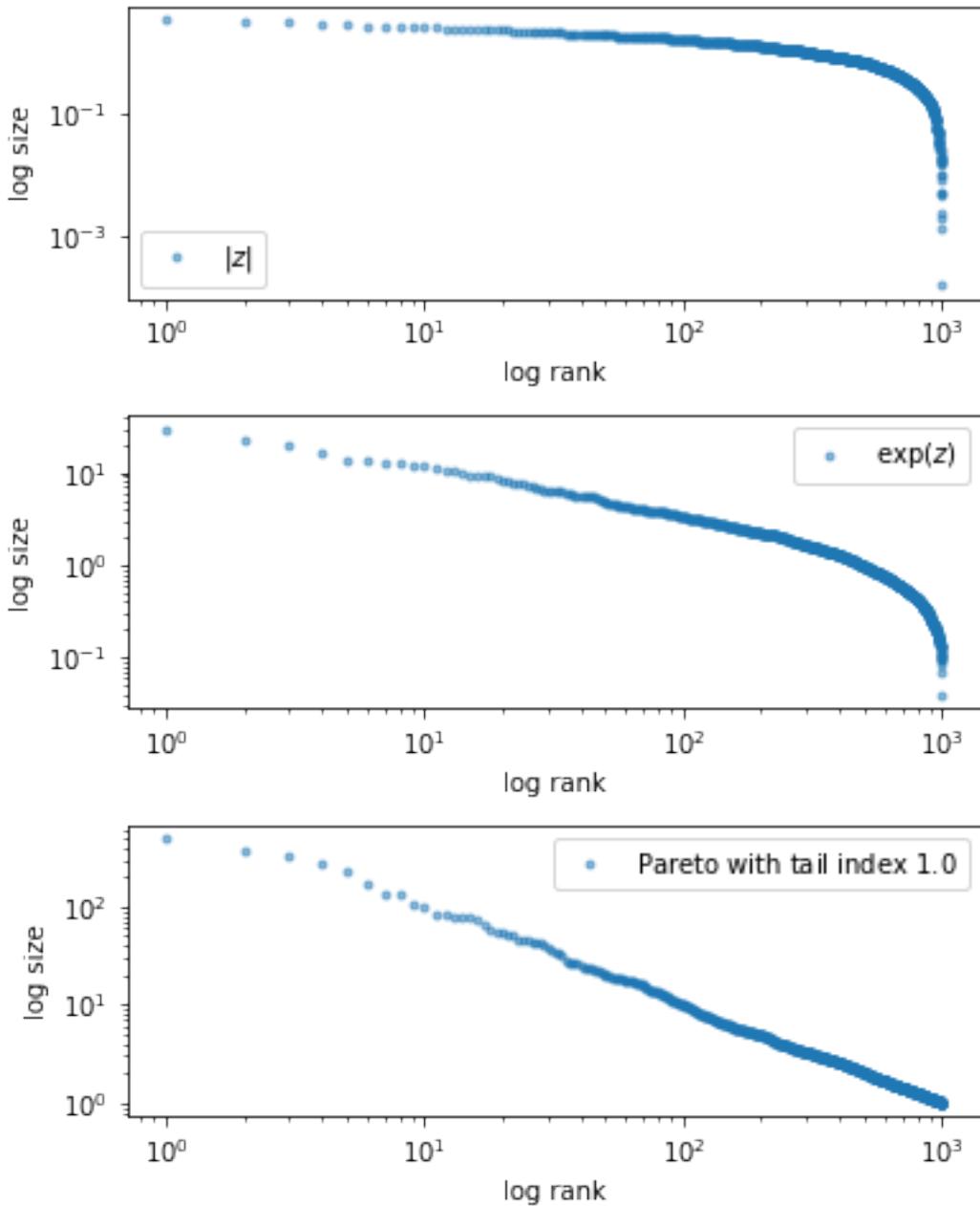
```
In [9]: fig, axes = plt.subplots(3, 1, figsize=(6, 8))
axes = axes.flatten()
labels = ['$|z|$', '$\exp(z)$', 'Pareto with tail index $1.0$']

for data, label, ax in zip(data_list, labels, axes):
    rank_data, size_data = qe.rank_size(data)

    ax.loglog(rank_data, size_data, 'o', markersize=3.0, alpha=0.5, label=label)
    ax.set_xlabel("log rank")
    ax.set_ylabel("log size")

    ax.legend()

fig.subplots_adjust(hspace=0.4)
plt.show()
```



7.7.5 Exercise 5

To do the exercise, we need to choose the parameters μ and σ of the lognormal distribution to match the mean and median of the Pareto distribution.

Here we understand the lognormal distribution as that of the random variable $\exp(\mu + \sigma Z)$ when Z is standard normal.

The mean and median of the Pareto distribution (5) with $\bar{x} = 1$ are

$$\text{mean} = \frac{\alpha}{\alpha - 1} \quad \text{and} \quad \text{median} = 2^{1/\alpha}$$

Using the corresponding expressions for the lognormal distribution leads us to the equations

$$\frac{\alpha}{\alpha - 1} = \exp(\mu + \sigma^2/2) \quad \text{and} \quad 2^{1/\alpha} = \exp(\mu)$$

which we solve for μ and σ given $\alpha = 1.05$.

Here is code that generates the two samples, produces the violin plot and prints the mean and standard deviation of the two samples.

```
In [10]: num_firms = 100_000
        num_years = 10
        tax_rate = 0.15
        r = 0.05

        β = 1 / (1 + r)      # discount factor

        x_bar = 1.0
        α = 1.05

        def pareto_rvs(n):
            "Uses a standard method to generate Pareto draws."
            u = np.random.uniform(size=n)
            y = x_bar / (u**(1/α))
            return y
```

Let's compute the lognormal parameters:

```
In [11]: μ = np.log(2) / α
        σ_sq = 2 * (np.log(α/(α - 1)) - np.log(2)/α)
        σ = np.sqrt(σ_sq)
```

Here's a function to compute a single estimate of tax revenue for a particular choice of distribution `dist`.

```
In [12]: def tax_rev(dist):
        tax_raised = 0
        for t in range(num_years):
            if dist == 'pareto':
                π = pareto_rvs(num_firms)
            else:
                π = np.exp(μ + σ * np.random.randn(num_firms))
            tax_raised += β**t * np.sum(π * tax_rate)
        return tax_raised
```

Now let's generate the violin plot.

```
In [13]: num_reps = 100
        np.random.seed(1234)

        tax_rev_lognorm = np.empty(num_reps)
        tax_rev_pareto = np.empty(num_reps)

        for i in range(num_reps):
```

```

tax_rev_pareto[i] = tax_rev('pareto')
tax_rev_lognorm[i] = tax_rev('lognorm')

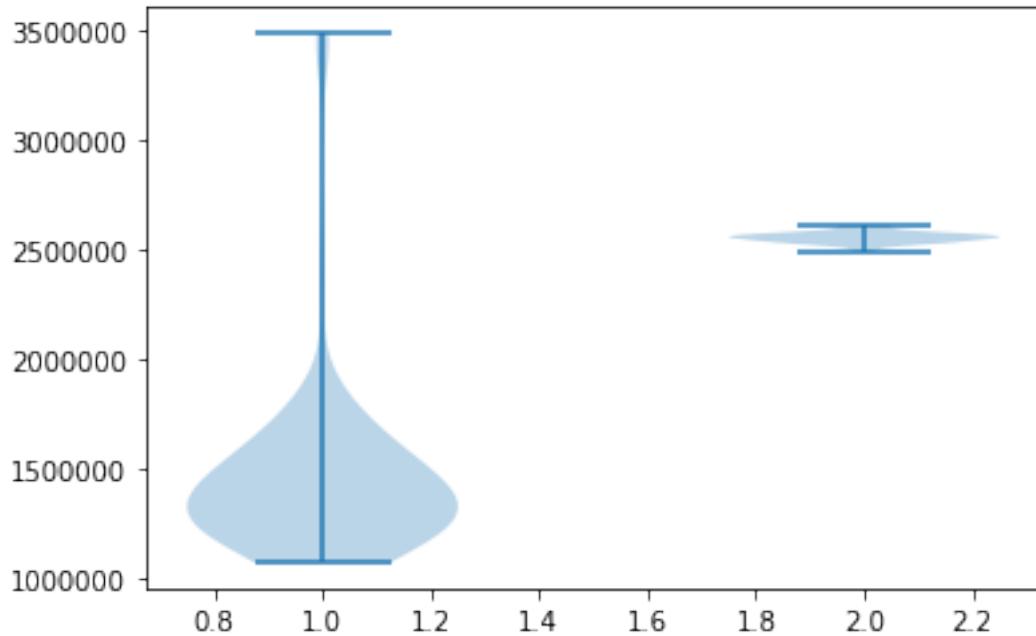
fig, ax = plt.subplots()

data = tax_rev_pareto, tax_rev_lognorm

ax.violinplot(data)

plt.show()

```



Finally, let's print the means and standard deviations.

In [14]: `tax_rev_pareto.mean(), tax_rev_pareto.std()`

Out[14]: (1458729.0546623734, 406089.3613661567)

In [15]: `tax_rev_lognorm.mean(), tax_rev_lognorm.std()`

Out[15]: (2556174.8615230713, 25586.44456513965)

Looking at the output of the code, our main conclusion is that the Pareto assumption leads to a lower mean and greater dispersion.

Chapter 8

Multivariate Normal Distribution

8.1 Contents

- Overview 8.2
- The Multivariate Normal Distribution 8.3
- Bivariate Example 8.4
- Trivariate Example 8.5
- One Dimensional Intelligence (IQ) 8.6
- Another representation 8.7
- Magic of the Cholesky factorization 8.8
- Math and Verbal Components of Intelligence 8.9
- Univariate Time Series Analysis 8.10
- Classic Factor Analysis Model 8.11
- PCA as Approximation to Factor Analytic Model 8.12
- Stochastic Difference Equation 8.13
- Application to Stock Price Model 8.14
- Filtering Foundations 8.15

8.2 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 classic models:

- a **factor analytic model** of an intelligence quotient, i.e., IQ
- a **factor analytic model** of two independent inherent abilities, mathematical and verbal.
- a more general factor analytic model
- PCA as an approximation to a factor analytic model
- time series generated by linear stochastic difference equations
- optimal linear filtering theory

8.3 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 classic examples.

We will use the following imports:

```
In [1]: import numpy as np
        from numba import njit
        import statsmodels.api as sm
        import matplotlib.pyplot as plt
        %matplotlib inline
```

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 $\mu = E z$ is the mean of the random vector z and $\Sigma = E(z - \mu)(z - \mu)'$ is the covariance matrix of z .

```
In [2]: @njit
def f(z, mu, Sigma):
    """
    The density function of multivariate normal distribution.

    Parameters
    -----
    z: ndarray(float, dim=2)
        random vector, N by 1
    mu: ndarray(float, dim=1 or 2)
        the mean of z, N by 1
    Sigma: ndarray(float, dim=2)
        the covariance matrix of z, N by 1
    """

    z = np.atleast_2d(z)
    mu = np.atleast_2d(mu)
    Sigma = np.atleast_2d(Sigma)

    N = z.size

    temp1 = np.linalg.det(Sigma) ** (-1/2)
    temp2 = np.exp(-.5 * (z - mu).T @ np.linalg.inv(Sigma) @ (z - mu))

    return (2 * np.pi) ** (-N/2) * temp1 * temp2
```

For some integer $k \in \{2, \dots, N - 1\}$, partition z as $z = \begin{bmatrix} z_1 \\ z_2 \end{bmatrix}$, where z_1 is an $(N - k) \times 1$ vector and z_2 is a $k \times 1$ vector.

Let

$$\mu = \begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix}, \quad \Sigma = \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix}$$

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

$$\hat{\mu}_1 = \mu_1 + \beta(z_2 - \mu_2)$$

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 $z_1 - \mu_1$ on $z_2 - \mu_2$.

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

```
In [3]: class MultivariateNormal:
    """
    Class of multivariate normal distribution.

    Parameters
    -----
    μ: ndarray(float, dim=1)
        the mean of  $z$ ,  $N$  by 1
    Σ: ndarray(float, dim=2)
        the covariance matrix of  $z$ ,  $N$  by  $N$ 

    Arguments
    -----
    μ, Σ:
        see parameters
```

```

 $\mu_s$ : list(ndarray(float, dim=1))
    list of mean vectors  $\mu_1$  and  $\mu_2$  in order
 $\Sigma_s$ : list(list(ndarray(float, dim=2)))
    2 dimensional list of covariance matrices
         $\Sigma_{11}$ ,  $\Sigma_{12}$ ,  $\Sigma_{21}$ ,  $\Sigma_{22}$  in order
 $\beta_s$ : list(ndarray(float, dim=1))
    list of regression coefficients  $\beta_1$  and  $\beta_2$  in order
"""

def __init__(self, mu, Sigma):
    "initialization"
    self.mu = np.array(mu)
    self.Sigma = np.atleast_2d(Sigma)

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 mu into
    mu1 and mu2, and the covariance matrix Sigma into Sigma11, Sigma12, Sigma21, Sigma22
    correspondingly. Compute the regression coefficients beta1 and beta2
    using the partitioned arrays.
    """
    mu = self.mu
    Sigma = self.Sigma

    self.mu_s = [mu[:k], mu[k:]]
    self.Sigma_s = [[Sigma[:k, :k], Sigma[:k, k:]],
                   [Sigma[k:, :k], Sigma[k:, k:]]]

    self.beta_s = [self.Sigma_s[0][1] @ np.linalg.inv(self.Sigma_s[1][1]),
                  self.Sigma_s[1][0] @ np.linalg.inv(self.Sigma_s[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
    -----
    mu_hat: ndarray(float, ndim=1)
        The conditional mean of z1 or z2.
    Sigma_hat: ndarray(float, ndim=2)
        The conditional covariance matrix of z1 or z2.
    """
    beta = self.beta_s[ind]
    mu_s = self.mu_s
    Sigma_s = self.Sigma_s

    mu_hat = mu_s[ind] + beta @ (z - mu_s[1-ind])
    Sigma_hat = Sigma_s[ind][ind] - beta @ Sigma_s[1-ind][1-ind] @ beta.T

    return mu_hat, Sigma_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.

Then for fun we'll 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.

8.4 Bivariate Example

We start with a bivariate normal distribution pinned down by

$$\mu = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad \Sigma = \begin{bmatrix} 1 & .2 \\ .2 & 1 \end{bmatrix}$$

```
In [4]: μ = np.array([0., 0.])
Σ = np.array([[1., .2], [.2, 1.]])

# construction of the multivariate normal instance
multi_normal = MultivariateNormal(μ, Σ)
```

```
In [5]: k = 1 # choose partition

# partition and compute regression coefficients
multi_normal.partition(k)
multi_normal.βs[0]
```

```
Out[5]: array([[0.2]])
```

Let's compute the mean and variance of the distribution of z_1 conditional on $z_2 = 5$.

```
In [6]: # 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 =  [1.] [[0.96]]
```

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

$$z_1 - \mu_1 = \beta(z_2 - \mu_2) + \epsilon,$$

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 $\hat{\Sigma}_1$.

In [7]: `n = 1_000_000 # sample size`

```
# simulate multivariate normal random vectors
data = np.random.multivariate_normal(mu, Sigma, size=n)
z1_data = data[:, 0]
z2_data = data[:, 1]

# OLS regression
mu1, mu2 = multi_normal.mus
results = sm.OLS(z1_data - mu1, z2_data - mu2).fit()
```

Let's compare the preceding population β with the OLS sample estimate on $z_2 - \mu_2$

In [8]: `multi_normal.betas[0], results.params`

Out[8]: `(array([[0.2]]), array([0.20036371]))`

Let's compare our population $\hat{\Sigma}_1$ with the degrees-of-freedom adjusted estimate of the variance of ϵ

In [9]: `Sigma_hat, results.resid @ results.resid.T / (n - 1)`

Out[9]: `(array([[0.96]]), 0.959416163313824)`

Lastly, let's compute the estimate of $E\hat{z}_1|z_2$ and compare it with $\hat{\mu}_1$

In [10]: `mu1_hat, results.predict(z2 - mu2) + mu1`

Out[10]: `(array([1.]), array([1.00181854]))`

Thus, in each case, for our very large sample size, the sample analogues closely approximate their population counterparts.

These close approximations are foretold by a version of a Law of Large Numbers.

8.5 Trivariate Example

Let's apply our code to a trivariate example.

We'll specify the mean vector and the covariance matrix as follows.

```
In [11]: mu = np.random.random(3)
C = np.random.random((3, 3))
Sigma = C @ C.T # positive semi-definite

multi_normal = MultivariateNormal(mu, Sigma)
```

In [12]: μ, Σ

```
Out[12]: (array([0.49949752, 0.43556324, 0.95564614]),
           array([[0.26223972, 0.5116022 , 0.5188636 ],
                  [0.5116022 , 1.21632313, 1.01434603],
                  [0.5188636 , 1.01434603, 1.02988457]]))
```

In [13]: $k = 1$
 $\text{multi_normal.partition}(k)$

Let's compute the distribution of z_1 conditional on $z_2 = \begin{bmatrix} 2 \\ 5 \end{bmatrix}$.

In [14]: $ind = 0$
 $z2 = np.array([2., 5.])$
 $\mu1_hat, \Sigma1_hat = \text{multi_normal.cond_dist}(ind, z2)$

In [15]: $n = 1_000_000$
 $\text{data} = np.random.multivariate_normal(\mu, \Sigma, size=n)$
 $z1_data = \text{data}[:, :k]$
 $z2_data = \text{data}[:, k:]$

In [16]: $\mu1, \mu2 = \text{multi_normal}\mu\varsigma$
 $\text{results} = sm.OLS(z1_data - \mu1, z2_data - \mu2).fit()$

As above, we compare population and sample regression coefficients, the conditional covariance matrix, and the conditional mean vector in that order.

In [17]: $\text{multi_normal}\beta\varsigma[0], \text{results.params}$

```
Out[17]: (array([[0.00260966, 0.50123724]]), array([0.00256933, 0.50129454]))
```

In [18]: $\Sigma1_hat, \text{results.resid} @ \text{results.resid.T} / (n - 1)$

```
Out[18]: (array([[0.00083085]]), 0.0008300920623381422)
```

In [19]: $\mu1_hat, \text{results.predict}(z2 - \mu2) + \mu1$

```
Out[19]: (array([2.53076095]), array([2.53092959]))
```

Once again, sample analogues do a good job of approximating their populations counterparts.

8.6 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 i th 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 the noise in the test scores is IID and not correlated with IQ.

In particular, we assume $\{w_i\}_{i=1}^{n+1}$ are i.i.d. standard normal:

$$w = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \\ w_{n+1} \end{bmatrix} \sim N(0, I_{n+1})$$

The following system describes the 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 .

In [20]: `def construct_moments_IQ(n, mu_theta, sigma_theta, sigma_y):`

```

mu_IQ = np.ones(n+1) * mu_theta

D_IQ = np.zeros((n+1, n+1))
D_IQ[range(n), range(n)] = sigma_y
D_IQ[:, n] = sigma_theta

Sigma_IQ = D_IQ @ D_IQ.T

return mu_IQ, Sigma_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.

```
In [21]: n = 50  
        mu_theta, sigma_theta, sigma_y = 100., 10., 10.  
  
        mu_IQ, Sigma_IQ, D_IQ = construct_moments_IQ(n, mu_theta, sigma_theta, sigma_y)  
        mu_IQ, Sigma_IQ, D_IQ
```

We can now use our `MultivariateNormal` class to construct an instance, then partition the mean vector and covariance matrix as we wish.

We choose $k=n$ so that $z_1 = y$ and $z_2 = \theta$.

```
In [22]: multi_normal_IQ = MultivariateNormal(mu_IQ, Sigma_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

```
In [23]: x = np.random.multivariate_normal(mu_IQ, Sigma_IQ)
        y = x[:-1] # test scores
        theta = x[-1] # IQ
```

In [24]: # the true value
θ

```
Out[24]: 113.24907291423337
```

The method `cond_dist` takes test scores as input and returns the conditional normal distribution of the IQ θ .

Note that now θ is what we denoted as z_2 in the general case so we need to set `ind=1`.

```
In [25]: ind = 1
        multi_normal_IQ.cond_dist(ind, y)
```

```
Out[25]: (array([112.10020058]), 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 the 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.

```
In [26]: # 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]

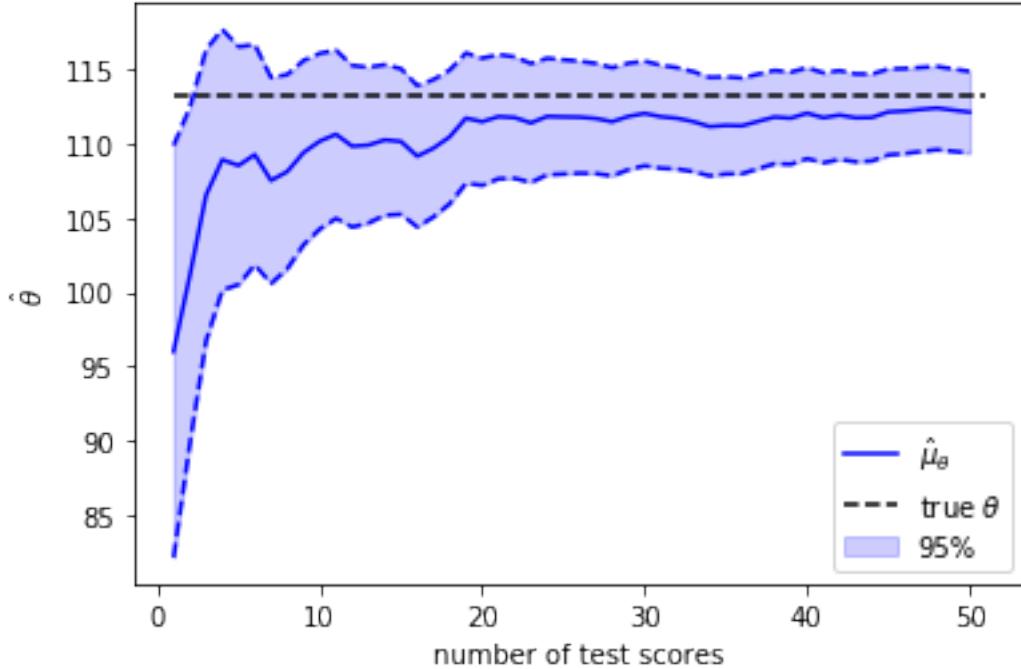
        # transform variance to standard deviation
        σθ_hat_arr = np.sqrt(Σθ_hat_arr)
```

```
In [27]: 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='$\hat{\mu}_\theta$')
        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('$\hat{\mu}_\theta$')
        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 deducing $1.96\hat{\sigma}_\theta$ 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 drove the number of tests $n \rightarrow +\infty$, the conditional standard deviation $\hat{\sigma}_\theta$ would converge to 0 at the rate $\frac{1}{n^{1/2}}$.

8.7 Another representation

By using a different representation, let's look at things from a different perspective.

We can represent the random vector X defined above as

$$X = \mu_\theta 1_{n+1} + C\epsilon, \quad \epsilon \sim N(0, I)$$

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

$$\epsilon = G(X - \mu_\theta 1_{n+1})$$

This formula confirms that the orthonormal vector ϵ contains the same information as the non-orthogonal vector $(X - \mu_\theta 1_{n+1})$.

We can say that ϵ is an orthogonal basis for $(X - \mu_\theta 1_{n+1})$.

Let c_i be the i th element in the last row of C .

Then we can write

$$\theta = \mu_\theta + c_1\epsilon_1 + c_2\epsilon_2 + \cdots + c_n\epsilon_n + c_{n+1}\epsilon_{n+1} \quad (1)$$

The mutual orthogonality of the ϵ_i 's provides us with an informative way to interpret them in light of equation (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 (1) also provides us with an enlightening way to express conditional means and conditional variances that we computed earlier.

In particular,

$$E[\theta | y_1, \dots, y_k] = \mu_\theta + c_1\epsilon_1 + \cdots + c_k\epsilon_k$$

and

$$Var(\theta | y_1, \dots, y_k) = c_{k+1}^2 + c_{k+2}^2 + \cdots + c_{n+1}^2.$$

```
In [28]: C = np.linalg.cholesky(Sigma_IQ)
G = np.linalg.inv(C)

ε = G @ (x - μθ)
```

```
In [29]: 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([np.sum(C[n, i+1:n+1]**2) 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.

In [30]: # conditional mean

```
np.max(np.abs(mu_theta_hat_arr - mu_theta_hat_arr_C)) < 1e-10
```

Out[30]: True

In [31]: # conditional variance

```
np.max(np.abs(Sigma_theta_hat_arr - Sigma_theta_hat_arr_C)) < 1e-10
```

Out[31]: True

8.8 Magic of the Cholesky factorization

Evidently, the Cholesky factorization is automatically computing the population **regression coefficients** and associated statistics that are produced by our `MultivariateNormal` class.

The Cholesky factorization is computing things **recursively**.

Indeed, in formula (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

8.9 Math and Verbal Components of 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.

In [32]: `def construct_moments_IQ2d(n, mu_theta, sigma_theta, mu_eta, sigma_eta, sigma_y):`

```

mu_IQ2d = np.empty(2*(n+1))
mu_IQ2d[:n] = mu_theta
mu_IQ2d[2*n] = mu_theta
mu_IQ2d[n:2*n] = mu_eta
mu_IQ2d[2*n+1] = mu_eta

D_IQ2d = np.zeros((2*(n+1), 2*(n+1)))
D_IQ2d[range(2*n), range(2*n)] = sigma_y
D_IQ2d[:n, 2*n] = sigma_theta
D_IQ2d[2*n, 2*n] = sigma_theta
D_IQ2d[n:2*n, 2*n+1] = sigma_eta
D_IQ2d[2*n+1, 2*n+1] = sigma_eta

Sigma_IQ2d = D_IQ2d @ D_IQ2d.T

return mu_IQ2d, Sigma_IQ2d, D_IQ2d

```

Let's put the function to work.

In [33]: `n = 2`

```

# mean and variance of theta, eta, and y
mu_theta, sigma_theta, mu_eta, sigma_eta, sigma_y = 100., 10., 100., 10, 10

mu_IQ2d, Sigma_IQ2d, D_IQ2d = construct_moments_IQ2d(n, mu_theta, sigma_theta, mu_eta, sigma_eta, sigma_y)
mu_IQ2d, Sigma_IQ2d, D_IQ2d

```

Out[33]: (`array([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., 10., 0.],`
`[0., 10., 0., 0., 10., 0.],`
`[0., 0., 10., 0., 0., 10.],`
`[0., 0., 0., 10., 0., 10.],`
`[0., 0., 0., 0., 10., 0.],`
`[0., 0., 0., 0., 0., 10.]])`)

In [34]: `# take one draw`

```

x = np.random.multivariate_normal(mu_IQ2d, Sigma_IQ2d)
y1 = x[:n]
y2 = x[n:2*n]
theta = x[2*n]

```

```
 $\eta = x[2*n+1]$ 
```

```
# the true values  
 $\theta, \eta$ 
```

Out[34]: (107.5379725643408, 99.94327875756062)

We first compute the joint normal distribution of (θ, η) .

In [35]: `multi_normal_IQ2d = MultivariateNormal(mu_IQ2d, Sigma_IQ2d)`

```
k = 2*n # the length of data vector  
multi_normal_IQ2d.partition(k)  
  
multi_normal_IQ2d.cond_dist(1, [*y1, *y2])
```

Out[35]: (array([108.18279866, 97.56009779]), 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.

In [36]: `def cond_dist_IQ2d(mu, Sigma, data):`

```
n = len(mu)  
  
multi_normal = MultivariateNormal(mu, Sigma)  
multi_normal.partition(n-1)  
mu_hat, Sigma_hat = multi_normal.cond_dist(1, data)  
  
return mu_hat, Sigma_hat
```

Let's see how things work for an example.

In [37]: `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')]:

 mu_hat, Sigma_hat = cond_dist_IQ2d(mu_IQ2d[indices], Sigma_IQ2d[indices][:, ↴indices],
 x[indices[:-1]])
 print(f'The mean and variance of {IQ} conditional on {conditions}:
↪<15> are '+
 f'{mu_hat[0]:1.2f} and {Sigma_hat[0, 0]:1.2f} respectively')`

The mean and variance of θ conditional on y_1, y_2, y_3, y_4 are 108.18 and 33.33 respectively

The mean and variance of θ conditional on y_1, y_2 respectively	are 108.18 and 33.33
The mean and variance of θ conditional on y_3, y_4 respectively	are 100.00 and 100.00
The mean and variance of η conditional on y_1, y_2, y_3, y_4 respectively	are 97.56 and 33.33
The mean and variance of η conditional on y_1, y_2 respectively	are 100.00 and 100.00
The mean and variance of η conditional on y_3, y_4 respectively	are 97.56 and 33.33

Evidently, math tests provide no information about μ and language tests provide no information about η .

8.10 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:

$$\begin{aligned}x_0 &\sim N(0, \sigma_0^2) \\x_{t+1} &= ax_t + bw_{t+1}, \quad w_{t+1} \sim N(0, 1), t \geq 0 \\y_t &= cx_t + dv_t, \quad v_t \sim N(0, 1), t \geq 0\end{aligned}$$

We can compute the moments of x_t

1. $Ex_{t+1}^2 = a^2Ex_t^2 + b^2, t \geq 0$, where $Ex_0^2 = \sigma_0^2$
2. $Ex_{t+j}x_t = a^jEx_t^2, \forall t \forall j$

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

$$\Sigma_y = EYY' = C\Sigma_x C' + DD'$$

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 $N(0, \Sigma_z)$.

```
In [38]: # as an example, consider the case where T = 3
T = 3
```

```
In [39]: # variance of the initial distribution x_0
sigma0 = 1.
```

```
# parameters of the equation system
a = .9
b = 1.
c = 1.0
d = .05
```

```
In [40]: # construct the covariance matrix of X
SigmaX = np.empty((T+1, T+1))

SigmaX[0, 0] = sigma0 ** 2
for i in range(T):
    SigmaX[i, i+1:] = SigmaX[i, i] * a ** np.arange(1, T+1-i)
    SigmaX[i+1:, i] = SigmaX[i, i+1:]

SigmaX[i+1, i+1] = a ** 2 * SigmaX[i, i] + b ** 2
```

```
In [41]: SigmaX
```

```
Out[41]: 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]])
```

```
In [42]: # 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
```

```
In [43]: # 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
```

```
In [44]: Σz
```

```
Out[44]: 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]])
```

```
In [45]: # 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.

```
In [46]: z = np.random.multivariate_normal(μz, Σz)

x = z[:T+1]
y = z[T+1:]
```

8.10.1 Smoothing Example

This is an instance of a classic **smoothing** calculation whose purpose is to compute $EX | Y$.

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

```
In [47]: # construct a MultivariateNormal instance
multi_normal_ex1 = MultivariateNormal(mu_z, Sigma_z)
x = z[:T+1]
y = z[T+1:]
```

```
In [48]: # partition Z into X and Y
multi_normal_ex1.partition(T+1)
```

```
In [49]: # 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 = [0.35046244 0.9239733 1.13800503 1.05240652]
Y = [0.31648301 0.86110137 1.11133608 1.06970771]
E [ X | Y] =
```

```
Out[49]: (array([0.3169846 , 0.86042015, 1.11065278, 1.06953285]),
 array([[2.48875094e-03, 5.57449314e-06, 1.24861729e-08, 2.80235835e-11],
 [5.57449314e-06, 2.48876343e-03, 5.57452116e-06, 1.25113941e-08],
 [1.24861729e-08, 5.57452116e-06, 2.48876346e-03, 5.58575339e-06],
 [2.80235835e-11, 1.25113941e-08, 5.58575339e-06, 2.49377812e-03]]))
```

8.10.2 Filtering Exercise

Compute $E[x_t | y_{t-1}, y_{t-2}, \dots, y_0]$.

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 .

```
In [50]: t = 3
```

```
In [51]: # mean of the subvector
sub_mu_z = np.zeros(t+1)

# covariance matrix of the subvector
sub_Sigma_z = np.empty((t+1, t+1))

sub_Sigma_z[0, 0] = Sigma_z[t, t] # x_t
sub_Sigma_z[0, 1:] = Sigma_z[t, T+1:T+t+1]
sub_Sigma_z[1:, 0] = Sigma_z[T+1:T+t+1, t]
sub_Sigma_z[1:, 1:] = Sigma_z[T+1:T+t+1, T+1:T+t+1]
```

```
In [52]: sub_Sigma_z
```

```
Out[52]: 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 ]])
```

```
In [53]: multi_normal_ex2 = MultivariateNormal(sub_mu_z, sub_Sigma_z)
multi_normal_ex2.partition(1)
```

```
In [54]: sub_y = y[:t]
multi_normal_ex2.cond_dist(0, sub_y)
```

```
Out[54]: (array([0.99944621]), array([[1.00201996]]))
```

8.10.3 Prediction Exercise

Compute $E[y_t | y_{t-j}, \dots, y_0]$.

As what we did in exercise 2, we will construct the mean vector and covariance matrix of the subvector $[y_t, y_0, \dots, y_{t-j-1}, y_{t-j}]$.

For example, we take a case in which $t = 3$ and $j = 2$.

```
In [55]: t = 3
j = 2
```

```
In [56]: sub_mu_z = np.zeros(t-j+2)
sub_Sigma_z = np.empty((t-j+2, t-j+2))

sub_Sigma_z[0, 0] = Sigma_z[T+t+1, T+t+1]
sub_Sigma_z[0, 1:] = Sigma_z[T+t+1, T+1:T+t-j+2]
sub_Sigma_z[1:, 0] = Sigma_z[T+1:T+t-j+2, T+t+1]
sub_Sigma_z[1:, 1:] = Sigma_z[T+1:T+t-j+2, T+1:T+t-j+2]
```

```
In [57]: sub_Sigma_z
```

```
Out[57]: array([[3.000041, 0.729, 1.4661],
[0.729, 1.0025, 0.9],
[1.4661, 0.9, 1.8125]])
```

```
In [58]: multi_normal_ex3 = MultivariateNormal(sub_mu_z, sub_Sigma_z)
multi_normal_ex3.partition(1)
```

```
In [59]: sub_y = y[:t-j+1]
multi_normal_ex3.cond_dist(0, sub_y)
```

```
Out[59]: (array([0.69632899]), array([[1.81413617]]))
```

8.10.4 Constructing a Wold Representation

Now we'll apply Cholesky decomposition to decompose $\Sigma_y = HH'$ and form

$$\epsilon = H^{-1}Y.$$

Then we can represent y_t as

$$y_t = h_{t,t}\epsilon_t + h_{t,t-1}\epsilon_{t-1} + \dots + h_{t,0}\epsilon_0.$$

```
In [60]: H = np.linalg.cholesky(Sigma)
H

Out[60]: 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]]))

In [61]: epsilon = np.linalg.inv(H) @ y
epsilon

Out[61]: array([0.31608815, 0.57567742, 0.33687673, 0.07010325])

In [62]: y
y

Out[62]: array([0.31648301, 0.86110137, 1.11133608, 1.06970771])
```

This example is an instance of what is known as a **Wold representation** in time series analysis.

8.11 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 $U \perp f$.
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 $k = 2$.

```
In [63]: N = 10
         k = 2
```

We set the coefficient matrix Λ and the covariance matrix of U to be

$$\Lambda = \begin{pmatrix} 1 & 0 \\ \vdots & \vdots \\ 1 & 0 \\ 0 & 1 \\ \vdots & \vdots \\ 0 & 1 \end{pmatrix}, \quad D = \begin{pmatrix} \sigma_u^2 & 0 & \cdots & 0 \\ 0 & \sigma_u^2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_u^2 \end{pmatrix}$$

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.

```
In [64]: Λ = np.zeros((N, k))
          Λ[:N//2, 0] = 1
          Λ[N//2:, 1] = 1

σu = .5
D = np.eye(N) * σu ** 2
```

```
In [65]: # compute Σy
Σy = Λ @ Λ.T + D
```

We can now construct the mean vector and the covariance matrix for Z .

```
In [66]: μz = np.zeros(k+N)
          Σz = np.empty((k+N, k+N))

Σz[:, :k] = np.eye(k)
Σz[:, k:] = Λ.T
Σz[k:, :k] = Λ
Σz[k:, k:] = Σy
```

```
In [67]: z = np.random.multivariate_normal(mu_z, Sigma_z)
f = z[:k]
y = z[k:]
```

```
In [68]: multi_normal_factor = MultivariateNormal(mu_z, Sigma_z)
multi_normal_factor.partition(k)
```

Let's compute the conditional distribution of the hidden factor f on the observations Y , namely, $f | Y = y$.

```
In [69]: multi_normal_factor.cond_dist(0, y)
```

```
Out[69]: (array([-0.2097539,  0.38269762]), array([[0.04761905, 0.          ],
[0.          , 0.04761905]]))
```

We can verify that the conditional mean $E[f | Y = y] = BY$ where $B = \Lambda' \Sigma_y^{-1}$.

```
In [70]: B = Lambda.T @ np.linalg.inv(Sigma_y)
```

```
B @ y
```

```
Out[70]: array([-0.2097539,  0.38269762])
```

Similarly, we can compute the conditional distribution $Y | f$.

```
In [71]: multi_normal_factor.cond_dist(1, f)
```

```
Out[71]: (array([-0.36090544, -0.36090544, -0.36090544, -0.36090544, -0.36090544,
0.3911745, 0.3911745, 0.3911745, 0.3911745, 0.3911745]),
array([[0.25, 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
[0., 0.25, 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
[0., 0., 0.25, 0., 0., 0., 0., 0., 0., 0., 0., 0.],
[0., 0., 0., 0.25, 0., 0., 0., 0., 0., 0., 0., 0.],
[0., 0., 0., 0., 0.25, 0., 0., 0., 0., 0., 0., 0.],
[0., 0., 0., 0., 0., 0.25, 0., 0., 0., 0., 0., 0.],
[0., 0., 0., 0., 0., 0., 0.25, 0., 0., 0., 0., 0.],
[0., 0., 0., 0., 0., 0., 0., 0.25, 0., 0., 0., 0.],
[0., 0., 0., 0., 0., 0., 0., 0., 0.25, 0., 0., 0.],
[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.25, 0., 0.],
[0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.25, 0.],
[0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.25]]))
```

It can be verified that the mean is $\Lambda I^{-1} f = \Lambda f$.

```
In [72]: Lambda @ f
```

```
Out[72]: array([-0.36090544, -0.36090544, -0.36090544, -0.36090544, -0.36090544,
0.3911745, 0.3911745, 0.3911745, 0.3911745, 0.3911745])
```

8.12 PCA as Approximation to Factor Analytic Model

For fun, let's apply a Principal Components Analysis (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 $E f_i | Y$ 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.

In [73]: `l_tilde, P = np.linalg.eigh(Sy)`

```
# arrange the eigenvectors by eigenvalues
ind = sorted(range(N), key=lambda x: l_tilde[x], reverse=True)

P = P[:, ind]
l_tilde = l_tilde[ind]
Lambda_tilde = np.diag(l_tilde)

print('l_tilde =', l_tilde)
```

`l_tilde = [5.25 5.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25]`

In [74]: `# verify the orthogonality of eigenvectors`
`np.abs(P @ P.T - np.eye(N)).max()`

Out[74]: 2.7639210781816965e-16

In [75]: `# verify the eigenvalue decomposition is correct`
`P @ Lambda_tilde @ P.T`

Out[75]: `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. , 0. ],
[1. , 1. , 1. , 1.25, 1. , 0. , 0. , 0. , 0. , 0. , 0. ],
[1. , 1. , 1. , 1. , 1.25, 0. , 0. , 0. , 0. , 0. , 0. ],
[0. , 0. , 0. , 0. , 0. , 1.25, 1. , 1. , 1. , 1. , 1. ],
[0. , 0. , 0. , 0. , 0. , 1. , 1.25, 1. , 1. , 1. , 1. ],
[0. , 0. , 0. , 0. , 0. , 1. , 1. , 1.25, 1. , 1. , 1. ],
[0. , 0. , 0. , 0. , 0. , 1. , 1. , 1. , 1.25, 1. , 1. ],
[0. , 0. , 0. , 0. , 0. , 1. , 1. , 1. , 1. , 1.25, 1. ],
[0. , 0. , 0. , 0. , 0. , 1. , 1. , 1. , 1. , 1. , 1.25]])
```

In [76]: $\epsilon = P.T @ y$

```
print("epsilon = ", epsilon)
```

```
epsilon = [-0.49247519  0.8985248   0.82499711 -0.7731567   0.0996241   -0.39475391
 0.60484801  0.72475496   0.27134959 -0.01516356]
```

In [77]: # print the values of the two factors

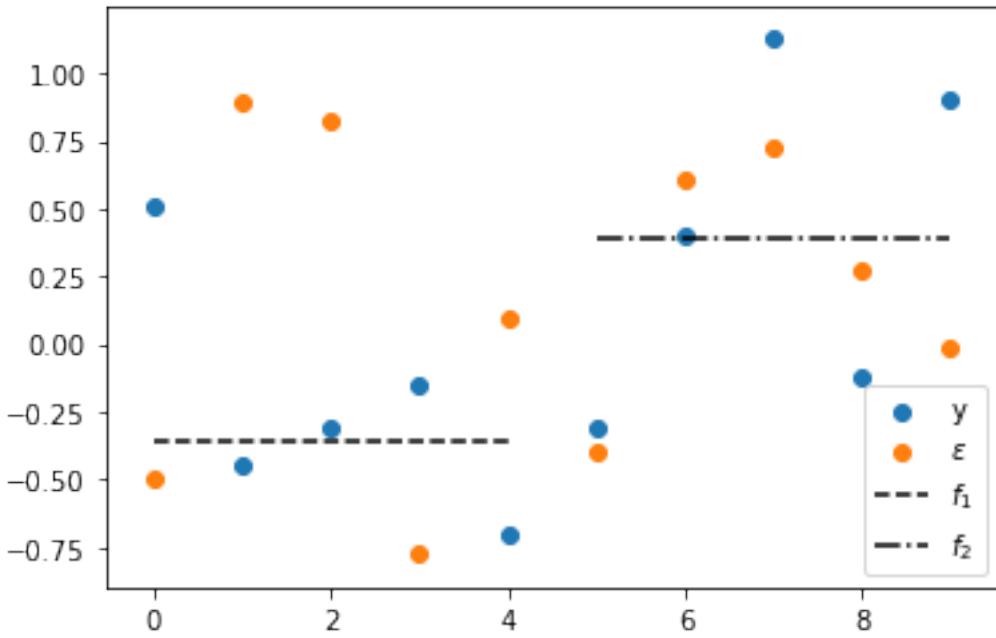
```
print('f = ', f)
```

```
f = [-0.36090544  0.3911745 ]
```

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

```
In [78]: plt.scatter(range(N), y, label='y')
plt.scatter(range(N), epsilon, label='$\epsilon$')
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 $Ef|y = By$

In [79]: `ε[:2]`

Out[79]: `array([-0.49247519, 0.8985248])`

In [80]: `# compare with Ef|y
B @ y`

Out[80]: `array([-0.2097539 , 0.38269762])`

The fraction of variance in y_t explained by the first two principal components can be computed as below.

In [81]: `l_tilde[:2].sum() / l_tilde.sum()`

Out[81]: `0.8400000000000001`

Compute

$$\hat{Y} = P_j \epsilon_j + P_k \epsilon_k$$

where P_j and P_k correspond to the largest two eigenvalues.

In [82]: `y_hat = P[:, :2] @ ε[:2]`

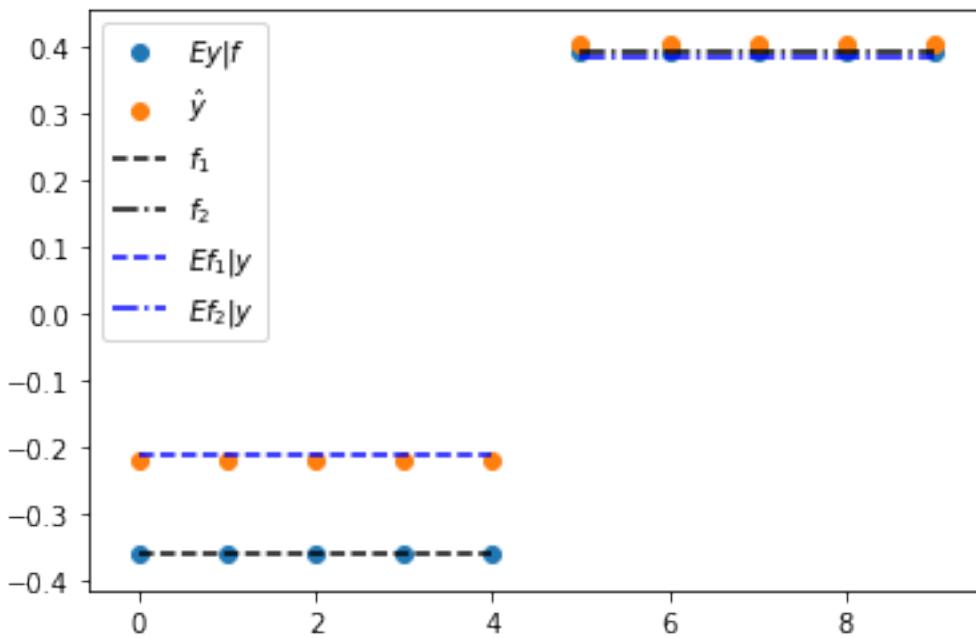
In this example, it turns out that the projection \hat{Y} of Y on the first two principal components does a good job of approximating $Ef \mid y$.

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.

```
In [83]: plt.scatter(range(N), f, label='$Ey|f$')
plt.scatter(range(N), y_hat, label='$\hat{y}$')
plt.hlines(f[0], 0, N//2-1, ls='--', label='$f_{\{1\}}$')
plt.hlines(f[1], N//2, N-1, ls='-.', label='$f_{\{2\}}$')

Ef_y = B @ y
plt.hlines(Ef_y[0], 0, N//2-1, ls='--', color='b', label='$Ef_{\{1\}}|y$')
plt.hlines(Ef_y[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 .

```
In [84]: Sigma_ijk = (P.T @ Sigma_y @ P)[:2, :2]
```

```
P_jk = P[:, :2]
```

```
Sigma_y_hat = P_jk @ Sigma_ijk @ P_jk.T
print('Sigma_y_hat = \n', Sigma_y_hat)
```

```
Sigma_y_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]
[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.  1.05 1.05 1.05 1.05 1.05]
```

8.13 Stochastic Difference Equation

Consider the stochastic second-order linear difference equation

$$y_t = \alpha_0 + \alpha_1 y_{t-1} + \alpha_2 y_{t-2} + u_t$$

where $u_t \sim N(0, \sigma_u^2)$ and

$$\begin{bmatrix} y_{-1} \\ y_0 \end{bmatrix} \sim N(\mu_{\tilde{y}}, \Sigma_{\tilde{y}})$$

It can be written as a stacked system

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ -\alpha_1 & 1 & 0 & 0 & \cdots & 0 & 0 & 0 \\ -\alpha_2 & -\alpha_1 & 1 & 0 & \cdots & 0 & 0 & 0 \\ 0 & -\alpha_2 & -\alpha_1 & 1 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & -\alpha_2 & -\alpha_1 & 1 \end{bmatrix}}_{\equiv A} \underbrace{\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ \vdots \\ y_T \end{bmatrix}}_{\equiv y} = \underbrace{\begin{bmatrix} \alpha_0 + \alpha_1 y_0 + \alpha_2 y_{-1} \\ \alpha_0 + \alpha_2 y_0 \\ \alpha_0 \\ \alpha_0 \\ \vdots \\ \alpha_0 \end{bmatrix}}_{\equiv b}$$

We can compute y by solving the system

$$y = A^{-1}(b + u)$$

We have

$$\begin{aligned} \mu_y &= A^{-1}\mu_b \\ \Sigma_y &= A^{-1}E[(b - \mu_b + u)(b - \mu_b + u)'] (A^{-1})' \\ &= A^{-1}(\Sigma_b + \Sigma_u)(A^{-1})' \end{aligned}$$

where

$$\mu_b = \begin{bmatrix} \alpha_0 + \alpha_1 \mu_{y_0} + \alpha_2 \mu_{y_{-1}} \\ \alpha_0 + \alpha_2 \mu_{y_0} \\ \alpha_0 \\ \vdots \\ \alpha_0 \end{bmatrix}$$

$$\Sigma_b = \begin{bmatrix} C\Sigma_{\tilde{y}}C' & 0_{N-2 \times N-2} \\ 0_{N-2 \times 2} & 0_{N-2 \times N-2} \end{bmatrix}, \quad C = \begin{bmatrix} \alpha_2 & \alpha_1 \\ 0 & \alpha_2 \end{bmatrix}$$

$$\Sigma_u = \begin{bmatrix} \sigma_u^2 & 0 & \cdots & 0 \\ 0 & \sigma_u^2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_u^2 \end{bmatrix}$$

```
In [85]: # set parameters
T = 80
T = 160
# coefficients of the second order difference equation
μ₀ = 10
μ₁ = 1.53
μ₂ = -.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]])
```

```
In [86]: # 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] = -μ₁

    if i-2 >= 0:
        A[i, i-2] = -μ₂

A_inv = np.linalg.inv(A)
```

```
In [87]: # compute the mean vectors of b and y
μb = np.ones(T) * μ₀
μb[0] += μ₁ * μy_tilde[1] + μ₂ * μy_tilde[0]
μb[1] += μ₂ * μy_tilde[1]

μy = A_inv @ μb
```

```
In [88]: # compute the covariance matrices of b and y
Σu = np.eye(T) * σu ** 2

Σb = np.zeros((T, T))

C = np.array([[μ₂, μ₁], [0, μ₂]])
Σb[:, :] = C @ Σy_tilde @ C.T

Σy = A_inv @ (Σb + Σu) @ A_inv.T
```

8.14 Application to Stock Price Model

Let

$$p_t = \sum_{j=0}^{T-t} \beta^j y_{t+j}$$

Form

$$\underbrace{\begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ \vdots \\ p_T \end{bmatrix}}_{\equiv p} = \underbrace{\begin{bmatrix} 1 & \beta & \beta^2 & \cdots & \beta^{T-1} \\ 0 & 1 & \beta & \cdots & \beta^{T-2} \\ 0 & 0 & 1 & \cdots & \beta^{T-3} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix}}_{\equiv B} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_T \end{bmatrix}$$

we have

$$\begin{aligned} \mu_p &= B\mu_y \\ \Sigma_p &= B\Sigma_y B' \end{aligned}$$

In [89]: $\beta = .96$

```
In [90]: # construct B
B = np.zeros((T, T))

for i in range(T):
    B[i, i:] = beta ** np.arange(0, T-i)
```

Denote

$$z = \begin{bmatrix} y \\ p \end{bmatrix} = \underbrace{\begin{bmatrix} I \\ B \end{bmatrix}}_{\equiv D} y$$

Thus, $\{y_t\}_{t=1}^T$ and $\{p_t\}_{t=1}^T$ jointly follow the multivariate normal distribution $N(\mu_z, \Sigma_z)$, where

$$\mu_z = D\mu_y$$

$$\Sigma_z = D\Sigma_y D'$$

In [91]: $D = np.vstack([np.eye(T), B])$

```
In [92]: muZ = D @ muY
SigmaZ = D @ SigmaY @ D.T
```

We can simulate paths of y_t and p_t and compute the conditional mean $E[p_t | y_{t-1}, y_t]$ using the `MultivariateNormal` class.

```
In [93]: z = np.random.multivariate_normal(mu_z, Sigma_z)
y, p = z[:T], z[T:]
```

```
In [94]: cond_Ep = np.empty(T-1)
```

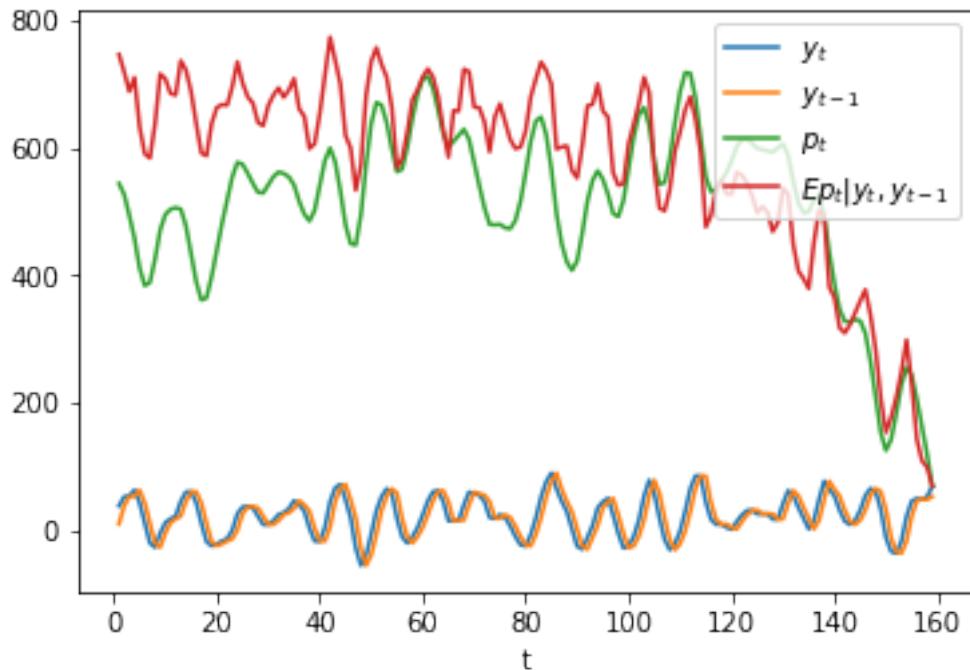
```
sub_mu = np.empty(3)
sub_Sigma = np.empty((3, 3))
for t in range(2, T+1):
    sub_mu[:] = mu_z[[t-2, t-1, T-1+t]]
    sub_Sigma[:, :] = Sigma_z[[t-2, t-1, T-1+t], :][:, [t-2, t-1, T-1+t]]

    multi_normal = MultivariateNormal(sub_mu, sub_Sigma)
    multi_normal.partition(2)

    cond_Ep[t-2] = multi_normal.cond_dist(1, y[t-2:t])[0][0]
```

```
In [95]: 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='$E_p_{\{t\}}|y_{\{t\}}, y_{\{t-1\}}$')

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 red line is the conditional expectation $E[p_t | y_{t-1}, y_t]$, 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 .

8.15 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 , who does not observe x_0 , who knows $\hat{x}_0, \Sigma_0, G, R$ – and therefore knows the joint probability distribution of the vector $\begin{bmatrix} x_0 \\ y_0 \end{bmatrix}$ – and who 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

$$\begin{aligned} \beta_0 &= \Sigma_0 G' (G\Sigma_0 G' + R)^{-1} \\ \tilde{x}_0 &= \hat{x}_0 + \beta_0(y_0 - G\hat{x}_0) \\ \tilde{\Sigma}_0 &= \Sigma_0 - \Sigma_0 G' (G\Sigma_0 G' + R)^{-1} G\Sigma_0 \end{aligned}$$

8.15.1 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.

It follows that the probability distribution of x_1 conditional on y_0 is

$$x_1 | y_0 \sim \mathcal{N}(A\tilde{x}_0, A\tilde{\Sigma}_0 A' + CC')$$

Define

$$\begin{aligned} \hat{x}_1 &= A\tilde{x}_0 \\ \Sigma_1 &= A\tilde{\Sigma}_0 A' + CC' \end{aligned}$$

8.15.2 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 t th 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 $y_0, y_1, \dots, y_{t-1} = y^{t-1}$ 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:

$$\begin{aligned} \Sigma_t &= A\tilde{\Sigma}_{t-1}A' + CC' \\ \hat{x}_t &= A\tilde{x}_{t-1} \\ \beta_t &= \Sigma_t G'(G\Sigma_t G' + R)^{-1} \\ \tilde{x}_t &= \hat{x}_t + \beta_t(y_t - G\hat{x}_t) \\ \tilde{\Sigma}_t &= \Sigma_t - \Sigma_t G'(G\Sigma_t G' + R)^{-1}G\Sigma_t \end{aligned}$$

We can use the Python class *MultivariateNormal* to construct examples.

Here is an example for a single period problem at time 0

```
In [96]: 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]])
```

```
In [97]: # construction of the multivariate normal instance
multi_normal = MultivariateNormal(μ, Σ)
```

```
In [98]: multi_normal.partition(2)
```

```
In [99]: # the observation of y
y0 = 2.3

# conditional distribution of x0
μ1_hat, Σ11 = multi_normal.cond_dist(0, y0)
μ1_hat, Σ11
```

```
Out[99]: (array([-0.078125,  0.803125]), array([[ 0.72098214, -0.203125],
   [-0.403125,   0.228125]]))
```

```
In [100]: 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
Σ1_cond = C @ C.T + A @ Σ11 @ A.T
x1_cond, Σ1_cond
```

```
Out[100]: (array([0.1215625, 0.24875]), array([[4.12874554, 1.95523214],
[1.92123214, 1.04592857]))
```

8.15.3 Code for Iterating

Here is code for solving a dynamic filtering problem by iterating on our equations, followed by an example.

```
In [101]: 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
```

```
In [102]: iterate(x0_hat, Σ0, A, C, G, R, [2.3, 1.2, 3.2])
```

```
Out[102]: (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]],  
  
[[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](#)

Chapter 9

Univariate Time Series with Matrix Algebra

9.1 Contents

- Overview 9.2
- Samuelson’s model 9.3
- Adding a random term 9.4
- A forward looking model 9.5

9.2 Overview

This lecture uses matrices to solve some linear difference equations.

As a running example, we’ll study a **second-order linear difference equation** that was the key technical tool in Paul Samuelson’s 1939 article [93] that introduced the **multiplier-accelerator** model.

This model became the workhorse that powered early econometric versions of Keynesian macroeconomic models in the United States.

You can read about the details of that model in [this QuantEcon lecture](#).

(That lecture also describes some technicalities about second-order linear difference equations.)

We’ll also study a “perfect foresight” model of stock prices that involves solving a “forward-looking” linear difference equation.

We will use the following imports:

```
In [1]: import numpy as np  
import matplotlib.pyplot as plt  
%matplotlib inline
```

9.3 Samuelson’s model

Let $t = 0, \pm 1, \pm 2, \dots$ index time.

For $t = 1, 2, 3, \dots, T$ suppose that

$$y_t = \alpha_0 + \alpha_1 y_{t-1} + \alpha_2 y_{t-2} \quad (1)$$

where we assume that y_0 and y_{-1} are given numbers that we take as **initial conditions**.

In Samuelson's model, y_t stood for **national income** or perhaps a different measure of aggregate activity called **gross domestic product** (GDP) at time t .

Equation (1) is called a **second-order linear difference equation**.

But actually, it is a collection of T simultaneous linear equations in the T variables y_1, y_2, \dots, y_T .

Note: To be able to solve a second-order linear difference equation, we require two **boundary conditions** that can take the form either of two **initial conditions** or two **terminal conditions** or possibly one of each.

Let's write our equations as a stacked system

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ -\alpha_1 & 1 & 0 & 0 & \cdots & 0 & 0 & 0 \\ -\alpha_2 & -\alpha_1 & 1 & 0 & \cdots & 0 & 0 & 0 \\ 0 & -\alpha_2 & -\alpha_1 & 1 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & -\alpha_2 & -\alpha_1 & 1 \end{bmatrix}}_{\equiv A} \underbrace{\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ \vdots \\ y_T \end{bmatrix}}_{\equiv y} = \underbrace{\begin{bmatrix} \alpha_0 + \alpha_1 y_0 + \alpha_2 y_{-1} \\ \alpha_0 + \alpha_2 y_0 \\ \alpha_0 \\ \alpha_0 \\ \vdots \\ \alpha_0 \end{bmatrix}}_{\equiv b}$$

or

$$Ay = b$$

where

$$y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_T \end{bmatrix}$$

Evidently y can be computed from

$$y = A^{-1}b$$

The vector y is a complete time path $\{y_t\}_{t=1}^T$.

Let's put Python to work on an example that captures the flavor of Samuelson's multiplier-accelerator model.

We'll set parameters equal to the same values we used in [this QuantEcon lecture](#).

In [2]: `T = 80`

```
# parameters
alpha0 = 10.0
```

```
l1 = 1.53
l2 = -.9

y_1 = 28. # y_{-1}
y0 = 24.
```

```
In [3]: # construct A and b
A = np.zeros((T, T))

for i in range(T):
    A[i, i] = 1

    if i-1 >= 0:
        A[i, i-1] = -l1

    if i-2 >= 0:
        A[i, i-2] = -l2

b = np.ones(T) * l0
b[0] = l0 + l1 * y0 + l2 * y_1
b[1] = l0 + l2 * y0
```

Let's look at the matrix A and the vector b for our example.

```
In [4]: A, b
```

```
Out[4]: (array([[ 1. ,  0. ,  0. , ...,  0. ,  0. ,  0. ],
   [-1.53,  1. ,  0. , ...,  0. ,  0. ,  0. ],
   [ 0.9 , -1.53,  1. , ...,  0. ,  0. ,  0. ],
   ...,
   [ 0. ,  0. ,  0. , ...,  1. ,  0. ,  0. ],
   [ 0. ,  0. ,  0. , ..., -1.53,  1. ,  0. ],
   [ 0. ,  0. ,  0. , ...,  0.9 , -1.53,  1. ]]),
 array([ 21.52, -11.6 ,  10. ,  10. ,  10. ,  10. ,  10. ,
        10. ,  10. ,  10. ,  10. ,  10. ,  10. ,  10. ,
        10. ,  10. ,  10. ,  10. ,  10. ,  10. ,  10. ,
        10. ,  10. ,  10. ,  10. ,  10. ,  10. ,  10. ,
        10. ,  10. ,  10. ,  10. ,  10. ,  10. ,  10. ,
        10. ,  10. ,  10. ,  10. ,  10. ,  10. ,  10. ,
        10. ,  10. ,  10. ,  10. ,  10. ,  10. ,  10. ,
        10. ,  10. ,  10. ,  10. ,  10. ,  10. ,  10. ,
        10. ,  10. ,  10. ,  10. ,  10. ,  10. ,  10. ,
        10. ,  10. ,  10. ,  10. ,  10. ,  10. ,  10. ,
        10. ,  10. ,  10. ,  10. ,  10. ,  10. ,  10. ]))
```

Now let's solve for the path of y .

If y_t is GNP at time t , then we have a version of Samuelson's model of the dynamics for GNP.

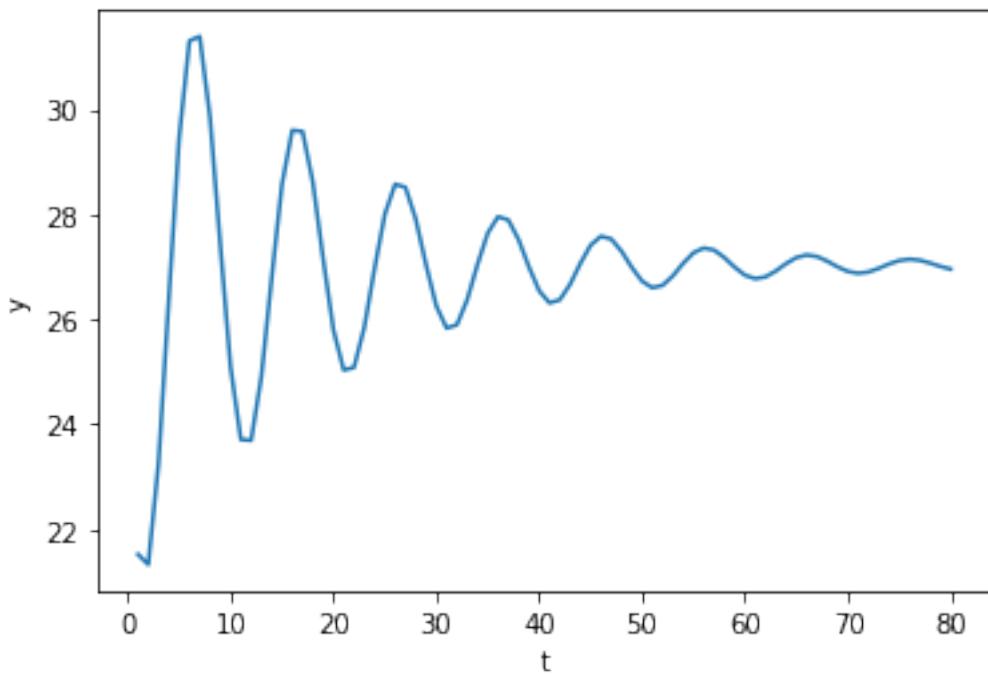
```
In [5]: A_inv = np.linalg.inv(A)
```

```
y = A_inv @ b
```

```
In [6]: plt.plot(np.arange(T)+1, y)
plt.xlabel('t')
```

```
plt.ylabel('y')
```

```
plt.show()
```



If we set both initial values at the **steady state** value of y_t , namely,

$$y_0 = y_{-1} = \frac{\alpha_0}{1 - \alpha_1 - \alpha_2}$$

then y_t will be constant

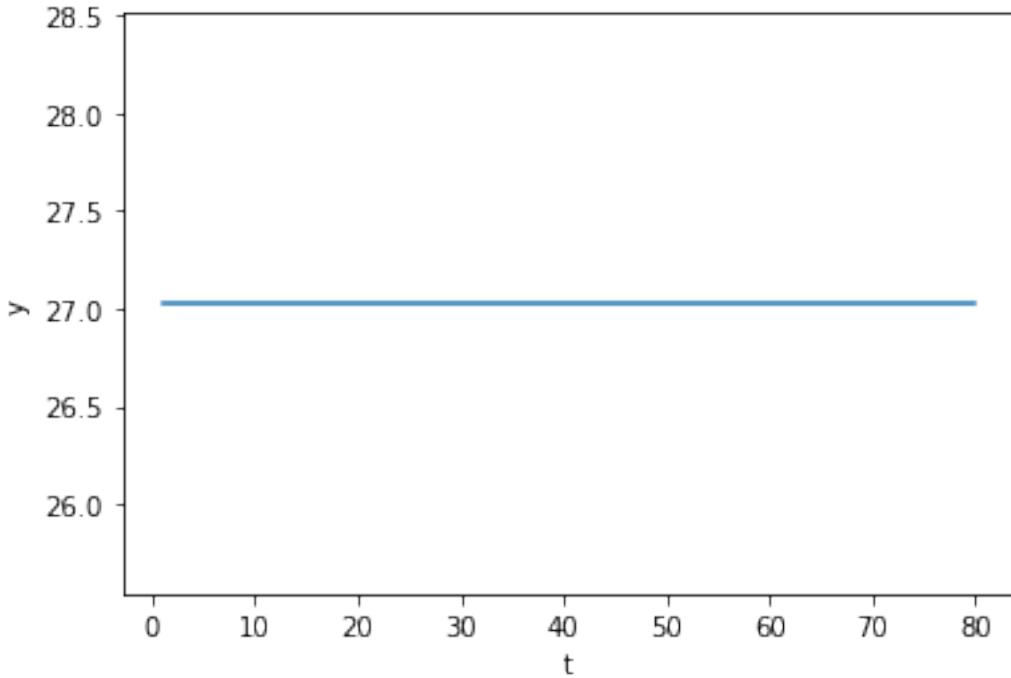
```
In [7]: y_1_steady = 0 / (1 - 1 - 2) # y_{-1}
y0_steady = 0 / (1 - 1 - 2)

b_steady = np.ones(T) * 0
b_steady[0] = 0 + 1 * y0_steady + 2 * y_1_steady
b_steady[1] = 0 + 2 * y0_steady
```

```
In [8]: y_steady = A_inv @ b_steady
```

```
In [9]: plt.plot(np.arange(T)+1, y_steady)
plt.xlabel('t')
plt.ylabel('y')

plt.show()
```



9.4 Adding a random term

To generate some excitement, we'll follow in the spirit of the great economists Eugen Slutsky and Ragnar Frisch and replace our original second-order difference equation with the following **second-order stochastic linear difference equation**:

$$y_t = \alpha_0 + \alpha_1 y_{t-1} + \alpha_2 y_{t-2} + u_t \quad (2)$$

where $u_t \sim N(0, \sigma_u^2)$ and is IID, meaning **independent** and **identically distributed**.

We'll stack these T equations into a system cast in terms of matrix algebra.

Let's define the random vector

$$u = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_T \end{bmatrix}$$

Where A, b, y are defined as above, now assume that y is governed by the system

$$Ay = b + u$$

The solution for y becomes

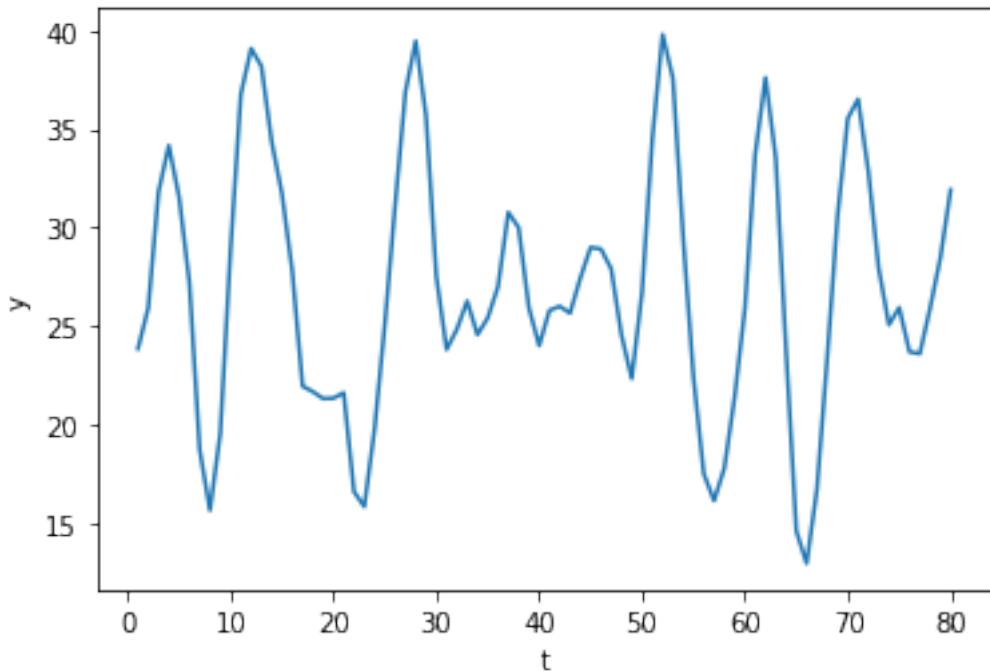
$$y = A^{-1}(b + u)$$

Let's try it out in Python.

In [10]: $\mathbb{I} u = 2.$

In [11]: $u = np.random.normal(0, \mathbb{I} u, size=T)$
 $y = A_inv @ (b + u)$

In [12]: $plt.plot(np.arange(T)+1, y)$
 $plt.xlabel('t')$
 $plt.ylabel('y')$
 $plt.show()$



The above time series looks a lot like (detrended) GDP series for a number of advanced countries in recent decades.

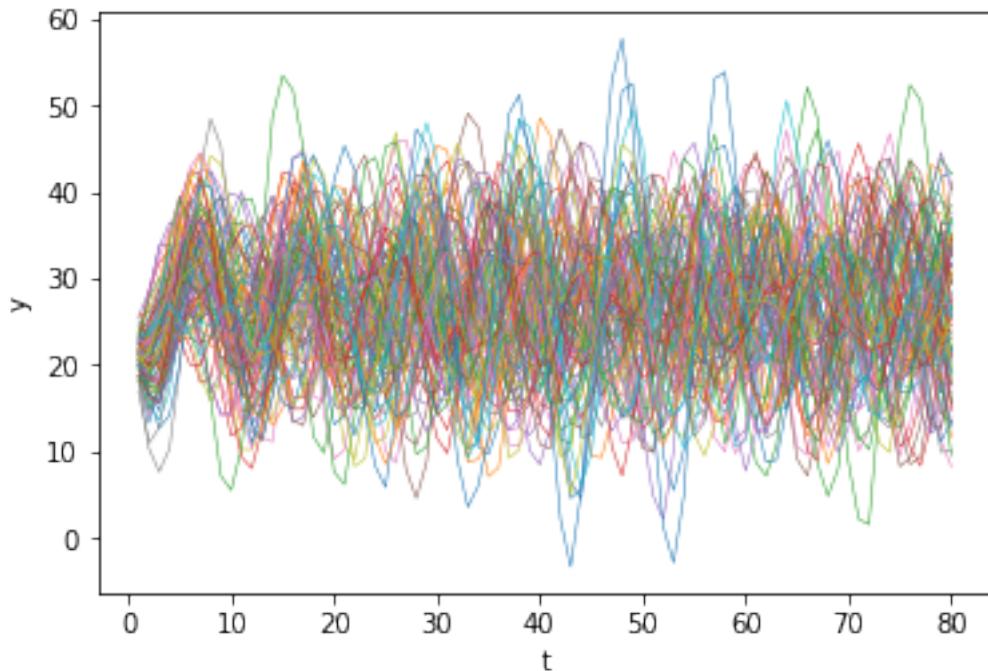
We can simulate N paths.

In [13]: $N = 100$

```
for i in range(N):
    u = np.random.normal(0, I_u, size=T)
    y = A_inv @ (b + u)
    plt.plot(np.arange(T)+1, y, lw=0.5)

plt.xlabel('t')
plt.ylabel('y')

plt.show()
```



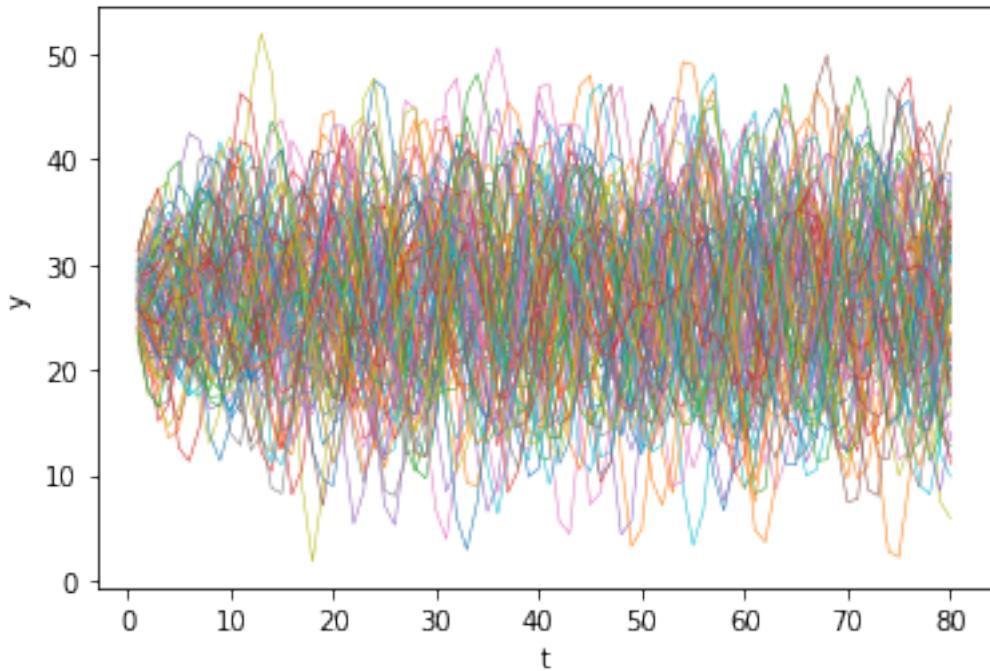
Also consider the case when y_0 and y_{-1} are at steady state.

In [14]: `N = 100`

```
for i in range(N):
    u = np.random.normal(0, 1, size=T)
    y_steady = A_inv @ (b_steady + u)
    plt.plot(np.arange(T)+1, y_steady, lw=0.5)

plt.xlabel('t')
plt.ylabel('y')

plt.show()
```



9.5 A forward looking model

Samuelson's model is **backwards looking** in the sense that we give it **initial conditions** and let it run.

Let's now turn to model that is **forward looking**.

We apply similar linear algebra machinery to study a **perfect foresight** model widely used as a benchmark in macroeconomics and finance.

As an example, we suppose that p_t is the price of a stock and that y_t is its dividend.

We assume that y_t is determined by second-order difference equation that we analyzed just above, so that

$$y = A^{-1} (b + u)$$

Our **perfect foresight** model of stock prices is

$$p_t = \sum_{j=0}^{T-t} \beta^j y_{t+j}, \quad \beta \in (0, 1)$$

where β is a discount factor.

The model asserts that the price of the stock at t equals the discounted present values of the (perfectly foreseen) future dividends.

Form

$$\underbrace{\begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ \vdots \\ p_T \end{bmatrix}}_{\equiv p} = \underbrace{\begin{bmatrix} 1 & \beta & \beta^2 & \cdots & \beta^{T-1} \\ 0 & 1 & \beta & \cdots & \beta^{T-2} \\ 0 & 0 & 1 & \cdots & \beta^{T-3} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix}}_{\equiv B} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_T \end{bmatrix}$$

In [15]: $\beta = .96$

```
In [16]: # construct B
B = np.zeros((T, T))

for i in range(T):
    B[i, i:] = beta ** np.arange(0, T-i)
```

In [17]: B

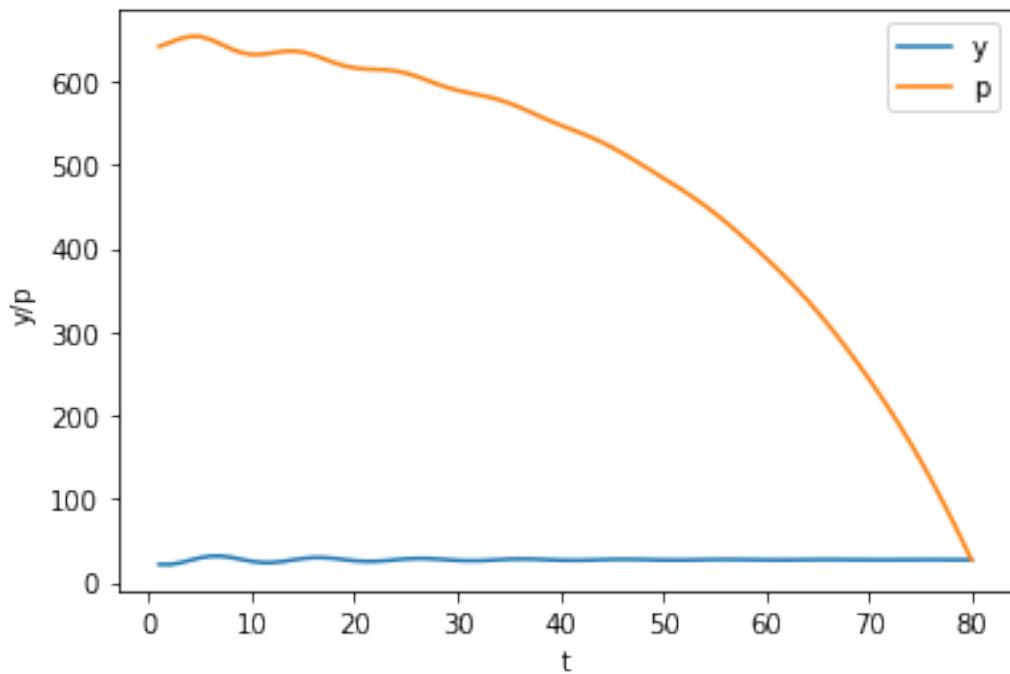
```
Out[17]: array([[1.          , 0.96        , 0.9216      , ... , 0.04314048, 0.04141486,
                  0.03975826],
                 [0.          , 1.          , 0.96        , ... , 0.044938  , 0.04314048,
                  0.04141486],
                 [0.          , 0.          , 1.          , ... , 0.04681041, 0.044938  ,
                  0.04314048],
                 ...,
                 [0.          , 0.          , 0.          , ... , 1.          , 0.96        ,
                  0.9216      ],
                 [0.          , 0.          , 0.          , ... , 0.          , 1.          ,
                  0.96        ],
                 [0.          , 0.          , 0.          , ... , 0.          , 0.          ,
                  1.          ]])
```

```
In [18]: beta_u = 0.
u = np.random.normal(0, beta_u, size=T)
y = A_inv @ (b + u)
y_steady = A_inv @ (b_steady + u)
```

In [19]: p = B @ y

```
In [20]: plt.plot(np.arange(0, T)+1, y, label='y')
plt.plot(np.arange(0, T)+1, p, label='p')
plt.xlabel('t')
plt.ylabel('y/p')
plt.legend()

plt.show()
```



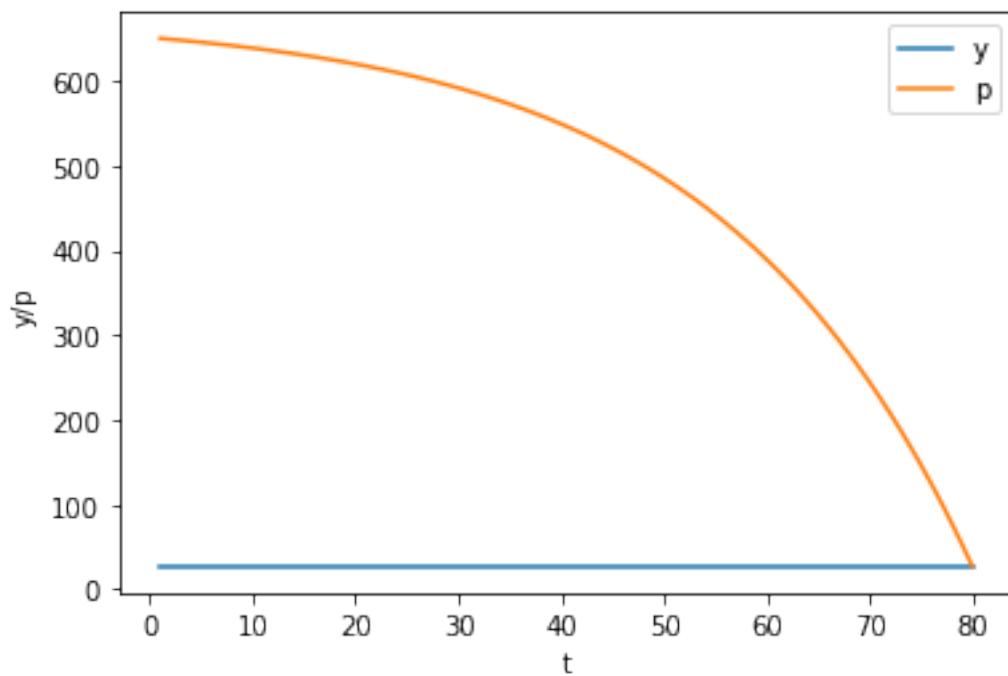
Can you explain why the trend of the price is downward over time?

Also consider the case when y_0 and y_{-1} are at the steady state.

```
In [21]: p_steady = B @ y_steady
```

```
plt.plot(np.arange(0, T)+1, y_steady, label='y')
plt.plot(np.arange(0, T)+1, p_steady, label='p')
plt.xlabel('t')
plt.ylabel('y/p')
plt.legend()

plt.show()
```



Part II

Introduction to Dynamics

Chapter 10

Dynamics in One Dimension

10.1 Contents

- Overview 10.2
- Some Definitions 10.3
- Graphical Analysis 10.4
- Exercises 10.5
- Solutions 10.6

10.2 Overview

In this lecture we give a quick introduction to discrete time dynamics in one dimension.

In one-dimensional models, the state of the system is described by a single variable.

Although most interesting dynamic models have two or more state variables, the one-dimensional setting is a good place to learn the foundations of dynamics and build intuition.

Let's start with some standard imports:

```
In [1]: import numpy as np  
import matplotlib.pyplot as plt  
%matplotlib inline
```

10.3 Some Definitions

This section sets out the objects of interest and the kinds of properties we study.

10.3.1 Difference Equations

A **time homogeneous first order difference equation** is an equation of the form

$$x_{t+1} = g(x_t) \tag{1}$$

where g is a function from some subset S of \mathbb{R} to itself.

Here S is called the **state space** and x is called the **state variable**.

In the definition,

- time homogeneity means that g is the same at each time t
- first order means dependence on only one lag (i.e., earlier states such as x_{t-1} do not enter into (1)).

If $x_0 \in S$ is given, then (1) recursively defines the sequence

$$x_0, \quad x_1 = g(x_0), \quad x_2 = g(x_1) = g(g(x_0)), \quad \text{etc.} \quad (2)$$

This sequence is called the **trajectory** of x_0 under g .

If we define g^n to be n compositions of g with itself, then we can write the trajectory more simply as $x_t = g^t(x_0)$ for $t \geq 0$.

10.3.2 Example: A Linear Model

One simple example is the **linear difference equation**

$$x_{t+1} = ax_t + b, \quad S = \mathbb{R}$$

where a, b are fixed constants.

In this case, given x_0 , the trajectory (2) is

$$x_0, \quad ax_0 + b, \quad a^2x_0 + ab + b, \quad \text{etc.} \quad (3)$$

Continuing in this way, and using our knowledge of [geometric series](#), we find that, for any $t \geq 0$,

$$x_t = a^t x_0 + b \frac{1 - a^t}{1 - a} \quad (4)$$

This is about all we need to know about the linear model.

We have an exact expression for x_t for all t and hence a full understanding of the dynamics.

Notice in particular that $|a| < 1$, then, by (4), we have

$$x_t \rightarrow \frac{b}{1 - a} \text{ as } t \rightarrow \infty \quad (5)$$

regardless of x_0

This is an example of what is called global stability, a topic we return to below.

10.3.3 Example: A Nonlinear Model

In the linear example above, we obtained an exact analytical expression for x_t in terms of arbitrary t and x_0 .

This made analysis of dynamics very easy.

When models are nonlinear, however, the situation can be quite different.

For example, recall how we previously studied the law of motion for the Solow growth model, a simplified version of which is

$$k_{t+1} = szk_t^\alpha + (1 - \delta)k_t \quad (6)$$

Here k is capital stock and s, z, α, δ are positive parameters with $0 < \alpha, \delta < 1$.

If you try to iterate like we did in (3), you will find that the algebra gets messy quickly.

Analyzing the dynamics of this model requires a different method (see below).

10.3.4 Stability

A **steady state** of the difference equation $x_{t+1} = g(x_t)$ is a point x^* in S such that $x^* = g(x^*)$.

In other words, x^* is a **fixed point** of the function g in S .

For example, for the linear model $x_{t+1} = ax_t + b$, you can use the definition to check that

- $x^* := b/(1 - a)$ is a steady state whenever $a \neq 1$.
- if $a = 1$ and $b = 0$, then every $x \in \mathbb{R}$ is a steady state.
- if $a = 1$ and $b \neq 0$, then the linear model has no steady state in \mathbb{R} .

A steady state x^* of $x_{t+1} = g(x_t)$ is called **globally stable** if, for all $x_0 \in S$,

$$x_t = g^t(x_0) \rightarrow x^* \text{ as } t \rightarrow \infty$$

For example, in the linear model $x_{t+1} = ax_t + b$ with $a \neq 1$, the steady state x^*

- is globally stable if $|a| < 1$ and
- fails to be globally stable otherwise.

This follows directly from (4).

A steady state x^* of $x_{t+1} = g(x_t)$ is called **locally stable** if there exists an $\epsilon > 0$ such that

$$|x_0 - x^*| < \epsilon \implies x_t = g^t(x_0) \rightarrow x^* \text{ as } t \rightarrow \infty$$

Obviously every globally stable steady state is also locally stable.

We will see examples below where the converse is not true.

10.4 Graphical Analysis

As we saw above, analyzing the dynamics for nonlinear models is nontrivial.

There is no single way to tackle all nonlinear models.

However, there is one technique for one-dimensional models that provides a great deal of intuition.

This is a graphical approach based on **45 degree diagrams**.

Let's look at an example: the Solow model with dynamics given in (6).

We begin with some plotting code that you can ignore at first reading.

The function of the code is to produce 45 degree diagrams and time series plots.

```
In [2]: def subplots(fs):
    "Custom subplots with axes through the origin"
    fig, ax = plt.subplots(figsize=fs)

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

    return fig, ax

def plot45(g, xmin, xmax, x0, num_arrows=6, var='x'):

    xgrid = np.linspace(xmin, xmax, 200)

    fig, ax = subplots((6.5, 6))
    ax.set_xlim(xmin, xmax)
    ax.set_ylim(xmin, xmax)

    hw = (xmax - xmin) * 0.01
    hl = 2 * hw
    arrow_args = dict(fc="k", ec="k", head_width=hw,
                      length_includes_head=True, lw=1,
                      alpha=0.6, head_length=hl)

    ax.plot(xgrid, g(xgrid), 'b-', lw=2, alpha=0.6, label='g')
    ax.plot(xgrid, xgrid, 'k-', lw=1, alpha=0.7, label='45')

    x = x0
    xticks = [xmin]
    xtick_labels = [xmin]

    for i in range(num_arrows):
        if i == 0:
            ax.arrow(x, 0.0, 0.0, g(x), **arrow_args) # x, y, dx, dy
        else:
            ax.arrow(x, x, 0.0, g(x) - x, **arrow_args)
            ax.plot((x, x), (0, x), 'k', ls='dotted')

        ax.arrow(x, g(x), g(x) - x, 0, **arrow_args)
        xticks.append(x)
        xtick_labels.append(r'$\{}_{}$'.format(var, str(i)))

        x = g(x)
        xticks.append(x)
        xtick_labels.append(r'$\{}_{}$'.format(var, str(i+1)))
        ax.plot((x, x), (0, x), 'k', ls='dotted')

    xticks.append(xmax)
    xtick_labels.append(xmax)
```

```

ax.set_xticks(xticks)
ax.set_yticks(xticks)
ax.set_xticklabels(xtick_labels)
ax.set_yticklabels(xtick_labels)

bbox = (0., 1.04, 1., .104)
legend_args = {'bbox_to_anchor': bbox, 'loc': 'upper right'}

ax.legend(ncol=2, frameon=False, **legend_args, fontsize=14)
plt.show()

def ts_plot(g, xmin, xmax, x0, ts_length=6, var='x'):
    fig, ax = subplots((7, 5.5))
    ax.set_xlim(xmin, xmax)
    ax.set_xlabel(r'$t$', fontsize=14)
    ax.set_ylabel(r'$\{ \}_t$'.format(var), fontsize=14)
    x = np.empty(ts_length)
    x[0] = x0
    for t in range(ts_length-1):
        x[t+1] = g(x[t])
    ax.plot(range(ts_length),
            x,
            'bo-',
            alpha=0.6,
            lw=2,
            label=r'$\{ \}_t$'.format(var))
    ax.legend(loc='best', fontsize=14)
    ax.set_xticks(range(ts_length))
    plt.show()

```

Let's create a 45 degree diagram for the Solow model with a fixed set of parameters

In [3]: `A, s, alpha, delta = 2, 0.3, 0.3, 0.4`

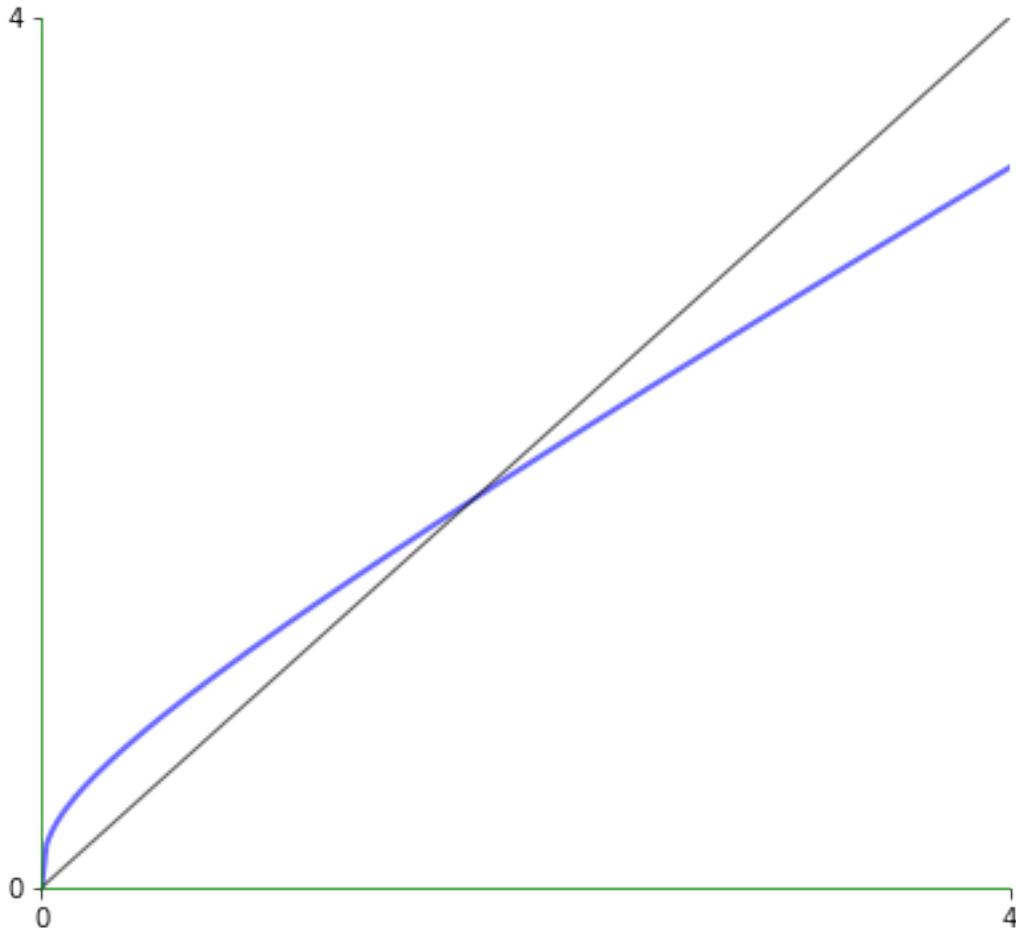
Here's the update function corresponding to the model.

In [4]: `def g(k):
 return A * s * k**alpha + (1 - delta) * k`

Here is the 45 degree plot.

In [5]: `xmin, xmax = 0, 4 # Suitable plotting region.
plot45(g, xmin, xmax, 0, num_arrows=0)`

— g — 45



The plot shows the function g and the 45 degree line.

Think of k_t as a value on the horizontal axis.

To calculate k_{t+1} , we can use the graph of g to see its value on the vertical axis.

Clearly,

- If g lies above the 45 degree line at this point, then we have $k_{t+1} > k_t$.
- If g lies below the 45 degree line at this point, then we have $k_{t+1} < k_t$.
- If g hits the 45 degree line at this point, then we have $k_{t+1} = k_t$, so k_t is a steady state.

For the Solow model, there are two steady states when $S = \mathbb{R}_+ = [0, \infty)$.

- the origin $k = 0$
- the unique positive number such that $k = szk^\alpha + (1 - \delta)k$.

By using some algebra, we can show that in the second case, the steady state is

$$k^* = \left(\frac{sz}{\delta} \right)^{1/(1-\alpha)}$$

10.4.1 Trajectories

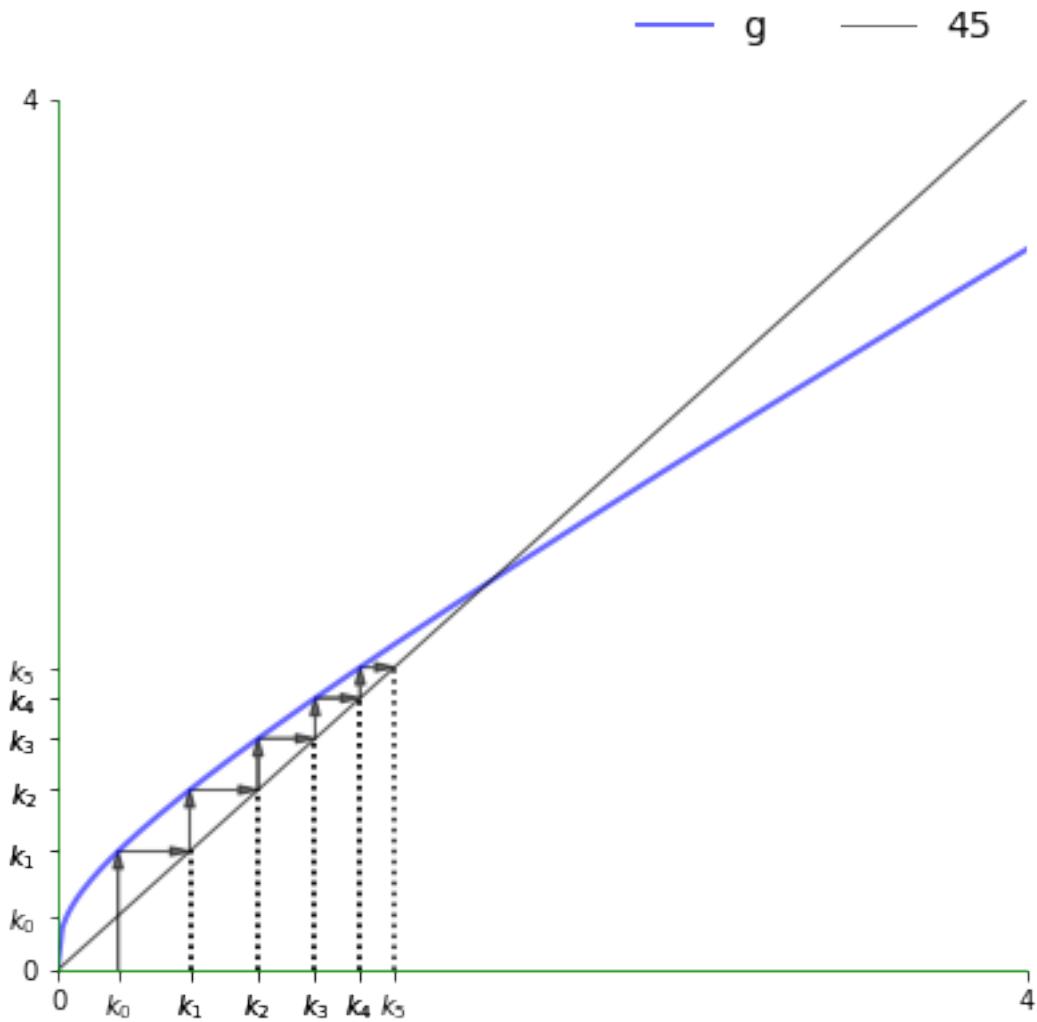
By the preceding discussion, in regions where g lies above the 45 degree line, we know that the trajectory is increasing.

The next figure traces out a trajectory in such a region so we can see this more clearly.

The initial condition is $k_0 = 0.25$.

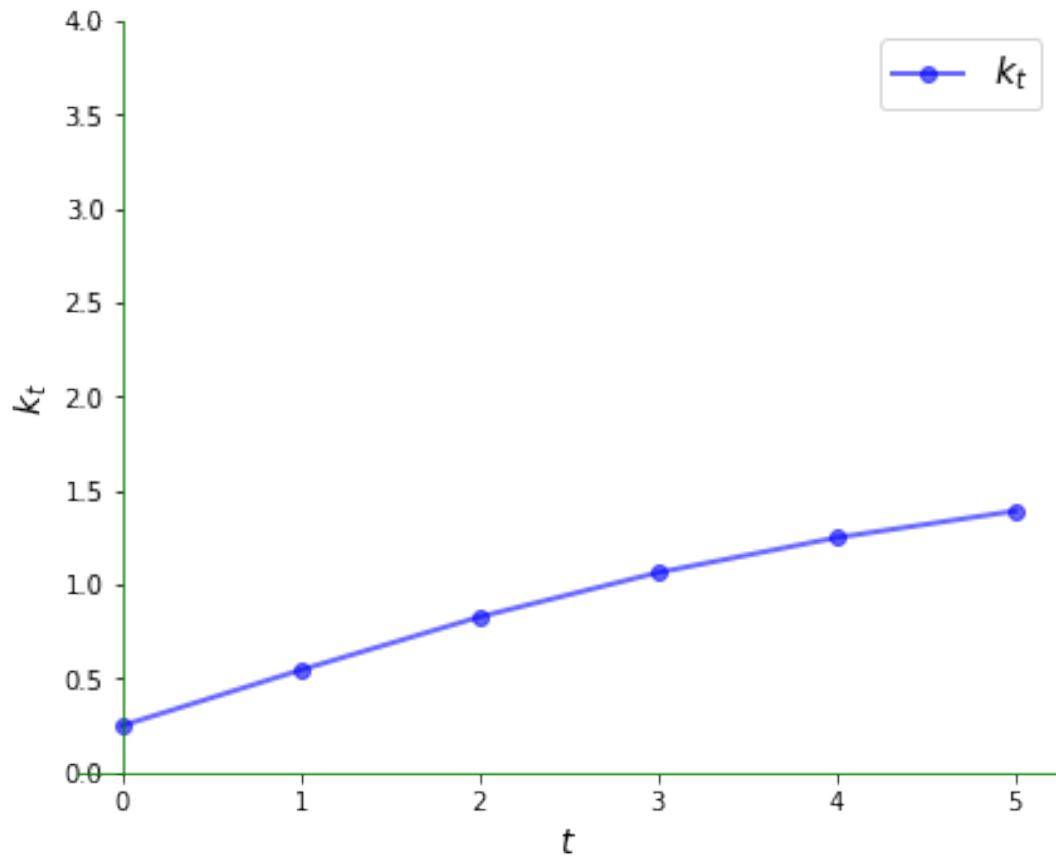
In [6]: $k_0 = 0.25$

```
plot45(g, xmin, xmax, k0, num_arrows=5, var='k')
```



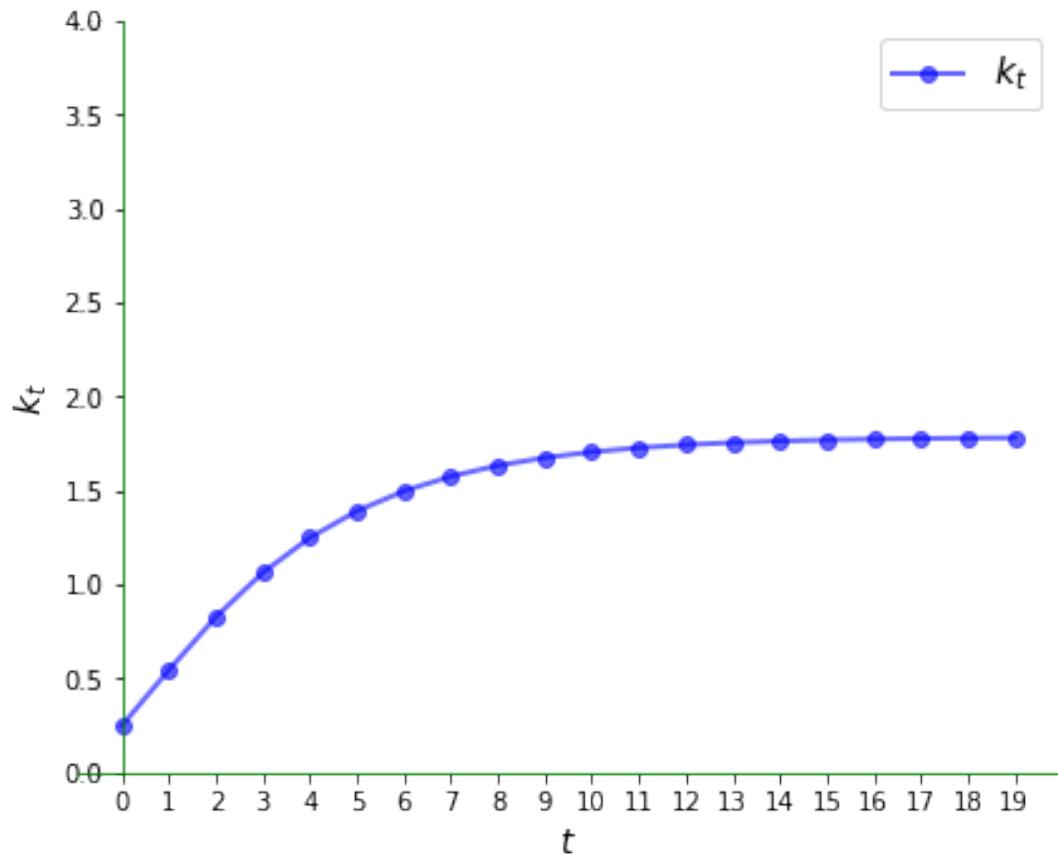
We can plot the time series of capital corresponding to the figure above as follows:

In [7]: $ts_plot(g, \text{xmin}, \text{xmax}, \text{k0}, \text{var}='k')$



Here's a somewhat longer view:

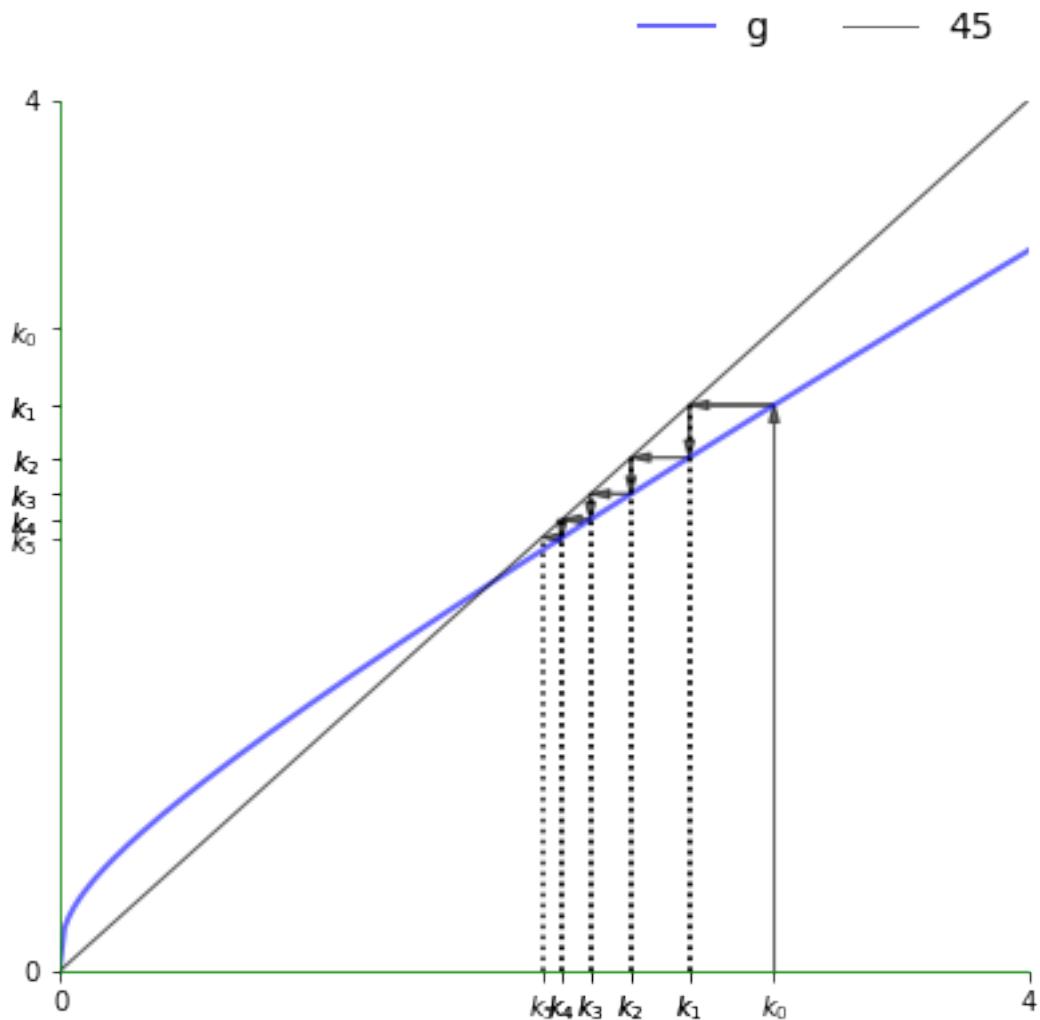
```
In [8]: ts_plot(g, xmin, xmax, k0, ts_length=20, var='k')
```



When capital stock is higher than the unique positive steady state, we see that it declines:

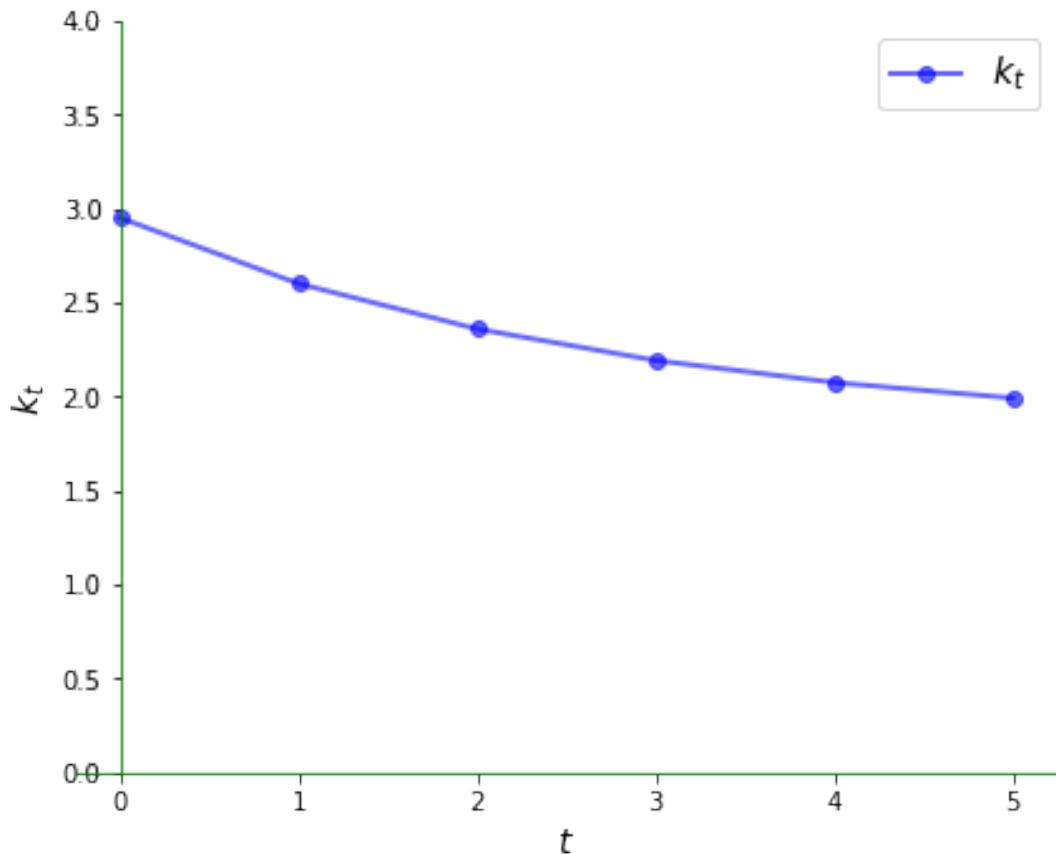
In [9]: $k_0 = 2.95$

```
plot45(g, xmin, xmax, k0, num_arrows=5, var='k')
```



Here is the time series:

```
In [10]: ts_plot(g, xmin, xmax, k0, var='k')
```



10.4.2 Complex Dynamics

The Solow model is nonlinear but still generates very regular dynamics.

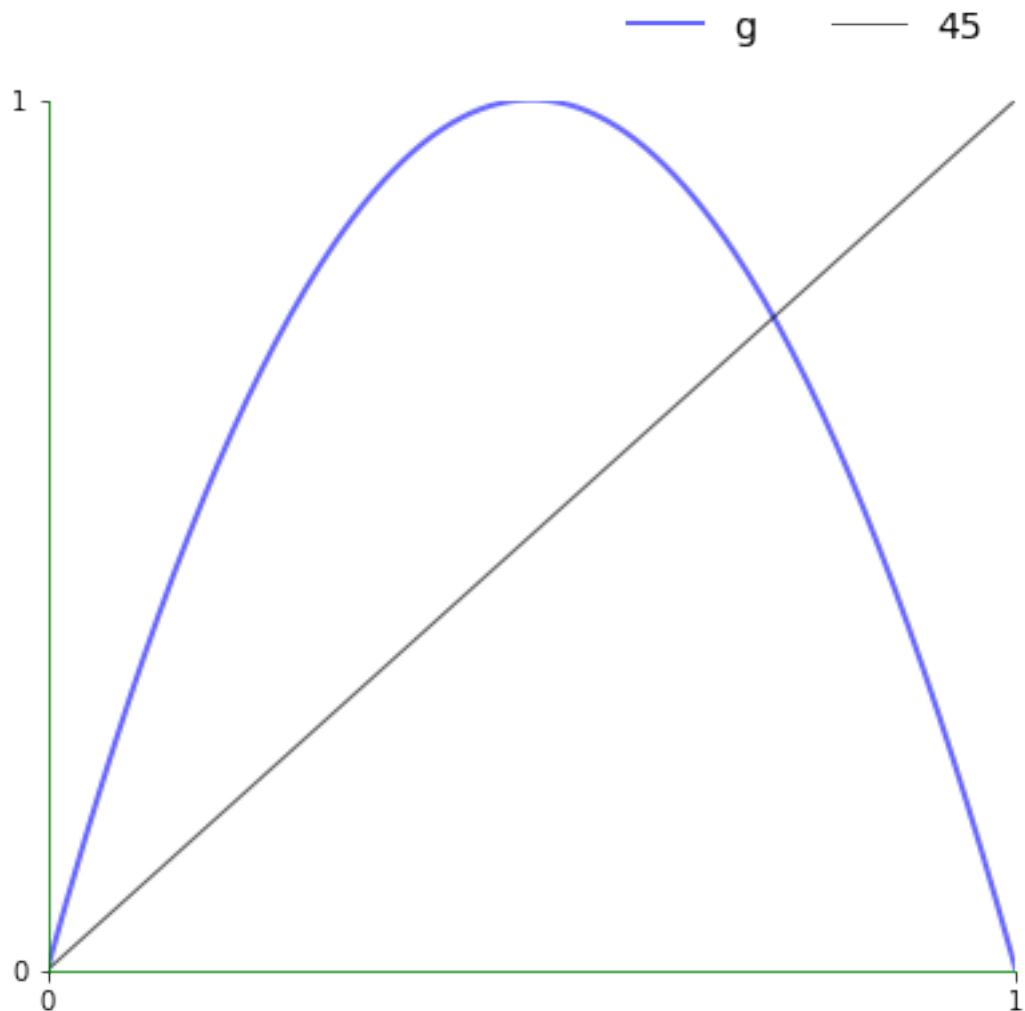
One model that generates irregular dynamics is the **quadratic map**

$$g(x) = 4x(1 - x), \quad x \in [0, 1]$$

Let's have a look at the 45 degree diagram.

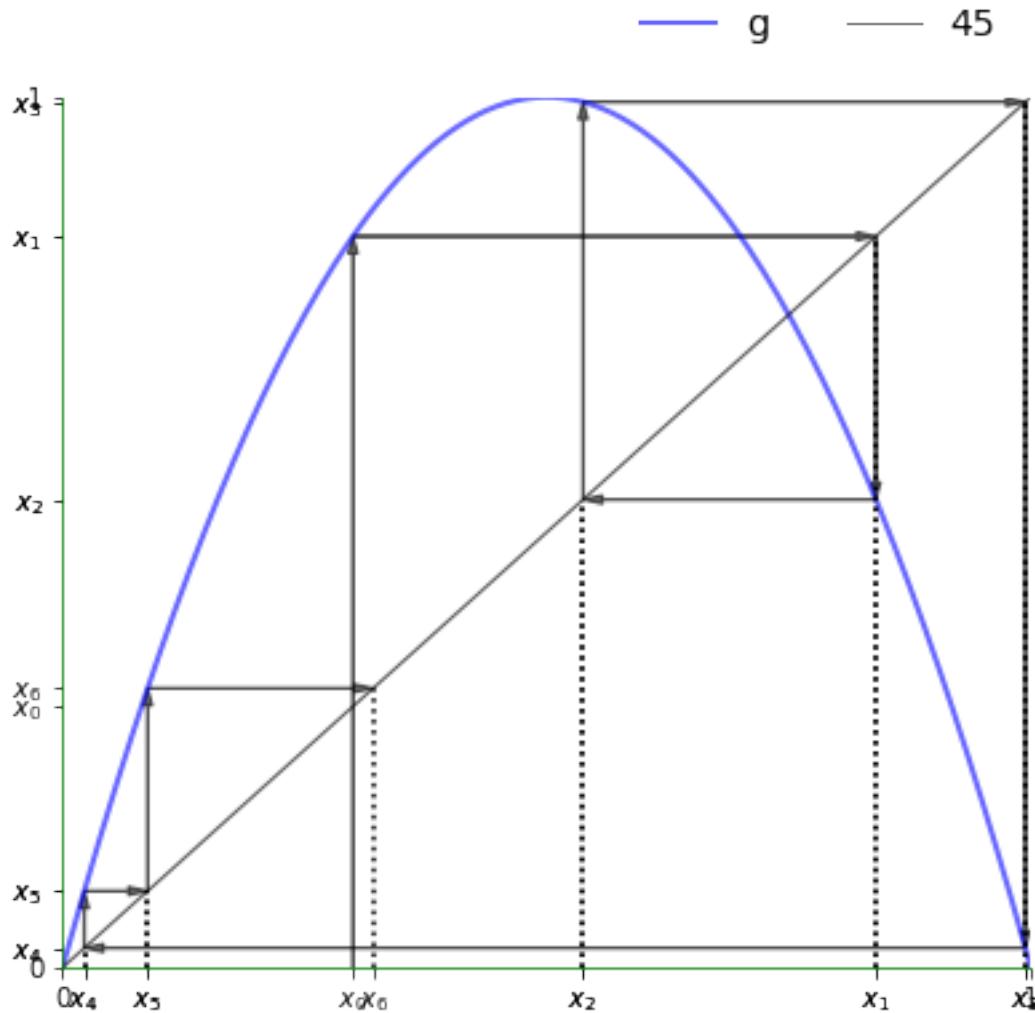
```
In [11]: xmin, xmax = 0, 1
g = lambda x: 4 * x * (1 - x)

x0 = 0.3
plot45(g, xmin, xmax, x0, num_arrows=0)
```



Now let's look at a typical trajectory.

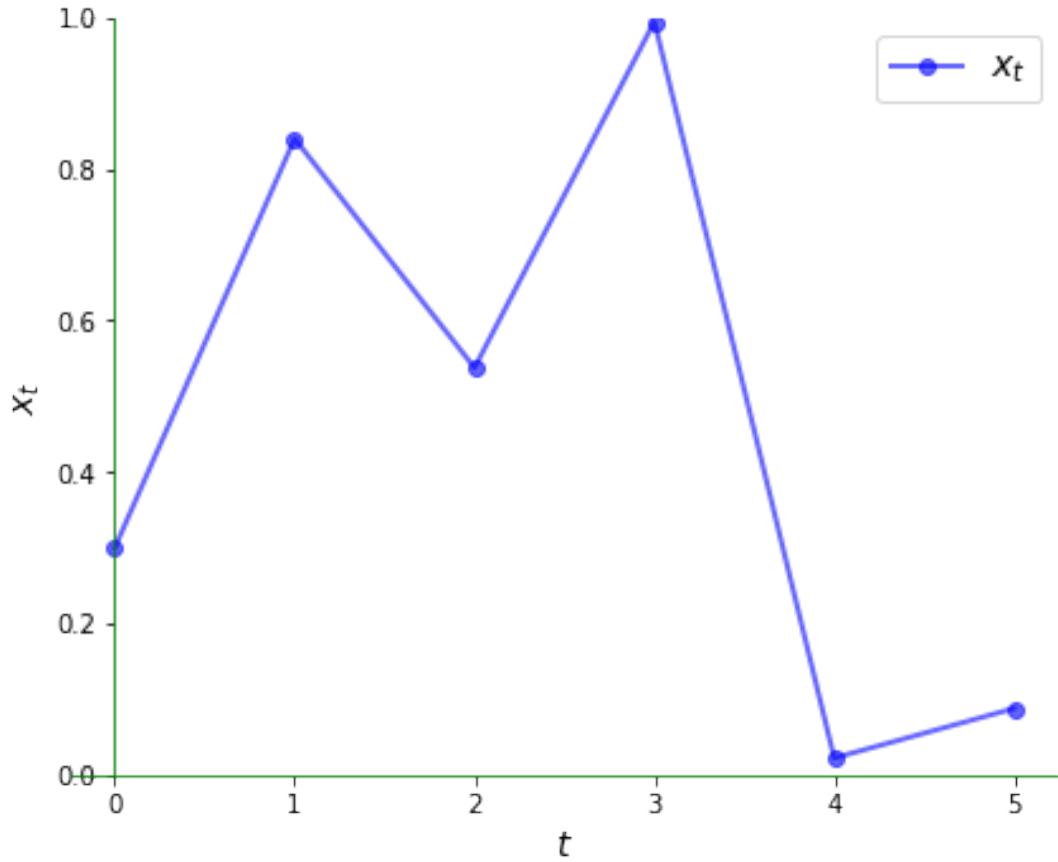
```
In [12]: plot45(g, xmin, xmax, x0, num_arrows=6)
```



Notice how irregular it is.

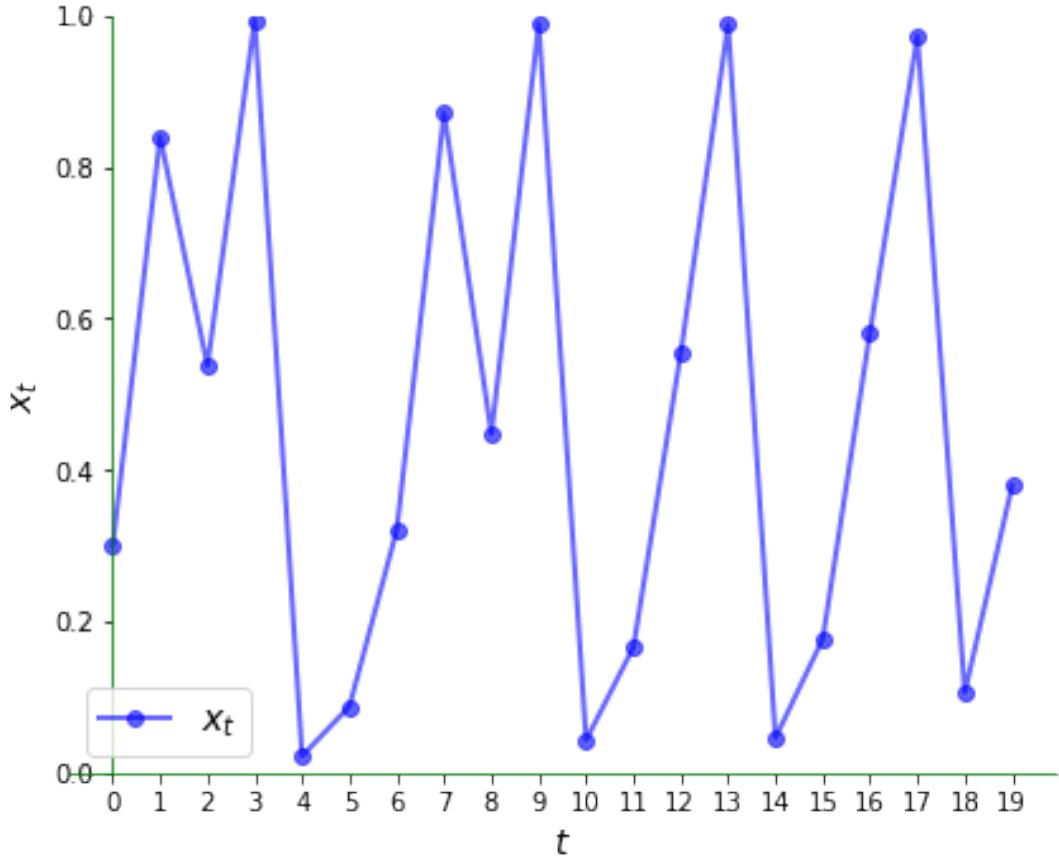
Here is the corresponding time series plot.

```
In [13]: ts_plot(g, xmin, xmax, x0, ts_length=6)
```



The irregularity is even clearer over a longer time horizon:

```
In [14]: ts_plot(g, xmin, xmax, x0, ts_length=20)
```



10.5 Exercises

10.5.1 Exercise 1

Consider again the linear model $x_{t+1} = ax_t + b$ with $a \neq 1$.

The unique steady state is $b/(1 - a)$.

The steady state is globally stable if $|a| < 1$.

Try to illustrate this graphically by looking at a range of initial conditions.

What differences do you notice in the cases $a \in (-1, 0)$ and $a \in (0, 1)$?

Use $a = 0.5$ and then $a = -0.5$ and study the trajectories

Set $b = 1$ throughout.

10.6 Solutions

10.6.1 Exercise 1

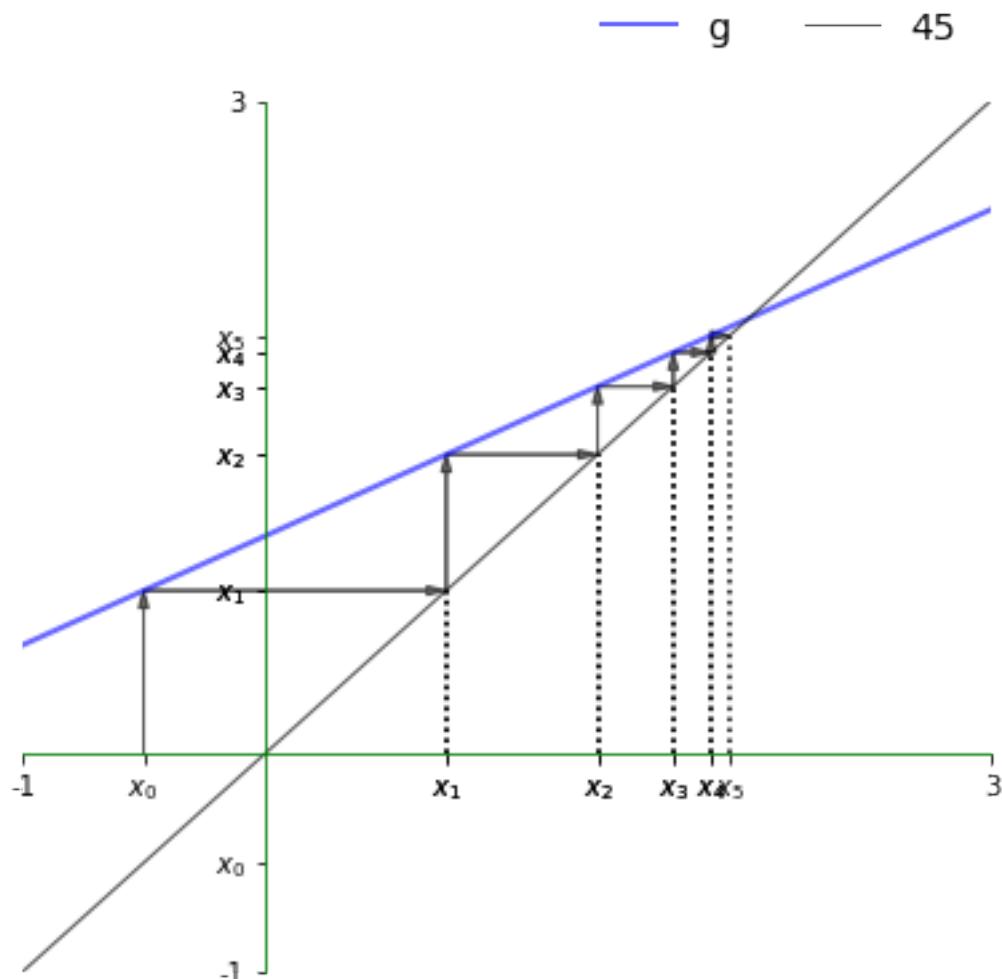
We will start with the case $a = 0.5$.

Let's set up the model and plotting region:

```
In [15]: a, b = 0.5, 1
xmin, xmax = -1, 3
g = lambda x: a * x + b
```

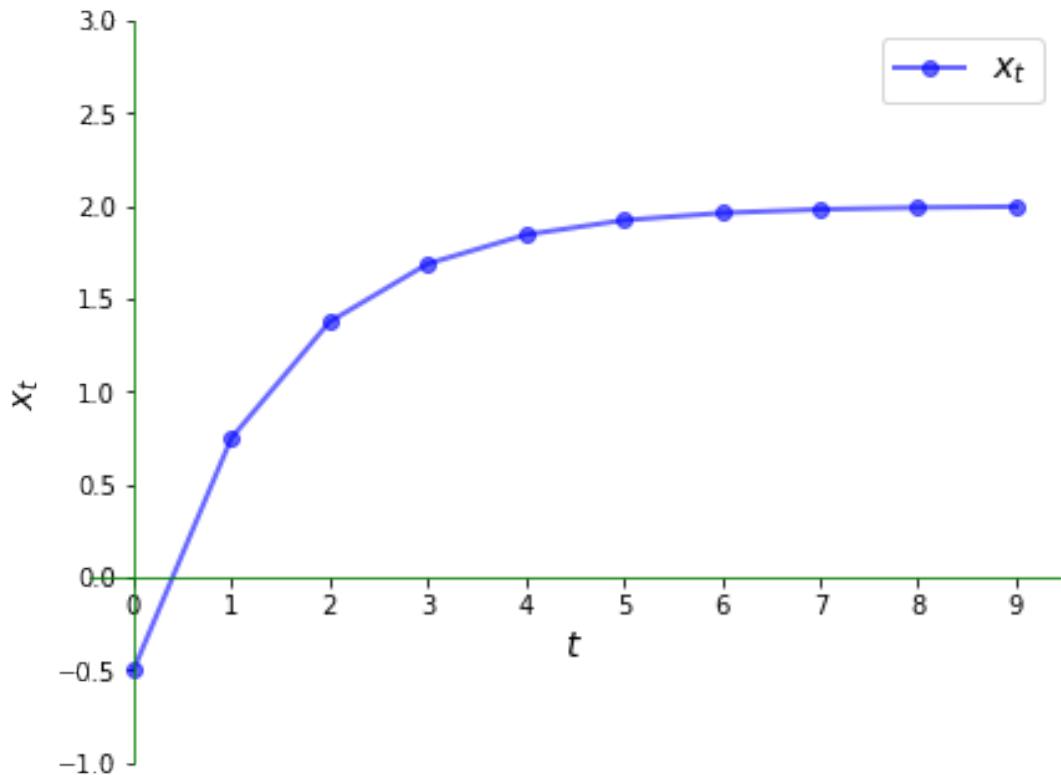
Now let's plot a trajectory:

```
In [16]: x0 = -0.5
plot45(g, xmin, xmax, x0, num_arrows=5)
```



Here is the corresponding time series, which converges towards the steady state.

```
In [17]: ts_plot(g, xmin, xmax, x0, ts_length=10)
```



Now let's try $a = -0.5$ and see what differences we observe.

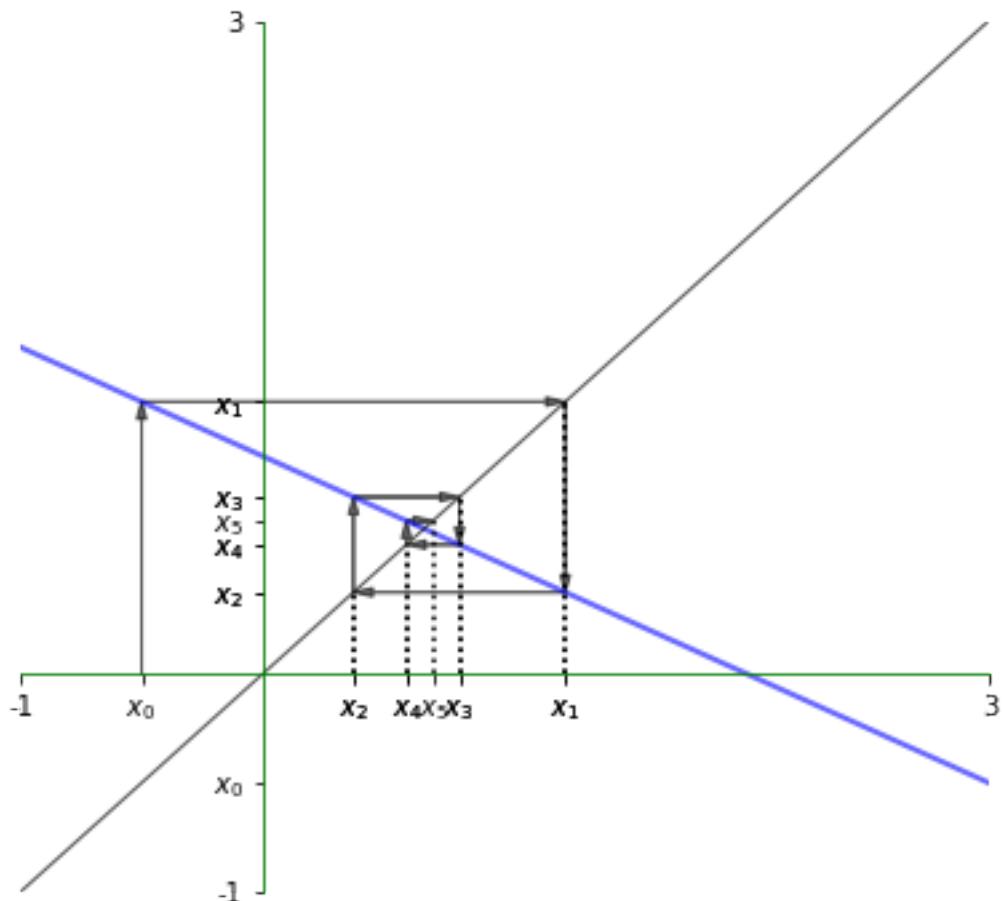
Let's set up the model and plotting region:

```
In [18]: a, b = -0.5, 1
xmin, xmax = -1, 3
g = lambda x: a * x + b
```

Now let's plot a trajectory:

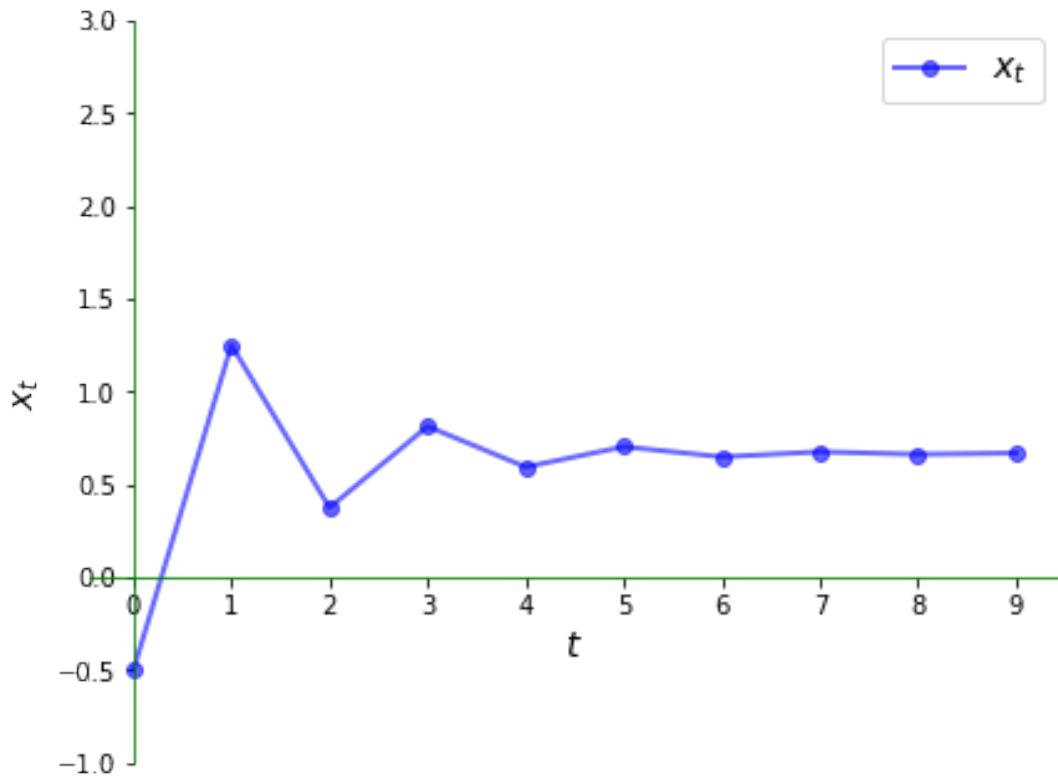
```
In [19]: x0 = -0.5
plot45(g, xmin, xmax, x0, num_arrows=5)
```

— g — 45



Here is the corresponding time series, which converges towards the steady state.

```
In [20]: ts_plot(g, xmin, xmax, x0, ts_length=10)
```



Once again, we have convergence to the steady state but the nature of convergence differs. In particular, the time series jumps from above the steady state to below it and back again. In the current context, the series is said to exhibit **damped oscillations**.

Chapter 11

AR1 Processes

11.1 Contents

- Overview 11.2
- The AR(1) Model 11.3
- Stationarity and Asymptotic Stability 11.4
- Ergodicity 11.5
- Exercises 11.6
- Solutions 11.7

11.2 Overview

In this lecture we are going to study a very simple class of stochastic models called AR(1) processes.

These simple models are used again and again in economic research to represent the dynamics of series such as

- labor income
- dividends
- productivity, etc.

AR(1) processes can take negative values but are easily converted into positive processes when necessary by a transformation such as exponentiation.

We are going to study AR(1) processes partly because they are useful and partly because they help us understand important concepts.

Let's start with some imports:

```
In [1]: import numpy as np  
import matplotlib.pyplot as plt  
%matplotlib inline
```

11.3 The AR(1) Model

The **AR(1) model** (autoregressive model of order 1) takes the form

$$X_{t+1} = aX_t + b + cW_{t+1} \quad (1)$$

where a, b, c are scalar-valued parameters.

This law of motion generates a time series $\{X_t\}$ as soon as we specify an initial condition X_0 .

This is called the **state process** and the state space is \mathbb{R} .

To make things even simpler, we will assume that

- the process $\{W_t\}$ is IID and standard normal,
- the initial condition X_0 is drawn from the normal distribution $N(\mu_0, v_0)$ and
- the initial condition X_0 is independent of $\{W_t\}$.

11.3.1 Moving Average Representation

Iterating backwards from time t , we obtain

$$X_t = aX_{t-1} + b + cW_t = a^2X_{t-2} + ab + acW_{t-1} + b + cW_t = \dots$$

If we work all the way back to time zero, we get

$$X_t = a^t X_0 + b \sum_{j=0}^{t-1} a^j + c \sum_{j=0}^{t-1} a^j W_{t-j} \quad (2)$$

Equation (2) shows that X_t is a well defined random variable, the value of which depends on

- the parameters,
- the initial condition X_0 and
- the shocks W_1, \dots, W_t from time $t = 1$ to the present.

Throughout, the symbol ψ_t will be used to refer to the density of this random variable X_t .

11.3.2 Distribution Dynamics

One of the nice things about this model is that it's so easy to trace out the sequence of distributions $\{\psi_t\}$ corresponding to the time series $\{X_t\}$.

To see this, we first note that X_t is normally distributed for each t .

This is immediate from (2), since linear combinations of independent normal random variables are normal.

Given that X_t is normally distributed, we will know the full distribution ψ_t if we can pin down its first two moments.

Let μ_t and v_t denote the mean and variance of X_t respectively.

We can pin down these values from (2) or we can use the following recursive expressions:

$$\mu_{t+1} = a\mu_t + b \quad \text{and} \quad v_{t+1} = a^2v_t + c^2 \quad (3)$$

These expressions are obtained from (1) by taking, respectively, the expectation and variance of both sides of the equality.

In calculating the second expression, we are using the fact that X_t and W_{t+1} are independent. (This follows from our assumptions and (2).)

Given the dynamics in (2) and initial conditions μ_0, v_0 , we obtain μ_t, v_t and hence

$$\psi_t = N(\mu_t, v_t)$$

The following code uses these facts to track the sequence of marginal distributions $\{\psi_t\}$.

The parameters are

In [2]: `a, b, c = 0.9, 0.1, 0.5`

```
mu, v = -3.0, 0.6 # initial conditions mu_0, v_0
```

Here's the sequence of distributions:

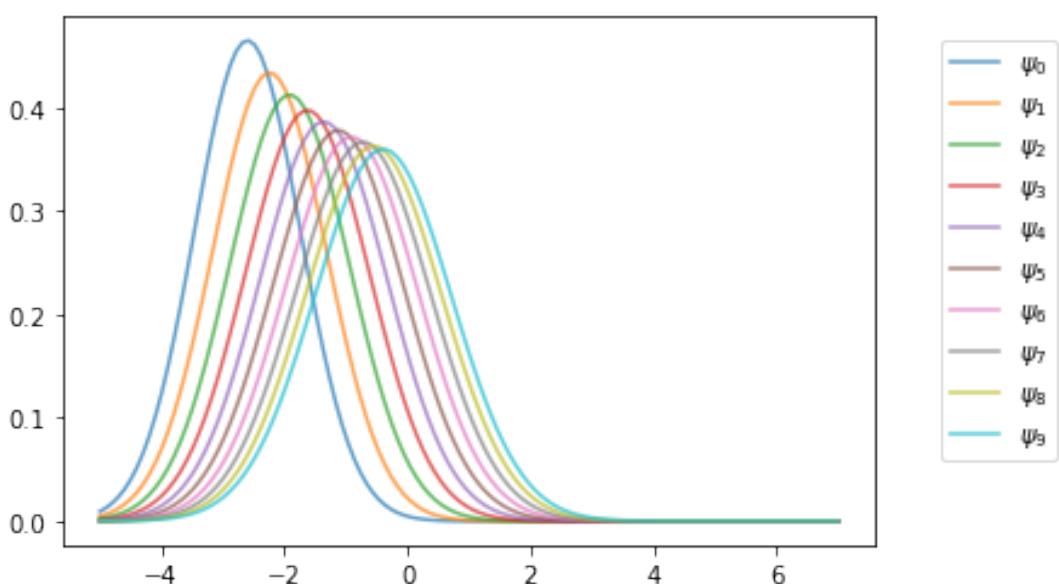
In [3]: `from scipy.stats import norm`

```
sim_length = 10
grid = np.linspace(-5, 7, 120)

fig, ax = plt.subplots()

for t in range(sim_length):
    mu = a * mu + b
    v = a**2 * v + c**2
    ax.plot(grid, norm.pdf(grid, loc=mu, scale=np.sqrt(v)),
            label=f"\psi_{t}",
            alpha=0.7)

ax.legend(bbox_to_anchor=[1.05, 1], loc=2, borderaxespad=1)
plt.show()
```



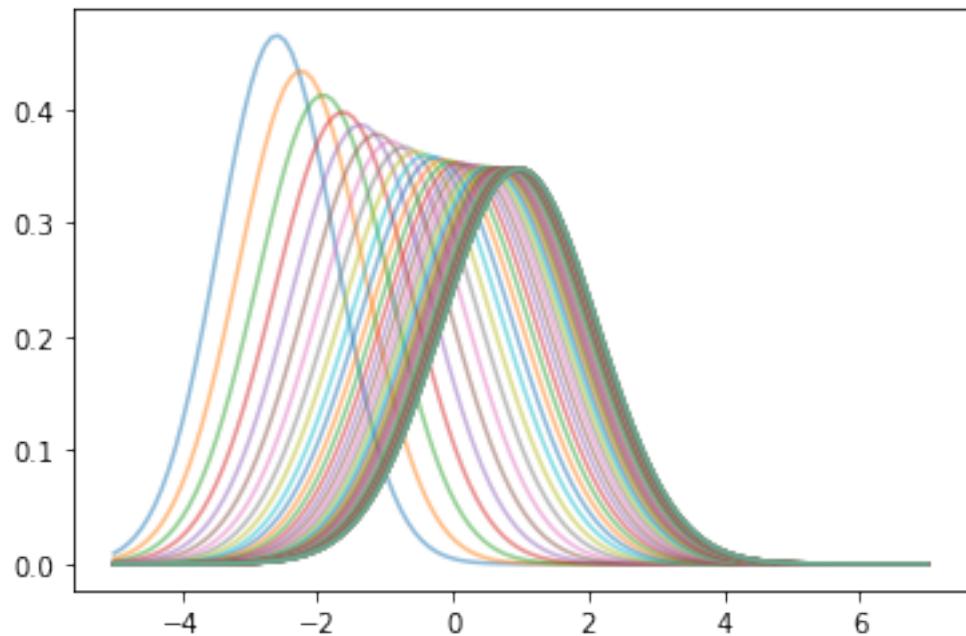
11.4 Stationarity and Asymptotic Stability

Notice that, in the figure above, the sequence $\{\psi_t\}$ seems to be converging to a limiting distribution.

This is even clearer if we project forward further into the future:

```
In [4]: def plot_density_seq(ax, mu_0=-3.0, v_0=0.6, sim_length=60):
    mu, v = mu_0, v_0
    for t in range(sim_length):
        mu = a * mu + b
        v = a**2 * v + c**2
        ax.plot(grid,
                 norm.pdf(grid, loc=mu, scale=np.sqrt(v)),
                 alpha=0.5)

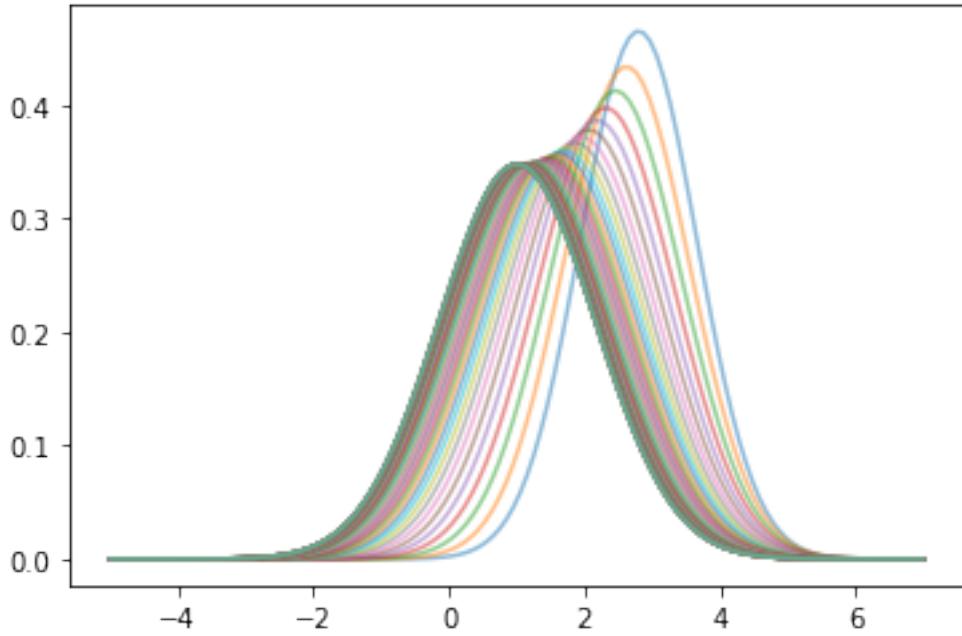
fig, ax = plt.subplots()
plot_density_seq(ax)
plt.show()
```



Moreover, the limit does not depend on the initial condition.

For example, this alternative density sequence also converges to the same limit.

```
In [5]: fig, ax = plt.subplots()
plot_density_seq(ax, mu_0=3.0)
plt.show()
```



In fact it's easy to show that such convergence will occur, regardless of the initial condition, whenever $|a| < 1$.

To see this, we just have to look at the dynamics of the first two moments, as given in (3).

When $|a| < 1$, these sequences converge to the respective limits

$$\mu^* := \frac{b}{1-a} \quad \text{and} \quad v^* = \frac{c^2}{1-a^2} \quad (4)$$

(See our [lecture on one dimensional dynamics](#) for background on deterministic convergence.)

Hence

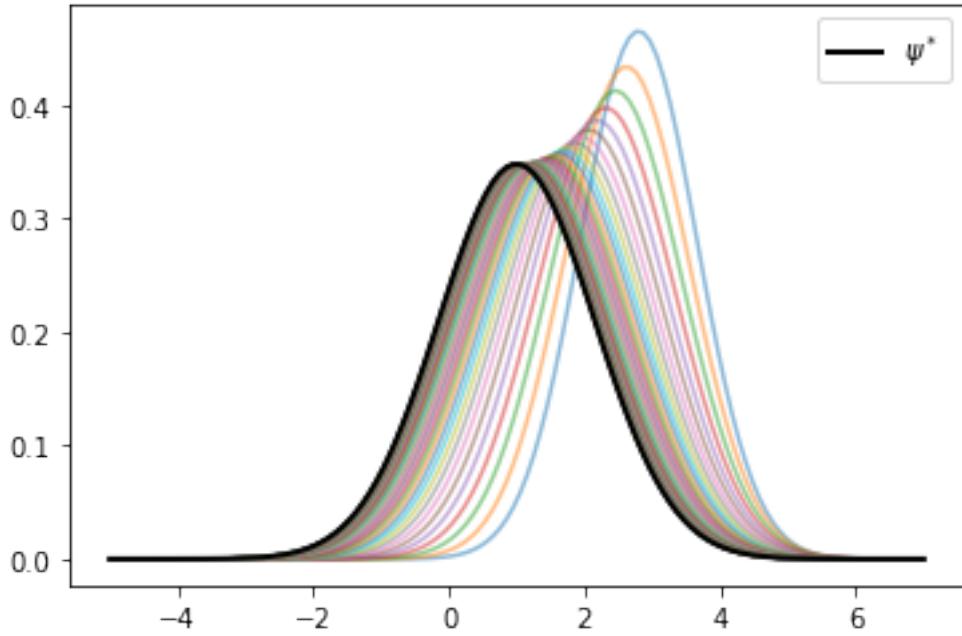
$$\psi_t \rightarrow \psi^* = N(\mu^*, v^*) \quad \text{as } t \rightarrow \infty \quad (5)$$

We can confirm this is valid for the sequence above using the following code.

```
In [6]: fig, ax = plt.subplots()
plot_density_seq(ax, mu_0=3.0)

mu_star = b / (1 - a)
std_star = np.sqrt(c**2 / (1 - a**2)) # square root of v_star
psi_star = norm.pdf(grid, loc=mu_star, scale=std_star)
ax.plot(grid, psi_star, 'k-', lw=2, label="$\psi^*$")
ax.legend()

plt.show()
```



As claimed, the sequence $\{\psi_t\}$ converges to ψ^* .

11.4.1 Stationary Distributions

A stationary distribution is a distribution that is a fixed point of the update rule for distributions.

In other words, if ψ_t is stationary, then $\psi_{t+j} = \psi_t$ for all j in \mathbb{N} .

A different way to put this, specialized to the current setting, is as follows: a density ψ on \mathbb{R} is **stationary** for the AR(1) process if

$$X_t \sim \psi \implies aX_t + b + cW_{t+1} \sim \psi$$

The distribution ψ^* in (5) has this property — checking this is an exercise.

(Of course, we are assuming that $|a| < 1$ so that ψ^* is well defined.)

In fact, it can be shown that no other distribution on \mathbb{R} has this property.

Thus, when $|a| < 1$, the AR(1) model has exactly one stationary density and that density is given by ψ^* .

11.5 Ergodicity

The concept of ergodicity is used in different ways by different authors.

One way to understand it in the present setting is that a version of the Law of Large Numbers is valid for $\{X_t\}$, even though it is not IID.

In particular, averages over time series converge to expectations under the stationary distribution.

Indeed, it can be proved that, whenever $|a| < 1$, we have

$$\frac{1}{m} \sum_{t=1}^m h(X_t) \rightarrow \int h(x)\psi^*(x)dx \quad \text{as } m \rightarrow \infty \quad (6)$$

whenever the integral on the right hand side is finite and well defined.

Notes:

- In (6), convergence holds with probability one.
- The textbook by [81] is a classic reference on ergodicity.

For example, if we consider the identity function $h(x) = x$, we get

$$\frac{1}{m} \sum_{t=1}^m X_t \rightarrow \int x\psi^*(x)dx \quad \text{as } m \rightarrow \infty$$

In other words, the time series sample mean converges to the mean of the stationary distribution.

As will become clear over the next few lectures, ergodicity is a very important concept for statistics and simulation.

11.6 Exercises

11.6.1 Exercise 1

Let k be a natural number.

The k -th central moment of a random variable is defined as

$$M_k := \mathbb{E}[(X - \mathbb{E}X)^k]$$

When that random variable is $N(\mu, \sigma^2)$, it is known that

$$M_k = \begin{cases} 0 & \text{if } k \text{ is odd} \\ \sigma^k (k-1)!! & \text{if } k \text{ is even} \end{cases}$$

Here $n!!$ is the double factorial.

According to (6), we should have, for any $k \in \mathbb{N}$,

$$\frac{1}{m} \sum_{t=1}^m (X_t - \mu^*)^k \approx M_k$$

when m is large.

Confirm this by simulation at a range of k using the default parameters from the lecture.

11.6.2 Exercise 2

Write your own version of a one dimensional [kernel density estimator](#), which estimates a density from a sample.

Write it as a class that takes the data X and bandwidth h when initialized and provides a method f such that

$$f(x) = \frac{1}{hn} \sum_{i=1}^n K\left(\frac{x - X_i}{h}\right)$$

For K use the Gaussian kernel (K is the standard normal density).

Write the class so that the bandwidth defaults to Silverman's rule (see the "rule of thumb" discussion on [this page](#)). Test the class you have written by going through the steps

1. simulate data X_1, \dots, X_n from distribution ϕ
2. plot the kernel density estimate over a suitable range
3. plot the density of ϕ on the same figure

for distributions ϕ of the following types

- [beta distribution](#) with $\alpha = \beta = 2$
- [beta distribution](#) with $\alpha = 2$ and $\beta = 5$
- [beta distribution](#) with $\alpha = \beta = 0.5$

Use $n = 500$.

Make a comment on your results. (Do you think this is a good estimator of these distributions?)

11.6.3 Exercise 3

In the lecture we discussed the following fact: for the $AR(1)$ process

$$X_{t+1} = aX_t + b + cW_{t+1}$$

with $\{W_t\}$ iid and standard normal,

$$\psi_t = N(\mu, s^2) \implies \psi_{t+1} = N(a\mu + b, a^2s^2 + c^2)$$

Confirm this, at least approximately, by simulation. Let

- $a = 0.9$
- $b = 0.0$
- $c = 0.1$
- $\mu = -3$
- $s = 0.2$

First, plot ψ_t and ψ_{t+1} using the true distributions described above.

Second, plot ψ_{t+1} on the same figure (in a different color) as follows:

1. Generate n draws of X_t from the $N(\mu, s^2)$ distribution
2. Update them all using the rule $X_{t+1} = aX_t + b + cW_{t+1}$
3. Use the resulting sample of X_{t+1} values to produce a density estimate via kernel density estimation.

Try this for $n = 2000$ and confirm that the simulation based estimate of ψ_{t+1} does converge to the theoretical distribution.

11.7 Solutions

11.7.1 Exercise 1

```
In [7]: from numba import njit
from scipy.special import factorial2

@njit
def sample_moments_ar1(k, m=100_000, mu_0=0.0, sigma_0=1.0, seed=1234):
    np.random.seed(seed)
    sample_sum = 0.0
    x = mu_0 + sigma_0 * np.random.randn()
    for t in range(m):
        sample_sum += (x - mu_star)**k
        x = a * x + b + c * np.random.randn()
    return sample_sum / m

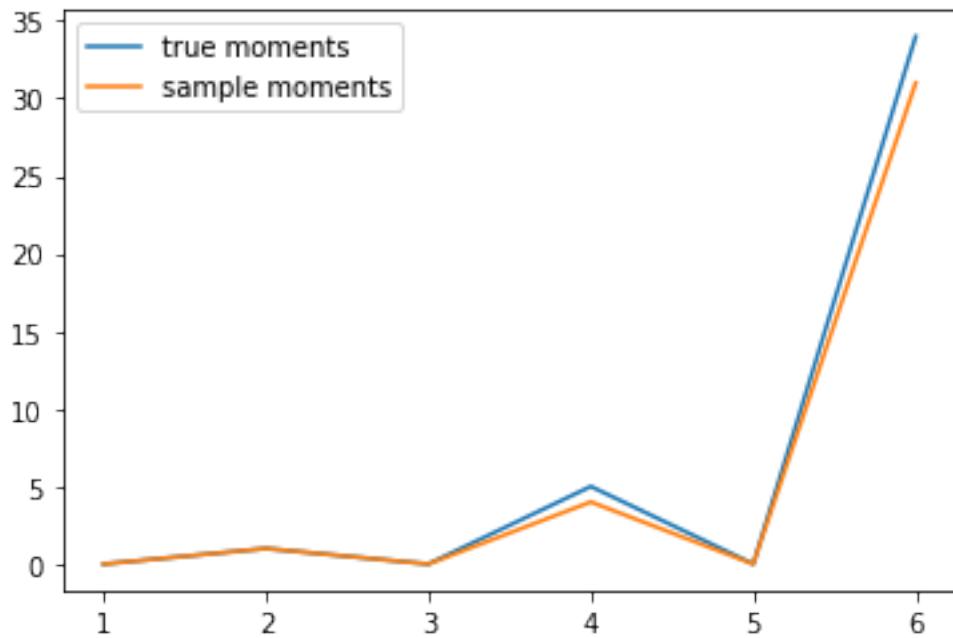
def true_moments_ar1(k):
    if k % 2 == 0:
        return std_star**k * factorial2(k - 1)
    else:
        return 0

k_vals = np.arange(6) + 1
sample_moments = np.empty_like(k_vals)
true_moments = np.empty_like(k_vals)

for k_idx, k in enumerate(k_vals):
    sample_moments[k_idx] = sample_moments_ar1(k)
    true_moments[k_idx] = true_moments_ar1(k)

fig, ax = plt.subplots()
ax.plot(k_vals, true_moments, label="true moments")
ax.plot(k_vals, sample_moments, label="sample moments")
ax.legend()

plt.show()
```



11.7.2 Exercise 2

Here is one solution:

```
In [8]: K = norm.pdf
```

```
class KDE:

    def __init__(self, x_data, h=None):
        if h is None:
            c = x_data.std()
            n = len(x_data)
            h = 1.06 * c * n**(-1/5)
        self.h = h
        self.x_data = x_data

    def f(self, x):
        if np.isscalar(x):
            return K((x - self.x_data) / self.h).mean() ** (1/self.h)
        else:
            y = np.empty_like(x)
            for i, x_val in enumerate(x):
                y[i] = K((x_val - self.x_data) / self.h).mean() ** (1/self.h)
            return y
```

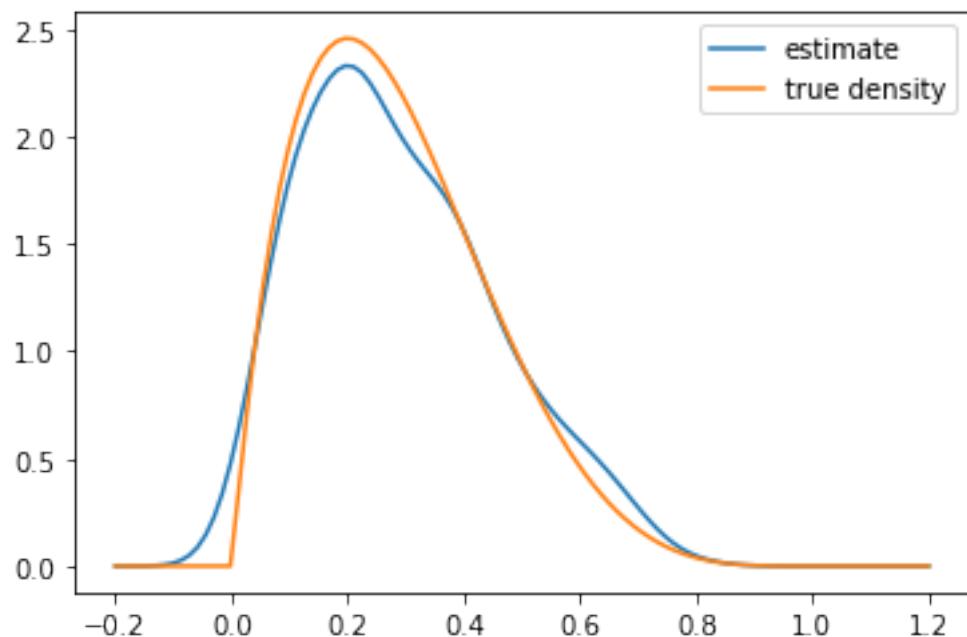
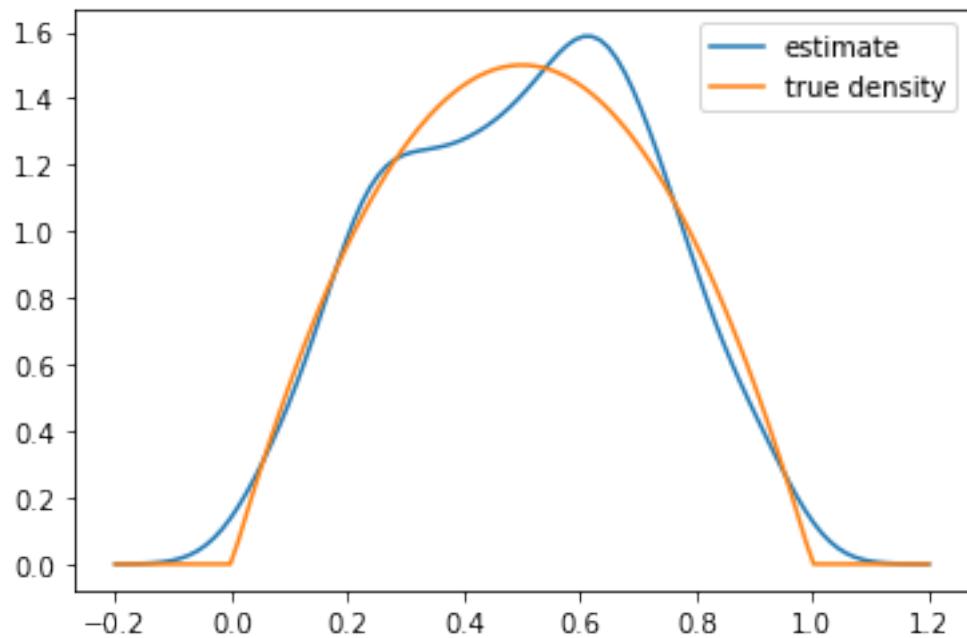
```
In [9]: def plot_kde(phi, x_min=-0.2, x_max=1.2):
    x_data = phi.rvs(n)
    kde = KDE(x_data)
```

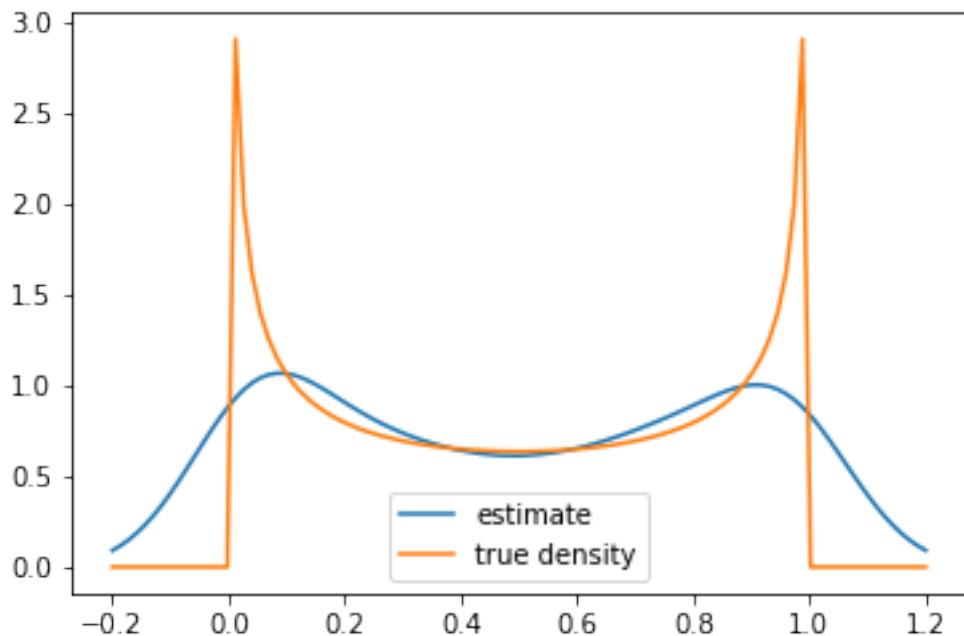
```
x_grid = np.linspace(-0.2, 1.2, 100)
fig, ax = plt.subplots()
```

```
ax.plot(x_grid, kde.f(x_grid), label="estimate")
ax.plot(x_grid, phi.pdf(x_grid), label="true density")
ax.legend()
plt.show()
```

In [10]: `from scipy.stats import beta`

```
n = 500
parameter_pairs= (2, 2), (2, 5), (0.5, 0.5)
for alpha, beta in parameter_pairs:
    plot_kde(beta(alpha, beta))
```





We see that the kernel density estimator is effective when the underlying distribution is smooth but less so otherwise.

11.7.3 Exercise 3

Here is our solution

```
In [11]: a = 0.9
b = 0.0
c = 0.1
μ = -3
s = 0.2
```

```
In [12]: μ_next = a * μ + b
s_next = np.sqrt(a**2 * s**2 + c**2)
```

```
In [13]: ψ = lambda x: K((x - μ) / s)
ψ_next = lambda x: K((x - μ_next) / s_next)
```

```
In [14]: ψ = norm(μ, s)
ψ_next = norm(μ_next, s_next)
```

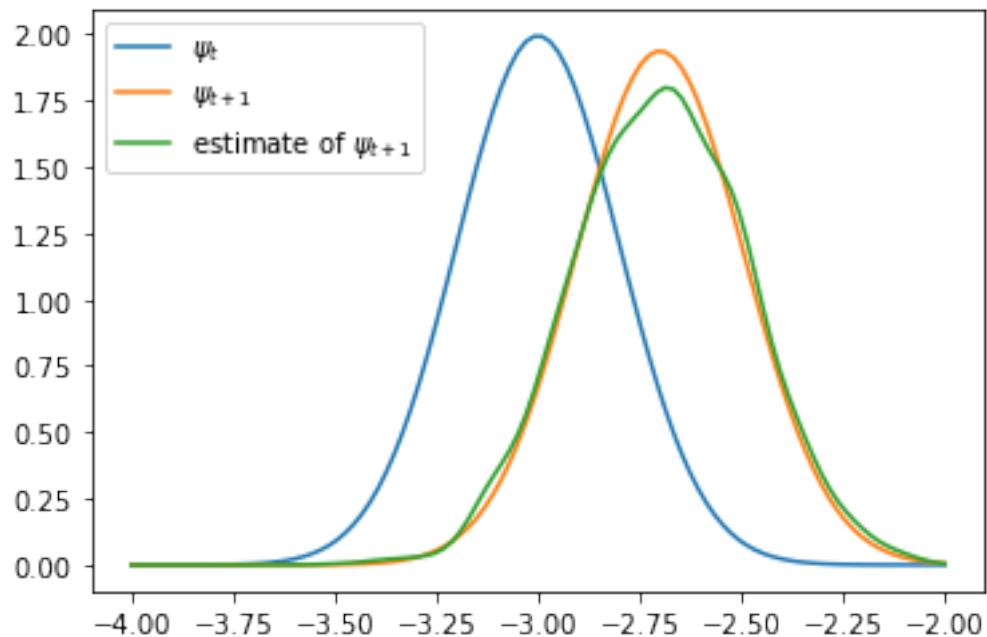
```
In [15]: n = 2000
x_draws = ψ.rvs(n)
x_draws_next = a * x_draws + b + c * np.random.randn(n)
kde = KDE(x_draws_next)

x_grid = np.linspace(μ - 1, μ + 1, 100)
```

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

ax.plot(x_grid, ψ.pdf(x_grid), label="$\psi_t$")
ax.plot(x_grid, ψ_next.pdf(x_grid), label="$\psi_{t+1}$")
ax.plot(x_grid, kde.f(x_grid), label="estimate of $\psi_{t+1}$")

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



The simulated distribution approximately coincides with the theoretical distribution, as predicted.

Chapter 12

Finite Markov Chains

12.1 Contents

- Overview 12.2
- Definitions 12.3
- Simulation 12.4
- Marginal Distributions 12.5
- Irreducibility and Aperiodicity 12.6
- Stationary Distributions 12.7
- Ergodicity 12.8
- Computing Expectations 12.9
- Exercises 12.10
- Solutions 12.11

In addition to what's in Anaconda, this lecture will need the following libraries:

```
In [1]: !pip install quantecon
```

12.2 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:

```
In [2]: import quantecon as qe
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
```

```

import matplotlib.pyplot as plt
%matplotlib inline

/home/ubuntu/anaconda3/lib/python3.7/site-packages/numba/np/ufunc/parallel.py:
  ↪355:
NumbaWarning: The TBB threading layer requires TBB version 2019.5 or later i.e.,
TBB_INTERFACE_VERSION >= 11005. Found TBB_INTERFACE_VERSION = 11004. The TBB
  ↪threading
layer is disabled.
warnings.warn(problem)

```

12.3 Definitions

The following concepts are fundamental.

12.3.1 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 Section ?? that if P is a stochastic matrix, then so is the k -th power P^k for all $k \in \mathbb{N}$.

12.3.2 Markov Chains

There is a close connection between stochastic matrices and Markov chains.

To begin, let S be a finite set with n elements $\{x_1, \dots, x_n\}$.

The set S is called the **state space** and x_1, \dots, 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 $y \in S$,

$$\mathbb{P}\{X_{t+1} = y | X_t\} = \mathbb{P}\{X_{t+1} = y | X_t, X_{t-1}, \dots\} \quad (1)$$

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

$$P(x, y) := \mathbb{P}\{X_{t+1} = y | X_t = x\} \quad (x, y \in S) \quad (2)$$

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

$$P_{ij} = P(x_i, x_j) \quad 1 \leq i, j \leq n$$

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 some specified distribution
- for each $t = 0, 1, \dots$, draw X_{t+1} from $P(X_t, \cdot)$

By construction, the resulting process satisfies (2).

12.3.3 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

$$P = \begin{pmatrix} 1 - \alpha & \alpha \\ \beta & 1 - \beta \end{pmatrix} \quad (3)$$

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.

12.3.4 Example 2

Using US unemployment data, Hamilton [49] estimated the stochastic matrix

$$P = \begin{pmatrix} 0.971 & 0.029 & 0 \\ 0.145 & 0.778 & 0.077 \\ 0 & 0.508 & 0.492 \end{pmatrix}$$

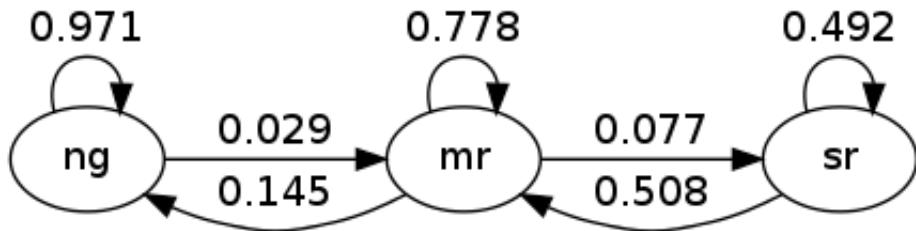
where

- 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.

12.4 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 $S = 0, \dots, n - 1$.

12.4.1 Rolling Our Own

To simulate a Markov chain, we need its stochastic matrix P and a probability distribution ψ for the initial state to be drawn from.

The Markov chain is then constructed as discussed above. To repeat:

1. At time $t = 0$, the X_0 is chosen from ψ .
2. At each subsequent time t , the new state X_{t+1} is drawn 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:

```
In [3]: ψ = (0.3, 0.7)          # probabilities over {0, 1}
      cdf = np.cumsum(ψ)        # convert into cumulative distribution
      qe.random.draw(cdf, 5)    # generate 5 independent draws from ψ

Out[3]: array([1, 1, 1, 1, 1])
```

We'll write our code as a function that takes 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

```
In [4]: def mc_sample_path(P, ψ_0=None, sample_size=1_000):
```

```
# set up
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

```
In [5]: 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

```
In [6]: X = mc_sample_path(P, ψ_0=[0.1, 0.9], sample_size=100_000)
np.mean(X == 0)
```

```
Out[6]: 0.24955
```

You can try changing the initial distribution to confirm that the output is always close to 0.25.

12.4.2 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

In [7]: `from quantecon import MarkovChain`

```
mc = qe.MarkovChain(P)
X = mc.simulate(ts_length=1_000_000)
np.mean(X == 0)
```

Out[7]: 0.250416

The `QuantEcon.py` routine is JIT compiled and much faster.

In [8]: `%time mc_sample_path(P, sample_size=1_000_000) # Our version`

```
CPU times: user 1.4 s, sys: 0 ns, total: 1.4 s
Wall time: 1.43 s
```

Out[8]: `array([0, 1, 1, ..., 0, 0, 1])`

In [9]: `%time mc.simulate(ts_length=1_000_000) # qe version`

```
CPU times: user 50.5 ms, sys: 2.94 ms, total: 53.4 ms
Wall time: 54.9 ms
```

Out[9]: `array([0, 0, 1, ..., 0, 0, 1])`

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

In [10]: `mc = qe.MarkovChain(P, state_values=('unemployed', 'employed'))
mc.simulate(ts_length=4, init='employed')`

Out[10]: `array(['employed', 'employed', 'employed', 'employed'], dtype='<U10')`

In [11]: `mc.simulate(ts_length=4, init='unemployed')`

Out[11]: `array(['unemployed', 'employed', 'employed', 'employed'], dtype='<U10')`

In [12]: `mc.simulate(ts_length=4) # Start at randomly chosen initial state`

```
Out[12]: array(['employed', 'employed', 'employed', 'unemployed'], dtype='<U10')
```

If we want to simulate with output as indices rather than state values we can use

```
In [13]: mc.simulate_indices(ts_length=4)
```

```
Out[13]: array([0, 1, 1, 1])
```

12.5 Marginal Distributions

Suppose that

1. $\{X_t\}$ is a Markov chain with stochastic matrix P
2. the distribution of X_t is known to be ψ_t

What then is the distribution of X_{t+1} , or, more generally, of X_{t+m} ?

To answer this, we let ψ_t be the distribution of X_t for $t = 0, 1, 2, \dots$.

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:

$$\mathbb{P}\{X_{t+1} = y\} = \sum_{x \in S} \mathbb{P}\{X_{t+1} = y \mid X_t = x\} \cdot \mathbb{P}\{X_t = x\}$$

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* (as is traditional in this literature), these n equations are summarized by the matrix expression

$$\psi_{t+1} = \psi_t P \tag{4}$$

In other words, to move the distribution forward one unit of time, we postmultiply by P .

By repeating this m times we move forward m steps into the future.

Hence, iterating on (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

$$X_0 \sim \psi_0 \implies X_m \sim \psi_0 P^m \quad (5)$$

and, more generally,

$$X_t \sim \psi_t \implies X_{t+m} \sim \psi_t P^m \quad (6)$$

12.5.1 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 (6), but now with ψ_t putting all probability on state x

- 1 in the x -th position and zero elsewhere

Inserting this into (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

$$\mathbb{P}\{X_{t+m} = y \mid X_t = x\} = P^m(x, y) = (x, y)\text{-th element of } P^m$$

12.5.2 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 estimate the probability that the economy is in state x to be $\psi(x)$.

The probability of being in recession (either mild or severe) in 6 months time is given by the inner product

$$\psi P^6 \cdot \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}$$

12.5.3 Example 2: Cross-Sectional Distributions

The marginal distributions we have been studying can be viewed either as probabilities or as cross-sectional frequencies in 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, independent of one another.

Let ψ be the current *cross-sectional* distribution over $\{0, 1\}$.

The cross-sectional distribution records the 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 (3).

This is because each worker is updated according to P , so ψP^{10} represents probabilities for a single randomly selected worker.

But when the sample is large, outcomes and probabilities are roughly equal (by the Law of Large Numbers).

So for a very large (tending to infinite) population, ψP^{10} also represents the fraction of workers in each state.

This is exactly the cross-sectional distribution.

12.6 Irreducibility and Aperiodicity

Irreducibility and aperiodicity are central concepts of modern Markov chain theory.

Let's see what they're about.

12.6.1 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

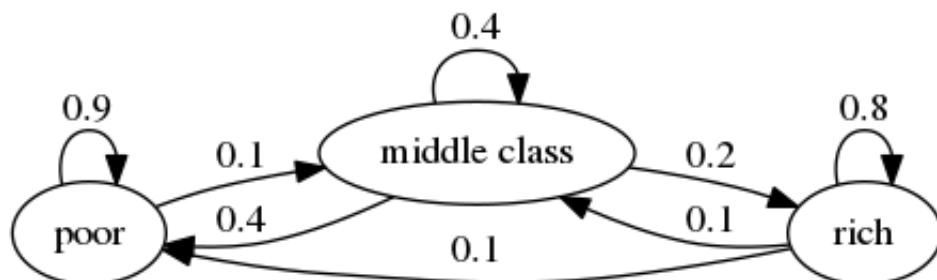
$$P^j(x, y) > 0 \quad \text{and} \quad P^k(y, x) > 0$$

In view of our discussion [above](#), this means precisely that

- state x can be reached eventually from state y , and
- state y can be reached eventually 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 reach any state from any other state eventually.

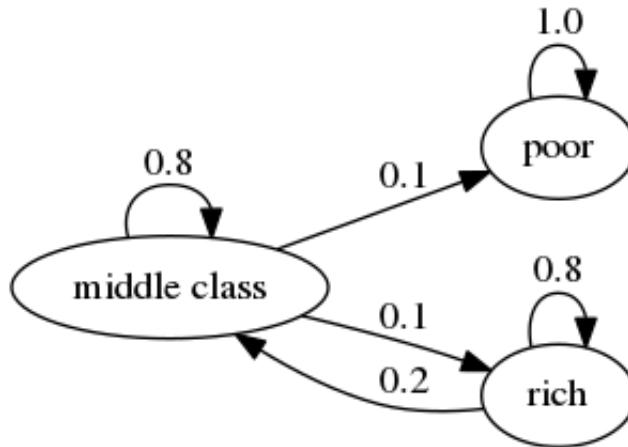
We can also test this using [QuantEcon.py](#)'s `MarkovChain` class

```
In [14]: 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
```

Out[14]: True

Here's a more pessimistic scenario, where the poor are poor forever



This stochastic matrix is not irreducible, since, for example, rich is not accessible from poor.

Let's confirm this

```
In [15]: 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
```

Out[15]: False

We can also determine the “communication classes”

In [16]: `mc.communication_classes`

Out[16]: `[array(['poor']), 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.

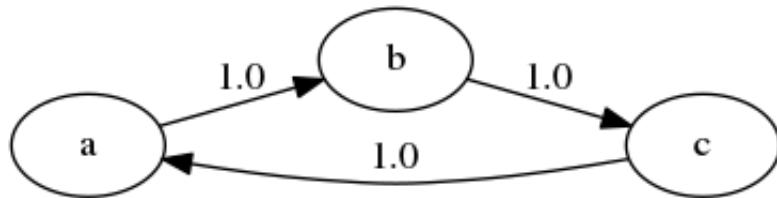
For example, poverty is a life sentence in the second graph but not the first.

We'll come back to this a bit later.

12.6.2 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:

In [17]: `P = [[0, 1, 0],
[0, 0, 1],
[1, 0, 0]]`

`mc = qe.MarkovChain(P)
mc.period`

Out[17]: 3

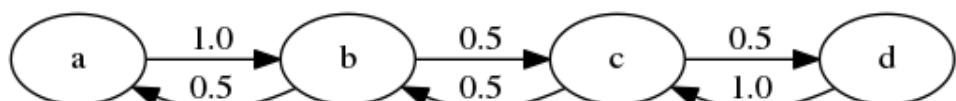
More formally, the **period** of a state x is the greatest common divisor of the set of integers

$$D(x) := \{j \geq 1 : P^j(x, x) > 0\}$$

In the last example, $D(x) = \{3, 6, 9, \dots\}$ 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 as follows

```
In [18]: 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
```

Out[18]: 2

```
In [19]: mc.is_aperiodic
```

Out[19]: False

12.7 Stationary Distributions

As seen in (4), we can shift probabilities forward one unit of time via postmultiplication by P .

Some distributions are invariant under this updating process — for example,

```
In [20]: P = np.array([[0.4, 0.6],
                      [0.2, 0.8]])
ψ = (0.25, 0.75)
ψ @ P
```

Out[20]: array([0.25, 0.75])

Such distributions are called **stationary**, or **invariant**.

Formally, a 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 in just a moment.

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 may in fact be many stationary distributions corresponding to a given stochastic matrix P .

- For example, if P is the identity matrix, then all distributions are stationary.

Since stationary distributions are long run equilibria, to get uniqueness we require that initial conditions are not infinitely persistent.

Infinite persistence of initial conditions occurs if certain regions of the state space cannot be accessed from other regions, which 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 distribution ψ_0 , we have $\|\psi_0 P^t - \psi^*\| \rightarrow 0$ as $t \rightarrow \infty$.

For a proof, see, for example, theorem 5.2 of [45].

(Note that part 1 of the theorem requires only irreducibility, whereas part 2 requires both irreducibility and aperiodicity)

A stochastic matrix satisfying the conditions of the theorem is sometimes called **uniformly ergodic**.

One easy sufficient condition for aperiodicity and irreducibility is that every element of P is strictly positive.

- Try to convince yourself of this.

12.7.1 Example

Recall our model of employment/unemployment dynamics for a given 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 on interpretation below.

Not surprisingly it tends to zero as $\beta \rightarrow 0$, and to one as $\alpha \rightarrow 0$.

12.7.2 Calculating Stationary Distributions

As discussed above, a given 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,

$$\psi_3 := \lambda\psi_1 + (1 - \lambda)\psi_2$$

is a stationary distribution for P for any $\lambda \in [0, 1]$.

If we restrict attention to the case where only one stationary distribution exists, one option for finding it is to try to solve the linear system $\psi(I_n - P) = 0$ for ψ , where I_n is the $n \times n$ identity.

But the zero vector solves this equation, so we need to proceed carefully.

In essence, we need to impose the restriction that the solution must be a probability distribution.

There are various ways to do this.

One option is to regard this as an eigenvector problem: a vector ψ such that $\psi = \psi P$ is a left eigenvector associated with the unit eigenvalue $\lambda = 1$.

A more stable and sophisticated algorithm is implemented in [QuantEcon.py](#).

This is the one we recommend you to use:

```
In [21]: P = [[0.4, 0.6],
             [0.2, 0.8]]

mc = qe.MarkovChain(P)
mc.stationary_distributions # Show all stationary distributions

Out[21]: array([[0.25, 0.75]])
```

12.7.3 Convergence to Stationarity

Part 2 of the Markov chain convergence theorem [stated above](#) tells us that the distribution of X_t converges to the stationary distribution regardless of where we start off.

This adds considerable weight to our interpretation of ψ^* as a stochastic steady state.

The convergence in the theorem is illustrated in the next figure

```
In [22]: 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):
```

```

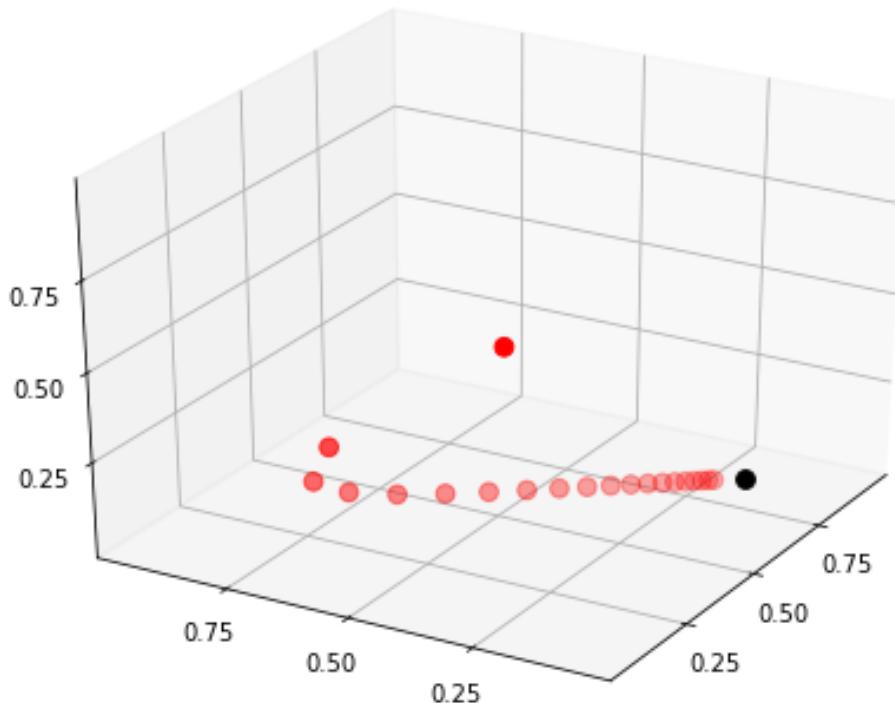
x_vals.append(ψ[0])
y_vals.append(ψ[1])
z_vals.append(ψ[2])
ψ = ψ @ P

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 probability distribution ψ , represented as a vector in \mathbb{R}^3 .
- The other red dots are the distributions ψP^t for $t = 1, 2, \dots$.
- The black dot is ψ^* .

You might like to try experimenting with different initial conditions.

12.8 Ergodicity

Under irreducibility, yet another important result obtains: For all $x \in S$,

$$\frac{1}{m} \sum_{t=1}^m \mathbf{1}\{X_t = x\} \rightarrow \psi^*(x) \quad \text{as } m \rightarrow \infty \quad (7)$$

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 distribution (or value) 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 (7) is valid.

The convergence in (7) is a special case of a law of large numbers result for Markov chains — see [EDTC](#), section 4.3.4 for some additional information.

12.8.1 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 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 interpretation of the notion of ergodicity.

12.9 Computing Expectations

We are interested in computing expectations of the form

$$\mathbb{E}[h(X_t)] \quad (8)$$

and conditional expectations such as

$$\mathbb{E}[h(X_{t+k}) \mid X_t = x] \quad (9)$$

where

- $\{X_t\}$ is a Markov chain generated by $n \times n$ stochastic matrix P

- h is a given function, which, in expressions involving matrix algebra, we'll think of as the column vector

$$h = \begin{pmatrix} h(x_1) \\ \vdots \\ h(x_n) \end{pmatrix}$$

The unconditional expectation (8) is easy: We just sum over the 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 (9), 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

$$\mathbb{E}[h(X_{t+k}) \mid X_t = x] = (P^k h)(x) \tag{10}$$

The vector $P^k h$ stores the conditional expectation $\mathbb{E}[h(X_{t+k}) \mid X_t = x]$ over all x .

12.9.1 Expectations of Geometric Sums

Sometimes we also want to compute expectations of a geometric sum, such as $\sum_t \beta^t h(X_t)$.

In view of the preceding discussion, this is

$$\mathbb{E}\left[\sum_{j=0}^{\infty} \beta^j h(X_{t+j}) \mid X_t = x\right] = [(I - \beta P)^{-1} h](x)$$

where

$$(I - \beta P)^{-1} = I + \beta P + \beta^2 P^2 + \dots$$

Premultiplication by $(I - \beta P)^{-1}$ amounts to “applying the **resolvent operator**”.

12.10 Exercises

12.10.1 Exercise 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 $\bar{X}_m \rightarrow p$ as $m \rightarrow \infty$, where

$$\bar{X}_m := \frac{1}{m} \sum_{t=1}^m \mathbf{1}\{X_t = 0\}$$

The exercise is to illustrate this 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 $(0, 1)$.

12.10.2 Exercise 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 [30].)

To understand the issue, consider the set of results returned by a query to a web search engine.

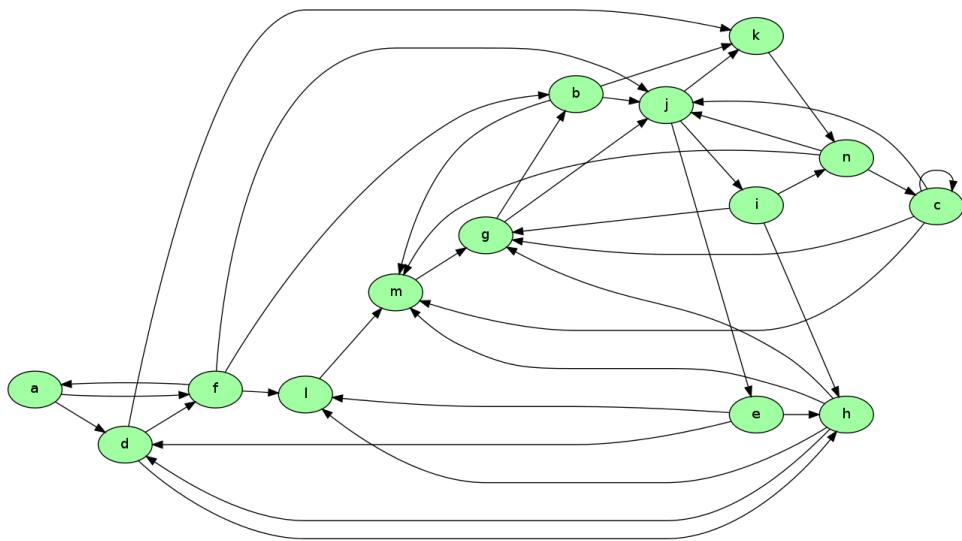
For the user, it is desirable to

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 $P(i, j) = \mathbf{1}\{i \rightarrow j\}/\ell_i$ where $\mathbf{1}\{i \rightarrow j\} = 1$ 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

$$r_j = \sum_{i \in L_j} \frac{r_i}{\ell_i} = \sum_{\text{all } i} \mathbf{1}\{i \rightarrow j\} \frac{r_i}{\ell_i} = \sum_{\text{all } i} P(i, j) r_i$$

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.

In [23]: %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;

```

Writing 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

```

In [24]: import re
re.findall('\w', 'x +++ y ***** z') # \w matches alphanumerics

Out[24]: ['x', 'y', 'z']

In [25]: re.findall('\w', 'a ^& b &&& $$ c')

Out[25]: ['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**.

12.10.3 Exercise 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 [105] 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 $\{x_0, \dots, x_{n-1}\} \subset \mathbb{R}$ 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

$$P(x_i, x_j) = P(x_i, x_0) = F(x_0 - \rho x_i + s/2)$$

1. If $j = n - 1$, then set

$$P(x_i, x_j) = P(x_i, x_{n-1}) = 1 - F(x_{n-1} - \rho x_i - s/2)$$

1. 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 $\{x_0, \dots, x_{n-1}\} \subset \mathbb{R}$ and $n \times n$ matrix P as described above.

- Even better, write a function that returns an instance of [QuantEcon.py](#)'s `MarkovChain` class.

12.11 Solutions

12.11.1 Exercise 1

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.

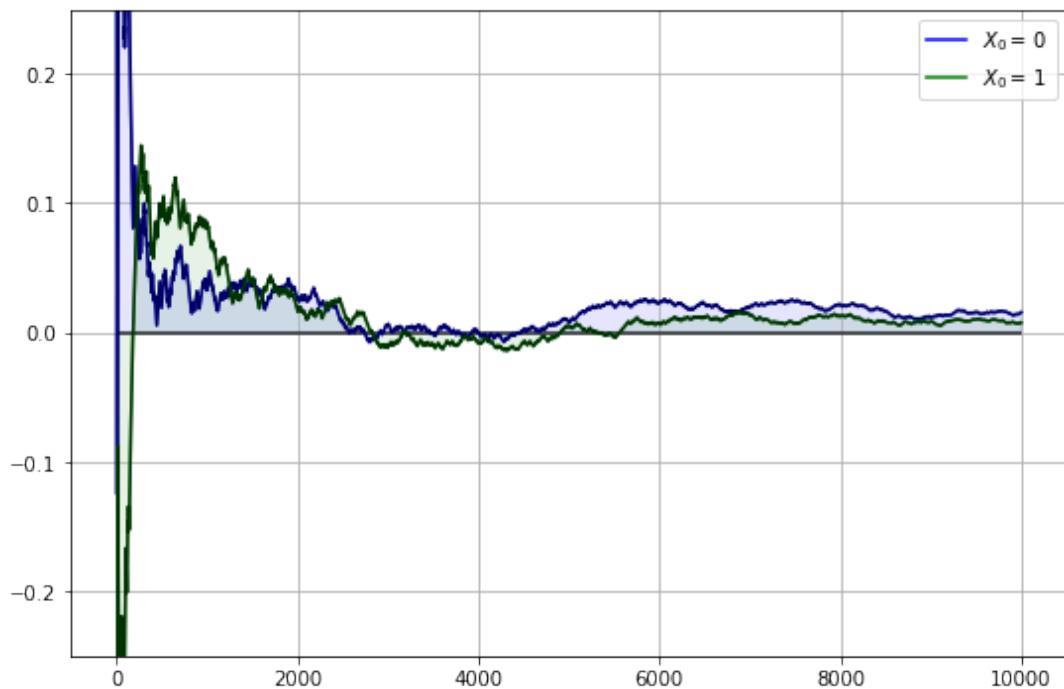
```
In [26]: α = β = 0.1
N = 10000
p = β / (α + β)

P = ((1 - α, α),
      (β, 1 - β))           # Careful: P and p are distinct
P = np.array(P)
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)    # Horizontal 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=f'$X_0 = \{x0\}$')
    # Overlay in black--make lines clearer
    ax.plot(X_bar - p, 'k-', alpha=0.6)

ax.legend(loc='upper right')
plt.show()
```



12.11.2 Exercise 2

```
In [27]: """
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)
f = open(infile, 'r')
edges = f.readlines()
f.close()
for edge in edges:
    from_node, to_node = re.findall('\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:.4f}')
```

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
```

12.11.3 Exercise 3

A solution from the [QuantEcon.py](#) library can be found [here](#).

Footnotes

[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.

Chapter 13

Inventory Dynamics

13.1 Contents

- Overview 13.2
- Sample Paths 13.3
- Marginal Distributions 13.4
- Exercises 13.5
- Solutions 13.6

13.2 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 [19].

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

```
In [1]: import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

from numba import njit, float64, prange
from numba.experimental import jitclass
```

13.3 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.

```
In [2]: 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:

```
In [3]: firm = Firm()

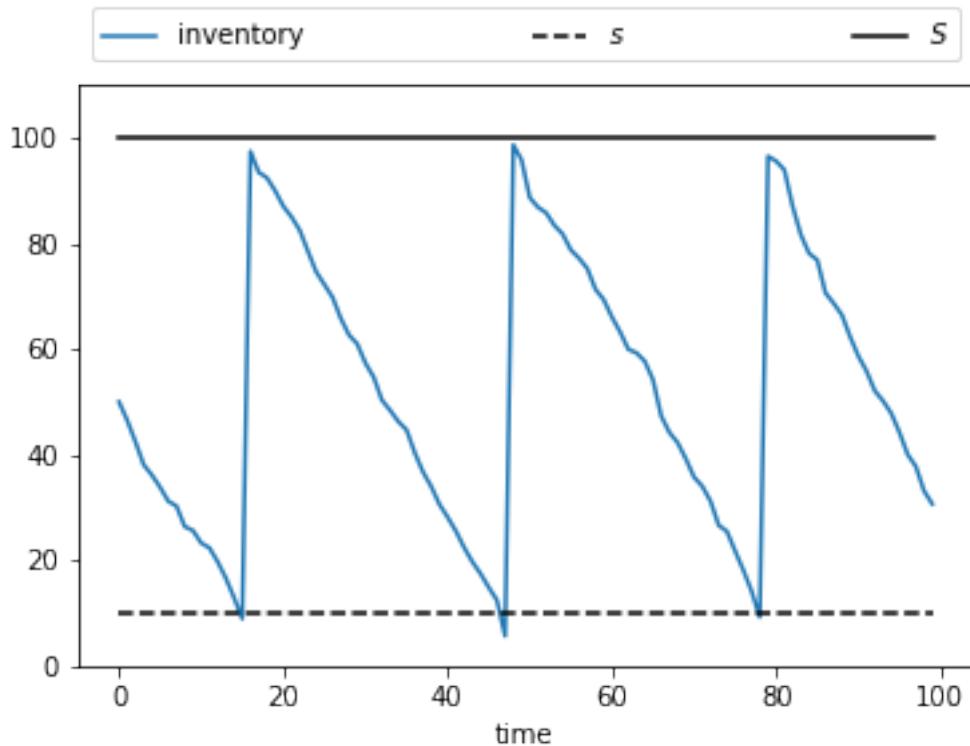
s, S = firm.s, firm.S
sim_length = 100
x_init = 50

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(s * np.ones(sim_length), 'k--', label="$s$")
ax.plot(S * np.ones(sim_length), 'k-', label="$S$")
ax.set_xlim(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:

```
In [4]: sim_length=200
fig, ax = plt.subplots()
```

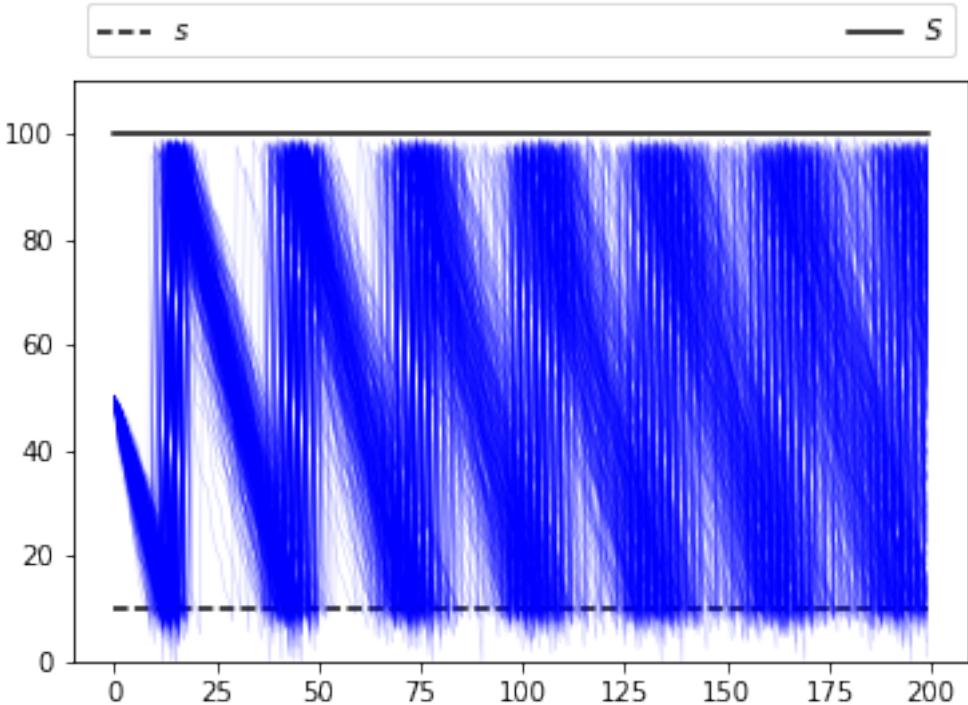
```

ax.plot(s * np.ones(sim_length), 'k--', label="$s$")
ax.plot(S * np.ones(sim_length), '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()

```



13.4 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 $T = 50$.

```

In [5]: 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('Xt', fontsize=16)
ax.vlines((T,), -1.5, 1.5)

ax.set_xticks((T,))
ax.set_xticklabels((r'$T$',))

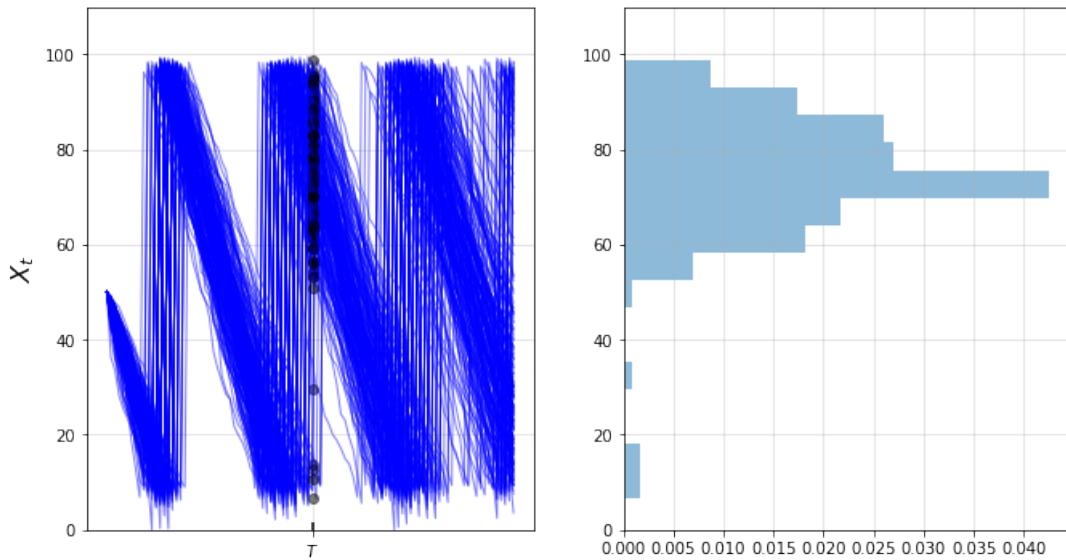
sample = np.empty(M)
for m in range(M):
    X = firm.sim_inventory_path(x_init, 2 * T)
    ax.plot(X, 'b-', lw=1, alpha=0.5)
    ax.plot((T,), (X[T+1],), 'ko', alpha=0.5)
    sample[m] = X[T+1]

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

In [6]: $T = 50$
 $M = 50_000$

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

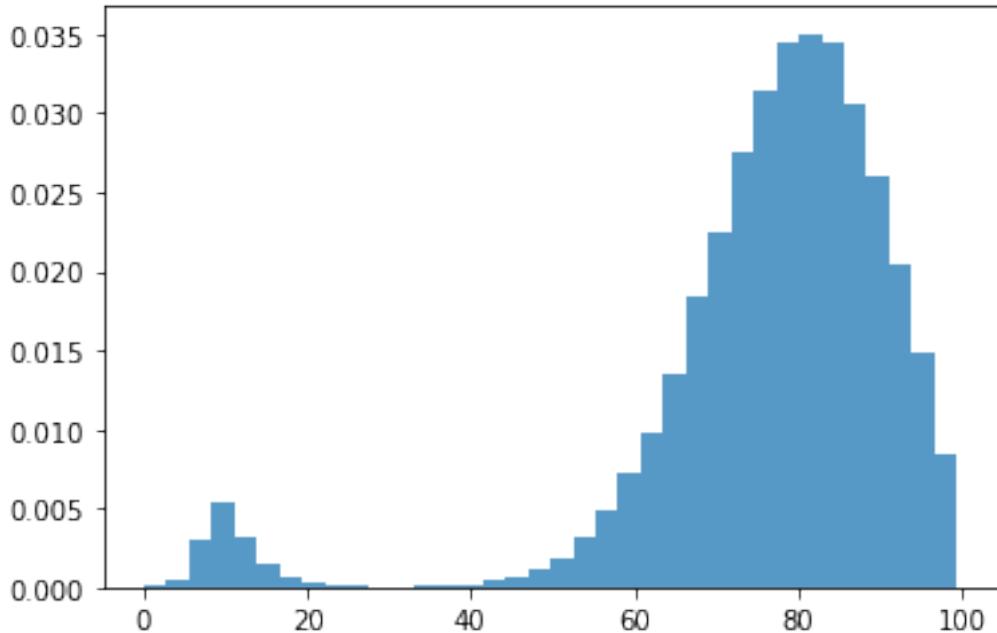
```

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](#)

```
In [7]: from sklearn.neighbors import KernelDensity

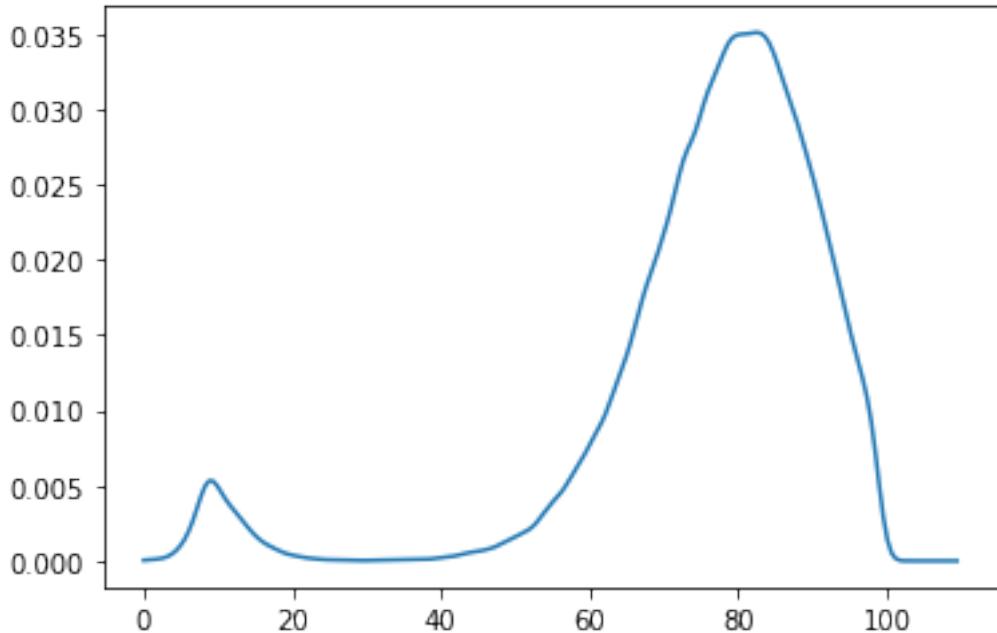
def plot_kde(sample, ax, label=''):
    xmin, xmax = 0.9 * min(sample), 1.1 * max(sample)
    xgrid = np.linspace(xmin, xmax, 200)
    kde = KernelDensity(kernel='gaussian').fit(sample[:, None])
```

```

log_dens = kde.score_samples(xgrid[:, None])
ax.plot(xgrid, np.exp(log_dens), label=label)

In [8]: 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.

13.5 Exercises

13.5.1 Exercise 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.

13.5.2 Exercise 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.

13.6 Solutions

13.6.1 Exercise 1

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.

```
In [9]: s, S, mu, sigma = firm.s, firm.S, firm.mu, firm.sigma
```

```
@njit(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
```

```
In [10]: 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.ones(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}')

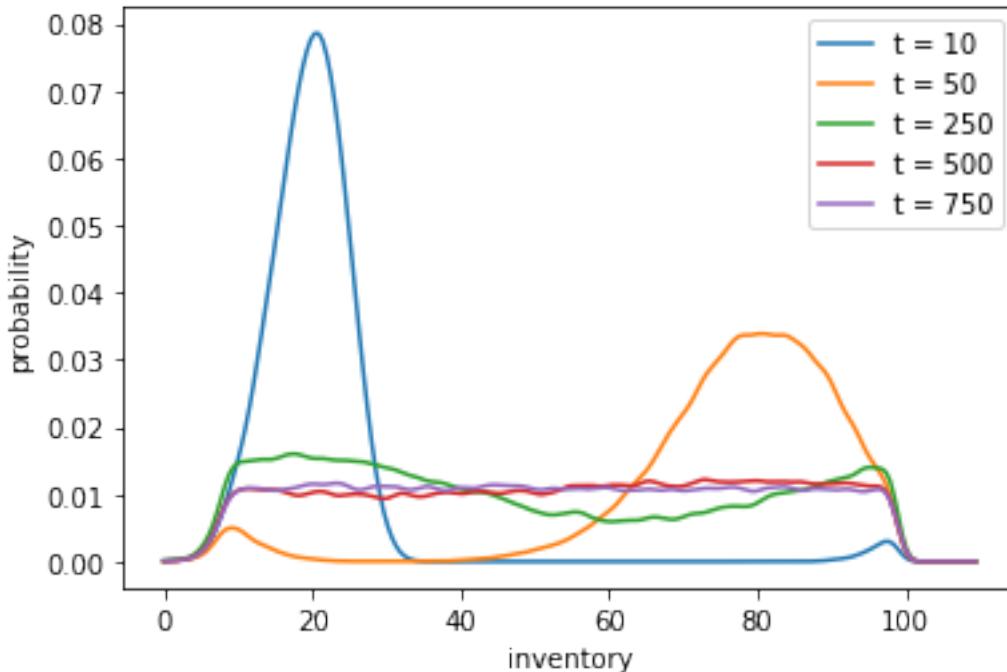
ax.set_xlabel('inventory')
```

```

ax.set_ylabel('probability')
ax.legend()
plt.show()

/home/ubuntu/anaconda3/lib/python3.7/site-packages/numba/np/ufunc/parallel.py:
 355:
NumbaWarning: The TBB threading layer requires TBB version 2019.5 or later i.e.,
TBB_INTERFACE_VERSION >= 11005. Found TBB_INTERFACE_VERSION = 11004. The TBB
threading
layer is disabled.
warnings.warn(problem)

```



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 will all firms starting at $X_0 = 20$ or $X_0 = 80$.

13.6.2 Exercise 2

Here is one solution.

Again, the computations are relatively intensive so we have written a specialized function rather than using the class above.

We will also use parallelization across firms.

```
In [11]: @njit(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.

In [12]: %time

```

freq = compute_freq()
print(f"Frequency of at least two stock outs = {freq}")

```

```

Frequency of at least two stock outs = 0.446537
CPU times: user 4.8 s, sys: 4.13 ms, total: 4.8 s
Wall time: 2.75 s

```

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 14

Linear State Space Models

14.1 Contents

- Overview 14.2
- The Linear State Space Model 14.3
- Distributions and Moments 14.4
- Stationarity and Ergodicity 14.5
- Noisy Observations 14.6
- Prediction 14.7
- Code 14.8
- Exercises 14.9
- Solutions 14.10

“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:

```
In [1]: !pip install quantecon
```

14.2 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:

```
In [2]: import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from quantecon import LinearStateSpace
from scipy.stats import norm
import random

/home/ubuntu/anaconda3/lib/python3.7/site-packages/numba/np/ufunc/parallel.py:
  ↵355:
NumbaWarning: The TBB threading layer requires TBB version 2019.5 or later i.e.,
TBB_INTERFACE_VERSION >= 11005. Found TBB_INTERFACE_VERSION = 11004. The TBB
  ↵threading
layer is disabled.
warnings.warn(problem)
```

14.3 The Linear State Space Model

The objects in play are:

- An $n \times 1$ vector x_t denoting the **state** at time $t = 0, 1, 2, \dots$
- 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 $t = 0, 1, 2, \dots$
- 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

$$\begin{aligned} x_{t+1} &= Ax_t + Cw_{t+1} \\ y_t &= Gx_t \\ x_0 &\sim N(\mu_0, \Sigma_0) \end{aligned} \tag{1}$$

14.3.1 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, \dots , the model (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

$$\mathbb{E}[w_{t+1}|x_t, x_{t-1}, \dots] = 0$$

This is a weaker condition than that $\{w_t\}$ is IID with $w_{t+1} \sim N(0, I)$.

14.3.2 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

$$y_{t+1} = \phi_0 + \phi_1 y_t + \phi_2 y_{t-1} \quad \text{s.t. } y_0, y_{-1} \text{ given} \quad (2)$$

To map (2) into our state space system (1), we set

$$x_t = \begin{bmatrix} 1 \\ y_t \\ y_{t-1} \end{bmatrix} \quad A = \begin{bmatrix} 1 & 0 & 0 \\ \phi_0 & \phi_1 & \phi_2 \\ 0 & 1 & 0 \end{bmatrix} \quad C = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \quad G = [0 \ 1 \ 0]$$

You can confirm that under these definitions, (1) and (2) agree.

The next figure shows the dynamics of this process when $\phi_0 = 1.1, \phi_1 = 0.8, \phi_2 = -0.8, y_0 = y_{-1} = 1$.

In [3]: `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()

```

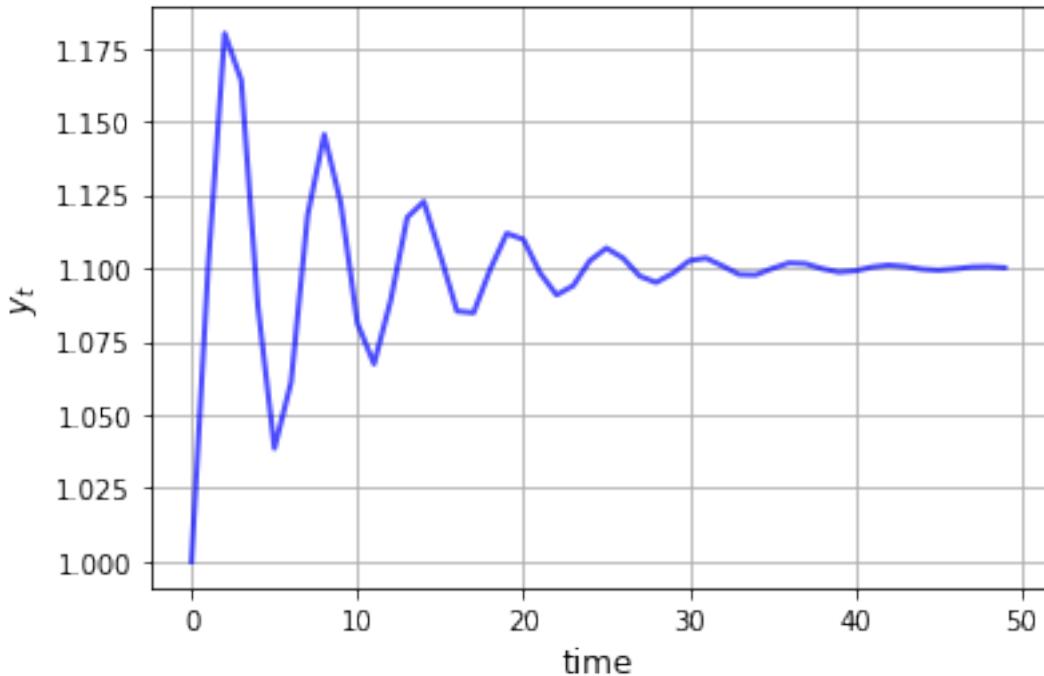
In [4]: $\phi_0, \phi_1, \phi_2 = 1.1, 0.8, -0.8$

```

A = [[1,      0,      0],
      [\phi_0,  \phi_1,  \phi_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 (1) to represent the model

$$y_{t+1} = \phi_1 y_t + \phi_2 y_{t-1} + \phi_3 y_{t-2} + \phi_4 y_{t-3} + \sigma w_{t+1} \quad (3)$$

where $\{w_t\}$ is IID and standard normal.

To put this in the linear state space format we take $x_t = [y_t \ y_{t-1} \ y_{t-2} \ y_{t-3}]'$ and

$$A = \begin{bmatrix} \phi_1 & \phi_2 & \phi_3 & \phi_4 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad C = \begin{bmatrix} \sigma \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad G = [1 \ 0 \ 0 \ 0]$$

The matrix A has the form of the *companion matrix* to the vector $[\phi_1 \ \phi_2 \ \phi_3 \ \phi_4]$.

The next figure shows the dynamics of this process when

$$\phi_1 = 0.5, \phi_2 = -0.2, \phi_3 = 0, \phi_4 = 0.5, \sigma = 0.2, y_0 = y_{-1} = y_{-2} = y_{-3} = 1$$

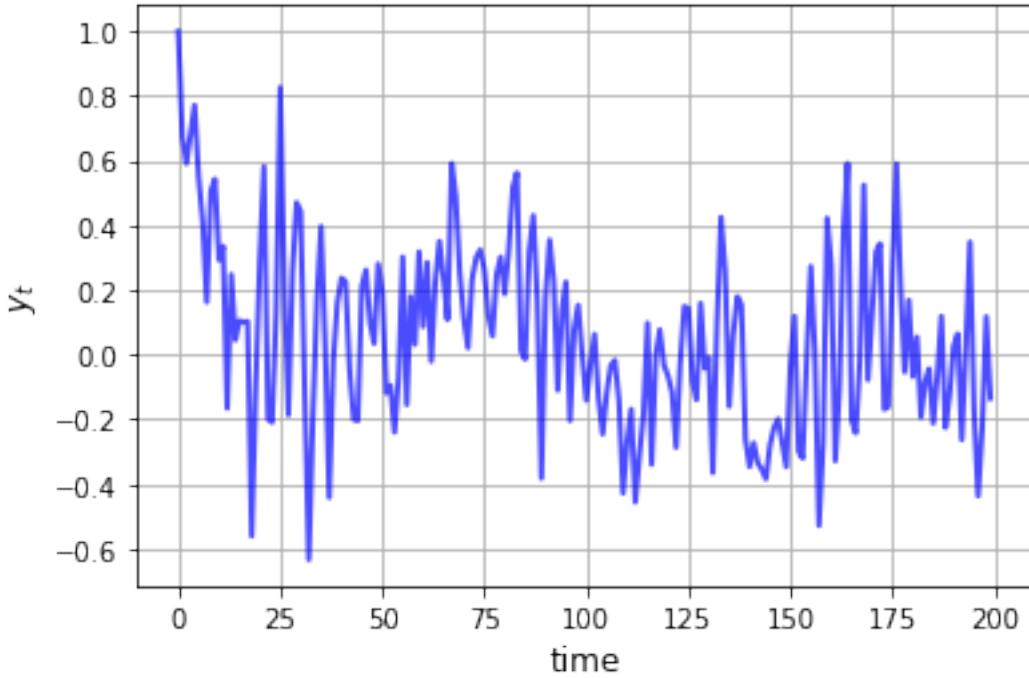
In [5]: $\phi_1, \phi_2, \phi_3, \phi_4 = 0.5, -0.2, 0, 0.5$
 $\sigma = 0.2$

```
A_1 = [[phi_1,     phi_2,     phi_3,     phi_4],
       [1,         0,         0,         0],
       [0,         1,         0,         0],
       [0,         0,         1,         0]]
```

```
C_1 = [[sigma],
       [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 (3) is termed a *vector autoregression*.

To map this into (1), we set

$$x_t = \begin{bmatrix} y_t \\ y_{t-1} \\ y_{t-2} \\ y_{t-3} \end{bmatrix} \quad A = \begin{bmatrix} \phi_1 & \phi_2 & \phi_3 & \phi_4 \\ I & 0 & 0 & 0 \\ 0 & I & 0 & 0 \\ 0 & 0 & I & 0 \end{bmatrix} \quad C = \begin{bmatrix} \sigma \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad G = [I \ 0 \ 0 \ 0]$$

where I is the $k \times k$ identity matrix and σ is a $k \times k$ matrix.

Seasonals

We can use (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 (3).

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:Section ??

$$x_{t+4} = x_t$$

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

$$A = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \quad C = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad G = [a \ b] \quad (4)$$

and starting at initial condition $x_0 = [0 \ 1]'$.

In fact, it's possible to use the state-space system to represent polynomial trends of any order.

For instance, we can represent the model $y_t = at^2 + bt + c$ in the linear state space form by taking

$$A = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \quad C = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \quad G = [2a \ a+b \ c]$$

and starting at initial condition $x_0 = [0 \ 0 \ 1]'$.

It follows that

$$A^t = \begin{bmatrix} 1 & t & t(t-1)/2 \\ 0 & 1 & t \\ 0 & 0 & 1 \end{bmatrix}$$

Then $x'_t = [t(t-1)/2 \ t \ 1]$. You can now confirm that $y_t = Gx_t$ has the correct form.

14.3.3 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 (1) repeatedly to obtain

$$\begin{aligned}
x_t &= Ax_{t-1} + Cw_t \\
&= A^2x_{t-2} + ACw_{t-1} + Cw_t \\
&\quad \vdots \\
&= \sum_{j=0}^{t-1} A^j C w_{t-j} + A^t x_0
\end{aligned} \tag{5}$$

Representation (5) 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

$$A = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \quad C = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

You will be able to show that $A^t = \begin{bmatrix} 1 & t \\ 0 & 1 \end{bmatrix}$ and $A^j C = [1 \ 0]'$.

Substituting into the moving average representation (5), we obtain

$$x_{1t} = \sum_{j=0}^{t-1} w_{t-j} + [1 \ t] x_0$$

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*.

14.4 Distributions and Moments

14.4.1 Unconditional Moments

Using (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

$$\mu_{t+1} = A\mu_t \quad \text{with} \quad \mu_0 \text{ given} \tag{6}$$

Here μ_0 is a primitive given in (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

$$\Sigma_{t+1} = A\Sigma_t A' + CC' \quad \text{with } \Sigma_0 \text{ given} \quad (7)$$

As with μ_0 , the matrix Σ_0 is a primitive given in (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 Observations

Using linearity of expectations again we have

$$\mathbb{E}[y_t] = \mathbb{E}[Gx_t] = G\mu_t \quad (8)$$

The variance-covariance matrix of y_t is easily shown to be

$$\text{Var}[y_t] = \text{Var}[Gx_t] = G\Sigma_t G' \quad (9)$$

14.4.2 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 **sufficient statistics** for the population distribution.

(Sufficient statistics form a list of objects that characterize a 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

$$u \sim N(\bar{u}, S) \quad \text{and} \quad v = a + Bu \implies v \sim N(a + B\bar{u}, BSB') \quad (10)$$

In particular, given our Gaussian assumptions on the primitives and the linearity of (1) we can see immediately that both x_t and y_t are Gaussian for all $t \geq 0$ Section ??.

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 (6) and (7).

Letting μ_t and Σ_t be as defined by these equations, we have

$$x_t \sim N(\mu_t, \Sigma_t) \quad (11)$$

By similar reasoning combined with (8) and (9),

$$y_t \sim N(G\mu_t, G\Sigma_t G') \quad (12)$$

14.4.3 Ensemble Interpretations

How should we interpret the distributions defined by (11)–(12)?

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 (3).

The values of y_T are represented by black dots in the left-hand figure

```
In [6]: 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_xlim(0, T)
        ax.set_ylim(ymin, ymax)

        ax = axes[0]
        ax.set_ylabel('$y_t$', fontsize=12)
        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_ymin(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()

```

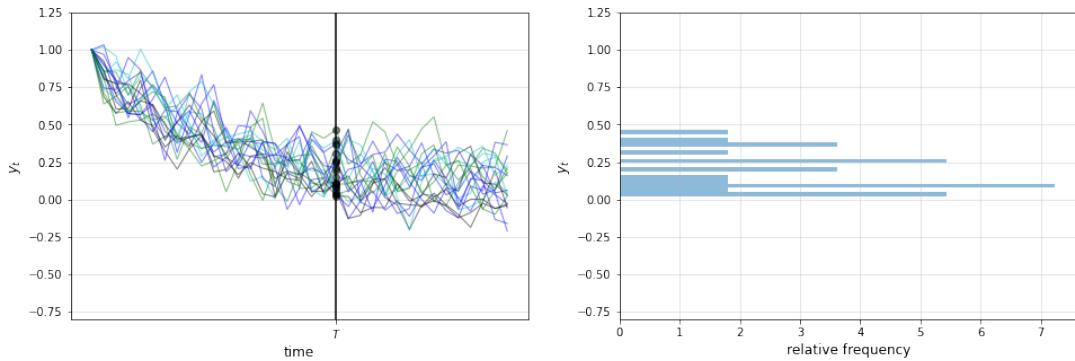
In [7]: $\phi_1, \phi_2, \phi_3, \phi_4 = 0.5, -0.2, 0, 0.5$
 $\sigma = 0.1$

```
A_2 = [[phi_1, phi_2, phi_3, phi_4],
       [1, 0, 0, 0],
       [0, 1, 0, 0],
       [0, 0, 1, 0]]
```

```
C_2 = [[sigma], [0], [0], [0]]
```

```
G_2 = [1, 0, 0, 0]
```

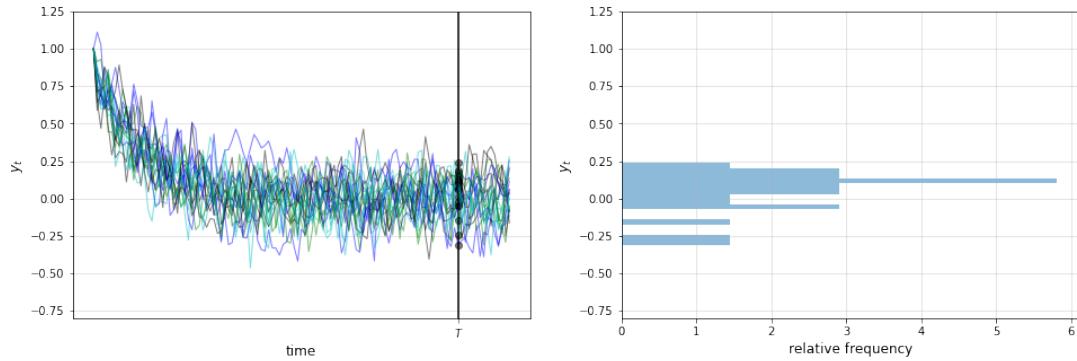
```
cross_section_plot(A_2, C_2, G_2)
```



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

In [8]: $t = 100$
 $\text{cross_section_plot}(A_2, C_2, G_2, T=t)$

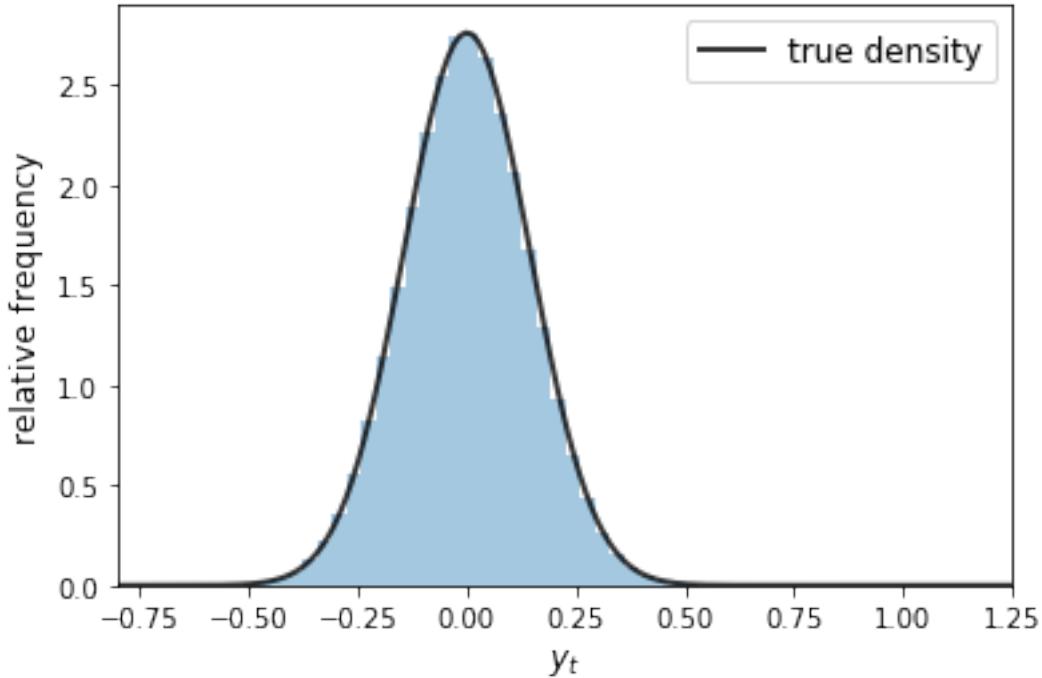


Let's now try with 500,000 observations, showing only the histogram (without rotation)

```
In [9]: 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 = ar.stationary_distributions()
f_y = norm(loc=float(mu_y), scale=float(np.sqrt(Sigma_y)))
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 (12).

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 sufficient statistics.

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_T^i
3. histogramming this sample

Just as the histogram approximates the population distribution, the *ensemble* or *cross-sectional average*

$$\bar{y}_T := \frac{1}{I} \sum_{i=1}^I y_T^i$$

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 $t = 0, \dots, 50$.

The parameters are the same as for the preceding figures, and the sample size is relatively small ($I = 20$).

```
In [10]: I = 20
T = 50
ymin = -0.5
ymax = 1.15

ar = LinearStateSpace(A_2, C_2, G_2, mu_0=np.ones(4))

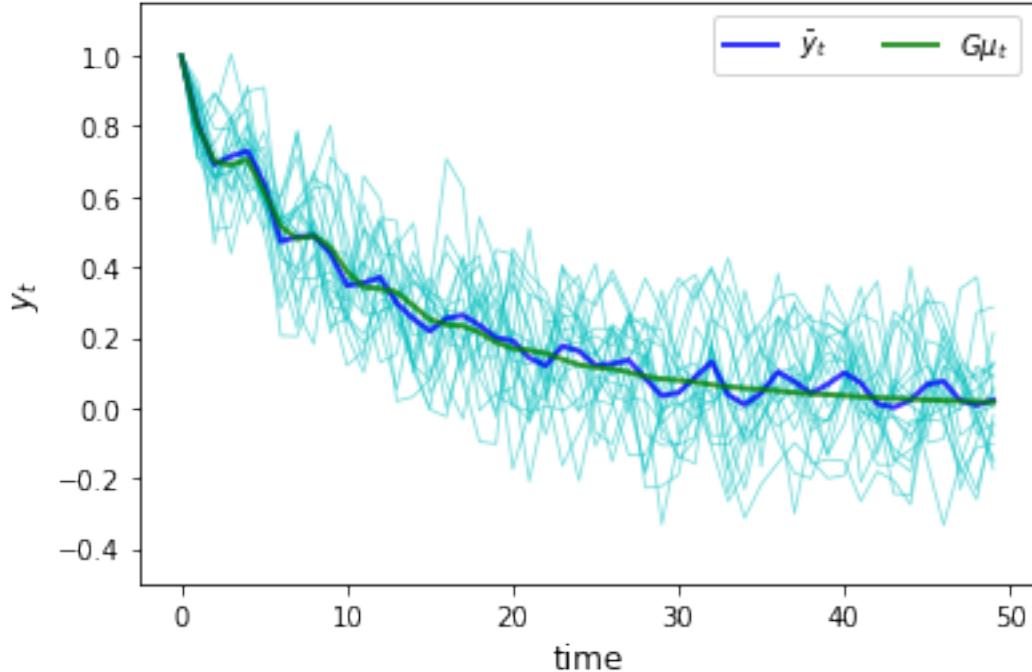
fig, ax = plt.subplots()

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):
    mu_x, mu_y, Sigma_x, Sigma_y = next(m)
    population_means.append(float(mu_y))

ax.plot(population_means, color='g', lw=2, alpha=0.8, label='$G\mu_t$')
ax.set_xlim(0, T)
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

$$\bar{x}_T := \frac{1}{I} \sum_{i=1}^I x_T^i \rightarrow \mu_T \quad (I \rightarrow \infty)$$

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

$$\frac{1}{I} \sum_{i=1}^I (x_T^i - \bar{x}_T)(x_T^i - \bar{x}_T)' \rightarrow \Sigma_T \quad (I \rightarrow \infty)$$

14.4.4 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, \dots, x_T , recall that joint and conditional densities are linked by the rule

$$p(x, y) = p(y | x)p(x) \quad (\text{joint} = \text{conditional} \times \text{marginal})$$

From this rule we get $p(x_0, x_1) = p(x_1 | x_0)p(x_0)$.

The Markov property $p(x_t | x_{t-1}, \dots, x_0) = p(x_t | x_{t-1})$ and repeated applications of the preceding rule lead us to

$$p(x_0, x_1, \dots, x_T) = p(x_0) \prod_{t=0}^{T-1} p(x_{t+1} | x_t)$$

The marginal $p(x_0)$ is just the primitive $N(\mu_0, \Sigma_0)$.

In view of (1), the conditional densities are

$$p(x_{t+1} | x_t) = N(Ax_t, CC')$$

Autocovariance Functions

An important object related to the joint distribution is the *autocovariance function*

$$\Sigma_{t+j,t} := \mathbb{E}[(x_{t+j} - \mu_{t+j})(x_t - \mu_t)'] \tag{13}$$

Elementary calculations show that

$$\Sigma_{t+j,t} = A^j \Sigma_t \quad (14)$$

Notice that $\Sigma_{t+j,t}$ in general depends on both j , the gap between the two dates, and t , the earlier date.

14.5 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.

14.5.1 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''

```
In [11]: 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':
        mu_x, mu_y, Sigma_x, Sigma_y = ar.stationary_distributions()
        ar_state = LinearStateSpace(A, C, G, mu_0=mu_x, Sigma_0=Sigma_x)

    ymin, ymax = -0.6, 0.6
    fig, ax = plt.subplots()
    ax.grid(alpha=0.4)
    ax.set_xlim(ymin, ymax)
    ax.set_ylabel('$y_t$', fontsize=12)
    ax.set_xlabel('time$', fontsize=12)

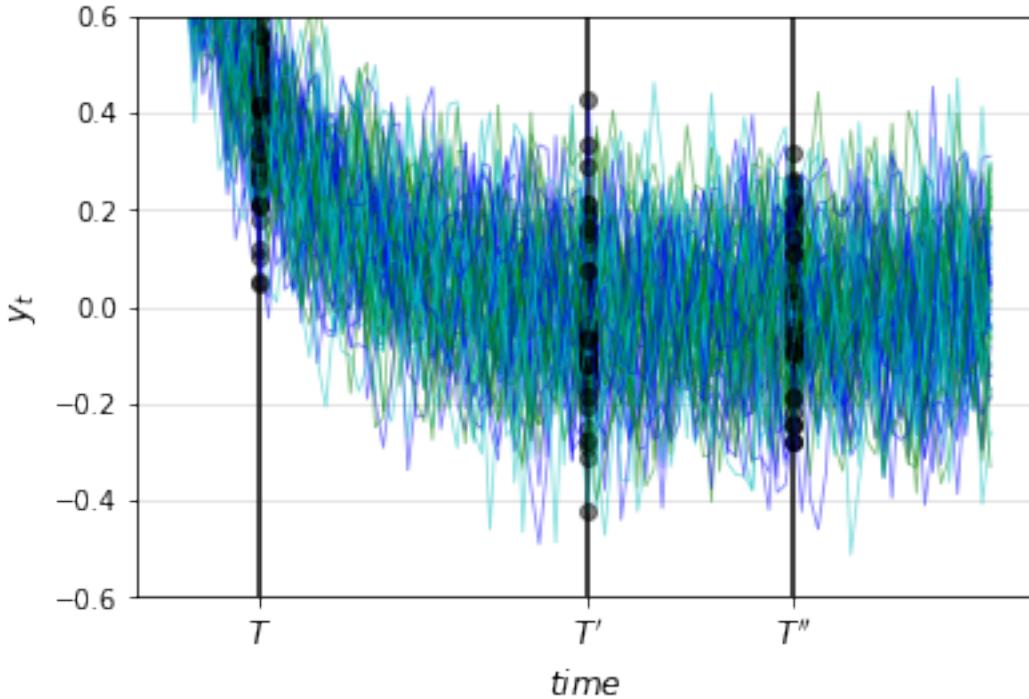
    ax.vlines((T0, T1, T2), -1.5, 1.5)
    ax.set_xticks((T0, T1, T2))
    ax.set_xticklabels(("T$", "T'$", "T''$"), fontsize=12)
    for i in range(80):
        rcolor = random.choice(('c', 'g', 'b'))

        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()
```

In [12]: `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 $t \rightarrow \infty$.

When such a distribution exists it is called a *stationary distribution*.

14.5.2 Stationary Distributions

In our setting, a distribution ψ_∞ is said to be *stationary* for x_t if

$$x_t \sim \psi_\infty \quad \text{and} \quad x_{t+1} = Ax_t + Cw_{t+1} \quad \Rightarrow \quad x_{t+1} \sim \psi_\infty$$

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

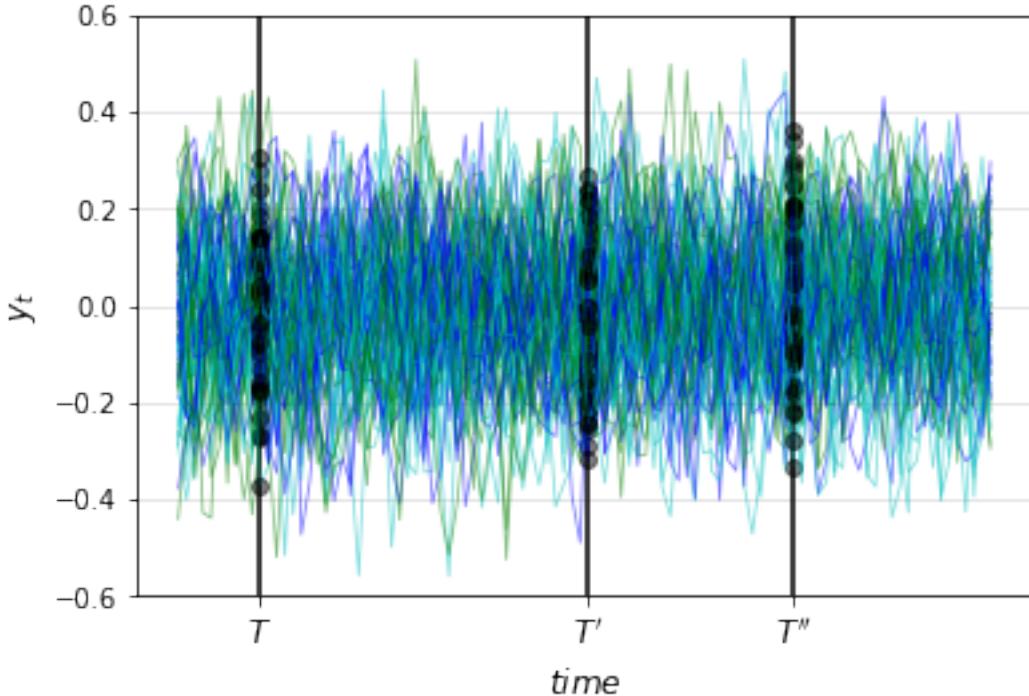
$$\psi_\infty = N(\mu_\infty, \Sigma_\infty)$$

where μ_∞ and Σ_∞ are fixed points of (6) and (7) respectively.

14.5.3 Covariance Stationary Processes

Let's see what happens to the preceding figure if we start x_0 at the stationary distribution.

```
In [13]: 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 (6) and (7) 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 (14), 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 .

14.5.4 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 (7) also has a unique fixed point in this case, and, moreover

$$\mu_t \rightarrow \mu_\infty = 0 \quad \text{and} \quad \Sigma_t \rightarrow \Sigma_\infty \quad \text{as} \quad t \rightarrow \infty$$

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 $x_t = [1 \ y_t \ y_{t-1}]'$.

Because of the constant first component in the state vector, we will never have $\mu_t \rightarrow 0$.

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

$$A = \begin{bmatrix} A_1 & a \\ 0 & 1 \end{bmatrix} \quad C = \begin{bmatrix} C_1 \\ 0 \end{bmatrix}$$

where

- A_1 is an $(n - 1) \times (n - 1)$ matrix
- a is an $(n - 1) \times 1$ column vector

Let $x_t = [x'_{1t} \ 1]'$ 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

$$\mu_{1,t+1} = A_1 \mu_{1,t} + a \tag{15}$$

Assume now that the moduli of the eigenvalues of A_1 are all strictly less than one.

Then (15) has a unique stationary solution, namely,

$$\mu_{1\infty} = (I - A_1)^{-1}a$$

The stationary value of μ_t itself is then $\mu_\infty := [\mu'_{1\infty} \quad 1]'$.

The stationary values of Σ_t and $\Sigma_{t+j,t}$ satisfy

$$\begin{aligned}\Sigma_\infty &= A\Sigma_\infty A' + CC' \\ \Sigma_{t+j,t} &= A^j \Sigma_\infty\end{aligned}\tag{16}$$

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 (7) converge to the fixed point of the *discrete Lyapunov equation* in the first line of (16).

14.5.5 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,

- $\frac{1}{T} \sum_{t=1}^T x_t \rightarrow \mu_\infty$
- $\frac{1}{T} \sum_{t=1}^T (x_t - \bar{x}_T)(x_t - \bar{x}_T)' \rightarrow \Sigma_\infty$
- $\frac{1}{T} \sum_{t=1}^T (x_{t+j} - \bar{x}_T)(x_t - \bar{x}_T)' \rightarrow A^j \Sigma_\infty$

In our linear Gaussian setting, any covariance stationary process is also ergodic.

14.6 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

$$\begin{aligned} x_{t+1} &= Ax_t + Cw_{t+1} \\ y_t &= Gx_t + Hv_t \\ x_0 &\sim N(\mu_0, \Sigma_0) \end{aligned} \tag{17}$$

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 (8) and (9) now become

$$\mathbb{E}[y_t] = \mathbb{E}[Gx_t + Hv_t] = G\mu_t \tag{18}$$

The variance-covariance matrix of y_t is easily shown to be

$$\text{Var}[y_t] = \text{Var}[Gx_t + Hv_t] = G\Sigma_t G' + HH' \tag{19}$$

The distribution of y_t is therefore

$$y_t \sim N(G\mu_t, G\Sigma_t G' + HH')$$

14.7 Prediction

The theory of prediction for linear state space systems is elegant and simple.

14.7.1 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

$$\mathbb{E}_t[x_{t+1}] := \mathbb{E}[x_{t+1} | x_t, x_{t-1}, \dots, x_0] = Ax_t$$

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 $\mathbb{E}_t[x_{t+1}] = \mathbb{E}[x_{t+1} | x_t]$ 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

$$x_{t+j} = A^j x_t + A^{j-1} C w_{t+1} + A^{j-2} C w_{t+2} + \cdots + A^0 C w_{t+j}$$

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

$$\mathbb{E}_t[y_{t+j}] = \mathbb{E}_t[Gx_{t+j} + Hv_{t+j}] = GA^j x_t$$

14.7.2 Covariance of Prediction Errors

It is useful to obtain the covariance matrix of the vector of j -step-ahead prediction errors

$$x_{t+j} - \mathbb{E}_t[x_{t+j}] = \sum_{s=0}^{j-1} A^s C w_{t-s+j} \quad (20)$$

Evidently,

$$V_j := \mathbb{E}_t[(x_{t+j} - \mathbb{E}_t[x_{t+j}])(x_{t+j} - \mathbb{E}_t[x_{t+j}])'] = \sum_{k=0}^{j-1} A^k C C' A^k \quad (21)$$

V_j defined in (21) can be calculated recursively via $V_1 = CC'$ and

$$V_j = CC' + AV_{j-1}A', \quad j \geq 2 \quad (22)$$

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

$$V_\infty = CC' + AV_\infty A' \quad (23)$$

Equation (23) 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.

14.8 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.

14.9 Exercises

14.9.1 Exercise 1

In several contexts, we want to compute forecasts of geometric sums of future random variables governed by the linear state-space system (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?

14.10 Solutions

14.10.1 Exercise 1

Suppose that every eigenvalue of A has modulus strictly less than $\frac{1}{\beta}$.

It then follows that $I + \beta A + \beta^2 A^2 + \dots = [I - \beta A]^{-1}$.

This leads to our formulas:

- Forecast of a geometric sum of future x 's

$$\mathbb{E}_t \left[\sum_{j=0}^{\infty} \beta^j x_{t+j} \right] = [I + \beta A + \beta^2 A^2 + \dots] x_t = [I - \beta A]^{-1} x_t$$

- Forecast of a geometric sum of future y 's

$$\mathbb{E}_t \left[\sum_{j=0}^{\infty} \beta^j y_{t+j} \right] = G[I + \beta A + \beta^2 A^2 + \dots] x_t = G[I - \beta A]^{-1} x_t$$

Footnotes

[1] The eigenvalues of A are $(1, -1, i, -i)$.

[2] The correct way to argue this is by induction. Suppose that x_t is Gaussian. Then (1) and (10) 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.

Chapter 15

Application: The Samuelson Multiplier-Accelerator

15.1 Contents

- Overview 15.2
- Details 15.3
- Implementation 15.4
- Stochastic Shocks 15.5
- Government Spending 15.6
- Wrapping Everything Into a Class 15.7
- Using the LinearStateSpace Class 15.8
- Pure Multiplier Model 15.9
- Summary 15.10

In addition to what's in Anaconda, this lecture will need the following libraries:

In [1]: `!pip install quantecon`

15.2 Overview

This lecture creates non-stochastic and stochastic versions of Paul Samuelson's celebrated multiplier accelerator model [93].

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:

In [2]: `import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline`

We'll also use the following for various tasks described below:

```
In [3]: from quantecon import LinearStateSpace
import cmath
import math
import sympy
from sympy import Symbol, init_printing
from cmath import sqrt

/home/ubuntu/anaconda3/lib/python3.7/site-packages/numba/np/ufunc/parallel.py:
  ↪355:
NumbaWarning: The TBB threading layer requires TBB version 2019.5 or later i.e.,
TBB_INTERFACE_VERSION >= 11005. Found TBB_INTERFACE_VERSION = 11004. The TBB
  ↪threading
layer is disabled.
warnings.warn(problem)
```

15.2.1 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 [95].)

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 [95].)

15.3 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.
- a is the marginal propensity to consume in the Keynesian consumption function $C_t = aY_{t-1} + \gamma$.
- b is the “accelerator coefficient” in the “investment accelerator” $I_t = b(Y_{t-1} - Y_{t-2})$.
- $\{\epsilon_t\}$ 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

$$C_t = aY_{t-1} + \gamma \quad (1)$$

with the investment accelerator

$$I_t = b(Y_{t-1} - Y_{t-2}) \quad (2)$$

and the national income identity

$$Y_t = C_t + I_t + G_t \quad (3)$$

- The parameter a is peoples' *marginal propensity to consume* out of income - equation (1) asserts that people consume a fraction of $a \in (0, 1)$ of each additional dollar of income.
- The parameter $b > 0$ is the investment accelerator coefficient - equation (2) asserts that people invest in physical capital when income is increasing and disinvest when it is decreasing.

Equations (1), (2), and (3) imply the following second-order linear difference equation for national income:

$$Y_t = (a + b)Y_{t-1} - bY_{t-2} + (\gamma + G_t)$$

or

$$Y_t = \rho_1 Y_{t-1} + \rho_2 Y_{t-2} + (\gamma + G_t) \quad (4)$$

where $\rho_1 = (a + b)$ and $\rho_2 = -b$.

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 (a, b) 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.

15.3.1 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 (4), leading to the **second-order scalar linear stochastic difference equation**:

$$Y_t = G_t + a(1-b)Y_{t-1} - abY_{t-2} + \sigma\epsilon_t \quad (5)$$

15.3.2 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 (5) as

$$Y_t = \rho_1 Y_{t-1} + \rho_2 Y_{t-2}$$

or

$$Y_{t+2} - \rho_1 Y_{t+1} - \rho_2 Y_t = 0 \quad (6)$$

To discover the properties of the solution of (6), it is useful first to form the **characteristic polynomial** for (6):

$$z^2 - \rho_1 z - \rho_2 \quad (7)$$

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 (7), the characteristic polynomial (7) equals zero:

$$z^2 - \rho_1 z - \rho_2 = (z - \lambda_1)(z - \lambda_2) = 0 \quad (8)$$

Equation (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

$$\lambda_1 = re^{i\omega}, \quad \lambda_2 = re^{-i\omega}$$

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 (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

$$\begin{aligned} Y_t &= c_1(re^{i\omega})^t + c_2(re^{-i\omega})^t \\ &= c_1 r^t e^{i\omega t} + c_2 r^t e^{-i\omega t} \\ &= c_1 r^t [\cos(\omega t) + i \sin(\omega t)] + c_2 r^t [\cos(\omega t) - i \sin(\omega t)] \\ &= (c_1 + c_2)r^t \cos(\omega t) + i(c_1 - c_2)r^t \sin(\omega t) \end{aligned}$$

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

$$c_1 = ve^{i\theta}, \quad c_2 = ve^{-i\theta}$$

So we can write

$$\begin{aligned} Y_t &= ve^{i\theta}r^t e^{i\omega t} + ve^{-i\theta}r^t e^{-i\omega t} \\ &= vr^t [e^{i(\omega t+\theta)} + e^{-i(\omega t+\theta)}] \\ &= 2vr^t \cos(\omega t + \theta) \end{aligned}$$

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 [93], we want to choose the parameters a, b 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:

$$|\lambda_j| < 1 \quad \text{for } j = 1, 2$$

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.

15.3.3 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 a, b 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 (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 :

$$A = \begin{bmatrix} 1 & 0 & 0 \\ \gamma + G & \rho_1 & \rho_2 \\ 0 & 1 & 0 \end{bmatrix}$$

15.4 Implementation

We'll start by drawing an informative graph from page 189 of [95]

In [4]: `def param_plot():`

```
"""This function creates the graph on page 189 of
Sargent Macroeconomic Theory, second edition, 1987.
"""

fig, ax = plt.subplots(figsize=(10, 6))
ax.set_aspect('equal')

# Set axis
xmin, ymin = -3, -2
xmax, ymax = -xmin, -ymin
plt.axis([xmin, xmax, ymin, ymax])

# Set axis labels
ax.set(xticks=[], yticks[])
ax.set_xlabel(r'$\rho_2$', fontsize=16)
ax.xaxis.set_label_position('top')
ax.set_ylabel(r'$\rho_1$', rotation=0, fontsize=16)
ax.yaxis.set_label_position('right')

# Draw (t1, t2) points
ρ1 = np.linspace(-2, 2, 100)
ax.plot(ρ1, -abs(ρ1) + 1, c='black')
ax.plot(ρ1, np.ones_like(ρ1) * -1, c='black')
ax.plot(ρ1, -(ρ1**2 / 4), c='black')

# Turn normal axes off
for spine in ['left', 'bottom', 'top', 'right']:
    ax.spines[spine].set_visible(False)

# Add arrows to represent axes
axes_arrows = {'arrowstyle': '<-|>', 'lw': 1.3}
ax.annotate(' ', xy=(xmin, 0), xytext=(xmax, 0), arrowprops=axes_arrows)
ax.annotate(' ', xy=(0, ymin), xytext=(0, ymax), arrowprops=axes_arrows)

# Annotate the plot with equations
plot_arrowsl = {'arrowstyle': '-|>', 'connectionstyle': "arc3, rad=-0.
˓→2"}
plot_arrowsr = {'arrowstyle': '-|>', 'connectionstyle': "arc3, rad=0.2"}
ax.annotate(r'$\rho_1 + \rho_2 < 1$', xy=(0.5, 0.3), xytext=(0.8, 0.6),
            arrowprops=plot_arrowsr, fontsize='12')
ax.annotate(r'$\rho_1 + \rho_2 = 1$', xy=(0.38, 0.6), xytext=(0.6, 0.8),
            arrowprops=plot_arrowsr, fontsize='12')
ax.annotate(r'$\rho_2 < 1 + \rho_1$', xy=(-0.5, 0.3), xytext=(-1.3, 0.
˓→6),
            arrowprops=plot_arrowsl, fontsize='12')
```

```

ax.annotate(r'$\rho_2 = 1 + \rho_1$', xy=(-0.38, 0.6), xytext=(-1, 0.8),
            arrowprops=plot_arrowsl, fontsize='12')
ax.annotate(r'$\rho_2 = -1$', xy=(1.5, -1), xytext=(1.8, -1.3),
            arrowprops=plot_arrowsl, fontsize='12')
ax.annotate(r'${\rho_1}^2 + 4\rho_2 = 0$', xy=(1.15, -0.35),
            xytext=(1.5, -0.3), arrowprops=plot_arrowsr, fontsize='12')
ax.annotate(r'${\rho_1}^2 + 4\rho_2 < 0$', xy=(1.4, -0.7),
            xytext=(1.8, -0.6), arrowprops=plot_arrowsr, fontsize='12')

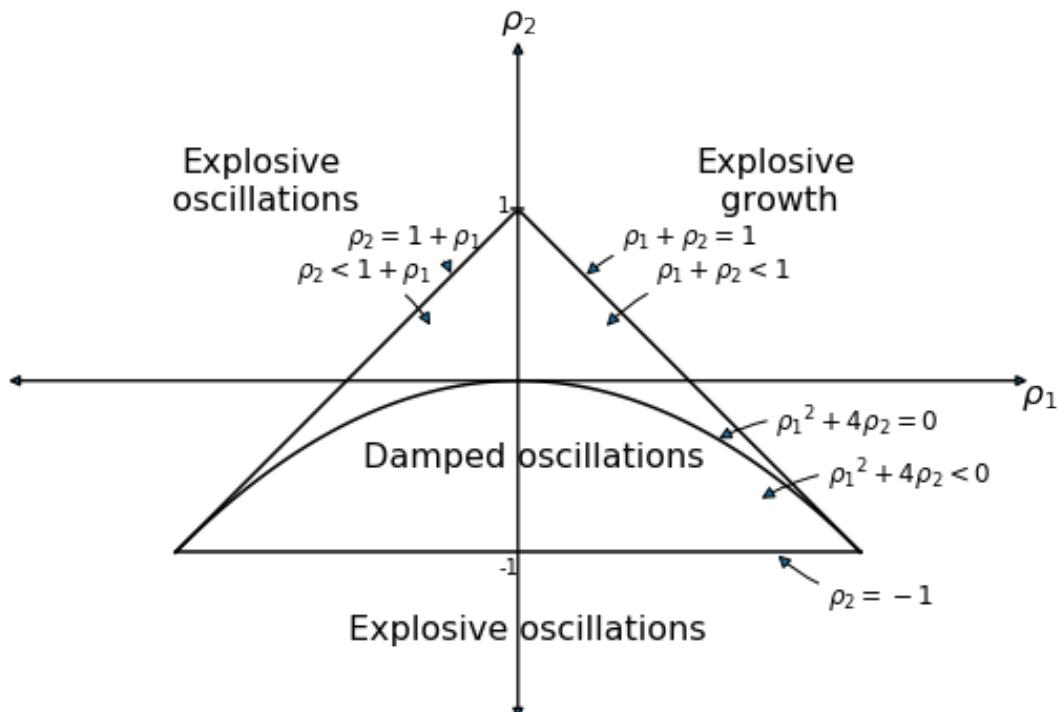
# Label categories of solutions
ax.text(1.5, 1, 'Explosive\n growth', ha='center', fontsize=16)
ax.text(-1.5, 1, 'Explosive\n oscillations', ha='center', fontsize=16)
ax.text(0.05, -1.5, 'Explosive oscillations', ha='center', fontsize=16)
ax.text(0.09, -0.5, 'Damped oscillations', ha='center', fontsize=16)

# Add small marker to y-axis
ax.axhline(y=1.005, xmin=0.495, xmax=0.505, c='black')
ax.text(-0.12, -1.12, '-1', fontsize=10)
ax.text(-0.12, 0.98, '1', fontsize=10)

return fig

```

param_plot()
plt.show()



The graph portrays regions in which the (λ_1, λ_2) root pairs implied by the $(\rho_1 = (a + b), \rho_2 = -b)$ 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 (a, b) .

15.4.1 Function to Describe Implications of Characteristic Polynomial

In [5]: `def categorize_solution(p1, p2):`

```
"""This function takes values of p1 and p2 and uses them
to classify the type of solution
"""

discriminant = p1 ** 2 + 4 * p2
if p2 > 1 + p1 or p2 < -1:
    print('Explosive oscillations')
elif p1 + p2 > 1:
    print('Explosive growth')
elif discriminant < 0:
    print('Roots are complex with modulus less than one; \
therefore damped oscillations')
else:
    print('Roots are real and absolute values are less than one; \
therefore get smooth convergence to a steady state')
```

In [6]: *## Test the categorize_solution function*

`categorize_solution(1.3, -.4)`

```
Roots are real and absolute values are less than one; therefore get smooth
convergence
to a steady state
```

15.4.2 Function for Plotting Paths

A useful function for our work below is

In [7]: `def plot_y(function=None):`

```
"""Function plots path of Y_t"""

plt.subplots(figsize=(10, 6))
plt.plot(function)
plt.xlabel('Time $t$')
plt.ylabel('$Y_t$', rotation=0)
plt.grid()
plt.show()
```

15.4.3 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)

The function also plots a Y_t starting from initial conditions that we set

In [8]: # This is a 'manual' method

```
def y_nonstochastic(y_0=100, y_1=80, α=.92, β=.5, γ=10, n=80):

    """Takes values of parameters and computes the roots of characteristic
    polynomial. It tells whether they are real or complex and whether they
    are less than unity in absolute value. It also computes a simulation of
    length n starting from the two given initial conditions for national
    income
    """

    roots = []

    ρ1 = α + β
    ρ2 = -β

    print(f'ρ_1 is {ρ1}')
    print(f'ρ_2 is {ρ2}')

    discriminant = ρ1 ** 2 + 4 * ρ2

    if discriminant == 0:
        roots.append(-ρ1 / 2)
        print('Single real root: ')
        print(''.join(str(roots)))
    elif discriminant > 0:
        roots.append((-ρ1 + sqrt(discriminant).real) / 2)
        roots.append((-ρ1 - sqrt(discriminant).real) / 2)
        print('Two real roots: ')
        print(''.join(str(roots)))
    else:
        roots.append((-ρ1 + sqrt(discriminant)) / 2)
        roots.append((-ρ1 - sqrt(discriminant)) / 2)
        print('Two complex roots: ')
        print(''.join(str(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]

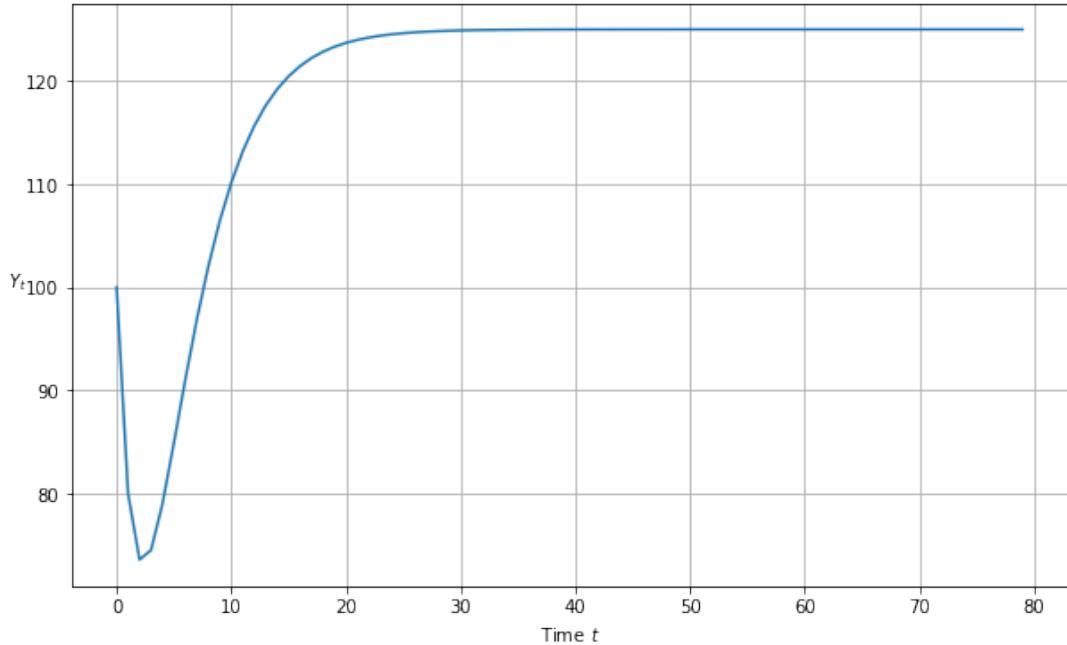
    for t in range(2, n):
        y_t.append(transition(y_t, t))

    return y_t
```

```
plot_y(y_nonstochastic())
```

```

ρ_1 is 1.42
ρ_2 is -0.5
Two real roots:
[-0.6459687576256715, -0.7740312423743284]
Absolute values of roots are less than one
```



15.4.4 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 (a, b) and (ρ_1, ρ_2) , pairs that would generate those roots

```
In [9]: ### code to reverse-engineer a cycle
### y_t = r^t (c_1 cos(ϕ t) + c2 sin(ϕ t))
###

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 a and the accelerator coefficient b that verifies those roots.

"""

```
g1 = cmath.rect(r, phi) # Generate two complex roots
g2 = cmath.rect(r, -phi)
rho1 = g1 + g2           # Implied rho1, rho2
rho2 = -g1 * g2
b = -rho2                # Reverse-engineer a and b that validate these
a = rho1 - b
return rho1, rho2, a, b
```

```
## Now let's use the function in an example
## Here are the example parameters
```

```
r = .95
period = 10               # Length of cycle in units of time
phi = 2 * math.pi/period
```

```
## Apply the function
```

```
rho1, rho2, a, b = f(r, phi)
```

```
print(f'a, b = {a}, {b}')
print(f'rho1, rho2 = {rho1}, {rho2}')
```

```
a, b = (0.6346322893124001+0j), (0.9024999999999999-0j)
rho1, rho2 = (1.5371322893124+0j), (-0.9024999999999999+0j)
```

In [10]: ## Print the real components of ρ_1 and ρ_2

```
rho1 = rho1.real
rho2 = rho2.real
```

```
rho1, rho2
```

Out[10]: (1.5371322893124, -0.9024999999999999)

15.4.5 Root Finding Using Numpy

Here we'll use numpy to compute the roots of the characteristic polynomial

In [11]: `r1, r2 = np.roots([1, -rho1, -rho2])`

```
p1 = cmath.polar(r1)
p2 = cmath.polar(r2)

print(f'r, phi = {r}, {phi}')
print(f'p1, p2 = {p1}, {p2}')
# print(f'g1, g2 = {g1}, {g2}')

print(f'a, b = {a}, {b}')
print(f'rho1, rho2 = {rho1}, {rho2}')
```

```
r, φ = 0.95, 0.6283185307179586
p1, p2 = (0.95, 0.6283185307179586), (0.95, -0.6283185307179586)
a, b = (0.6346322893124001+0j), (0.9024999999999999-0j)
ρ1, ρ2 = 1.5371322893124, -0.9024999999999999
```

In [12]: ##### This method uses numpy to calculate roots ####

```
def y_nonstochastic(y_0=100, y_1=80, α=.9, β=.8, γ=10, n=80):
    """ Rather than computing the roots of the characteristic
    polynomial by hand as we did earlier, this function
    enlists numpy to do the work for us
    """

    # Useful constants
    ρ1 = α + β
    ρ2 = -β

    categorize_solution(ρ1, ρ2)

    # Find roots of polynomial
    roots = np.roots([1, -ρ1, -ρ2])
    print(f'Roots are {roots}')

    # Check if real or complex
    if all(isinstance(root, complex) for root in roots):
        print('Roots are complex')
    else:
        print('Roots are real')

    # Check if roots are less than one
    if all(abs(root) < 1 for root in roots):
        print('Roots are less than one')
    else:
        print('Roots are not less than one')

    # Define transition equation
    def transition(x, t): return ρ1 * x[t - 1] + ρ2 * x[t - 2] + γ

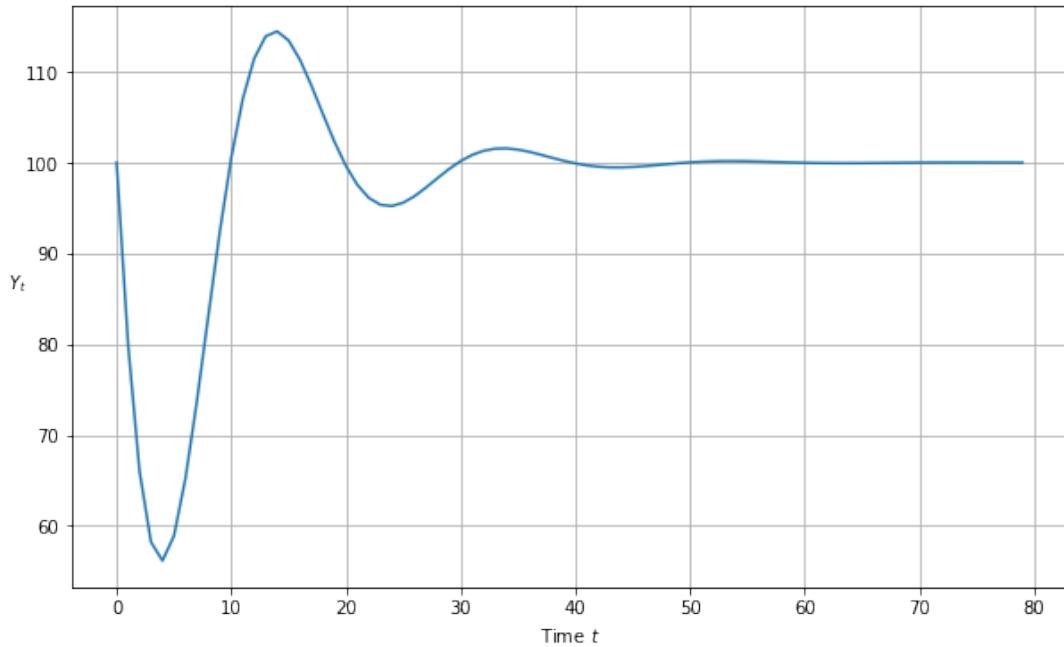
    # Set initial conditions
    y_t = [y_0, y_1]

    # Generate y_t series
    for t in range(2, n):
        y_t.append(transition(y_t, t))

    return y_t

plot_y(y_nonstochastic())
```

```
Roots are complex with modulus less than one; therefore damped oscillations
Roots are [0.85+0.27838822j 0.85-0.27838822j]
Roots are complex
Roots are less than one
```



15.4.6 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

```
In [13]: r = 1    # Generates undamped, nonexplosive cycles
period = 10    # Length of cycle in units of time
ϕ = 2 * math.pi/period

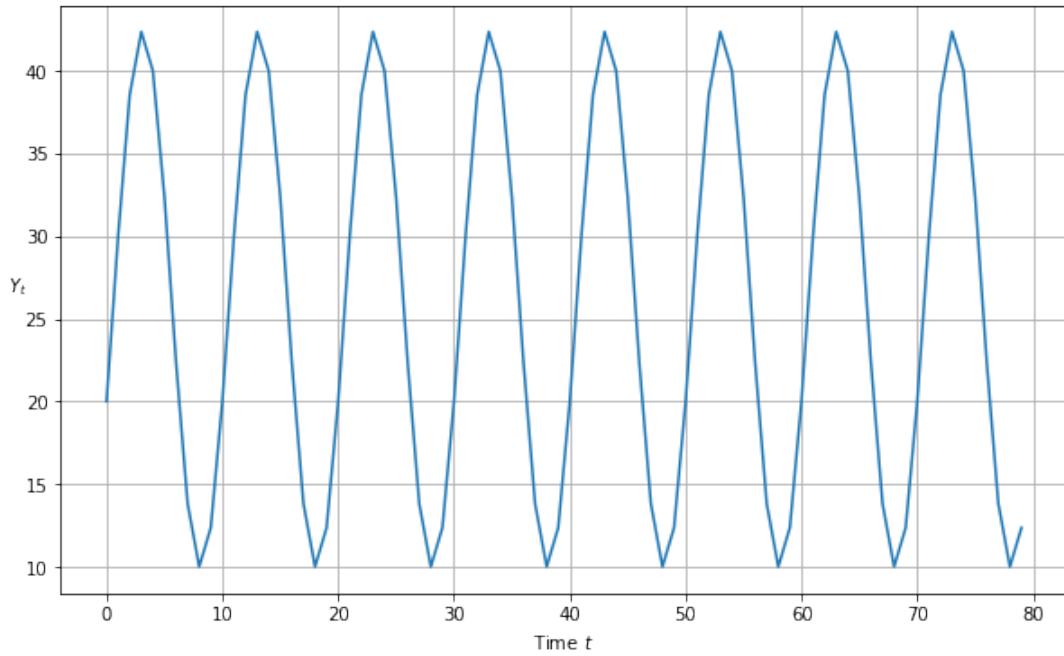
## Apply the reverse-engineering function f
ρ1, ρ2, a, b = f(r, ϕ)

# Drop the imaginary part so that it is a valid input into y_nonstochastic
a = a.real
b = b.real

print(f"a, b = {a}, {b}")

ytemp = y_nonstochastic(α=a, β=b, y_0=20, y_1=30)
plot_y(ytemp)

a, b = 0.6180339887498949, 1.0
Roots are complex with modulus less than one; therefore damped oscillations
Roots are [0.80901699+0.58778525j 0.80901699-0.58778525j]
Roots are complex
Roots are less than one
```



15.4.7 Digression: Using Sympy to Find Roots

We can also use sympy to compute analytic formulas for the roots

In [14]: `init_printing()`

```
r1 = Symbol("ρ_1")
r2 = Symbol("ρ_2")
z = Symbol("z")

sympy.solve(z**2 - r1*z - r2, z)
```

Out[14]: $\left[\frac{\rho_1}{2} - \frac{\sqrt{\rho_1^2 + 4\rho_2}}{2}, \quad \frac{\rho_1}{2} + \frac{\sqrt{\rho_1^2 + 4\rho_2}}{2} \right]$

In [15]: `a = Symbol("α")
b = Symbol("β")
r1 = a + b
r2 = -b`

```
sympy.solve(z**2 - r1*z - r2, z)
```

Out[15]: $\left[\frac{\alpha}{2} + \frac{\beta}{2} - \frac{\sqrt{\alpha^2 + 2\alpha\beta + \beta^2 - 4\beta}}{2}, \quad \frac{\alpha}{2} + \frac{\beta}{2} + \frac{\sqrt{\alpha^2 + 2\alpha\beta + \beta^2 - 4\beta}}{2} \right]$

15.5 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

In [16]: `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 and proceeds to analyze the roots of the characteristic
polynomial and also generate a simulation.
"""

# Useful constants
ρ1 = α + β
ρ2 = -β

# Categorize solution
categorize_solution(ρ1, ρ2)

# Find roots of polynomial
roots = np.roots([1, -ρ1, -ρ2])
print(roots)

# Check if real or complex
if all(isinstance(root, complex) for root in roots):
    print('Roots are complex')
else:
    print('Roots are real')

# Check if roots are less than one
if all(abs(root) < 1 for root in roots):
    print('Roots are less than one')
else:
    print('Roots are not less than one')

# Generate shocks
ε = np.random.normal(0, 1, n)

# Define transition equation
def transition(x, t): return ρ1 * \
    x[t - 1] + ρ2 * x[t - 2] + γ + σ * ε[t]

# Set initial conditions
y_t = [y_0, y_1]

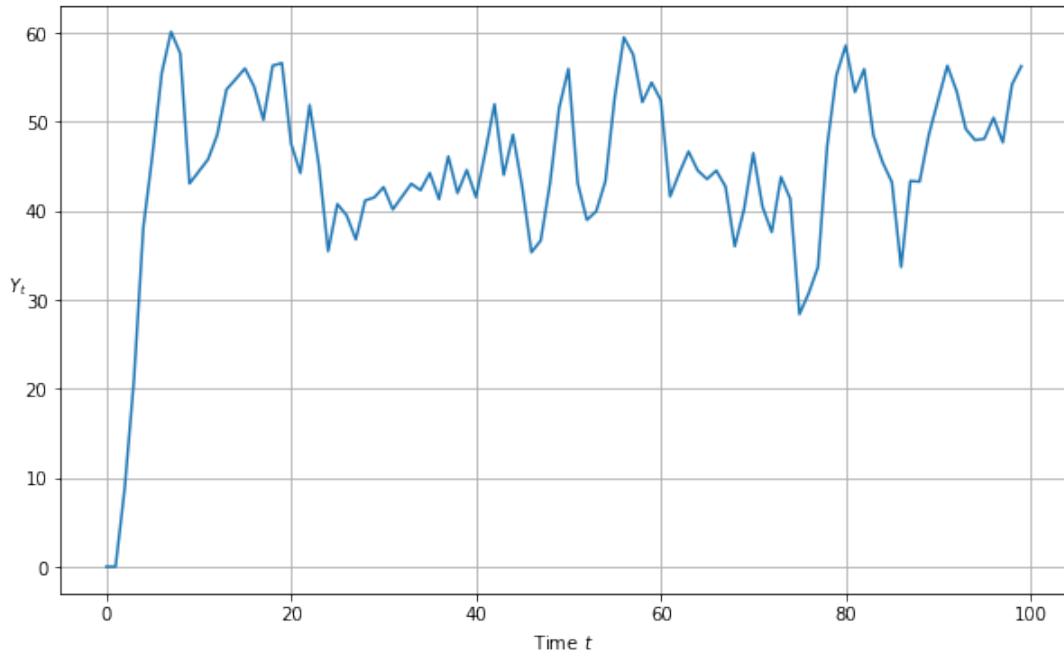
# Generate y_t series
for t in range(2, n):
    y_t.append(transition(y_t, t))

return y_t

plot_y(y_stochastic())
```

Roots are real and absolute values are less than one; therefore get smooth convergence
to a steady state
[0.7236068 0.2763932]

```
Roots are real
Roots are less than one
```



Let's do a simulation in which there are shocks and the characteristic polynomial has complex roots

In [17]: $r = .97$

```
period = 10    # Length of cycle in units of time
ϕ = 2 * math.pi/period

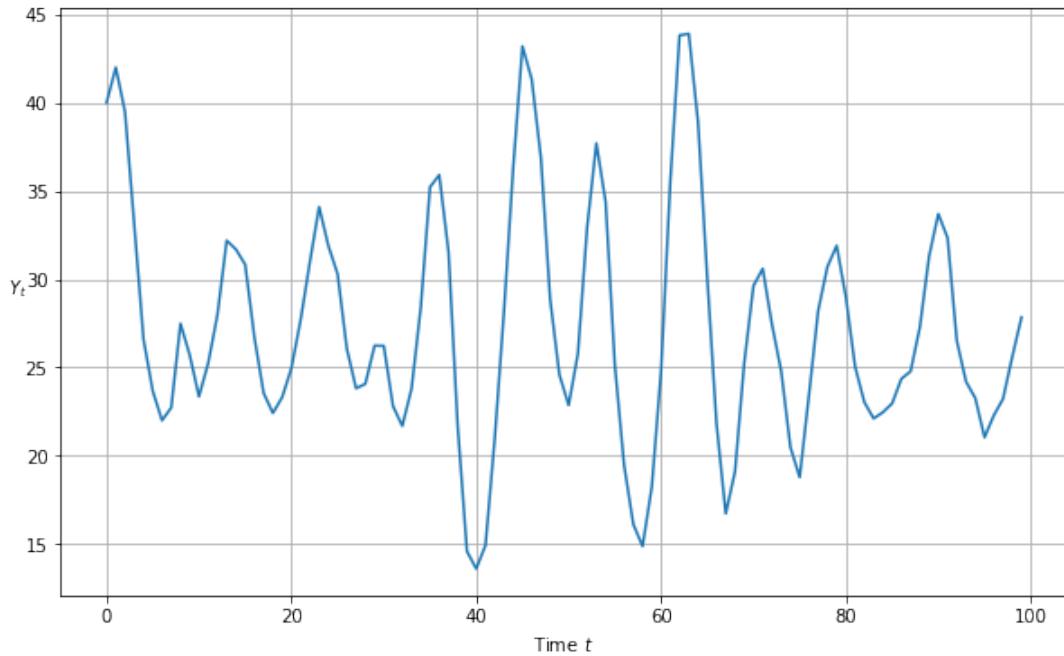
### Apply the reverse-engineering function f

ρ₁, ρ₂, a, b = f(r, ϕ)

# Drop the imaginary part so that it is a valid input into y_nonstochastic
a = a.real
b = b.real

print(f"a, b = {a}, {b}")
plot_y(y_stochastic(y_0=40, y_1 = 42, α=a, β=b, σ=2, n=100))
```

```
a, b = 0.6285929690873979, 0.9409000000000001
Roots are complex with modulus less than one; therefore damped oscillations
[0.78474648+0.57015169j 0.78474648-0.57015169j]
Roots are complex
Roots are less than one
```



15.6 Government Spending

This function computes a response to either a permanent or one-off increase in government expenditures

```
In [18]: def y_stochastic_g(y_0=20,
                           y_1=20,
                           alpha=0.8,
                           beta=0.2,
                           gamma=10,
                           n=100,
                           sigma=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
    """

    # Useful constants
    rho1 = alpha + beta
    rho2 = -beta

    # Categorize solution
    categorize_solution(rho1, rho2)

    # Find roots of polynomial
    roots = np.roots([1, -rho1, -rho2])
    print(roots)
```

```

# Check if real or complex
if all(isinstance(root, complex) for root in roots):
    print('Roots are complex')
else:
    print('Roots are real')

# Check if roots are less than one
if all(abs(root) < 1 for root in roots):
    print('Roots are less than one')
else:
    print('Roots are not less than one')

# Generate shocks
epsilon = np.random.normal(0, 1, n)

def transition(x, t, g):

    # Non-stochastic - separated to avoid generating random series
    # when not needed
    if sigma == 0:
        return rho1 * x[t - 1] + rho2 * x[t - 2] + gamma + g

    # Stochastic
    else:
        epsilon = np.random.normal(0, 1, n)
        return rho1 * x[t - 1] + rho2 * x[t - 2] + gamma + g + sigma * epsilon[t]

# Create list and set initial conditions
y_t = [y_0, y_1]

# Generate y_t series
for t in range(2, n):

    # No government spending
    if g == 0:
        y_t.append(transition(y_t, t))

    # Government spending (no shock)
    elif g != 0 and duration == None:
        y_t.append(transition(y_t, t))

    # Permanent government spending shock
    elif duration == 'permanent':
        if t < g_t:
            y_t.append(transition(y_t, t, g=0))
        else:
            y_t.append(transition(y_t, t, g=g))

    # One-off government spending shock
    elif duration == 'one-off':
        if t == g_t:
            y_t.append(transition(y_t, t, g=g))
        else:
            y_t.append(transition(y_t, t, g=0))

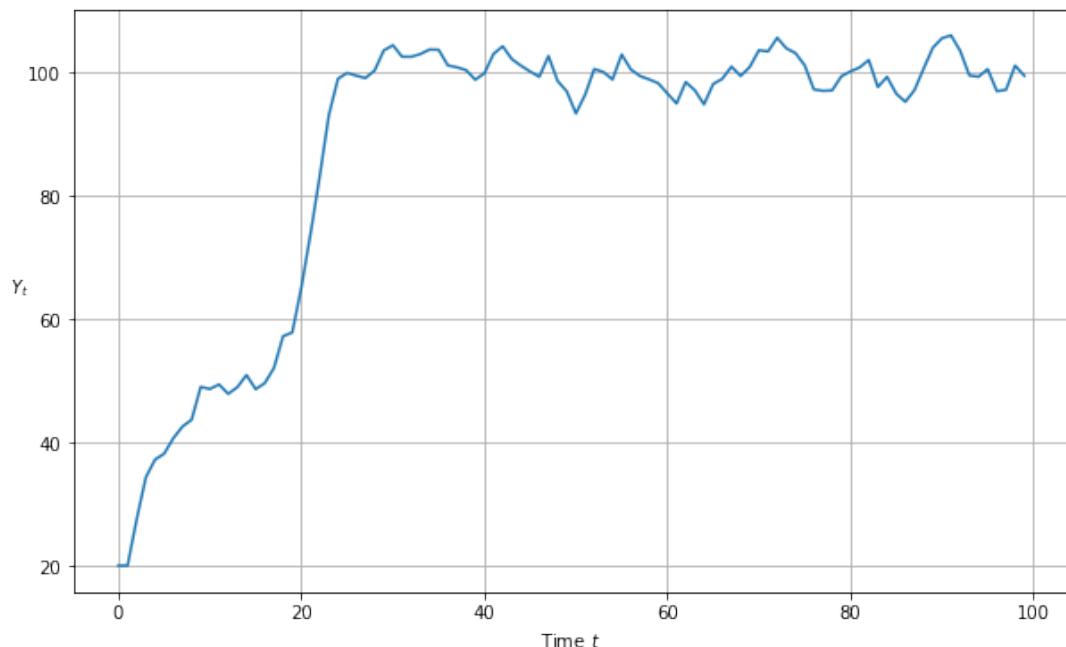
return y_t

```

A permanent government spending shock can be simulated as follows

```
In [19]: plot_y(y_stochastic_g(g=10, g_t=20, duration='permanent'))
```

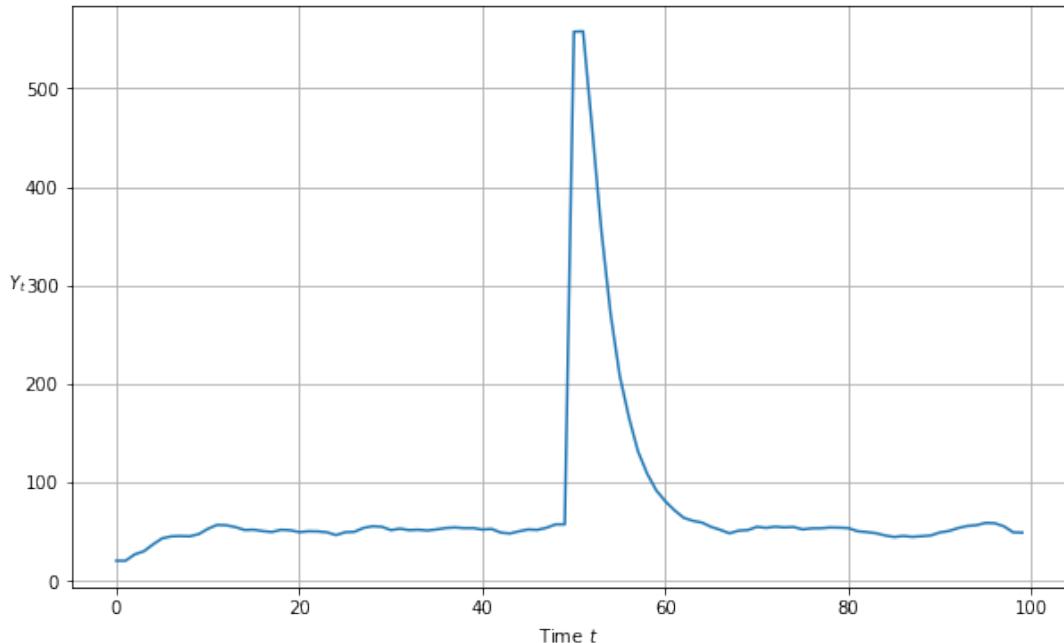
Roots are real and absolute values are less than one; therefore get smooth
 ↵convergence
 to a steady state
 $[0.7236068 \ 0.2763932]$
 Roots are real
 Roots are less than one



We can also see the response to a one time jump in government expenditures

```
In [20]: plot_y(y_stochastic_g(g=500, g_t=50, duration='one-off'))
```

Roots are real and absolute values are less than one; therefore get smooth
 ↵convergence
 to a steady state
 $[0.7236068 \ 0.2763932]$
 Roots are real
 Roots are less than one



15.7 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

In [21]: `class Samuelson():`

"""This class represents the Samuelson model, otherwise known as the multiple-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

\alpha : scalar

Marginal propensity to consume

\beta : scalar

Accelerator coefficient

n : int

```

Number of iterations
 $\sigma$  : 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,
             alpha=1.3,
             beta=0.2,
             gamma=10,
             n=100,
             sigma=0,
             g=0,
             g_t=0,
             duration=None):
    self.y_0, self.y_1, self.alpha, self.beta = y_0, y_1, alpha, beta
    self.n, self.g, self.g_t, self.duration = n, g, g_t, duration
    self.gamma, self.sigma = gamma, sigma
    self.rho1 = alpha + beta
    self.rho2 = -beta
    self.roots = np.roots([1, -self.rho1, -self.rho2])

def root_type(self):
    if all(isinstance(root, complex) for root in self.roots):
        return 'Complex conjugate'
    elif len(self.roots) > 1:
        return 'Double real'
    else:
        return 'Single real'

def root_less_than_one(self):
    if all(abs(root) < 1 for root in self.roots):
        return True

def solution_type(self):
    rho1, rho2 = self.rho1, self.rho2
    discriminant = rho1 ** 2 + 4 * rho2
    if rho2 >= 1 + rho1 or rho2 <= -1:
        return 'Explosive oscillations'
    elif rho1 + rho2 >= 1:
        return 'Explosive growth'
    elif discriminant < 0:
        return 'Damped oscillations'
    else:
        return 'Steady state'

```

```

def _transition(self, x, t, g):
    # Non-stochastic - separated to avoid generating random series
    # when not needed
    if self.σ == 0:
        return self.ρ1 * x[t - 1] + self.ρ2 * x[t - 2] + self.γ + g

    # Stochastic
    else:
        ε = np.random.normal(0, 1, self.n)
        return self.ρ1 * x[t - 1] + self.ρ2 * x[t - 2] + self.γ + g \
            + self.σ * ε[t]

def generate_series(self):
    # Create list and set initial conditions
    y_t = [self.y_0, self.y_1]

    # Generate y_t series
    for t in range(2, self.n):

        # No government spending
        if self.g == 0:
            y_t.append(self._transition(y_t, t))

        # Government spending (no shock)
        elif self.g != 0 and self.duration == None:
            y_t.append(self._transition(y_t, t))

        # Permanent government spending shock
        elif self.duration == 'permanent':
            if t < self.g_t:
                y_t.append(self._transition(y_t, t, g=0))
            else:
                y_t.append(self._transition(y_t, t, g=self.g))

        # One-off government spending shock
        elif self.duration == 'one-off':
            if t == self.g_t:
                y_t.append(self._transition(y_t, t, g=self.g))
            else:
                y_t.append(self._transition(y_t, t, g=0))
    return y_t

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:
        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):
    fig, ax = plt.subplots(figsize=(10, 6))
    ax.plot(self.generate_series())
    ax.set(xlabel='Iteration', xlim=(0, self.n))
    ax.set_ylabel('$Y_t$', rotation=0)
    ax.grid()

    # Add parameter values to plot
    paramstr = f'$\alpha={self.α:.2f}$ \n $\beta={self.β:.2f}$ \n \
$\\gamma={self.γ:.2f}$ \n $\\sigma={self.σ:.2f}$ \n \
$\\rho_1={self.ρ1:.2f}$ \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):

    # Uses the param_plot() function defined earlier (it is then able
    # to be used standalone or as part of the model)

    fig = param_plot()
    ax = fig.gca()

    # Add λ values to legend
    for i, root in enumerate(self.roots):
        if isinstance(root, complex):
            # Need to fill operator for positive as string is split apart
            operator = ['+', '']
            label = rf'$\lambda_{i+1} = {sam.roots[i].real:.2f} \ 
{operator[i]} {sam.roots[i].imag:.2f}i$'
        else:
            label = rf'$\lambda_{i+1} = {sam.roots[i].real:.2f}$'
        ax.scatter(0, 0, 0, label=label) # dummy to add to legend

    # Add ρ pair to plot
    ax.scatter(self.ρ1, self.ρ2, 100, 'red', '+',
               label=r'$(\rho_1, \rho_2)$', zorder=5)

    plt.legend(fontsize=12, loc=3)

    return fig

```

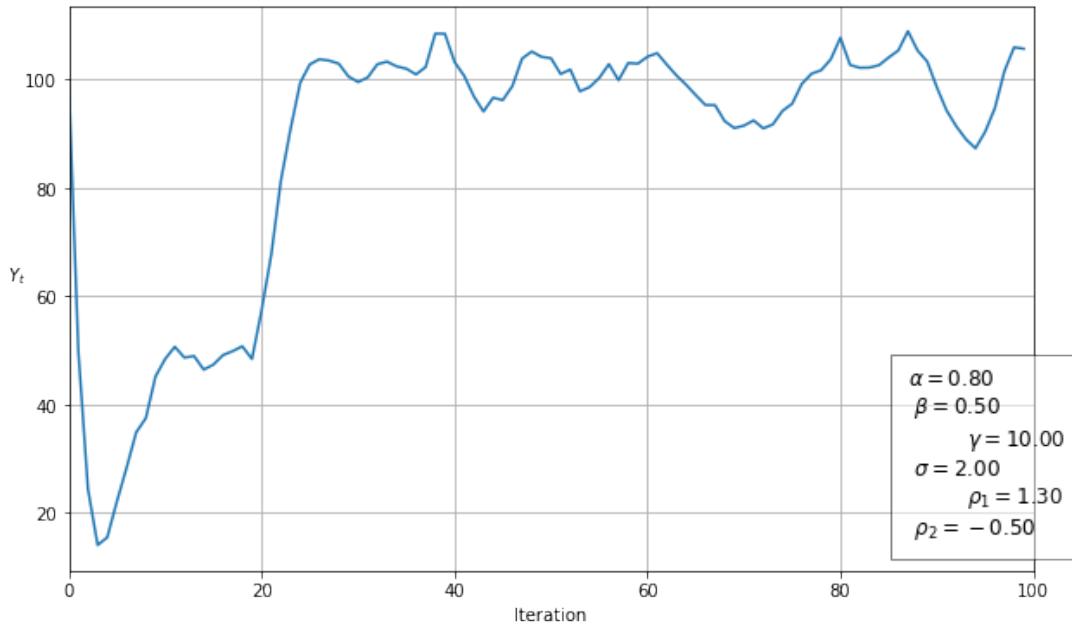
15.7.1 Illustration of Samuelson Class

Now we'll put our Samuelson class to work on an example

```
In [22]: sam = Samuelson(alpha=0.8, beta=0.5, sigma=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 sigma = 2
Government spending equal to 10
Permanent government spending shock at t = 20
```

```
In [23]: sam.plot()
plt.show()
```

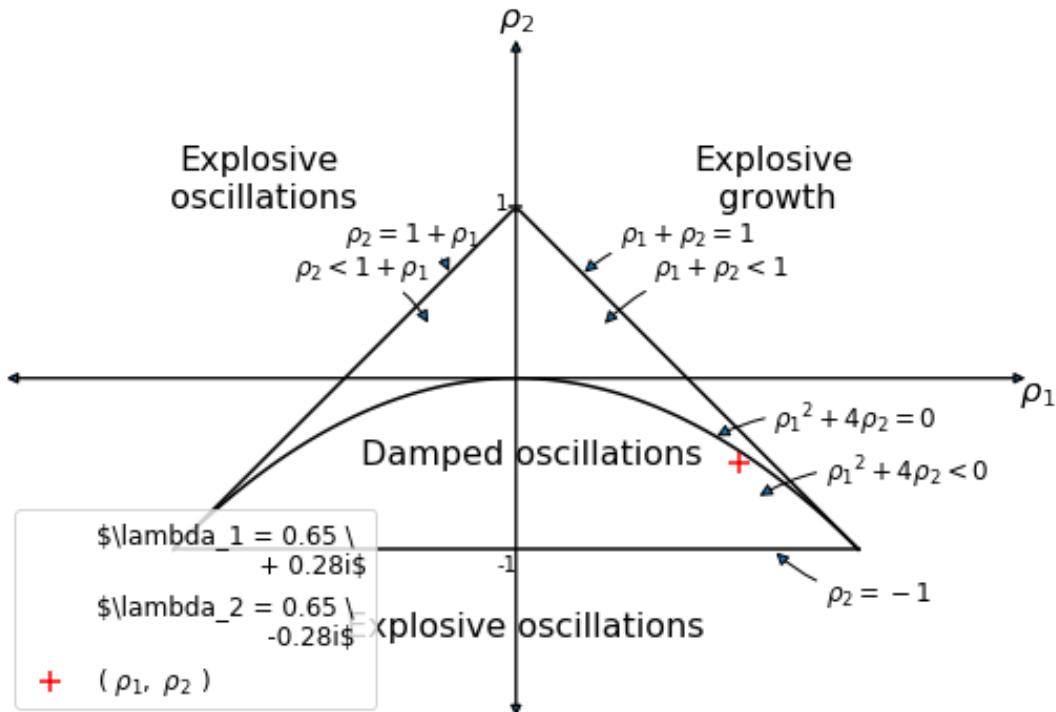


15.7.2 Using the Graph

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

```
In [24]: sam.param_plot()
plt.show()
```



15.8 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

```
In [25]: """This script maps the Samuelson model in the the
``LinearStateSpace`` class
"""

alpha = 0.8
beta = 0.9
rho1 = alpha + beta
rho2 = -beta
gamma = 10
sigma = 1
g = 10
n = 100

A = [[1, 0, 0],
      [gamma + g, rho1, rho2],
      [0, 1, 0]]
# this is Y_{t+1}

G = [[gamma + g, rho1, rho2], # this is C_{t+1}
      [gamma, alpha, 0], # this is I_{t+1}
      [0, beta, -beta]]
# this is I_{t+1}

mu_0 = [1, 100, 100]
C = np.zeros((3,1))
```

```
C[1] = σ # stochastic

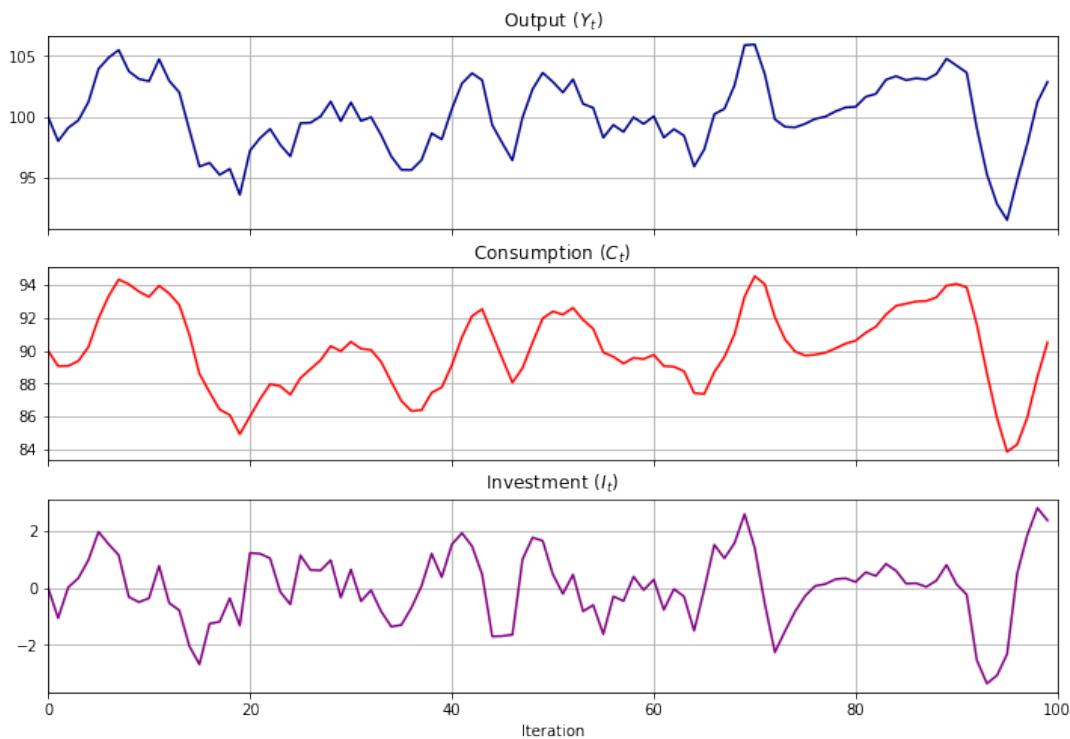
sam_t = LinearStateSpace(A, C, G, mu_0=μ_0)

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))
    ax.grid()

axes[-1].set_xlabel('Iteration')

plt.show()
```



15.8.1 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

```
In [26]: imres = sam_t.impulse_response()
imres = np.asarray(imres)
y1 = imres[:, :, 0]
y2 = imres[:, :, 1]
y1.shape
```

Out[26]: (2, 6, 1)

Now let's compute the zeros of the characteristic polynomial by simply calculating the eigenvalues of A

```
In [27]: A = np.asarray(A)
w, v = np.linalg.eig(A)
print(w)

[0.85+0.42130749j 0.85-0.42130749j 1. +0.j]
```

15.8.2 Inheriting Methods from `LinearStateSpace`

We could also create a subclass of `LinearStateSpace` (inheriting all its methods and attributes) to add more functions to use

In [28]: `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=100,
             alpha=0.8,
             beta=0.9,
             gamma=10,
             sigma=1,
             g=10):

    self.alpha, self.beta = alpha, beta
    self.y_0, self.y_1, self.g = y_0, y_1, g
    self.gamma, self.sigma = gamma, sigma

    # Define initial conditions
    self.u_0 = [1, y_0, y_1]

    self.rho1 = alpha + beta
    self.rho2 = -beta

    # Define transition matrix
    self.A = [[1, 0, 0],
              [gamma + g, self.rho1, self.rho2],
              [0, 1, 0]]

    # Define output matrix
    self.G = [[gamma + g, self.rho1, self.rho2],           # this is Y_{t+1}
              [gamma, alpha, 0],                      # this is C_{t+1}
              [0, beta, -beta]]                      # this is I_{t+1}

    self.C = np.zeros((3, 1))
    self.C[1] = sigma # stochastic
```

```

# Initialize LSS with parameters from Samuelson model
LinearStateSpace.__init__(self, self.A, self.C, self.G, mu_0=self.
                           ↵μ_0)

def plot_simulation(self, ts_length=100, stationary=True):

    # Temporarily store original parameters
    temp_μ = self.μ_0
    temp_Σ = self.Sigma_0

    # Set distribution parameters equal to their stationary
    # values for simulation
    if stationary == True:
        try:
            self.μ_x, self.μ_y, self.σ_x, self.σ_y = \
                self.stationary_distributions()
            self.μ_0 = self.μ_y
            self.Σ_0 = self.σ_y
        # Exception where no convergence achieved when
        # calculating stationary distributions
        except ValueError:
            print('Stationary distribution does not exist')

    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))
        ax.grid()

    axes[-1].set_xlabel('Iteration')

    # Reset distribution parameters to their initial values
    self.μ_0 = temp_μ
    self.Sigma_0 = temp_Σ

    return fig

def plot_irf(self, j=5):

    x, y = self.impulse_response(j)

    # Reshape into 3 x j matrix for plotting purposes
    yimf = np.array(y).flatten().reshape(j+1, 3).T

    fig, axes = plt.subplots(3, 1, sharex=True, figsize=(12, 8))
    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)
        ax.grid()

```

```

        axes[0].set_title('Impulse Response Functions')
        axes[-1].set_xlabel('Iteration')

    return fig

def multipliers(self, j=5):
    x, y = self.impulse_response(j)
    return np.sum(np.array(y).flatten().reshape(j+1, 3), axis=0)

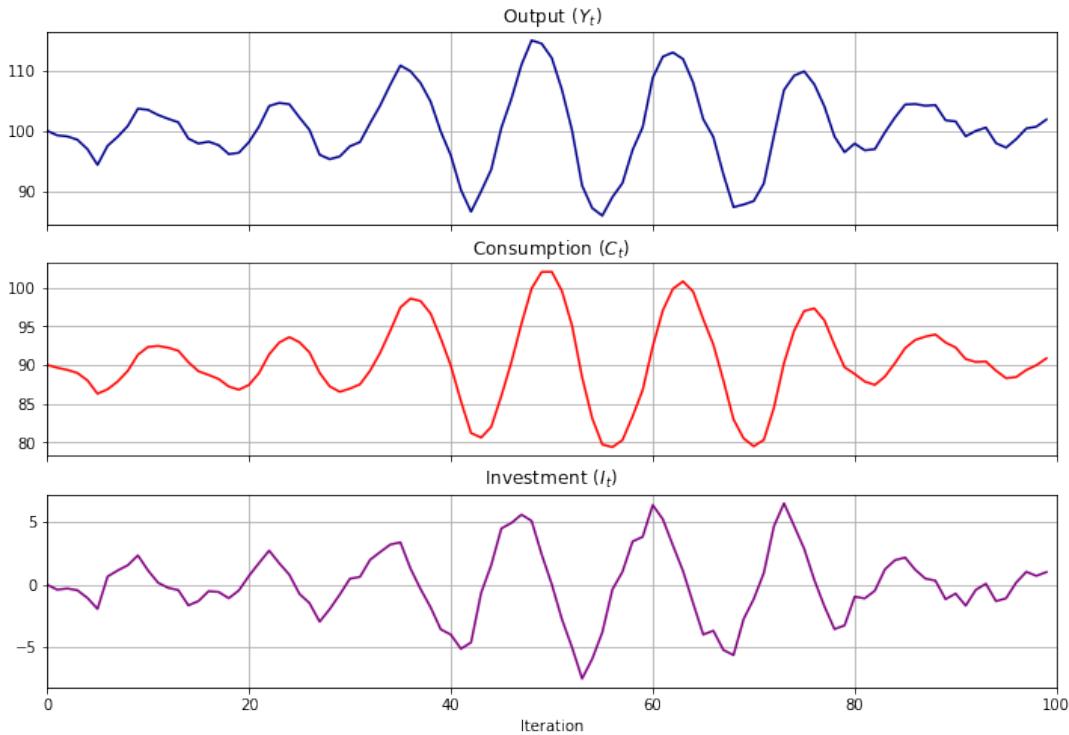
```

15.8.3 Illustrations

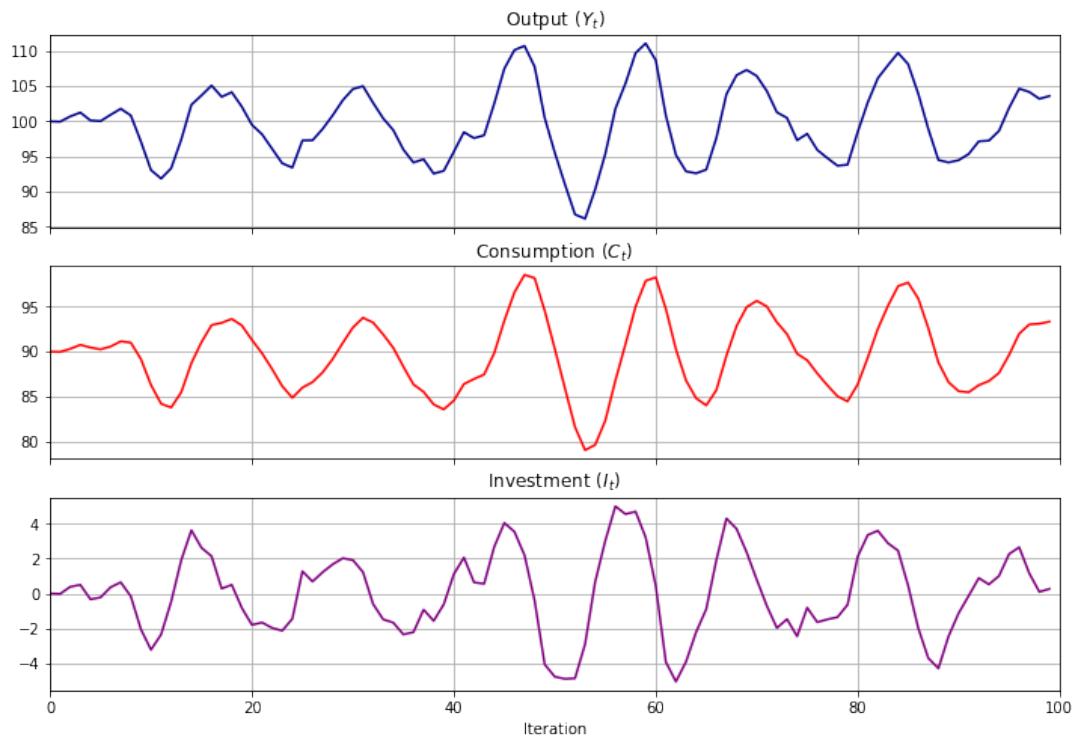
Let's show how we can use the `SamuelsonLSS`

In [29]: `samlss = SamuelsonLSS()`

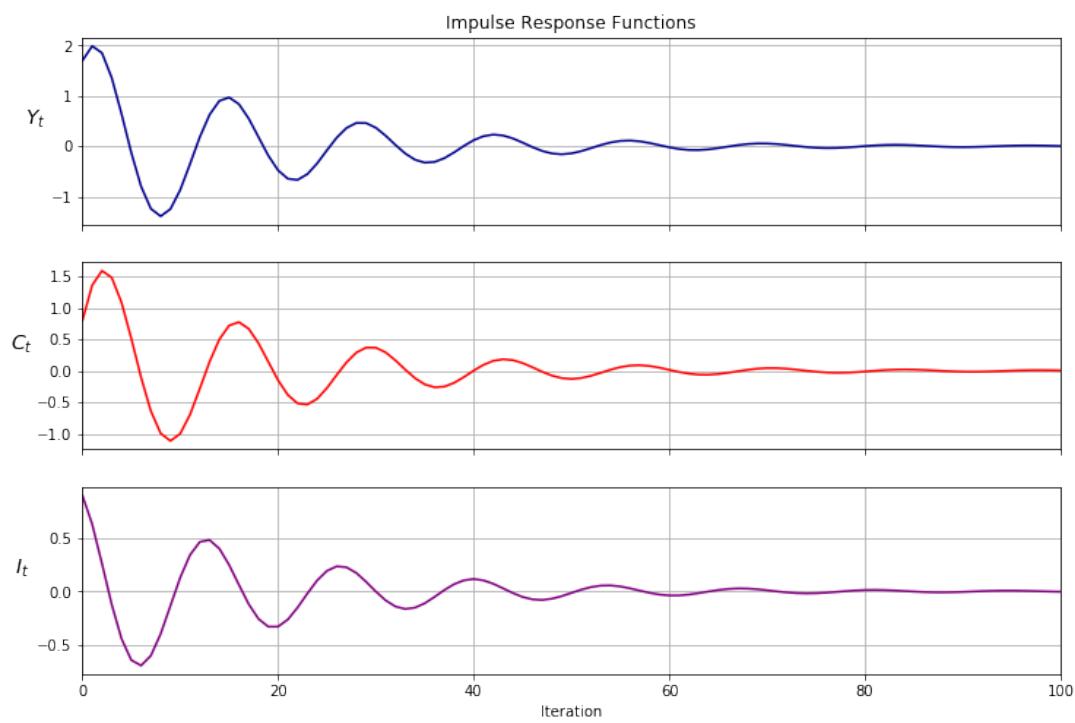
In [30]: `samlss.plot_simulation(100, stationary=False)`
`plt.show()`



In [31]: `samlss.plot_simulation(100, stationary=True)`
`plt.show()`



```
In [32]: samlss.plot_irf(100)  
plt.show()
```



```
In [33]: samlss.multipliers()
```

```
Out[33]: array([7.414389, 6.835896, 0.578493])
```

15.9 Pure Multiplier Model

Let's shut down the accelerator by setting $b = 0$ to get a pure multiplier model

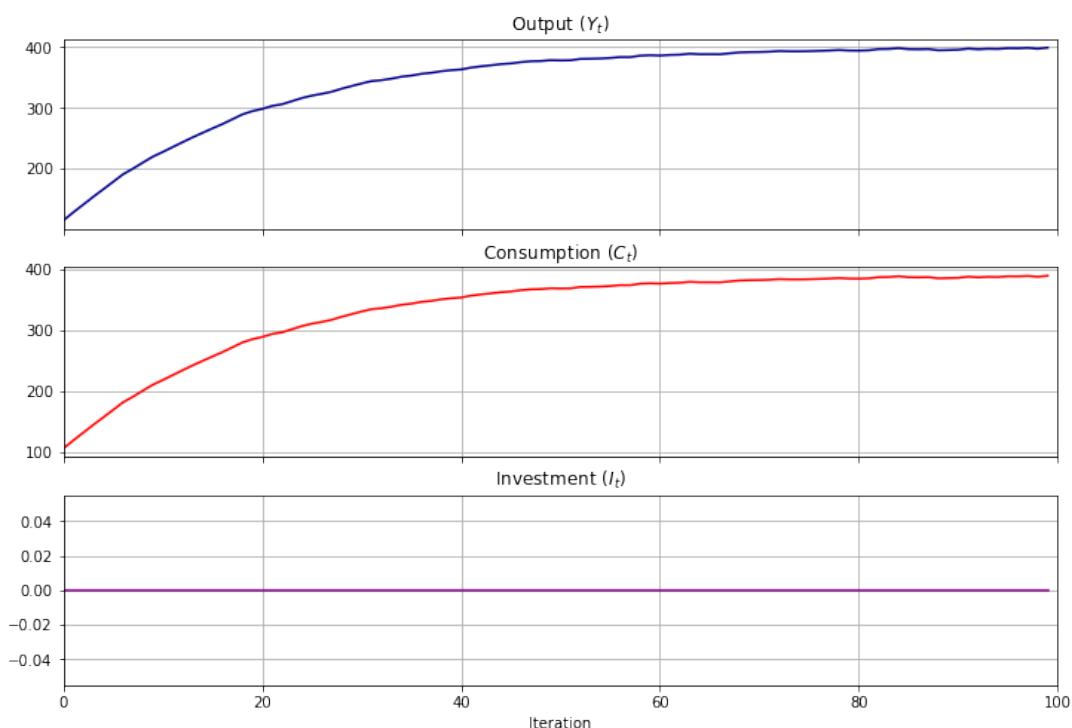
- the absence of cycles gives an idea about why Samuelson included the accelerator

```
In [34]: pure_multiplier = SamuelsonLSS(alpha=0.95, beta=0)
```

```
In [35]: pure_multiplier.plot_simulation()
```

Stationary distribution does not exist

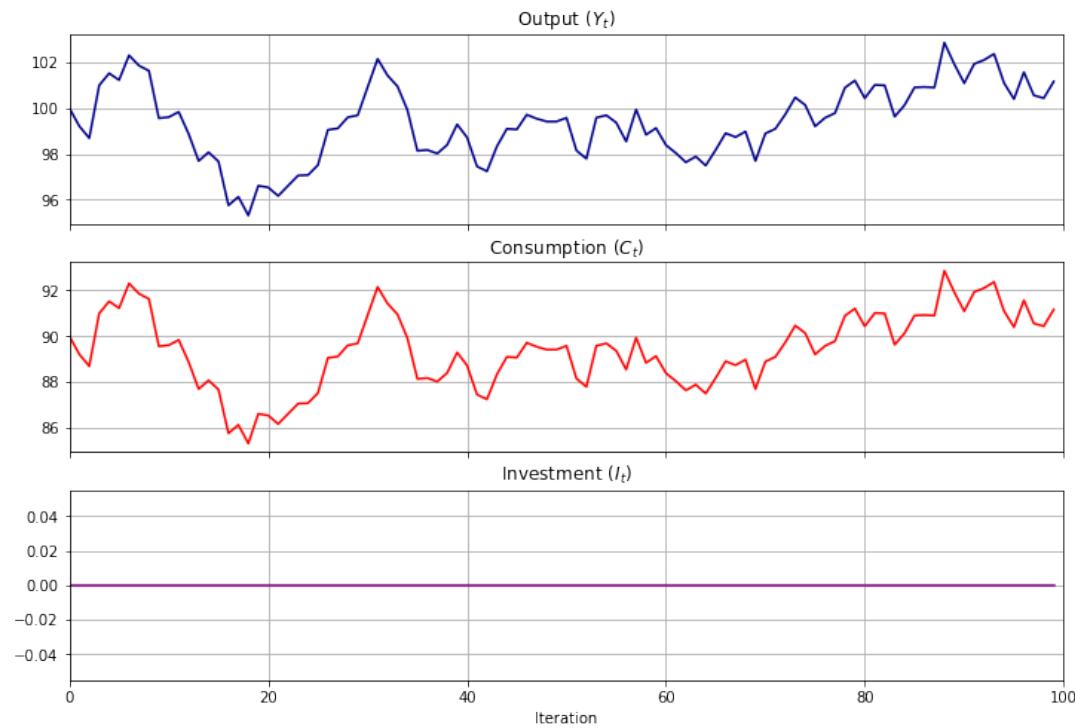
Out[35]:



```
In [36]: pure_multiplier = SamuelsonLSS(alpha=0.8, beta=0)
```

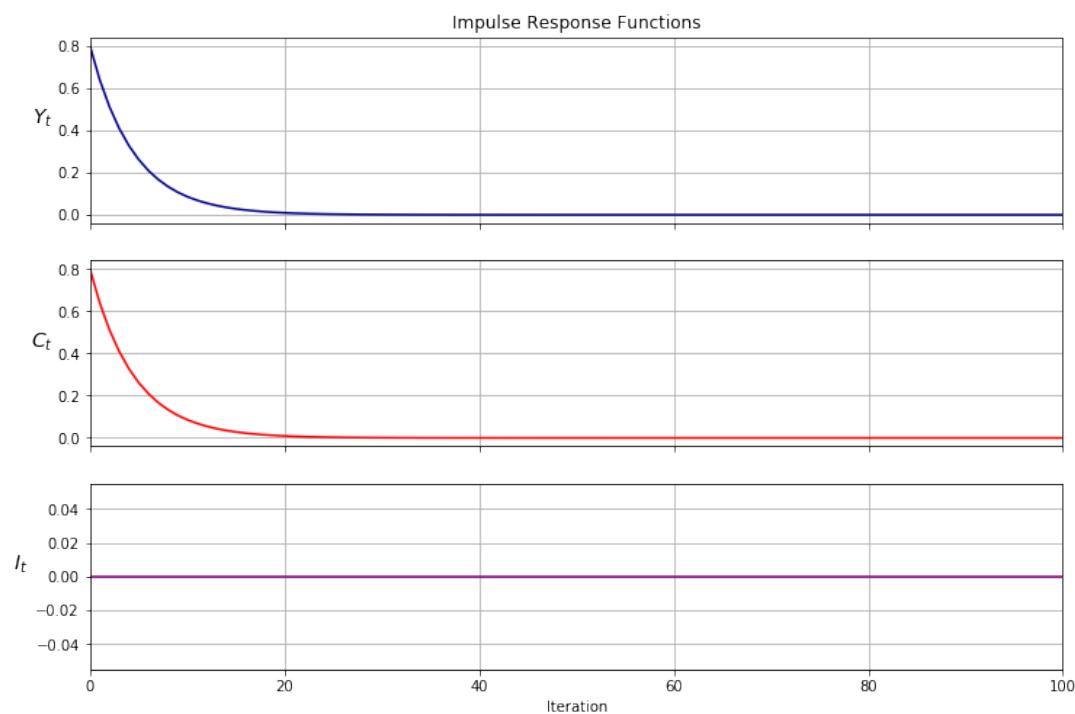
```
In [37]: pure_multiplier.plot_simulation()
```

Out[37]:



In [38]: `pure_multiplier.plot_irf(100)`

Out[38]:



15.10 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 [93].

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 16

Kesten Processes and Firm Dynamics

16.1 Contents

- Overview 16.2
- Kesten Processes 16.3
- Heavy Tails 16.4
- Application: Firm Dynamics 16.5
- Exercises 16.6
- Solutions 16.7

In addition to what's in Anaconda, this lecture will need the following libraries:

```
In [1]: !pip install quantecon  
!pip install --upgrade yfinance
```

16.2 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:

```
In [2]: import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

import quantecon as qe

/home/ubuntu/anaconda3/lib/python3.7/site-packages/numba/np/ufunc/parallel.py:
 355:
NumbaWarning: The TBB threading layer requires TBB version 2019.5 or later i.e.,
TBB_INTERFACE_VERSION >= 11005. Found TBB_INTERFACE_VERSION = 11004. The TBB
threading
layer is disabled.
warnings.warn(problem)
```

The following two lines are only added to avoid a `FutureWarning` caused by compatibility issues between pandas and matplotlib.

```
In [3]: from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters()
```

Additional technical background related to this lecture can be found in the monograph of [17].

16.3 Kesten Processes

A **Kesten process** is a stochastic process of the form

$$X_{t+1} = a_{t+1}X_t + \eta_{t+1} \quad (1)$$

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.

16.3.1 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.

```
In [4]: import yfinance as yf
import pandas as pd
```

```
s = yf.download('^IXIC', '2006-1-1', '2019-11-1')['Adj Close']

r = s.pct_change()

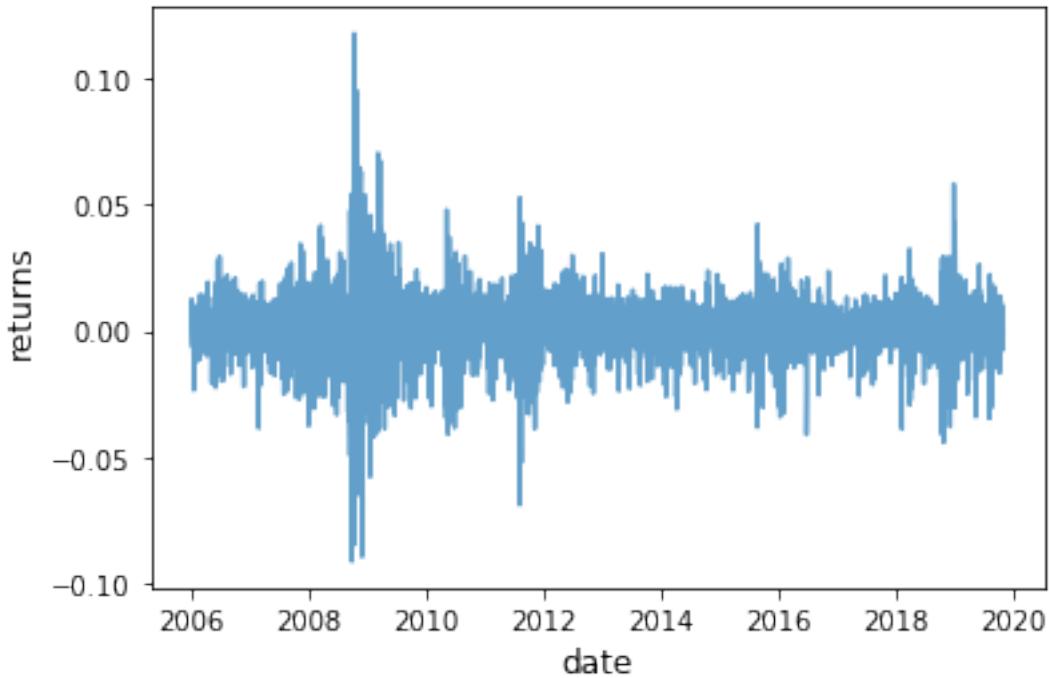
fig, ax = plt.subplots()

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

$$\sigma_{t+1}^2 = \alpha_0 + \sigma_t^2(\alpha_1 \xi_{t+1}^2 + \beta) \quad (2)$$

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

$$r_t = \sigma_t \zeta_t \quad (3)$$

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.

16.3.2 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

$$w_{t+1} = R_{t+1}sw_t + y_{t+1} \quad (4)$$

where $\{R_t\}$ is the gross rate of return on assets.

If $\{R_t\}$ and $\{y_t\}$ are both IID, then (4) is a Kesten process.

16.3.3 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 (1) if

$$X_t \sim F^* \implies a_{t+1}X_t + \eta_{t+1} \sim F^* \quad (5)$$

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

$$F^*(y) = \int \mathbb{P}\{a_{t+1}x + \eta_{t+1} \leq y\}F^*(dx) \quad \text{for all } y \geq 0. \quad (6)$$

The left hand side is the distribution of the next period state when the current state is drawn from F^* .

The equality in (6) states that this distribution is unchanged.

16.3.4 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 (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

$$\int \mathbb{P}\{R_{t+1}sw + y_{t+1} \leq y\} F^*(dw)$$

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.

16.3.5 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

$$\mathbb{E} \ln a_t < 0 \quad \text{and} \quad \mathbb{E} \eta_t < \infty \tag{7}$$

then a unique stationary distribution exists on \mathbb{R}_+ .

- See, for example, theorem 2.1.3 of [17], which provides slightly weaker conditions.

As one application of this result, we see that the wealth process (4) will have a unique stationary distribution whenever labor income has finite mean and $\mathbb{E} \ln R_t + \ln s < 0$.

16.4 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.

16.4.1 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., [17], theorem 2.4.4) states that if

1. the stationarity conditions in (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

$$\lim_{x \rightarrow \infty} x^\alpha \mathbb{P}\{X^* > x\} = c$$

for some positive constant c .

16.4.2 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.

```
In [5]:  $\mu = -0.5$ 
 $\sigma = 1.0$ 
```

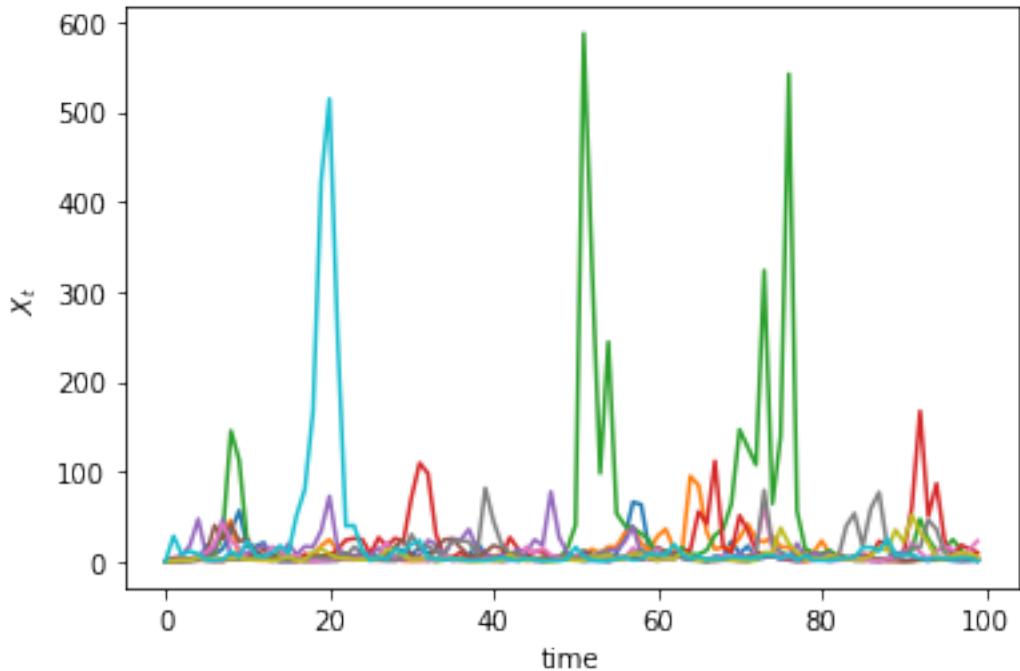
```
def kesten_ts(ts_length=100):
    x = np.zeros(ts_length)
    for t in range(ts_length-1):
        a = np.exp( $\mu + \sigma * \text{np.random.randn()}$ )
        b = np.exp(np.random.randn())
        x[t+1] = a * x[t] + b
    return x

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()
```



16.5 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., [7], [41]).

Let us try to explain this rather striking fact using the Kesten–Goldie Theorem.

16.5.1 Gibrat's Law

It was postulated many years ago by Robert Gibrat [42] 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

$$\frac{s_{t+1}}{s_t} = a_{t+1} \quad (8)$$

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., [35] and [46]) and
2. the growth rate of small firms is more volatile than that of large firms [32].

On the other hand, Gibrat's law is generally found to be a reasonable approximation for large firms [35].

We can accommodate these empirical findings by modifying (8) to

$$s_{t+1} = a_{t+1}s_t + b_{t+1} \quad (9)$$

where $\{a_t\}$ and $\{b_t\}$ are both IID and independent of each other.

In the exercises you are asked to show that (9) is more consistent with the empirical findings presented above than Gibrat's law in (8).

16.5.2 Heavy Tails

So what has this to do with Pareto tails?

The answer is that (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.

16.6 Exercises

16.6.1 Exercise 1

Simulate and plot 15 years of daily returns (consider each year as having 250 working days) using the GARCH(1, 1) process in (2)–(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.

16.6.2 Exercise 2

In our discussion of firm dynamics, it was claimed that (9) is more consistent with the empirical literature than Gibrat's law in (8).

(The empirical literature was reviewed immediately above (9).)

In what sense is this true (or false)?

16.6.3 Exercise 3

Consider an arbitrary Kesten process as given in (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.

16.6.4 Exercise 4

One unrealistic aspect of the firm dynamics specified in (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 [57].

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

$$s_{t+1} = e_{t+1} \mathbb{1}\{s_t < \bar{s}\} + (a_{t+1}s_t + b_{t+1}) \mathbb{1}\{s_t \geq \bar{s}\} \quad (10)$$

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 (10) 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 (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 (10) 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

```
In [6]: 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
```

16.7 Solutions

16.7.1 Exercise 1

Here is one solution:

```
In [7]: α_0 = 1e-5
α_1 = 0.1
```

```

 $\beta = 0.9$ 

years = 15
days = years * 250

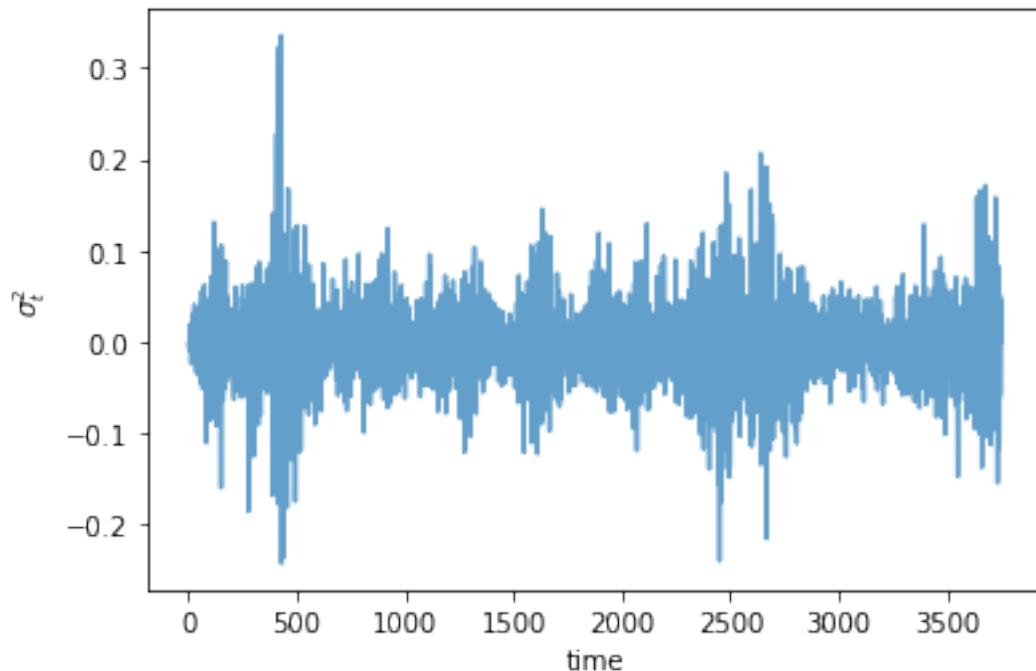
def garch_ts(ts_length=days):
    sigma2 = 0
    r = np.zeros(ts_length)
    for t in range(ts_length-1):
        xi = np.random.randn()
        sigma2 = alpha_0 + sigma2 * (alpha_1 * xi**2 + beta)
        r[t] = np.sqrt(sigma2) * np.random.randn()
    return r

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()

```



16.7.2 Exercise 2

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 (9) are more consistent with points 1-2 than Gibrat's law.

To see why, we rewrite (9) in terms of growth dynamics:

$$\frac{s_{t+1}}{s_t} = a_{t+1} + \frac{b_{t+1}}{s_t} \quad (11)$$

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 (11) clearly approaches Gibrat's law (8) as s_t gets large.

16.7.3 Exercise 3

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 $\alpha = -2\mu/\sigma^2$.

16.7.4 Exercise 4

Here's one solution. First we generate the observations:

```
In [8]: from numba import njit, prange
from numpy.random import randn
```

```
@njit(parallel=True)
def generate_draws(mu_a=-0.5,
                   sigma_a=0.1,
                   mu_b=0.0,
                   sigma_b=0.5,
                   mu_e=0.0,
```

```

sigma_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(mu_e + sigma_e * randn())
        else:
            a = np.exp(mu_a + sigma_a * randn())
            b = np.exp(mu_b + sigma_b * randn())
            new_s = a * s + b
        s = new_s
    draws[m] = s

return draws

data = generate_draws()

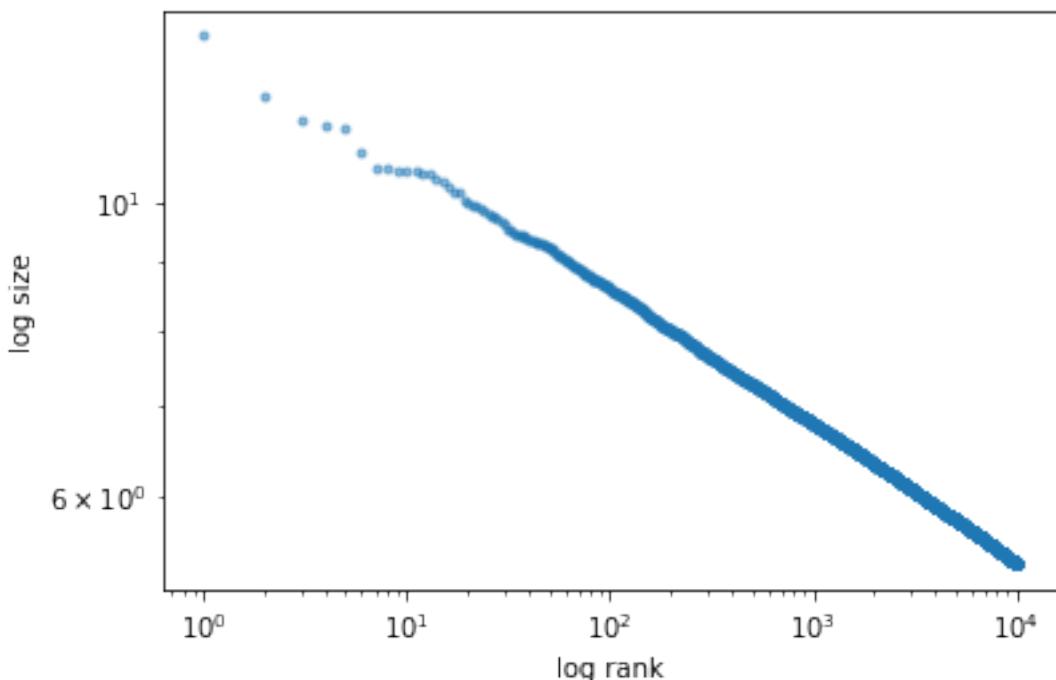
```

Now we produce the rank-size plot:

```
In [9]: 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 17

Wealth Distribution Dynamics

17.1 Contents

- Overview 17.2
- Lorenz Curves and the Gini Coefficient 17.3
- A Model of Wealth Dynamics 17.4
- Implementation 17.5
- Applications 17.6
- Exercises 17.7
- Solutions 17.8

In addition to what's in Anaconda, this lecture will need the following libraries:

```
In [1]: !pip install quantecon
```

17.2 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, [\[9\]](#).

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.

17.2.1 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.

```
In [2]: import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

import quantecon as qe
from numba import njit, float64, prange
from numba.experimental import jitclass

/home/ubuntu/anaconda3/lib/python3.7/site-packages/numba/np/ufunc/parallel.py:
 355:
NumbaWarning: The TBB threading layer requires TBB version 2019.5 or later i.e.,
TBB_INTERFACE_VERSION >= 11005. Found TBB_INTERFACE_VERSION = 11004. The TBB
threading
layer is disabled.
warnings.warn(problem)
```

17.3 Lorenz Curves and the Gini Coefficient

Before we investigate wealth dynamics, we briefly review some measures of inequality.

17.3.1 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

```
In [3]: 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:

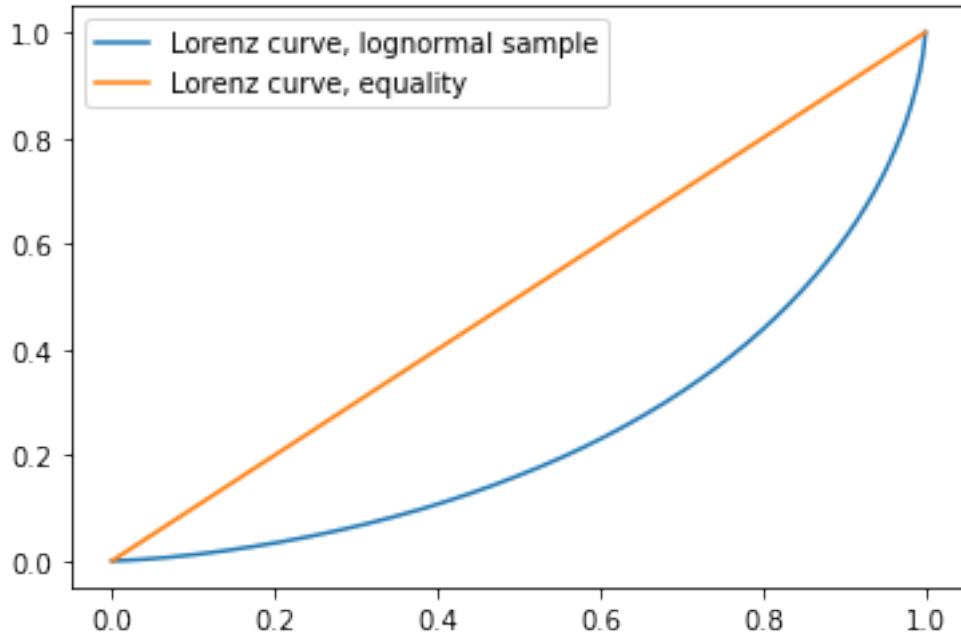
```
In [4]: f_vals, l_vals = qe.lorenz_curve(w)

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.

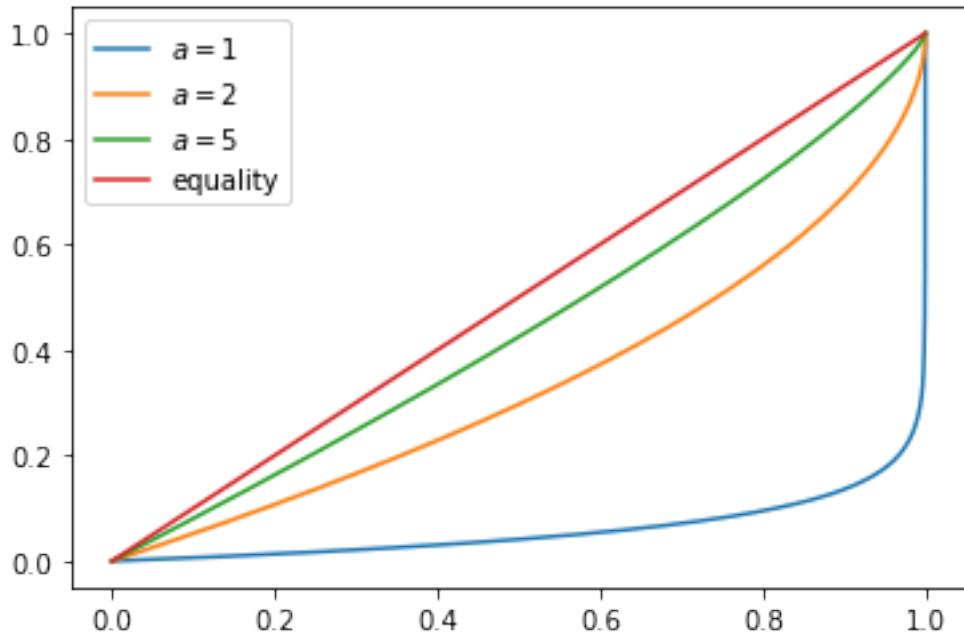
Here is another example, which shows how the Lorenz curve shifts as the underlying distribution changes.

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.

```

In [5]: 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.

17.3.2 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 .

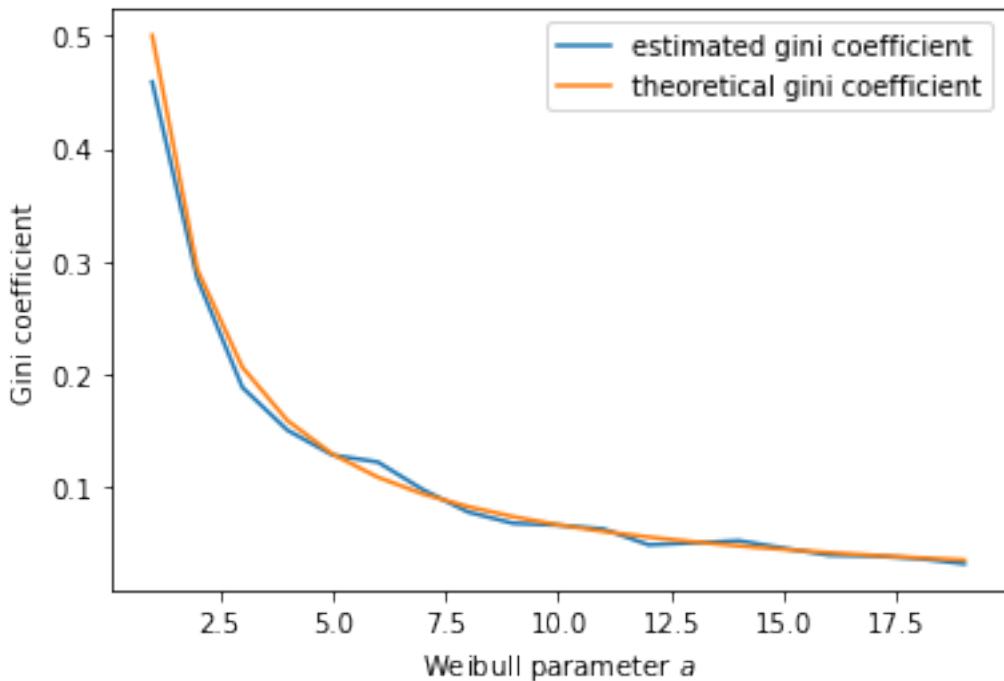
```
In [6]: a_vals = range(1, 20)
ginis = []
ginis_theoretical = []
n = 100

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')
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.

17.4 A Model of Wealth Dynamics

Having discussed inequality measures, let us now turn to wealth dynamics.

The model we will study is

$$w_{t+1} = (1 + r_{t+1})s(w_t) + y_{t+1} \quad (1)$$

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

$$z_{t+1} = az_t + b + \sigma_z \epsilon_{t+1}$$

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

$$s(w) = s_0 w \cdot \mathbb{1}\{w \geq \hat{w}\} \quad (2)$$

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.

17.5 Implementation

Here's some type information to help Numba.

```
In [7]: wealth_dynamics_data = [
    ('w_hat', float64),      # savings parameter
    ('s_0', float64),        # savings parameter
    ('c_y', float64),        # labor income parameter
    ('mu_y', float64),       # labor income parameter
    ('sigma_y', float64),    # labor income parameter
    ('c_r', float64),        # rate of return parameter
    ('mu_r', float64),       # rate of return parameter
    ('sigma_r', float64),    # rate of return parameter
    ('a', float64),          # aggregate shock parameter
    ('b', float64),          # aggregate shock parameter
    ('sigma_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.

```
In [8]: @jitclass(wealth_dynamics_data)
class WealthDynamics:

    def __init__(self,
                 w_hat=1.0,
                 s_0=0.75,
                 c_y=1.0,
                 mu_y=1.0,
                 sigma_y=0.2,
                 c_r=0.05,
                 mu_r=0.1,
                 sigma_r=0.5,
                 a=0.5,
                 b=0.0,
                 sigma_z=0.1):

        self.w_hat, self.s_0 = w_hat, s_0
        self.c_y, self.mu_y, self.sigma_y = c_y, mu_y, sigma_y
        self.c_r, self.mu_r, self.sigma_r = c_r, mu_r, sigma_r
        self.a, self.b, self.sigma_z = a, b, sigma_z

        # Record stationary moments
        self.z_mean = b / (1 - a)
        self.z_var = sigma_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(mu_r + sigma_r**2 / 2)
        self.y_mean = c_y * exp_z_mean + np.exp(mu_y + sigma_y**2 / 2)

        # Test a stability condition that ensures wealth does not diverge
        # to infinity.
        alpha = self.R_mean * self.s_0
        if alpha >= 1:
            raise ValueError("Stability condition failed.")

    def parameters(self):
        """
        Collect and return parameters.
        """
        parameters = (self.w_hat, self.s_0,
                      self.c_y, self.mu_y, self.sigma_y,
                      self.c_r, self.mu_r, self.sigma_r,
                      self.a, self.b, self.sigma_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, mu_y, sigma_y, c_r, mu_r, sigma_r, a, b, sigma_z = params
        zp = a * z + b + sigma_z * np.random.randn()

        # Update wealth
        y = c_y * np.exp(zp) + np.exp(mu_y + sigma_y * np.random.randn())
        wp = y
```

```

if w >= w_hat:
    R = c_r * np.exp(zp) + np.exp(mu_r + sigma_r * np.random.randn())
    wp += R * s_0 * w
return wp, zp

```

Here's function to simulate the time series of wealth for individual households.

In [9]: `@njit`

```

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.

In [10]: `@njit(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.

17.6 Applications

Let's try simulating the model at different parameter values and investigate the implications for the wealth distribution.

17.6.1 Time Series

Let's look at the wealth dynamics of an individual household.

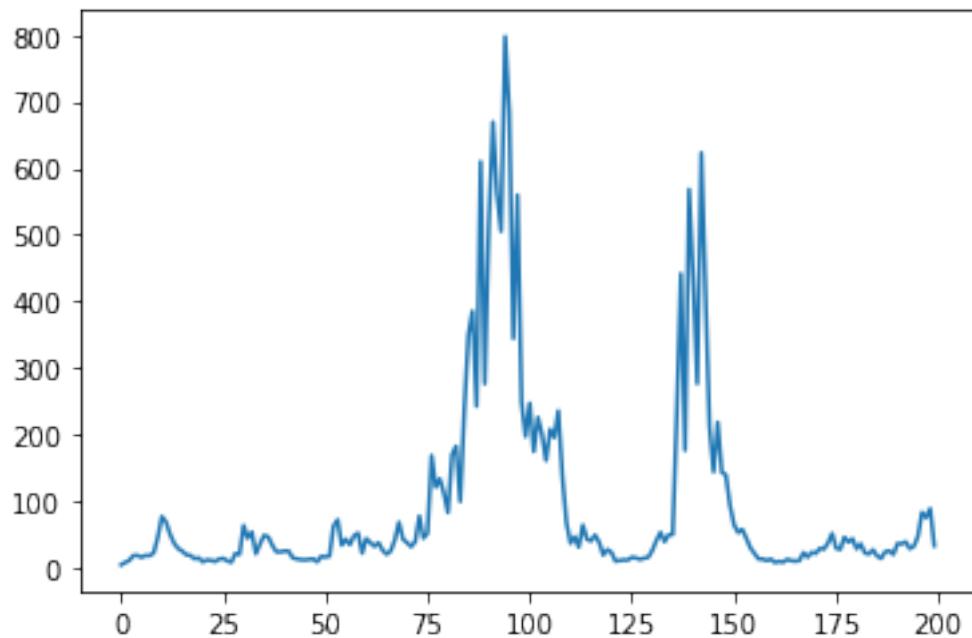
In [11]: `wdy = WealthDynamics()`

```

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.

17.6.2 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.

```
In [12]: 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.ones(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.

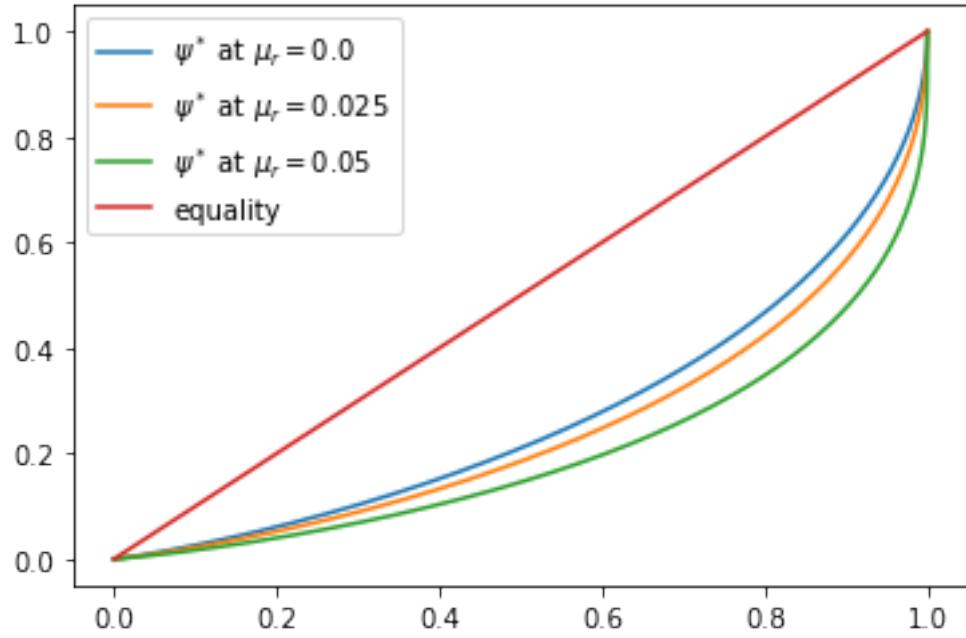
This is unavoidable because we are executing a CPU intensive task.

In fact the code, which is JIT compiled and parallelized, runs extremely fast relative to the number of computations.

```
In [13]: fig, ax = plt.subplots()
μ_r_vals = (0.0, 0.025, 0.05)
gini_vals = []

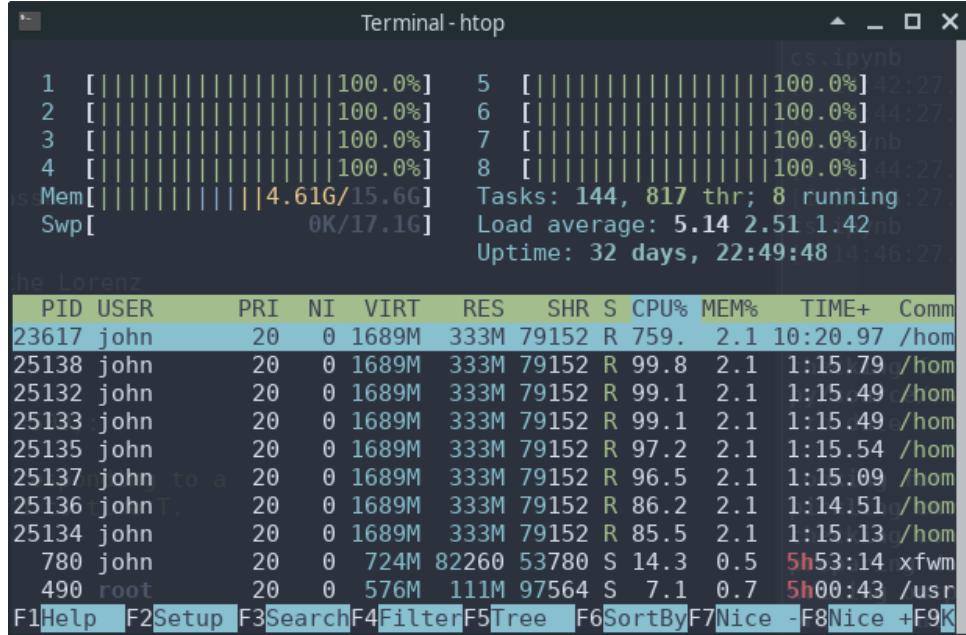
for μ_r in μ_r_vals:
    wdy = WealthDynamics(μ_r=μ_r)
    gv, (f_vals, l_vals) = generate_lorenz_and_gini(wdy)
    ax.plot(f_vals, l_vals, label=f'$\psi^*$ at $\mu_r = {μ_r:0.2}$')
    gini_vals.append(gv)

ax.plot(f_vals, f_vals, label='equality')
ax.legend(loc="upper left")
plt.show()
```



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:

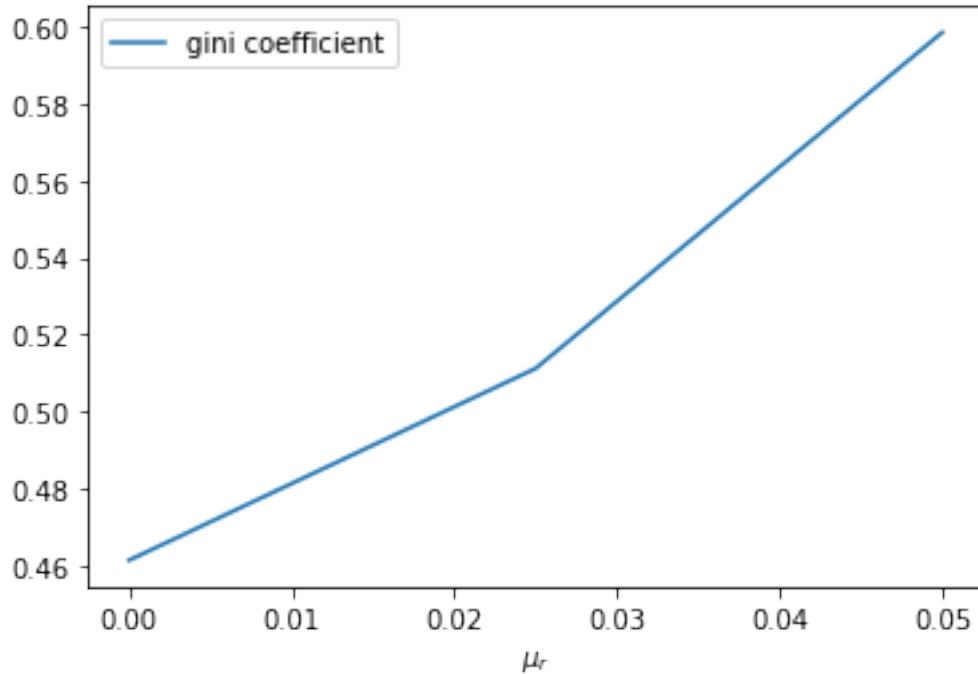


Notice how effectively Numba has implemented multithreading for this routine: all 8 CPUs on our workstation are running at maximum capacity (even though four of them are virtual).

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.

```
In [14]: fig, ax = plt.subplots()
ax.plot(mu_r_vals, gini_vals, label='gini coefficient')
ax.set_xlabel("$\mu_r$")
ax.legend()
plt.show()
```



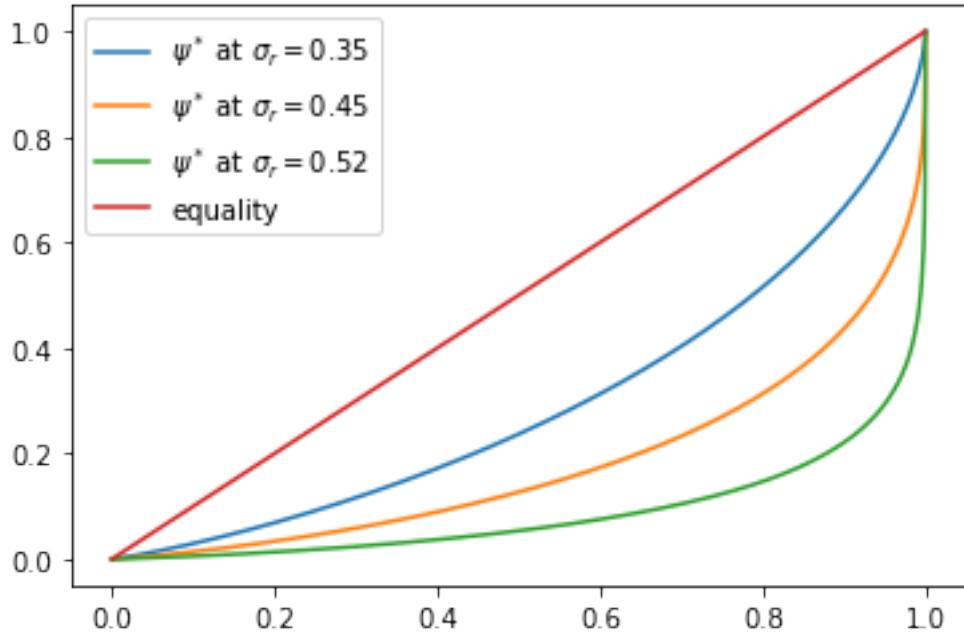
Once again, we see that inequality increases as returns on financial income rise.

Let's finish this section by investigating what happens when we change the volatility term σ_r in financial returns.

```
In [15]: fig, ax = plt.subplots()
sigma_r_vals = (0.35, 0.45, 0.52)
gini_vals = []

for sigma_r in sigma_r_vals:
    wdy = WealthDynamics(sigma_r=sigma_r)
    gv, (f_vals, l_vals) = generate_lorenz_and_gini(wdy)
    ax.plot(f_vals, l_vals, label=f'$\psi^*$ at $\sigma_r = {sigma_r}$')
    gini_vals.append(gv)

ax.plot(f_vals, f_vals, label='equality')
ax.legend(loc="upper left")
plt.show()
```



We see that greater volatility has the effect of increasing inequality in this model.

17.7 Exercises

17.7.1 Exercise 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 .

17.7.2 Exercise 2

The wealth process (1) is similar to a [Kesten process](#).

This is because, according to (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 (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

```
In [16]: num_households = 250_000
T = 500                                     # shift forward T periods
ψ_0 = np.ones(num_households) * wdy.y_mean   # initial distribution
z_0 = wdy.z_mean
```

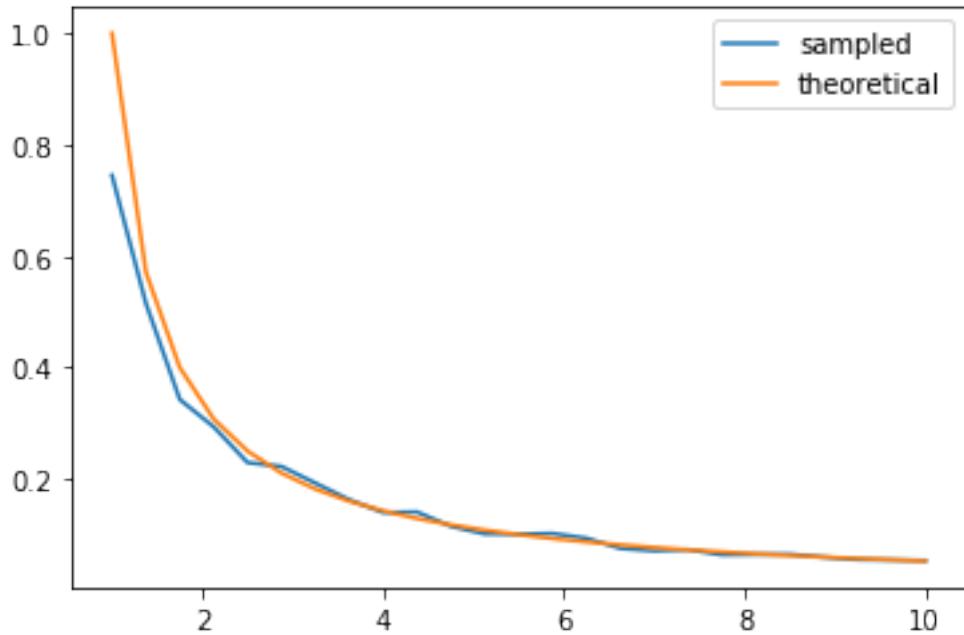
17.8 Solutions

Here is one solution, which produces a good match between theory and simulation.

17.8.1 Exercise 1

```
In [17]: 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.

17.8.2 Exercise 2

First let's generate the distribution:

```
In [18]: num_households = 250_000
T = 500 # how far to shift forward in time
ψ_θ = np.ones(num_households) * wdy.y_mean
z_θ = wdy.z_mean

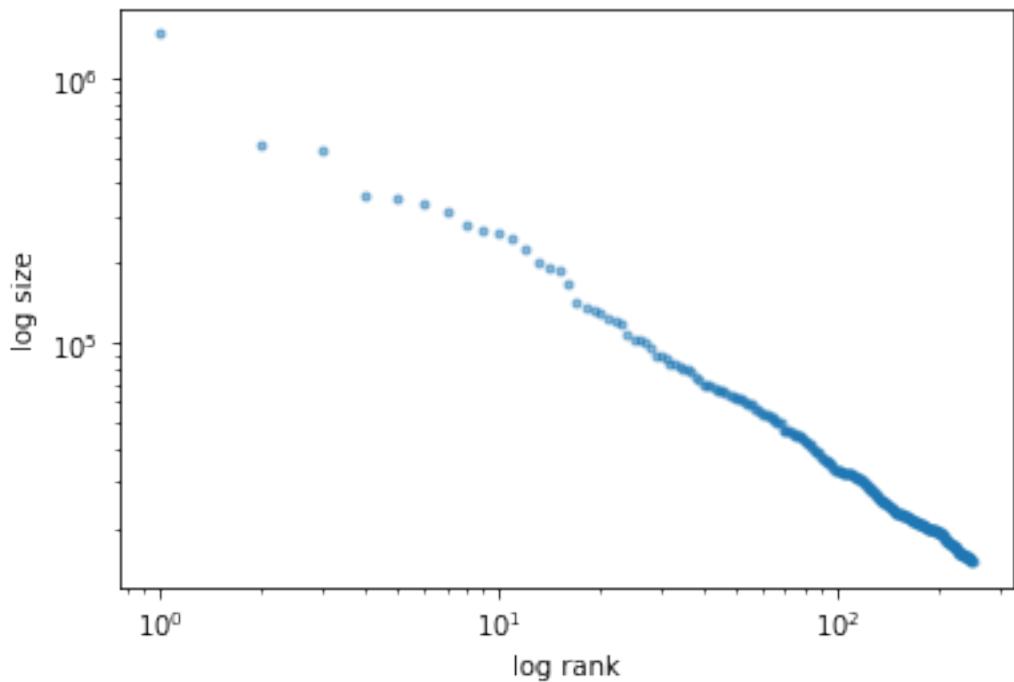
ψ_star = update_cross_section(wdy, ψ_θ, shift_length=T)
```

Now let's see the rank-size plot:

```
In [19]: 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()
```



Chapter 18

A First Look at the Kalman Filter

18.1 Contents

- Overview 18.2
- The Basic Idea 18.3
- Convergence 18.4
- Implementation 18.5
- Exercises 18.6
- Solutions 18.7

In addition to what's in Anaconda, this lecture will need the following libraries:

```
In [1]: !pip install quantecon
```

18.2 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

- [72], section 2.7
- [5]

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:

```
In [2]: from scipy import linalg
import numpy as np
import matplotlib.cm as cm
import matplotlib.pyplot as plt
%matplotlib inline
from quantecon import Kalman, LinearStateSpace
```

```

from scipy.stats import norm
from scipy.integrate import quad
from numpy.random import multivariate_normal
from scipy.linalg import eigvals

/home/ubuntu/anaconda3/lib/python3.7/site-packages/numba/np/ufunc/parallel.py:
  ↳355:
NumbaWarning: The TBB threading layer requires TBB version 2019.5 or later i.e.,
TBB_INTERFACE_VERSION >= 11005. Found TBB_INTERFACE_VERSION = 11004. The TBB
  ↳threading
layer is disabled.
warnings.warn(problem)

```

18.3 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

$$p = N(\hat{x}, \Sigma) \tag{1}$$

where \hat{x} is the mean of the distribution and Σ is a 2×2 covariance matrix. In our simulations, we will suppose that

$$\hat{x} = \begin{pmatrix} 0.2 \\ -0.2 \end{pmatrix}, \quad \Sigma = \begin{pmatrix} 0.4 & 0.3 \\ 0.3 & 0.45 \end{pmatrix} \tag{2}$$

This density $p(x)$ is shown below as a contour map, with the center of the red ellipse being equal to \hat{x} .

```
In [3]: # 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  $y = G x + N(\theta, R)$ 
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):
    """
    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  $N(\mu, C)$ "
    m_x, m_y = float(μ[0]), float(μ[1])

```

```

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)

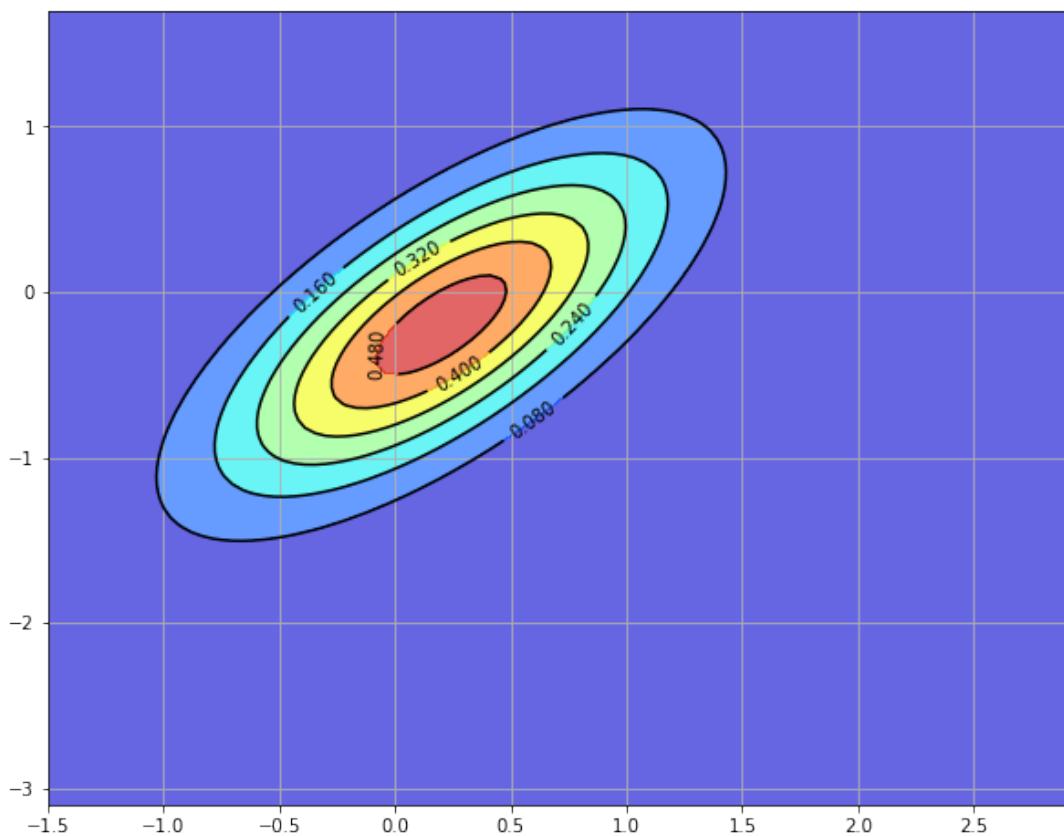
# Plot the figure

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()

```



18.3.1 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

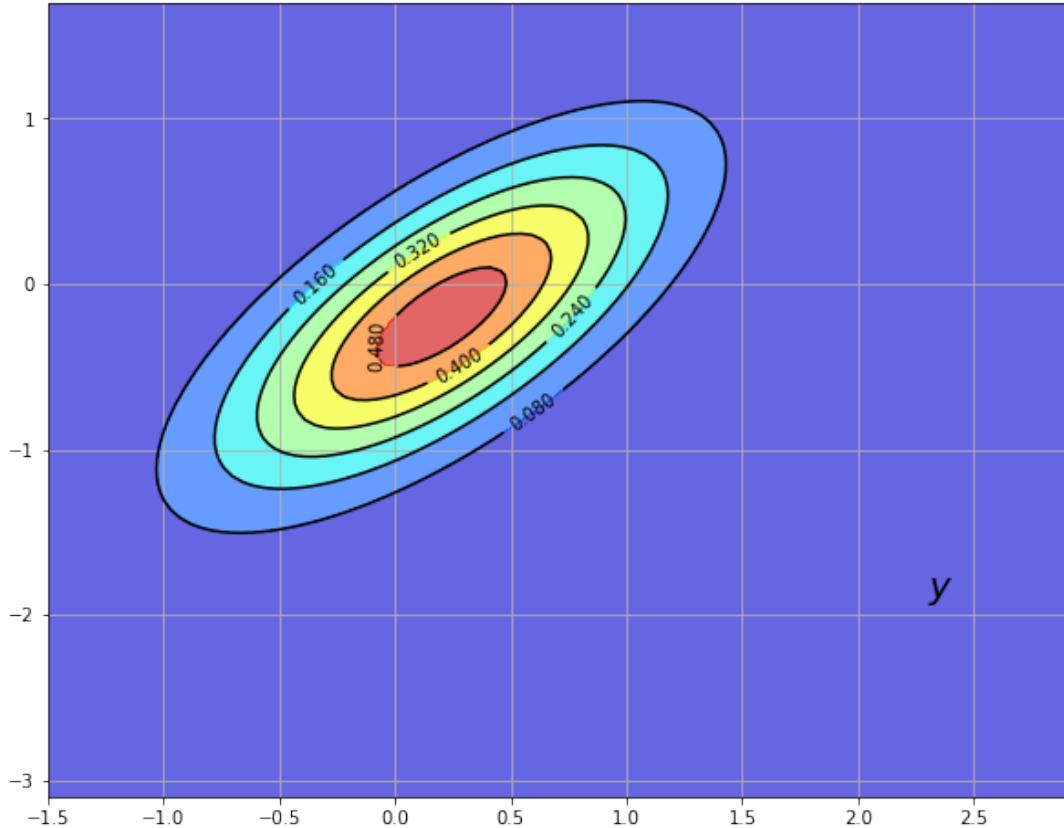
```
In [4]: fig, ax = plt.subplots(figsize=(10, 8))
ax.grid()
```

```

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]), float(y[1]), "$y$", 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

$$y = Gx + v, \quad \text{where } v \sim N(0, R) \quad (3)$$

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 (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 Section ?? to be

$$p(x | y) = N(\hat{x}^F, \Sigma^F)$$

where

$$\hat{x}^F := \hat{x} + \Sigma G' (G \Sigma G' + R)^{-1} (y - G\hat{x}) \quad \text{and} \quad \Sigma^F := \Sigma - \Sigma G' (G \Sigma G' + R)^{-1} G \Sigma \quad (4)$$

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 $y - G\hat{x}$.

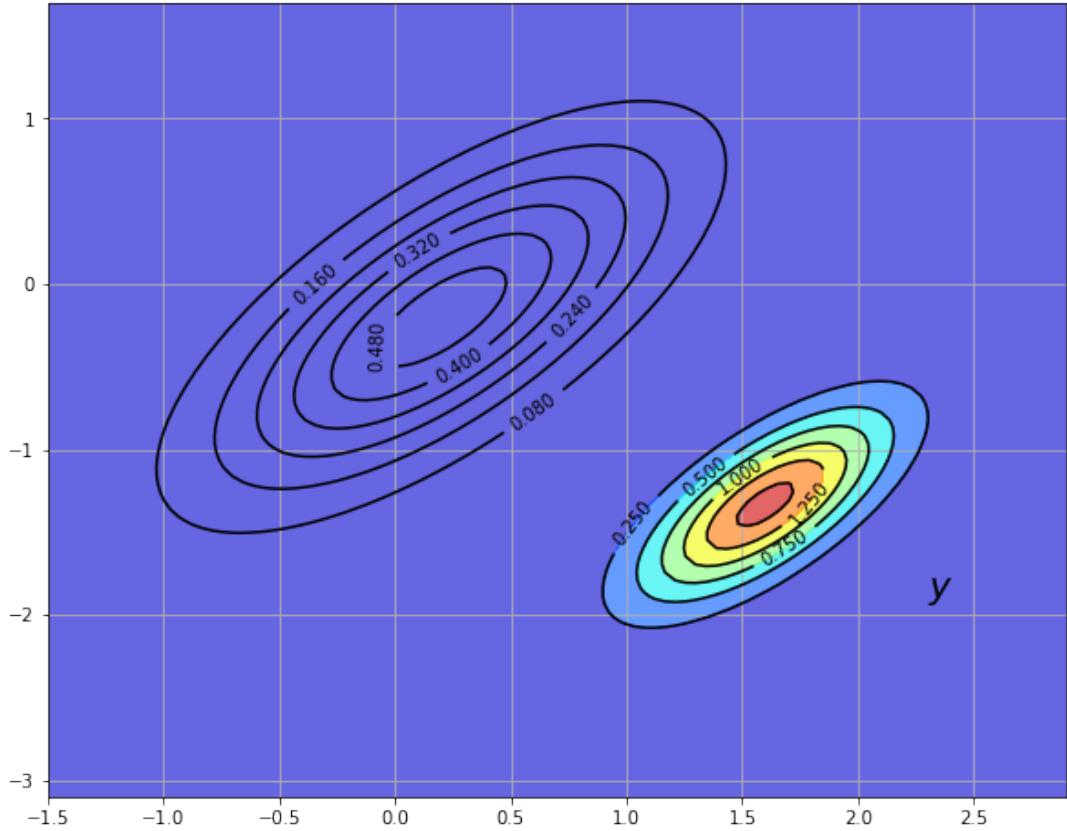
This new density $p(x | y) = N(\hat{x}^F, \Sigma^F)$ is shown in the next figure via contour lines and the color map.

The original density is left in as contour lines for comparison

```
In [5]: fig, ax = plt.subplots(figsize=(10, 8))
ax.grid()

Z = gen_gaussian_plot_vals(x_hat, Sigma)
cs1 = ax.contour(X, Y, Z, 6, colors="black")
ax.clabel(cs1, inline=1, fontsize=10)
M = Sigma * G.T * linalg.inv(G * Sigma * G.T + R)
x_hat_F = x_hat + M * (y - G * x_hat)
Sigma_F = Sigma - M * G * Sigma
new_Z = gen_gaussian_plot_vals(x_hat_F, Sigma_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]), float(y[1]), "$y$", fontsize=20, color="black")

plt.show()
```



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 (2).

18.3.2 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.

- $p(x|y) = N(\hat{x}^F, \Sigma^F)$ 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,

$$x_{t+1} = Ax_t + w_{t+1}, \quad \text{where } w_t \sim N(0, Q) \quad (5)$$

Our aim is to combine this law of motion and our current distribution $p(x|y) = N(\hat{x}^F, \Sigma^F)$ to come up with a new *predictive* distribution for the location in one unit of time.

In view of (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 (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

$$\text{Var}[Ax^F + w] = A \text{Var}[x^F]A' + Q = A\Sigma^F A' + Q = A\Sigma A' - A\Sigma G'(G\Sigma G' + R)^{-1}G\Sigma A' + Q$$

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

$$\begin{aligned}\hat{x}_{new} &:= A\hat{x} + K_\Sigma(y - G\hat{x}) \\ \Sigma_{new} &:= A\Sigma A' - K_\Sigma G\Sigma A' + Q\end{aligned}\tag{6}$$

- 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.

$$A = \begin{pmatrix} 1.2 & 0.0 \\ 0.0 & -0.2 \end{pmatrix}, \quad Q = 0.3 * \Sigma$$

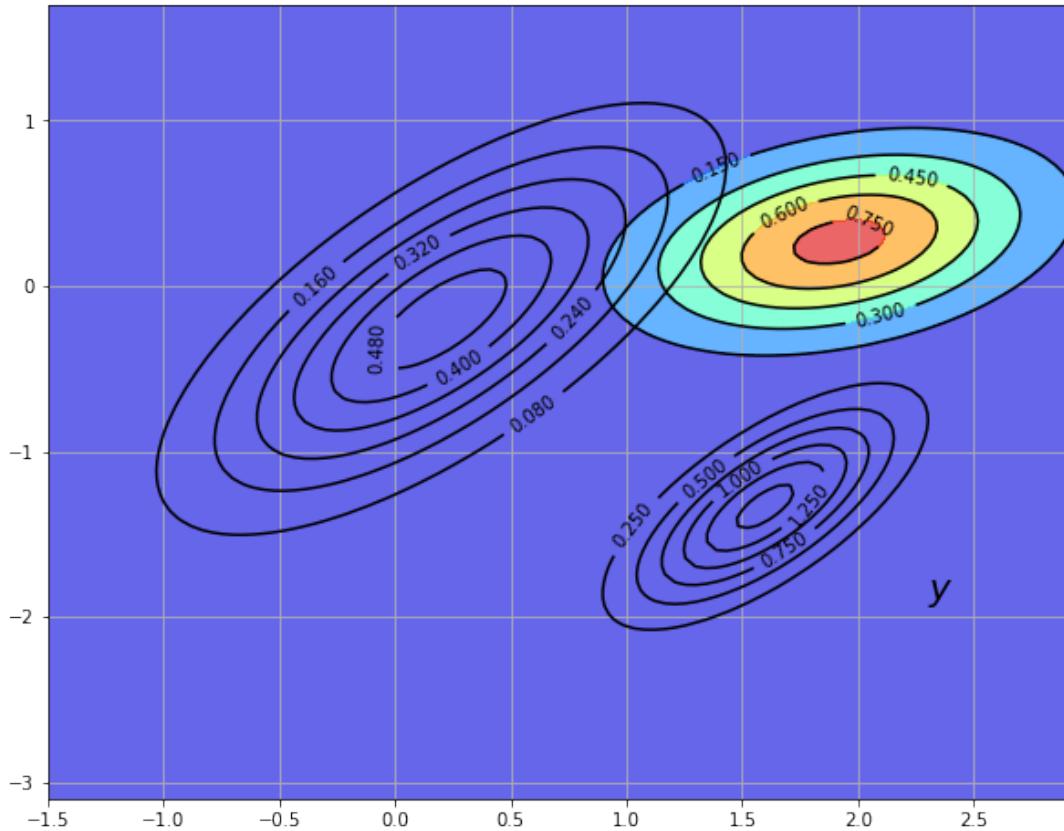
```
In [6]: fig, ax = plt.subplots(figsize=(10, 8))
ax.grid()

# Density 1
Z = gen_gaussian_plot_vals(x_hat, Sigma)
cs1 = ax.contour(X, Y, Z, 6, colors="black")
ax.clabel(cs1, inline=1, fontsize=10)

# Density 2
M = Sigma * G.T * linalg.inv(G * Sigma * G.T + R)
x_hat_F = x_hat + M * (y - G * x_hat)
Sigma_F = Sigma - M * G * Sigma
Z_F = gen_gaussian_plot_vals(x_hat_F, Sigma_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_Sigma = A * Sigma_F * A.T + Q
new_Z = gen_gaussian_plot_vals(new_x_hat, new_Sigma)
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]), float(y[1]), "$y$", fontsize=20, color="black")
plt.show()
```



18.3.3 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 $p(x | y)$.

Finally, we used the law of motion (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 $p_t(x | y) = N(\hat{x}_t^F, \Sigma_t^F)$ from $p_t(x)$ and y_t , applying Bayes rule and the conditional distribution (3).
4. Compute the predictive distribution $p_{t+1}(x) = N(\hat{x}_{t+1}, \Sigma_{t+1})$ from the filtering distribution and (5).

5. Increment t by one and go to step 1.

Repeating (6), the dynamics for \hat{x}_t and Σ_t are as follows

$$\begin{aligned}\hat{x}_{t+1} &= A\hat{x}_t + K_{\Sigma_t}(y_t - G\hat{x}_t) \\ \Sigma_{t+1} &= A\Sigma_t A' - K_{\Sigma_t} G\Sigma_t A' + Q\end{aligned}\tag{7}$$

These are the standard dynamic equations for the Kalman filter (see, for example, [72], page 58).

18.4 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 (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 $t \rightarrow \infty$.

To study this topic, let's expand the second equation in (7):

$$\Sigma_{t+1} = A\Sigma_t A' - A\Sigma_t G'(G\Sigma_t G' + R)^{-1}G\Sigma_t A' + Q\tag{8}$$

This is a nonlinear difference equation in Σ_t .

A fixed point of (8) is a constant matrix Σ such that

$$\Sigma = A\Sigma A' - A\Sigma G'(G\Sigma G' + R)^{-1}G\Sigma A' + Q\tag{9}$$

Equation (8) is known as a discrete-time Riccati difference equation.

Equation (9) 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 [6] and [5], chapter 4.

A sufficient (but not necessary) condition is that all the eigenvalues λ_i of A satisfy $|\lambda_i| < 1$ (cf. e.g., [5], p. 77).

(This strong condition assures that the unconditional distribution of x_t converges as $t \rightarrow +\infty$.)

In this case, for any initial choice of Σ_0 that is both non-negative and symmetric, the sequence $\{\Sigma_t\}$ in (8) converges to a non-negative symmetric matrix Σ that solves (9).

18.5 Implementation

The class `Kalman` from the `QuantEcon.py` package implements the Kalman filter

- Instance data consists of:
 - the moments (\hat{x}_t, Σ_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, Σ_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 (9) and the corresponding (stationary) Kalman gain

You can view the program [on GitHub](#).

18.6 Exercises

18.6.1 Exercise 1

Consider the following simple application of the Kalman filter, loosely based on [72], 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 (5) with $A = 1$, $Q = 0$ and $x_0 = \theta$.

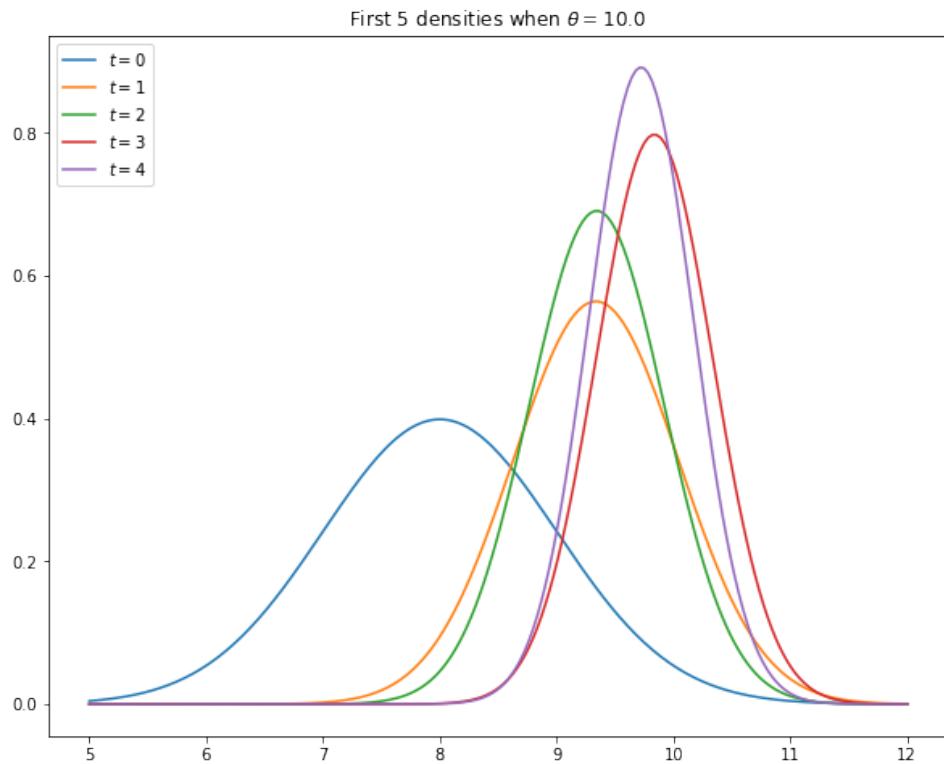
The measurement equation is $y_t = \theta + v_t$ where v_t is $N(0, 1)$ and IID.

The task of this exercise is 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 [72], 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



18.6.2 Exercise 2

The preceding figure gives some support to the idea that probability mass converges to θ .

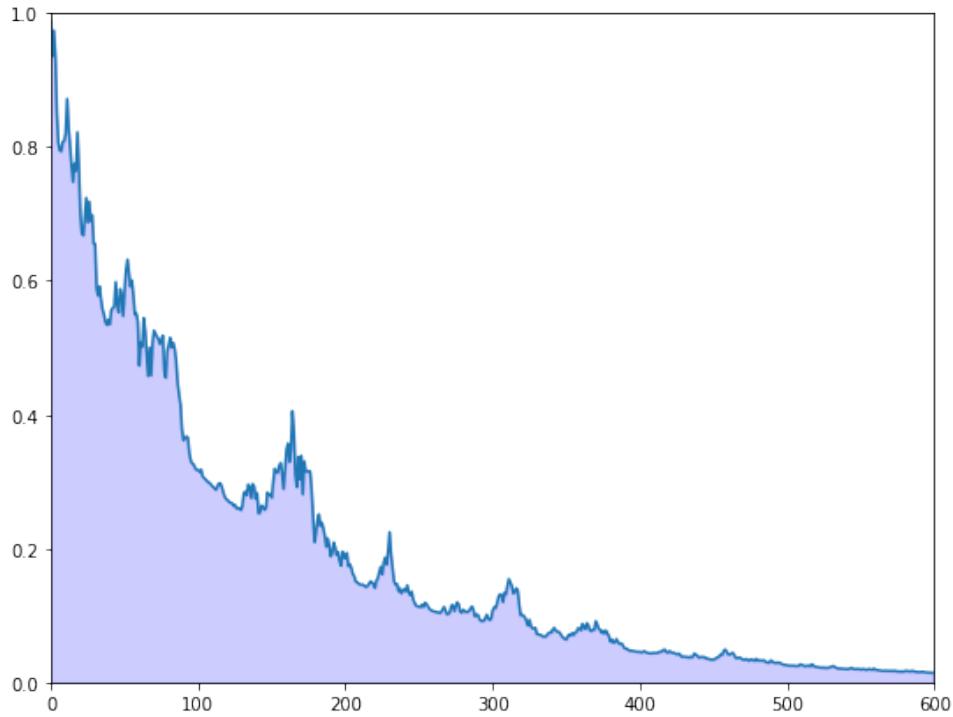
To get a better idea, choose a small $\epsilon > 0$ and calculate

$$z_t := 1 - \int_{\theta-\epsilon}^{\theta+\epsilon} p_t(x) dx$$

for $t = 0, 1, 2, \dots, T$.

Plot z_t against T , setting $\epsilon = 0.1$ and $T = 600$.

Your figure should show error erratically declining something like this



18.6.3 Exercise 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 $\mathbb{E}[x_t | x_{t-1}]$, 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 $g^*(x_{t-1}) := \mathbb{E}[x_t | x_{t-1}]$)

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 $t = 1, \dots, 50$.

For the parameters, set $G = I$, $R = 0.5I$ and $Q = 0.3I$, where I is the 2×2 identity.

Set

$$A = \begin{pmatrix} 0.5 & 0.4 \\ 0.6 & 0.3 \end{pmatrix}$$

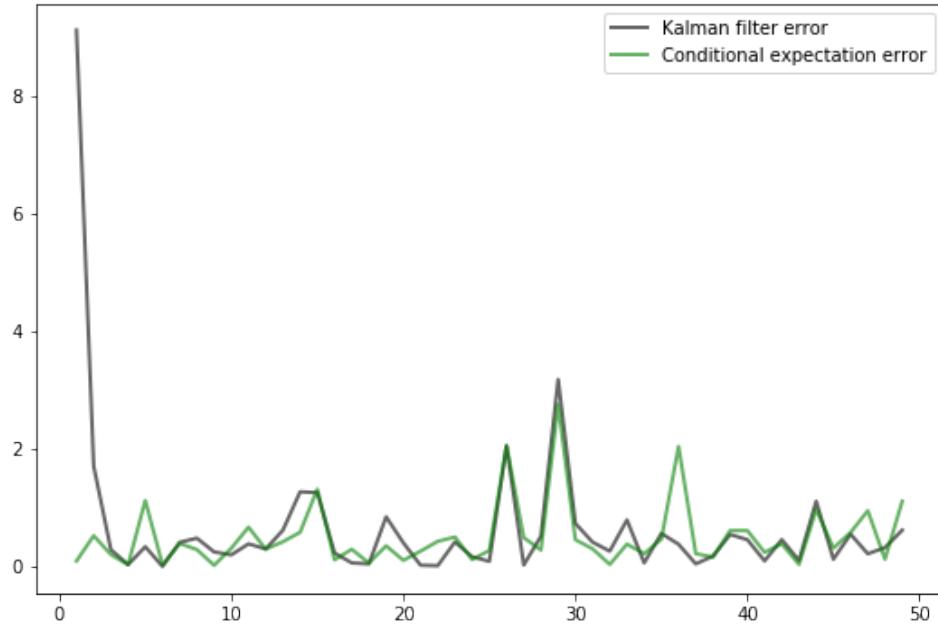
To initialize the prior density, set

$$\Sigma_0 = \begin{pmatrix} 0.9 & 0.3 \\ 0.3 & 0.9 \end{pmatrix}$$

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.

18.6.4 Exercise 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 (9)) 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.

18.7 Solutions

18.7.1 Exercise 1

```
In [7]: # Parameters
θ = 10 # Constant value of state x_t
A, C, G, H = 1, θ, 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)

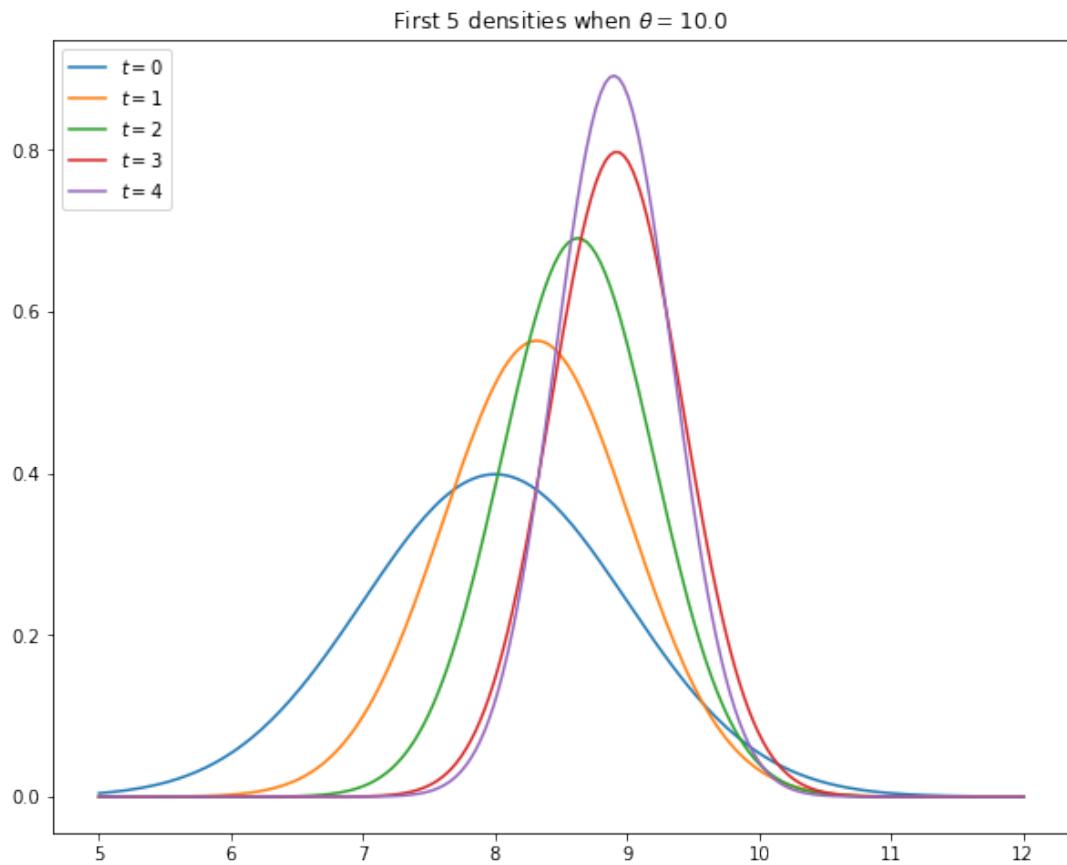
# Draw observations of y from state space model
N = 5
x, y = ss.simulate(N)
y = y.flatten()

# 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, kalman.Sigma)]
    # Plot, update filter
    ax.plot(xgrid, norm.pdf(xgrid, loc=m, scale=np.sqrt(v)), color='purple')
    kalman.update(y[i])

ax.set_title(f'First {N} densities when $\theta = {θ:.1f}$')
ax.legend(loc='upper left')
plt.show()

```



18.7.2 Exercise 2

```
In [8]: 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)

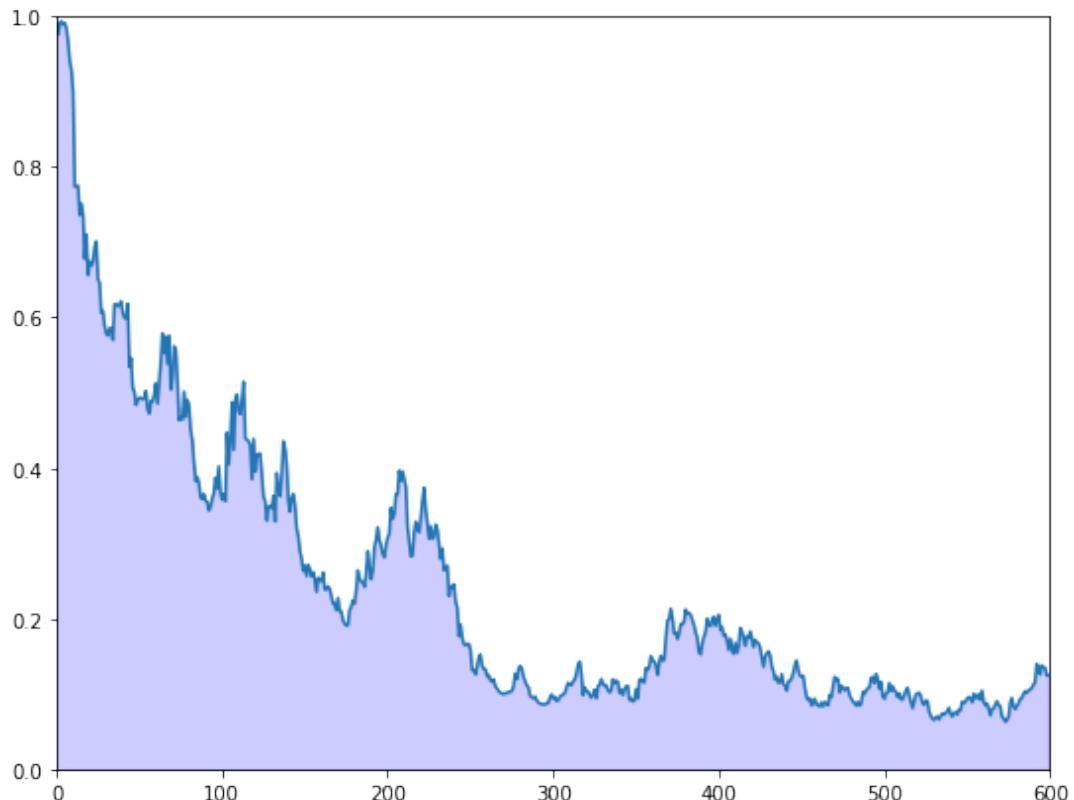
T = 600
z = np.empty(T)
x, y = ss.simulate(T)
y = y.flatten()

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, kalman.Sigma)]

    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_xlim(0, T)
ax.set_ylim(0, 1)
ax.plot(range(T), z)
ax.fill_between(range(T), np.zeros(T), z, color="blue", alpha=0.2)
plt.show()
```



18.7.3 Exercise 3

```
In [9]: # 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)

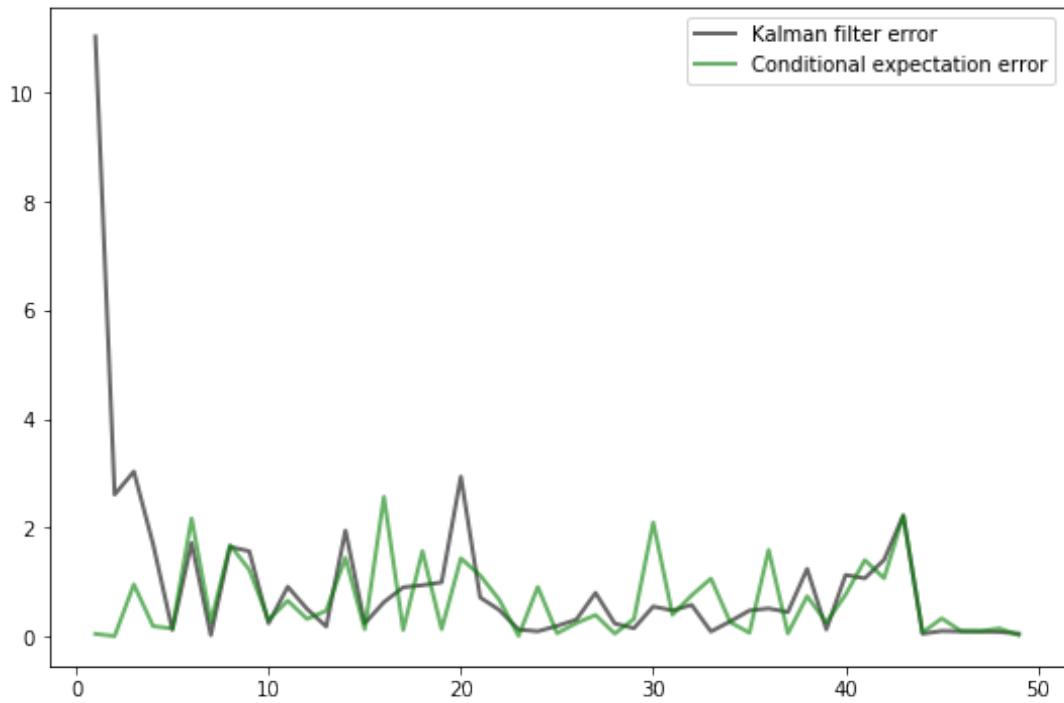
e1 = np.empty(T-1)
e2 = np.empty(T-1)

for t in range(1, T):
    kn.update(y[:, t])
    e1[t-1] = np.sum((x[:, t] - kn.x_hat.flatten())**2)
    e2[t-1] = np.sum((x[:, t] - A @ x[:, t-1])**2)

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]]



Footnotes

[1] See, for example, page 93 of [16]. To get from his expressions to the ones used above, you will also need to apply the [Woodbury matrix identity](#).

Chapter 19

Shortest Paths

19.1 Contents

- Overview 19.2
- Outline of the Problem 19.3
- Finding Least-Cost Paths 19.4
- Solving for Minimum Cost-to-Go 19.5
- Exercises 19.6
- Solutions 19.7

19.2 Overview

The shortest path problem is a [classic problem](#) in mathematics and computer science with applications in

- Economics (sequential decision making, analysis of social networks, etc.)
- Operations research and transportation
- Robotics and artificial intelligence
- Telecommunication network design and routing
- etc., etc.

Variations of the methods we discuss in this lecture are used millions of times every day, in applications such as

- Google Maps
- routing packets on the internet

For us, the shortest path problem also provides a nice introduction to the logic of **dynamic programming**.

Dynamic programming is an extremely powerful optimization technique that we apply in many lectures on this site.

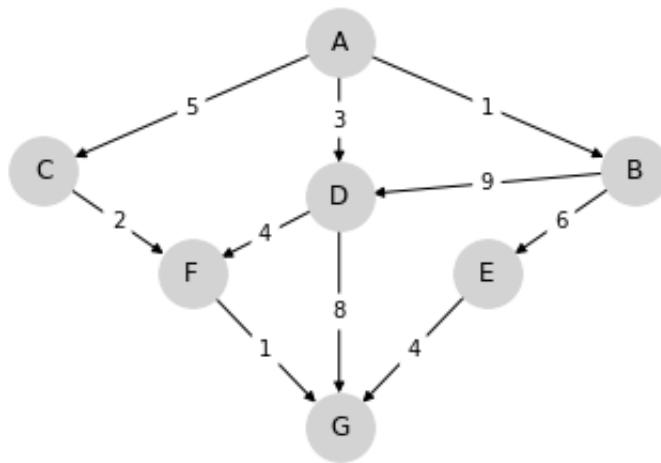
The only scientific library we'll need in what follows is NumPy:

```
In [1]: import numpy as np
```

19.3 Outline of the Problem

The shortest path problem is one of finding how to traverse a [graph](#) from one specified node to another at minimum cost.

Consider the following graph



We wish to travel from node (vertex) A to node G at minimum cost

- Arrows (edges) indicate the movements we can take.
- Numbers on edges indicate the cost of traveling that edge.

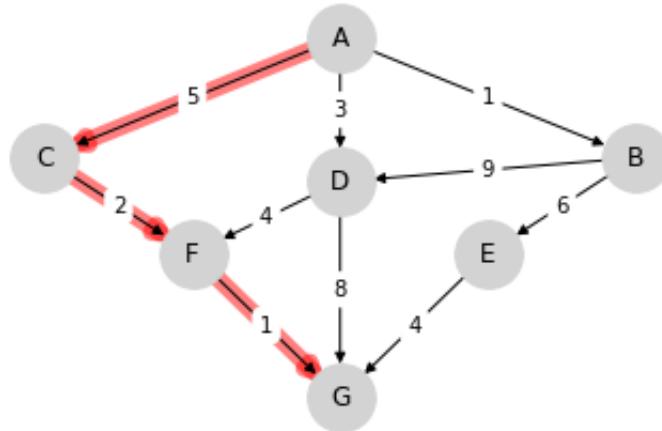
(Graphs such as the one above are called [weighted directed graphs](#).)

Possible interpretations of the graph include

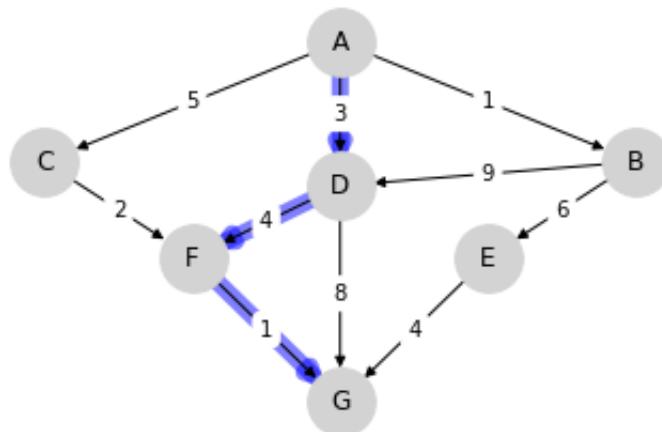
- Minimum cost for supplier to reach a destination.
- Routing of packets on the internet (minimize time).
- Etc., etc.

For this simple graph, a quick scan of the edges shows that the optimal paths are

- A, C, F, G at cost 8



- A, D, F, G at cost 8

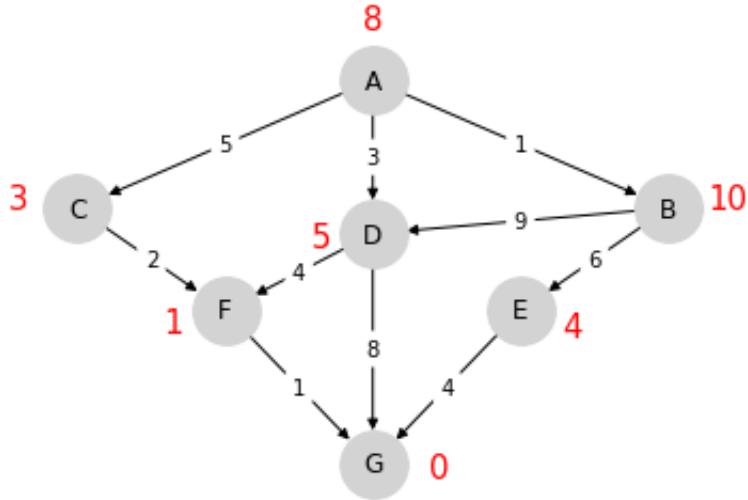


19.4 Finding Least-Cost Paths

For large graphs, we need a systematic solution.

Let $J(v)$ denote the minimum cost-to-go from node v , understood as the total cost from v if we take the best route.

Suppose that we know $J(v)$ for each node v , as shown below for the graph from the preceding example



Note that $J(G) = 0$.

The best path can now be found as follows

1. Start at node $v = A$
2. From current node v , move to any node that solves

$$\min_{w \in F_v} \{c(v, w) + J(w)\} \quad (1)$$

where

- F_v is the set of nodes that can be reached from v in one step.
- $c(v, w)$ is the cost of traveling from v to w .

Hence, if we know the function J , then finding the best path is almost trivial.

But how can we find the cost-to-go function J ?

Some thought will convince you that, for every node v , the function J satisfies

$$J(v) = \min_{w \in F_v} \{c(v, w) + J(w)\} \quad (2)$$

This is known as the *Bellman equation*, after the mathematician Richard Bellman.

The Bellman equation can be thought of as a restriction that J must satisfy.

What we want to do now is use this restriction to compute J .

19.5 Solving for Minimum Cost-to-Go

Let's look at an algorithm for computing J and then think about how to implement it.

19.5.1 The Algorithm

The standard algorithm for finding J is to start an initial guess and then iterate.

This is a standard approach to solving nonlinear equations, often called the method of **successive approximations**.

Our initial guess will be

$$J_0(v) = 0 \text{ for all } v \quad (3)$$

Now

1. Set $n = 0$
2. Set $J_{n+1}(v) = \min_{w \in F_v} \{c(v, w) + J_n(w)\}$ for all v
3. If J_{n+1} and J_n are not equal then increment n , go to 2

This sequence converges to J .

Although we omit the proof, we'll prove similar claims in our other lectures on dynamic programming.

19.5.2 Implementation

Having an algorithm is a good start, but we also need to think about how to implement it on a computer.

First, for the cost function c , we'll implement it as a matrix Q , where a typical element is

$$Q(v, w) = \begin{cases} c(v, w) & \text{if } w \in F_v \\ +\infty & \text{otherwise} \end{cases}$$

In this context Q is usually called the **distance matrix**.

We're also numbering the nodes now, with $A = 0$, so, for example

$$Q(1, 2) = \text{the cost of traveling from B to C}$$

For example, for the simple graph above, we set

In [2]: `from numpy import inf`

```
Q = np.array([[inf, 1, 5, 3, inf, inf, inf],
              [inf, inf, inf, 9, 6, inf, inf],
              [inf, inf, inf, inf, inf, 2, inf],
              [inf, inf, inf, inf, inf, 4, 8],
              [inf, inf, inf, inf, inf, inf, 4],
              [inf, inf, inf, inf, inf, inf, 1],
              [inf, inf, inf, inf, inf, inf, 0]])
```

Notice that the cost of staying still (on the principle diagonal) is set to

- np.inf for non-destination nodes — moving on is required.
- 0 for the destination node — here is where we stop.

For the sequence of approximations $\{J_n\}$ of the cost-to-go functions, we can use NumPy arrays.

Let's try with this example and see how we go:

```
In [3]: nodes = range(7)                      # Nodes = 0, 1, ..., 6
J = np.zeros_like(nodes, dtype=np.int)          # Initial guess
next_J = np.empty_like(nodes, dtype=np.int)      # Stores updated guess

max_iter = 500
i = 0

while i < max_iter:
    for v in nodes:
        # minimize  $Q[v, w] + J[w]$  over all choices of  $w$ 
        lowest_cost = inf
        for w in nodes:
            cost = Q[v, w] + J[w]
            if cost < lowest_cost:
                lowest_cost = cost
        next_J[v] = lowest_cost
    if np.equal(next_J, J).all():
        break
    else:
        J[:] = next_J    # Copy contents of next_J to J
        i += 1

print("The cost-to-go function is", J)
```

The cost-to-go function is [8 10 3 5 4 1 0]

This matches with the numbers we obtained by inspection above.

But, importantly, we now have a methodology for tackling large graphs.

19.6 Exercises

19.6.1 Exercise 1

The text below describes a weighted directed graph.

The line `node0, node1 0.04, node8 11.11, node14 72.21` means that from node0 we can go to

- node1 at cost 0.04
- node8 at cost 11.11
- node14 at cost 72.21

No other nodes can be reached directly from node0.

Other lines have a similar interpretation.

Your task is to use the algorithm given above to find the optimal path and its cost.

Note: You will be dealing with floating point numbers now, rather than integers, so consider replacing `np.equal()` with `np.allclose()`.

```
In [4]: %%file graph.txt
node0, node1 0.04, node8 11.11, node14 72.21
node1, node46 1247.25, node6 20.59, node13 64.94
node2, node66 54.18, node31 166.80, node45 1561.45
node3, node20 133.65, node6 2.06, node11 42.43
node4, node75 3706.67, node5 0.73, node7 1.02
node5, node45 1382.97, node7 3.33, node11 34.54
node6, node31 63.17, node9 0.72, node10 13.10
node7, node50 478.14, node9 3.15, node10 5.85
node8, node69 577.91, node11 7.45, node12 3.18
node9, node70 2454.28, node13 4.42, node20 16.53
node10, node89 5352.79, node12 1.87, node16 25.16
node11, node94 4961.32, node18 37.55, node20 65.08
node12, node84 3914.62, node24 34.32, node28 170.04
node13, node60 2135.95, node38 236.33, node40 475.33
node14, node67 1878.96, node16 2.70, node24 38.65
node15, node91 3597.11, node17 1.01, node18 2.57
node16, node36 392.92, node19 3.49, node38 278.71
node17, node76 783.29, node22 24.78, node23 26.45
node18, node91 3363.17, node23 16.23, node28 55.84
node19, node26 20.09, node20 0.24, node28 70.54
node20, node98 3523.33, node24 9.81, node33 145.80
node21, node56 626.04, node28 36.65, node31 27.06
node22, node72 1447.22, node39 136.32, node40 124.22
node23, node52 336.73, node26 2.66, node33 22.37
node24, node66 875.19, node26 1.80, node28 14.25
node25, node70 1343.63, node32 36.58, node35 45.55
node26, node47 135.78, node27 0.01, node42 122.00
node27, node65 480.55, node35 48.10, node43 246.24
node28, node82 2538.18, node34 21.79, node36 15.52
node29, node64 635.52, node32 4.22, node33 12.61
node30, node98 2616.03, node33 5.61, node35 13.95
node31, node98 3350.98, node36 20.44, node44 125.88
node32, node97 2613.92, node34 3.33, node35 1.46
node33, node81 1854.73, node41 3.23, node47 111.54
node34, node73 1075.38, node42 51.52, node48 129.45
node35, node52 17.57, node41 2.09, node50 78.81
node36, node71 1171.60, node54 101.08, node57 260.46
node37, node75 269.97, node38 0.36, node46 80.49
node38, node93 2767.85, node40 1.79, node42 8.78
node39, node50 39.88, node40 0.95, node41 1.34
node40, node75 548.68, node47 28.57, node54 53.46
node41, node53 18.23, node46 0.28, node54 162.24
node42, node59 141.86, node47 10.08, node72 437.49
node43, node98 2984.83, node54 95.06, node60 116.23
node44, node91 807.39, node46 1.56, node47 2.14
node45, node58 79.93, node47 3.68, node49 15.51
node46, node52 22.68, node57 27.50, node67 65.48
node47, node50 2.82, node56 49.31, node61 172.64
node48, node99 2564.12, node59 34.52, node60 66.44
node49, node78 53.79, node50 0.51, node56 10.89
node50, node85 251.76, node53 1.38, node55 20.10
node51, node98 2110.67, node59 23.67, node60 73.79
node52, node94 1471.80, node64 102.41, node66 123.03
node53, node72 22.85, node56 4.33, node67 88.35
```

```

node54, node88 967.59, node59 24.30, node73 238.61
node55, node84 86.09, node57 2.13, node64 60.80
node56, node76 197.03, node57 0.02, node61 11.06
node57, node86 701.09, node58 0.46, node60 7.01
node58, node83 556.70, node64 29.85, node65 34.32
node59, node90 820.66, node60 0.72, node71 0.67
node60, node76 48.03, node65 4.76, node67 1.63
node61, node98 1057.59, node63 0.95, node64 4.88
node62, node91 132.23, node64 2.94, node76 38.43
node63, node66 4.43, node72 70.08, node75 56.34
node64, node80 47.73, node65 0.30, node76 11.98
node65, node94 594.93, node66 0.64, node73 33.23
node66, node98 395.63, node68 2.66, node73 37.53
node67, node82 153.53, node68 0.09, node70 0.98
node68, node94 232.10, node70 3.35, node71 1.66
node69, node99 247.80, node70 0.06, node73 8.99
node70, node76 27.18, node72 1.50, node73 8.37
node71, node89 104.50, node74 8.86, node91 284.64
node72, node76 15.32, node84 102.77, node92 133.06
node73, node83 52.22, node76 1.40, node90 243.00
node74, node81 1.07, node76 0.52, node78 8.08
node75, node92 68.53, node76 0.81, node77 1.19
node76, node85 13.18, node77 0.45, node78 2.36
node77, node80 8.94, node78 0.98, node86 64.32
node78, node98 355.90, node81 2.59
node79, node81 0.09, node85 1.45, node91 22.35
node80, node92 121.87, node88 28.78, node98 264.34
node81, node94 99.78, node89 39.52, node92 99.89
node82, node91 47.44, node88 28.05, node93 11.99
node83, node94 114.95, node86 8.75, node88 5.78
node84, node89 19.14, node94 30.41, node98 121.05
node85, node97 94.51, node87 2.66, node89 4.90
node86, node97 85.09
node87, node88 0.21, node91 11.14, node92 21.23
node88, node93 1.31, node91 6.83, node98 6.12
node89, node97 36.97, node99 82.12
node90, node96 23.53, node94 10.47, node99 50.99
node91, node97 22.17
node92, node96 10.83, node97 11.24, node99 34.68
node93, node94 0.19, node97 6.71, node99 32.77
node94, node98 5.91, node96 2.03
node95, node98 6.17, node99 0.27
node96, node98 3.32, node97 0.43, node99 5.87
node97, node98 0.30
node98, node99 0.33
node99,

```

Writing graph.txt

19.7 Solutions

19.7.1 Exercise 1

First let's write a function that reads in the graph data above and builds a distance matrix.

```
In [5]: num_nodes = 100
destination_node = 99

def map_graph_to_distance_matrix(in_file):

    # First let's set up the distance matrix Q with inf everywhere
    Q = np.ones((num_nodes, num_nodes))
    Q = Q * np.inf

    # Now we read in the data and modify Q
    infile = open(in_file)
    for line in infile:
        elements = line.split(',')
        node = elements.pop(0)
        node = int(node[4:])      # convert node description to integer
        if node != destination_node:
            for element in elements:
                destination, cost = element.split()
                destination = int(destination[4:])
                Q[node, destination] = float(cost)
                Q[destination_node, destination_node] = 0

    infile.close()
    return Q
```

In addition, let's write

1. a “Bellman operator” function that takes a distance matrix and current guess of J and returns an updated guess of J , and
2. a function that takes a distance matrix and returns a cost-to-go function.

We'll use the algorithm described above.

The minimization step is vectorized to make it faster.

```
In [6]: def bellman(J, Q):
    num_nodes = Q.shape[0]
    next_J = np.empty_like(J)
    for v in range(num_nodes):
        next_J[v] = np.min(Q[v, :] + J)
    return next_J

def compute_cost_to_go(Q):
    J = np.zeros(num_nodes)      # Initial guess
    next_J = np.empty(num_nodes) # Stores updated guess
    max_iter = 500
    i = 0

    while i < max_iter:
        next_J = bellman(J, Q)
        if np.allclose(next_J, J):
            break
        else:
            J[:] = next_J      # Copy contents of next_J to J
```

```
i += 1
return(J)
```

We used `np.allclose()` rather than testing exact equality because we are dealing with floating point numbers now.

Finally, here's a function that uses the cost-to-go function to obtain the optimal path (and its cost).

```
In [7]: def print_best_path(J, Q):
    sum_costs = 0
    current_node = 0
    while current_node != destination_node:
        print(current_node)
        # Move to the next node and increment costs
        next_node = np.argmin(Q[current_node, :] + J)
        sum_costs += Q[current_node, next_node]
        current_node = next_node

    print(destination_node)
    print('Cost: ', sum_costs)
```

Okay, now we have the necessary functions, let's call them to do the job we were assigned.

```
In [8]: Q = map_graph_to_distance_matrix('graph.txt')
J = compute_cost_to_go(Q)
print_best_path(J, Q)
```

```
0
8
11
18
23
33
41
53
56
57
60
67
70
73
76
85
87
88
93
94
96
97
98
99
Cost: 160.55000000000007
```

Chapter 20

Cass-Koopmans Planning Problem

20.1 Contents

- Overview 20.2
- The Model 20.3
- Planning Problem 20.4
- Shooting Algorithm 20.5
- Setting Initial Capital to Steady State Capital 20.6
- A Turnpike Property 20.7
- A Limiting Economy 20.8
- Concluding Remarks 20.9

20.2 Overview

This lecture and lecture [Cass-Koopmans Competitive Equilibrium](#) describe a model that Tjalling Koopmans [66] and David Cass [22] used to analyze optimal growth.

The model can be viewed as an extension of the model of Robert Solow described in [an earlier lecture](#) but adapted to make the saving rate the outcome of an optimal choice.

(Solow assumed a constant saving rate determined outside the model.)

We describe two versions of the model, one in this lecture and the other in [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.

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 that describes optimal paths for long but finite-horizon economies.

Let's start with some standard imports:

```
In [1]: from numba import njit, float64
from numba.experimental import jitclass
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

20.3 The Model

Time is discrete and takes values $t = 0, 1, \dots, T$ where T is finite.

(We'll study a limiting case in which $T = +\infty$ before concluding).

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 some each period.

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 $\vec{C} = \{C_0, \dots, C_T\}$ and $\vec{K} = \{K_0, \dots, K_{T+1}\}$.

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 $t \in [0, T]$.

The representative household has preferences over consumption bundles ordered by the utility functional:

$$U(\vec{C}) = \sum_{t=0}^T \beta^t \frac{C_t^{1-\gamma}}{1-\gamma} \quad (1)$$

where $\beta \in (0, 1)$ is a discount factor and $\gamma > 0$ governs the curvature of the one-period utility function with larger γ implying more curvature.

Note that

$$u(C_t) = \frac{C_t^{1-\gamma}}{1-\gamma} \quad (2)$$

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

$$F(K_t, N_t) = AK_t^\alpha N_t^{1-\alpha} \quad (3)$$

with $0 < \alpha < 1, A > 0$.

A feasible allocation \vec{C}, \vec{K} satisfies

$$C_t + K_{t+1} \leq F(K_t, N_t) + (1 - \delta)K_t, \quad \text{for all } t \in [0, T] \quad (4)$$

where $\delta \in (0, 1)$ is a depreciation rate of capital.

20.4 Planning Problem

A planner chooses an allocation $\{\vec{C}, \vec{K}\}$ to maximize (1) subject to (4).

Let $\vec{\mu} = \{\mu_0, \dots, \mu_T\}$ be a sequence of nonnegative **Lagrange multipliers**.

To find an optimal allocation, form a Lagrangian

$$\mathcal{L}(\vec{C}, \vec{K}, \vec{\mu}) = \sum_{t=0}^T \beta^t \{u(C_t) + \mu_t (F(K_t, 1) + (1 - \delta)K_t - C_t - K_{t+1})\}$$

and then pose the following min-max problem:

$$\min_{\vec{\mu}} \max_{\vec{C}, \vec{K}} \mathcal{L}(\vec{C}, \vec{K}, \vec{\mu}) \quad (5)$$

- **Extremization** means maximization with respect to \vec{C}, \vec{K} and minimization with respect to $\vec{\mu}$.
- Our problem satisfies conditions that assure that required second-order conditions are satisfied at an allocation that satisfies the first-order conditions that we are about to compute.

Before computing first-order conditions, we present some handy formulas.

20.4.1 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

$$\begin{aligned}
\frac{\partial F(K_t, N_t)}{\partial K_t} &= \frac{\partial N_t f\left(\frac{K_t}{N_t}\right)}{\partial K_t} \\
&= N_t f'\left(\frac{K_t}{N_t}\right) \frac{1}{N_t} \quad (\text{Chain rule}) \\
&= f'\left(\frac{K_t}{N_t}\right) \Big|_{N_t=1} \\
&= f'(K_t)
\end{aligned} \tag{6}$$

and the marginal product of labor

$$\begin{aligned}
\frac{\partial F(K_t, N_t)}{\partial N_t} &= \frac{\partial N_t f\left(\frac{K_t}{N_t}\right)}{\partial N_t} \quad (\text{Product rule}) \\
&= f\left(\frac{K_t}{N_t}\right) + N_t f'\left(\frac{K_t}{N_t}\right) \frac{-K_t}{N_t^2} \quad (\text{Chain rule}) \\
&= f\left(\frac{K_t}{N_t}\right) - \frac{K_t}{N_t} f'\left(\frac{K_t}{N_t}\right) \Big|_{N_t=1} \\
&= f(K_t) - f'(K_t) K_t
\end{aligned}$$

20.4.2 First-order necessary conditions

We now compute **first order necessary conditions** for extremization of the Lagrangian:

$$C_t : \quad u'(C_t) - \mu_t = 0 \quad \text{for all } t = 0, 1, \dots, T \tag{7}$$

$$K_t : \quad \beta \mu_t [(1 - \delta) + f'(K_t)] - \mu_{t-1} = 0 \quad \text{for all } t = 1, 2, \dots, T \tag{8}$$

$$\mu_t : \quad F(K_t, 1) + (1 - \delta)K_t - C_t - K_{t+1} = 0 \quad \text{for all } t = 0, 1, \dots, T \tag{9}$$

$$K_{T+1} : \quad -\mu_T \leq 0, \leq 0 \text{ if } K_{T+1} = 0; = 0 \text{ if } K_{T+1} > 0 \tag{10}$$

In computing (9) we recognize that K_t appears in both the time t and time $t - 1$ feasibility constraints.

(10) comes from differentiating with respect to K_{T+1} and applying the following **Karush-Kuhn-Tucker condition** (KKT) (see [Karush-Kuhn-Tucker conditions](#)):

$$\mu_T K_{T+1} = 0 \tag{11}$$

Combining (7) and (8) gives

$$u'(C_t) [(1 - \delta) + f'(K_t)] - u'(C_{t-1}) = 0 \quad \text{for all } t = 1, 2, \dots, T + 1$$

which can be rearranged to become

$$u'(C_{t+1})[(1 - \delta) + f'(K_{t+1})] = u'(C_t) \quad \text{for all } t = 0, 1, \dots, T \quad (12)$$

Applying the inverse of the utility 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 (2) becomes the consumption **Euler equation**

$$\begin{aligned} C_{t+1} &= (\beta C_t^\gamma [f'(K_{t+1}) + (1 - \delta)])^{1/\gamma} \\ &= C_t (\beta [f'(K_{t+1}) + (1 - \delta)])^{1/\gamma} \end{aligned}$$

Below we define a `jitclass` that stores parameters and functions that define our economy.

```
In [2]: 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
]

In [3]: @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'
     $\alpha, A = \text{self.}\alpha, \text{self.}A$ 

    return A * k **  $\alpha$ 

def f_prime(self, k):
    'Derivative of production function'
     $\alpha, A = \text{self.}\alpha, \text{self.}A$ 

    return  $\alpha * A * k^{*\alpha - 1}$ 

def f_prime_inv(self, k):
    'Inverse of derivative of production function'
     $\alpha, A = \text{self.}\alpha, \text{self.}A$ 

    return (k / (A **  $\alpha$ )) ** (1 / ( $\alpha - 1$ ))

def next_k_c(self, k, c):
    '''
        Given the current capital  $K_t$  and an arbitrary feasible
        consumption choice  $C_t$ , computes  $K_{t+1}$  by state transition law
        and optimal  $C_{t+1}$  by Euler equation.
    '''
     $\beta, \delta = \text{self.}\beta, \text{self.}\delta$ 
     $u_{\text{prime}}, u_{\text{prime\_inv}} = \text{self.}u_{\text{prime}}, \text{self.}u_{\text{prime\_inv}}$ 
     $f, f_{\text{prime}} = \text{self.}f, \text{self.}f_{\text{prime}}$ 

     $k_{\text{next}} = f(k) + (1 - \delta) * k - c$ 
     $c_{\text{next}} = u_{\text{prime\_inv}}(u_{\text{prime}}(c)) / (\beta * (f_{\text{prime}}(k_{\text{next}}) + (1 - \delta)))$ 

    return k_next, c_next

```

We can construct an economy with the Python code:

In [4]: pp = PlanningProblem()

20.5 Shooting Algorithm

We use **shooting** to compute an optimal allocation \vec{C}, \vec{K} and an associated Lagrange multiplier sequence $\vec{\mu}$.

The first-order necessary conditions (7), (8), and (9) 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 (11)

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 (7) and then k_1 from equation (9) and μ_1 from equation (8).
- We could continue in this way to compute the remaining elements of $\vec{C}, \vec{K}, \vec{\mu}$.

But 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.

We actually modify the algorithm slightly by starting with a guess for c_0 instead of μ_0 in the following code.

```
In [5]: @njit
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
```

We'll start with an incorrect guess.

```
In [6]: paths = shooting(pp, 0.2, 0.3, T=10)
```

```
In [7]: fig, axs = plt.subplots(1, 2, figsize=(14, 5))
```

```
colors = ['blue', 'red']
titles = ['Consumption', 'Capital']
ylabels = ['$c_t$', '$k_t$']
```

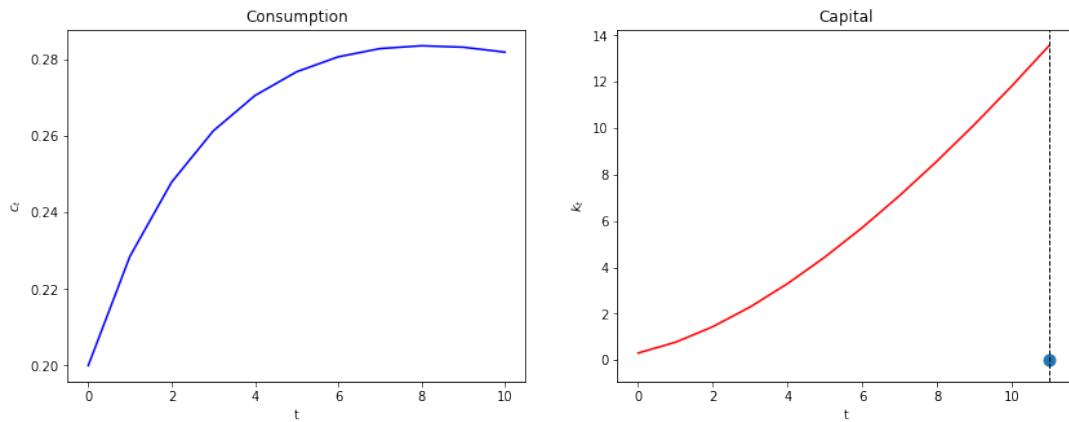
```

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 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.

```

In [8]: @njit
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:
            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

```

```

In [9]: 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 = ['$c_t$', '$k_t$', '$\mu_t$']
    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)

        mu_vec = pp.u_prime(c_vec)
        paths = [c_vec, k_vec, mu_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)

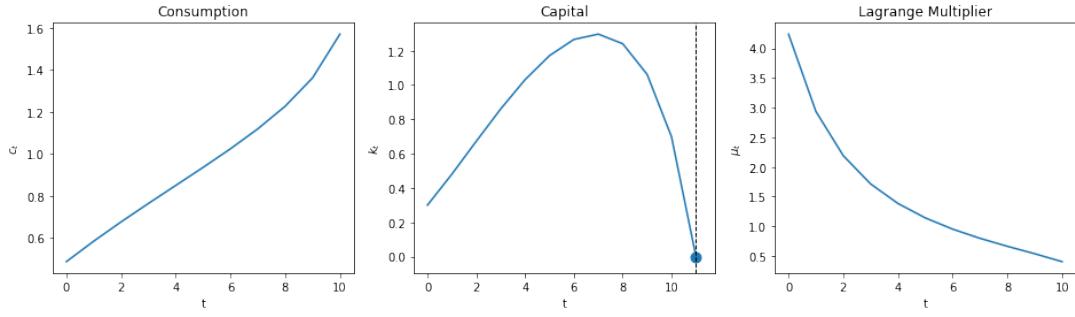
    axs[1].axvline(T+1, c='k', ls='--', lw=1)
    axs[1].scatter(T+1, paths[1][-1], s=80)

    return c_paths, k_paths

```

Now we can solve the model and plot the paths of consumption, capital, and Lagrange multiplier.

In [10]: `plot_paths(pp, 0.3, 0.3, [10]);`



20.6 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 .

Evaluating the feasibility constraint (4) at \bar{K} gives

$$f(\bar{K}) - \delta \bar{K} = \bar{C} \quad (13)$$

Substituting $K_t = \bar{K}$ and $C_t = \bar{C}$ for all t into (12) 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 the production function (3) 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 .

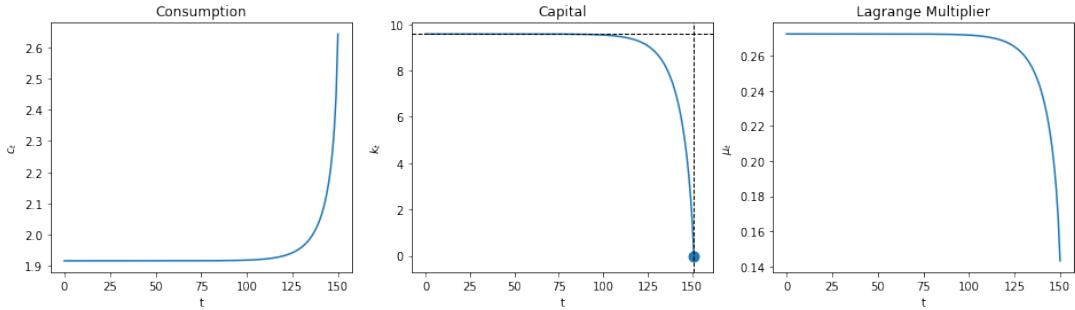
```
In [11]: ρ = 1 / pp.β - 1
k_ss = pp.f_prime_inv(ρ+pp.δ)

print(f'steady state for capital is: {k_ss}')
```

steady state for capital is: 9.57583816331462

Now we plot

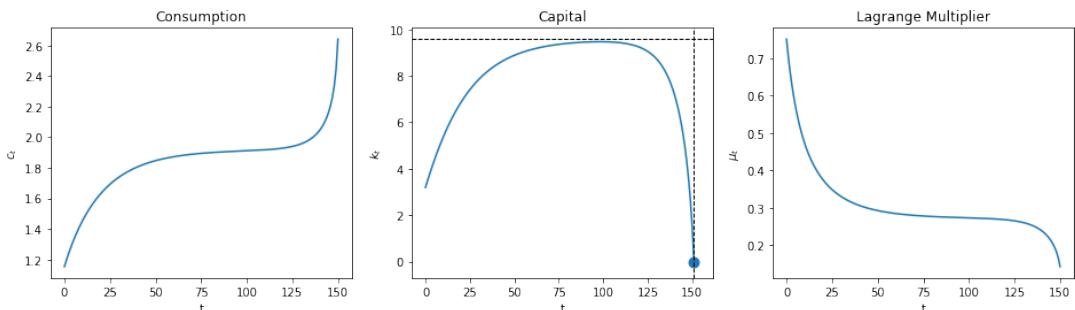
```
In [12]: plot_paths(pp, 0.3, k_ss, [150], k_ss=k_ss);
```



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} .

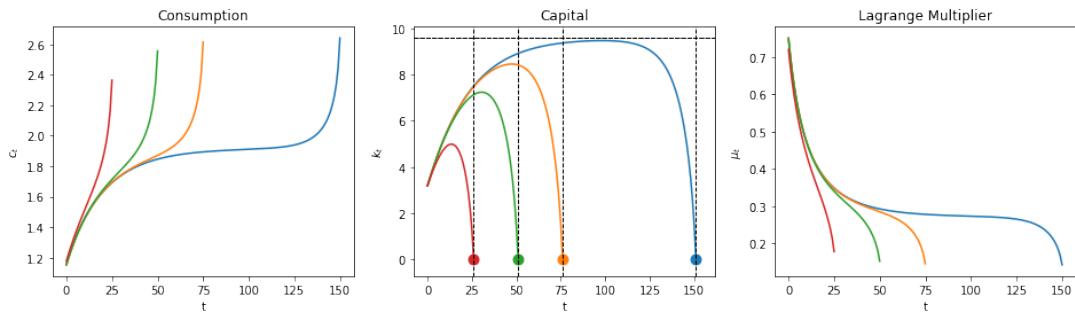
```
In [13]: plot_paths(pp, 0.3, k_ss/3, [150], k_ss=k_ss);
```



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 .

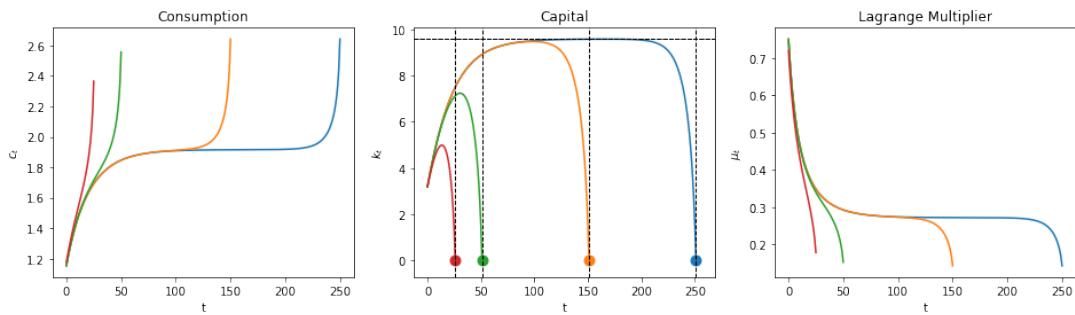
In [14]: `plot_paths(pp, 0.3, k_ss/3, [150, 75, 50, 25], k_ss=k_ss);`



20.7 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.

In [15]: `plot_paths(pp, 0.3, k_ss/3, [250, 150, 50, 25], k_ss=k_ss);`



Different colors in the above graphs are associated with different horizons T .

Notice that as the horizon increases, the planner puts K_t closer to the steady state value 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.

Let's calculate and plot the saving rate.

```
In [16]: @njit
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
```

```
In [17]: 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))

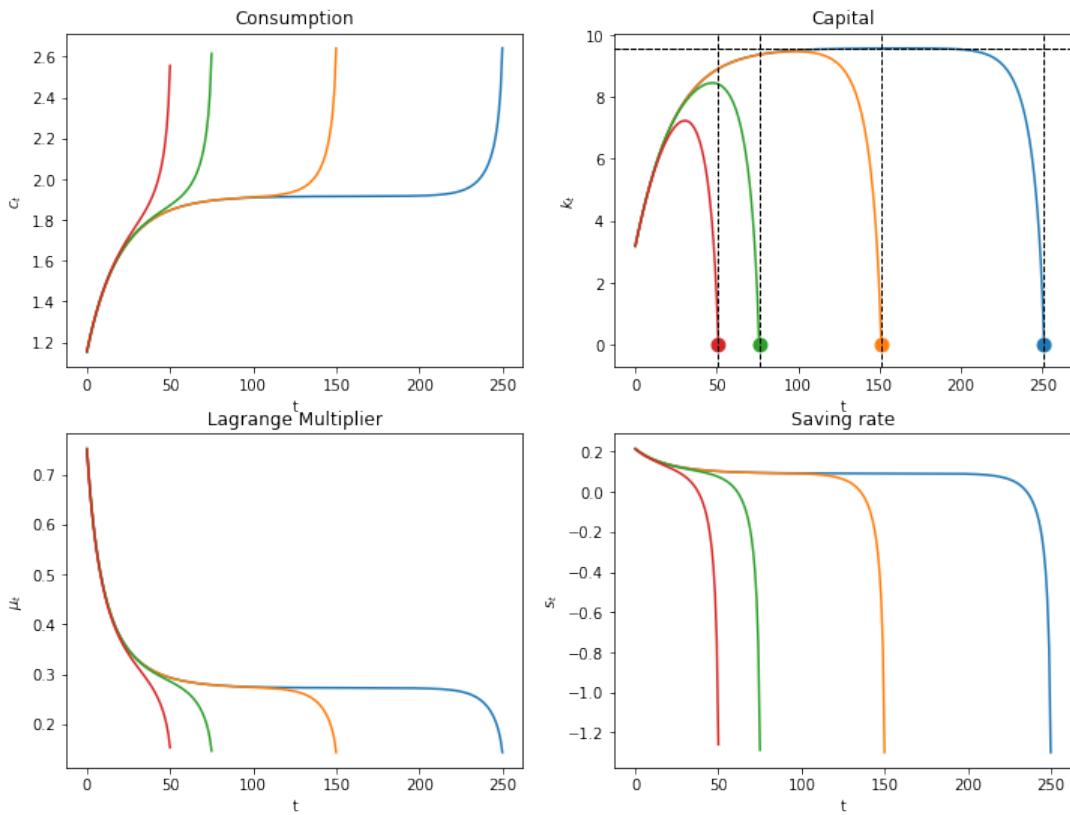
    c_paths, k_paths = plot_paths(pp, c0, k0, T_arr, k_ter=k_ter, k_ss=k_ss,
                                   axs=axs.flatten())

    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='--')

In [18]: plot_saving_rate(pp, 0.3, k_ss/3, [250, 150, 75, 50], k_ss=k_ss)
```



20.8 A Limiting Economy

We want to set $T = +\infty$.

The appropriate thing to do is to replace terminal condition (10) with

$$\lim_{T \rightarrow +\infty} \beta^T u'(C_T) K_{T+1} = 0,$$

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 (13) the steady state saving rate equals

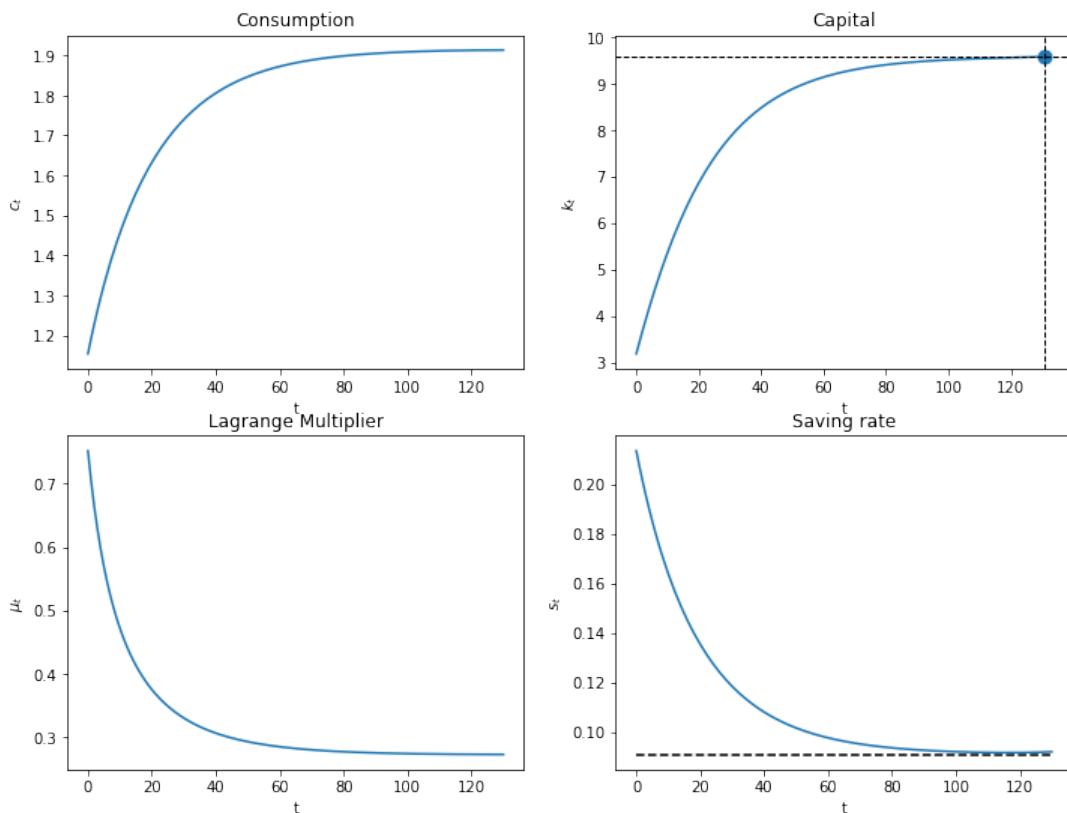
$$\bar{s} = \frac{\delta \bar{K}}{f'(\bar{K})}$$

The steady state saving rate $\bar{s} f(\bar{K})$ is the amount required to offset capital depreciation each period.

We first study optimal capital paths that start below the steady state.

```
In [19]: # 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, $f''(K) < 0$, so as K rises, $f'(K)$ declines.

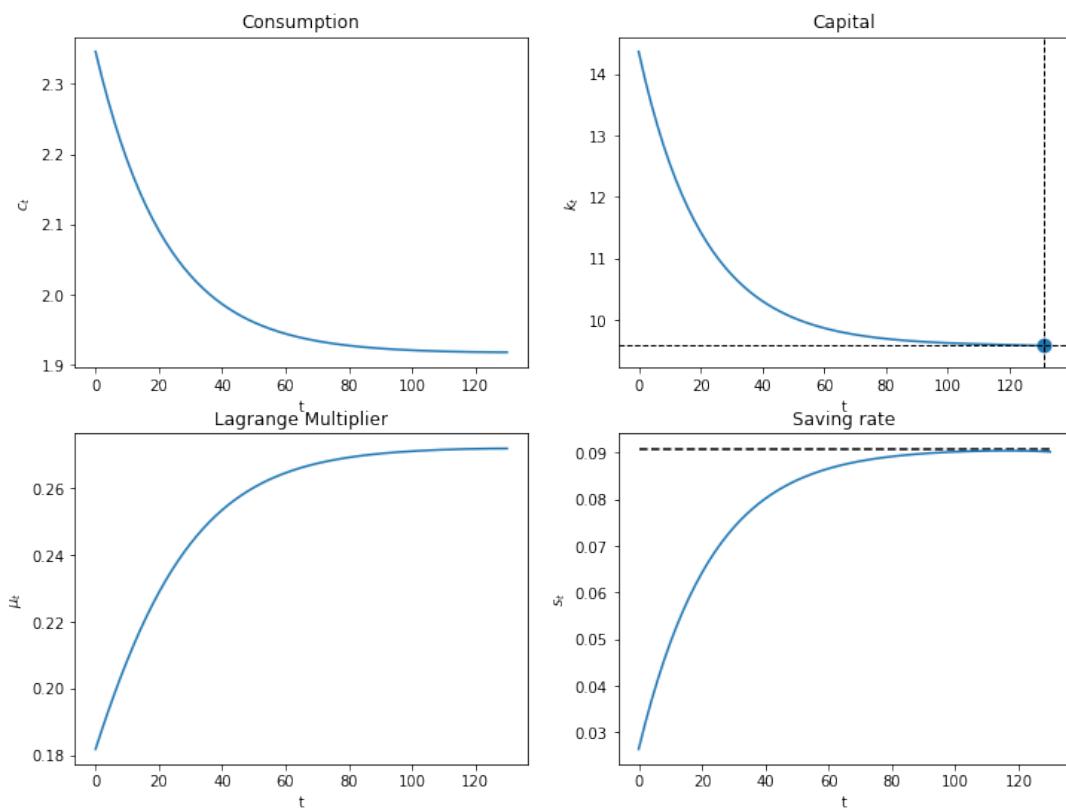
The planner slowly lowers the saving rate until reaching a steady state in which $f'(K) = \rho + \delta$.

20.8.1 Exercise

- 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?

20.8.2 Solution

```
In [20]: plot_saving_rate(pp, 0.3, k_ss*1.5, [130], k_ter=k_ss, k_ss=k_ss, s_ss=s_ss)
```



20.9 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 somewhat produced by the invisible hand.

Market prices must adjust to 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.

Chapter 21

Cass-Koopmans Competitive Equilibrium

21.1 Contents

- Overview 21.2
- Review of Cass-Koopmans Model 21.3
- Competitive Equilibrium 21.4
- Market Structure 21.5
- Firm Problem 21.6
- Household Problem 21.7
- Computing a Competitive Equilibrium 21.8
- Yield Curves and Hicks-Arrow Prices 21.9

21.2 Overview

This lecture continues our analysis in this lecture [Cass-Koopmans Planning Model](#) about the model that Tjalling Koopmans [66] and David Cass [22] used to study optimal growth.

This lecture illustrates what is, in fact, a more general connection between a **planned economy** and an economy organized as a **competitive equilibrium**.

The earlier lecture [Cass-Koopmans Planning Model](#) studied a planning problem and used ideas including

- A min-max problem for solving the planning problem.
- 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 in the min-max 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 the cases for the optimality of two competing visions of good ways to organize an economy, namely:
 - **socialism** in which a central planner commands the allocation of resources, and
 - **capitalism** (also known as a **market economy**) in which competitive equilibrium **prices** induce individual consumers and producers to choose a socially optimal allocation as an unintended consequence of their selfish decisions

Let's start with some standard imports:

```
In [1]: from numba import njit, float64
from numba.experimental import jitclass
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

21.3 Review of Cass-Koopmans Model

The physical setting is identical with that in [Cass-Koopmans Planning Model](#).

Time is discrete and takes values $t = 0, 1, \dots, T$.

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 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 $\vec{C} = \{C_0, \dots, C_T\}$ and $\vec{K} = \{K_0, \dots, K_{T+1}\}$.

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 $t \in [0, T]$.

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

$$F(K_t, N_t) = AK_t^\alpha N_t^{1-\alpha}$$

with $0 < \alpha < 1$, $A > 0$.

A feasible allocation \vec{C}, \vec{K} satisfies

$$C_t + K_{t+1} \leq F(K_t, N_t) + (1 - \delta)K_t, \quad \text{for all } t \in [0, T]$$

where $\delta \in (0, 1)$ is a depreciation rate of capital.

21.3.1 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 (1) subject to (4).

The allocation that solves the planning problem plays an important role in a competitive equilibrium as we shall see below.

21.4 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.

Market prices adjust to reconcile distinct decisions that are made separately by a representative household and a representative firm.

There is a representative consumer who has the same preferences over consumption plans as did the consumer in the planned economy.

Instead of being told what to consume and save by a planner, the household chooses for itself subject to a budget constraint

- At each time t , the household 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 all 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: We can think of there being a large number M of identical representative consumers and M identical representative firms.

21.5 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 can trade date 0 goods for goods at all other dates $t = 1, 2, \dots, T$.

21.5.1 Prices

There are sequences of prices $\{w_t, \eta_t\}_{t=0}^T = \{\vec{w}, \vec{\eta}\}$ where w_t is a wage or rental rate for labor at time t and η_t is a rental rate for capital at time t .

In addition there are intertemporal prices that work as follows.

Let q_t^0 be the price of a good at date t relative to a good at date 0.

We call $\{q_t^0\}_{t=0}^T$ a vector of **Hicks-Arrow prices**, named after the 1972 economics Nobel prize winners.

Evidently,

$$q_t^0 = \frac{\text{number of time 0 goods}}{\text{number of time } t \text{ goods}}$$

Because q_t^0 is a **relative price**, the units in terms of which prices are quoted are arbitrary – we are free to normalize them.

21.6 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

$$F(\tilde{k}_t, \tilde{n}_t) = A \tilde{k}_t^\alpha \tilde{n}_t^{1-\alpha}$$

21.6.1 Zero Profit Conditions

Zero-profits condition for capital and labor are

$$F_k(\tilde{k}_t, \tilde{n}_t) = \eta_t$$

and

$$F_n(\tilde{k}_t, \tilde{n}_t) = w_t \quad (1)$$

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 it assumed 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 the partial derivative $\frac{\partial F}{\partial \alpha}$ on both sides of the above equation gives

$$F(\tilde{k}_t, \tilde{n}_t) =_{\text{chain rule}} \frac{\partial F}{\partial \tilde{k}_t} \tilde{k}_t + \frac{\partial F}{\partial \tilde{n}_t} \tilde{n}_t$$

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 will want to make \tilde{k}_t arbitrarily large.

But setting $\tilde{k}_t = +\infty$ is not physically feasible, so presumably **equilibrium** prices will assume values that prevent 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 will set \tilde{k}_t to zero, something that is not feasible.

It is convenient to define $\vec{w} = \{w_0, \dots, w_T\}$ and $\vec{\eta} = \{\eta_0, \dots, \eta_T\}$.

21.7 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

$$(c_t + (k_{t+1} - (1 - \delta)k_t))$$

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 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 household'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 states that the present value of the household's net excess demands must be zero:

$$\sum_{t=0}^T q_t^0 e_t \leq 0$$

or

$$\sum_{t=0}^T q_t^0 (c_t + (k_{t+1} - (1 - \delta)k_t) - (w_t 1 + \eta_t k_t)) \leq 0$$

The household chooses an allocation to solve the constrained optimization problem:

$$\begin{aligned} & \max_{\vec{c}, \vec{k}} \sum_{t=0}^T \beta^t u(c_t) \\ \text{subject to } & \sum_{t=0}^T q_t^0 (c_t + (k_{t+1} - (1 - \delta)k_t) - w_t - \eta_t k_t) \leq 0 \end{aligned}$$

21.7.1 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 for which
 - Given the price system, the allocation solves the household's problem.
 - Given the price system, the allocation solves the firm's problem.

21.8 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.

21.8.1 Guess for Price System

In this lecture [Cass-Koopmans Planning Model](#), we computed an allocation $\{\vec{C}, \vec{K}, \vec{N}\}$ that solves the planning problem.

(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](#).)

We use that allocation to construct a guess for the equilibrium price system.

In particular, we guess that for $t = 0, \dots, T$:

$$\lambda q_t^0 = \beta^t u'(K_t) = \beta^t \mu_t \quad (2)$$

$$w_t = f(K_t) - K_t f'(K_t) \quad (3)$$

$$\eta_t = f'(K_t) \quad (4)$$

At these prices, let the capital chosen by the household be

$$k_t^*(\vec{q}, \vec{w}, \vec{\eta}), \quad t \geq 0 \quad (5)$$

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

$$k_t^* = \tilde{k}_t^* \quad (6)$$

$$1 = \tilde{n}_t^* \quad (7)$$

$$c_t^* + k_{t+1}^* - (1 - \delta)k_t^* = F(\tilde{k}_t^*, \tilde{n}_t^*)$$

We shall verify that for $t = 0, \dots, T$ the allocations chosen by the household and the firm both equal the allocation that solves the planning problem:

$$k_t^* = \tilde{k}_t^* = K_t, \tilde{n}_t = 1, c_t^* = C_t \quad (8)$$

21.8.2 Verification Procedure

Our approach is to stare at first-order necessary conditions for the 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.

21.8.3 Household's Lagrangian

To solve the household's problem, we formulate the Lagrangian

$$\mathcal{L}(\vec{c}, \vec{k}, \lambda) = \sum_{t=0}^T \beta^t u(c_t) + \lambda \left(\sum_{t=0}^T q_t^0 (((1-\delta)k_t - w_t) + \eta_t k_t - c_t - k_{t+1}) \right)$$

and attack the min-max problem:

$$\min_{\lambda} \max_{\vec{c}, \vec{k}} \mathcal{L}(\vec{c}, \vec{k}, \lambda)$$

First-order conditions are

$$c_t : \quad \beta^t u'(c_t) - \lambda q_t^0 = 0 \quad t = 0, 1, \dots, T \quad (9)$$

$$k_t : \quad -\lambda q_t^0 [(1-\delta) + \eta_t] + \lambda q_{t-1}^0 = 0 \quad t = 1, 2, \dots, T+1 \quad (10)$$

$$\lambda : \quad \left(\sum_{t=0}^T q_t^0 (c_t + (k_{t+1} - (1-\delta)k_t) - w_t - \eta_t k_t) \right) \leq 0 \quad (11)$$

$$k_{T+1} : \quad -\lambda q_0^{T+1} \leq 0, \quad \leq 0 \text{ if } k_{T+1} = 0; = 0 \text{ if } k_{T+1} > 0 \quad (12)$$

Now we plug in our guesses of prices and embark on some algebra in the hope of derived all first-order necessary conditions (7)-(10) for the planning problem from this lecture [Cass-Koopmans Planning Model](#).

Combining (9) and (2), we get:

$$u'(C_t) = \mu_t$$

which is (7).

Combining (10), (2), and (4) we get:

$$-\lambda \beta^t \mu_t [(1-\delta) + f'(K_t)] + \lambda \beta^{t-1} \mu_{t-1} = 0 \quad (13)$$

Rewriting (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 (8).

Combining (11), (2), (3) and (4) after multiplying both sides of (11) by λ , we get

$$\sum_{t=0}^T \beta^t \mu_t (C_t + (K_{t+1} - (1-\delta)K_t) - f(K_t) + K_t f'(K_t) - f'(K_t)K_t) \leq 0$$

which simplifies

$$\sum_{t=0}^T \beta^t \mu_t (C_t + K_{t+1} - (1-\delta)K_t - F(K_t, 1)) \leq 0$$

Since $\beta^t \mu_t > 0$ for $t = 0, \dots, T$, it follows that

$$C_t + K_{t+1} - (1-\delta)K_t - F(K_t, 1) = 0 \quad \text{for all } t \text{ in } 0, \dots, T$$

which is (9).

Combining (12) and (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 (10) 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.

We now turn to the problem faced by a firm in a competitive equilibrium:

If we plug (8) into (1) for all t , we get

$$\frac{\partial F(K_t, 1)}{\partial K_t} = f'(K_t) = \eta_t$$

which is (4).

If we now plug (8) into (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 (5).

So 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 (6) and (7) this allocation is identical to the one that solves the consumer's problem.

Note: Because budget sets are affected only by relative prices, $\{q_0^t\}$ is determined only up to multiplication by a positive constant.

Normalization: We are free to choose a $\{q_0^t\}$ that makes $\lambda = 1$ so that we are measuring q_0^t 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 that characterize an economy.

```
In [2]: 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
]

In [3]: @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'
     $\alpha, A = \text{self.}\alpha, \text{self.}A$ 

    return A * k **  $\alpha$ 

def f_prime(self, k):
    'Derivative of production function'
     $\alpha, A = \text{self.}\alpha, \text{self.}A$ 

    return  $\alpha * A * k^{*\alpha - 1}$ 

def f_prime_inv(self, k):
    'Inverse of derivative of production function'
     $\alpha, A = \text{self.}\alpha, \text{self.}A$ 

    return (k / (A *  $\alpha$ )) ** (1 / ( $\alpha - 1$ ))

def next_k_c(self, k, c):
    """
    Given the current capital  $K_t$  and an arbitrary feasible
    consumption choice  $C_t$ , computes  $K_{t+1}$  by state transition law
    and optimal  $C_{t+1}$  by Euler equation.
    """
     $\beta, \delta = \text{self.}\beta, \text{self.}\delta$ 
    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 -  $\delta$ ) * k - c
    c_next = u_prime_inv(u_prime(c)) / ( $\beta * (f'_\text{prime}(k_{t+1}) + (1 - \delta))$ )

    return k_next, c_next

```

```

In [4]: @njit
def shooting(pp, c0, k0, T=10):
    """
    Given the initial condition of capital  $k_0$  and an initial guess
    of consumption  $c_0$ , 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. $\delta$ ) * k_vec[T] - c_vec[T]

    return c_vec, k_vec

```

```
In [5]: @njit
    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:
                    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
```

```
In [6]: 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 that turns out to be the allocation associated with a competitive equilibrium.

Now we're ready to bring in Python code that we require to compute additional objects that appear in a competitive equilibrium.

```
In [7]: @njit
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
```

```

@njit
def w(pp, k_path):
    w_path = pp.f(k_path) - k_path * pp.f_prime(k_path)
    return w_path

@njit
def η(pp, k_path):
    η_path = pp.f_prime(k_path)
    return η_path

```

Now we calculate and plot for each T

```

In [8]: T_arr = [250, 150, 75, 50]

fix, axs = plt.subplots(2, 3, figsize=(13, 6))
titles = ['Arrow-Hicks Prices', 'Labor Rental Rate', 'Capital Rental Rate',
          'Consumption', 'Capital', 'Lagrange Multiplier']
ylabes = ['$q_t$', '$w_t$', '$\eta_t$', '$c_t$', '$k_t$', '$\mu_t$']

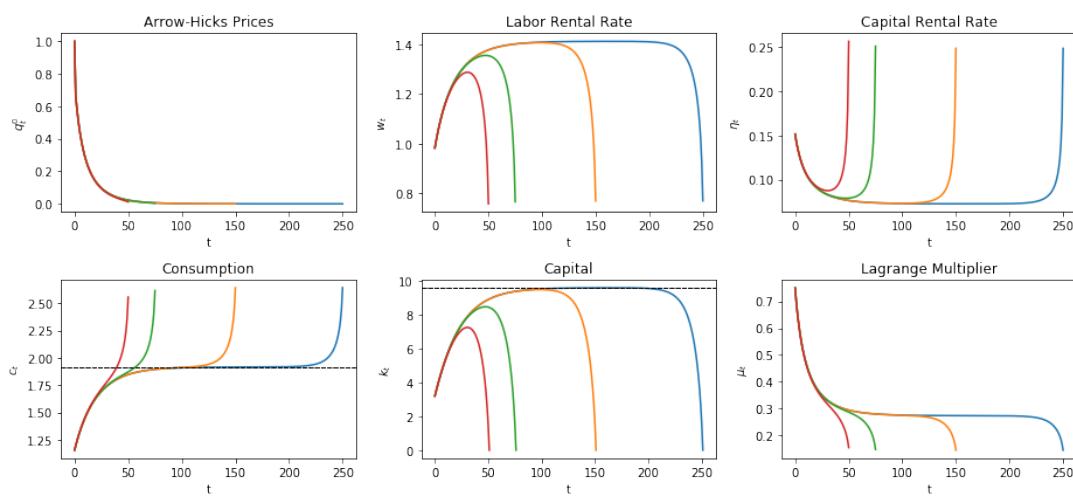
for T in T_arr:
    c_path, k_path = bisection(pp, 0.3, k_ss/3, T, verbose=False)
    μ_path = pp.u_prime(c_path)

    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=ylabes[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 $T = 150$

```
In [9]: T = 150
γ_arr = [1.1, 4, 6, 8]

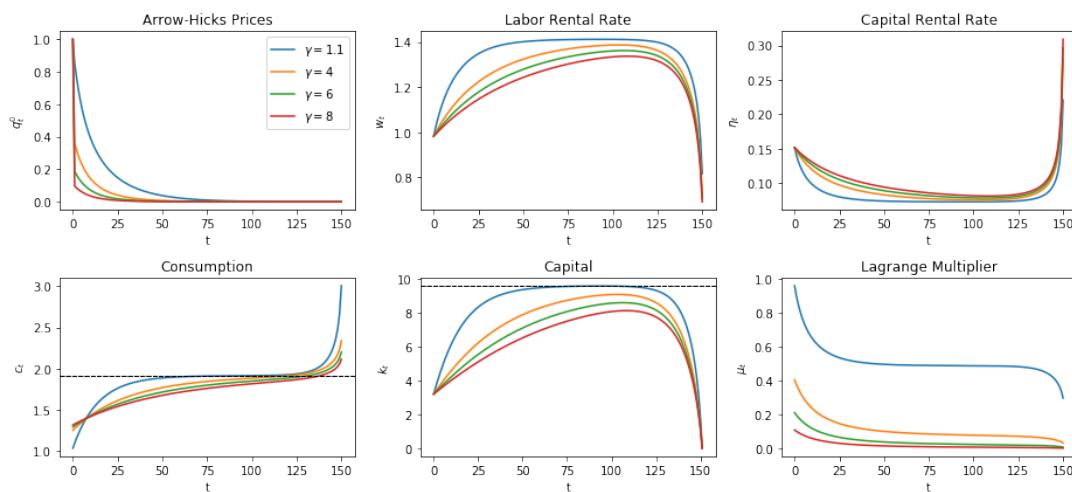
fix, axs = plt.subplots(2, 3, figsize=(13, 6))

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=f'$\gamma = {γ}$')
        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 adjustments to the steady state allocations.

Vice-versa for lower γ .

21.9 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 $j = 1, 2, \dots$ against $j = 1, 2, \dots$

The formulas we want are:

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 for a base-year $t_0 \leq t$

$$q_t^{t_0} = \beta^{t-t_0} \frac{u'(c_t)}{u'(c_{t_0})} = \beta^{t-t_0} \frac{c_t^{-\gamma}}{c_{t_0}^{-\gamma}}$$

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

```
In [10]: @njit
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

@njit
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)
```

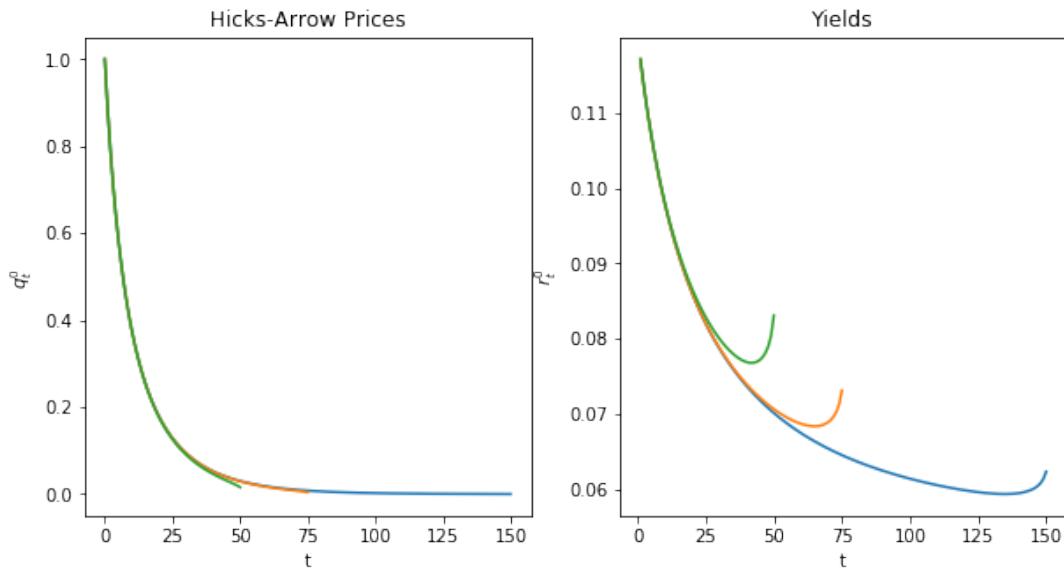
```

    axs[0].set(xlabel='t', ylabel='$q_t$', title='Hicks-Arrow
    ↪Prices')

    axs[1].plot(range(t0+1, T+1), r_path)
    axs[1].set(xlabel='t', ylabel='$r_t$', title='Yields')

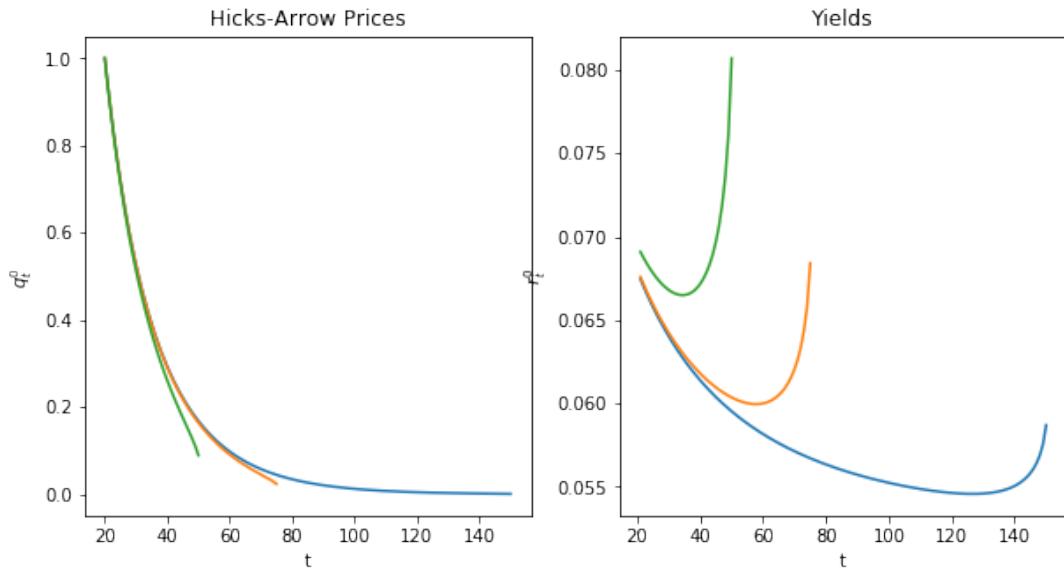
```

In [11]: `T_arr = [150, 75, 50]`
`plot_yield_curves(pp, 0, 0.3, k_ss/3, T_arr)`



Now we plot when $t_0 = 20$

In [12]: `plot_yield_curves(pp, 20, 0.3, k_ss/3, T_arr)`



We aim to have more to say about the term structure of interest rates in a planned lecture on the topic.

Part III

Search

Chapter 22

Job Search I: The McCall Search Model

22.1 Contents

- Overview 22.2
- The McCall Model 22.3
- Computing the Optimal Policy: Take 1 22.4
- Computing the Optimal Policy: Take 2 22.5
- Exercises 22.6
- Solutions 22.7

“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:

```
In [1]: !pip install quantecon
```

22.2 Overview

The McCall search model [80] helped transform economists’ way of thinking about labor markets.

To clarify vague notions such as “involuntary” unemployment, McCall modeled the decision problem of unemployed agents directly, in terms of factors such as

- current and likely future wages
- impatience
- unemployment compensation

To solve the decision problem he used dynamic programming.

Here we set up McCall's model and adopt the same solution method.

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:

```
In [2]: import numpy as np
from numba import jit, float64
from numba.experimental import jitclass
import matplotlib.pyplot as plt
%matplotlib inline
import quantecon as qe
from quantecon.distributions import BetaBinomial

/home/ubuntu/anaconda3/lib/python3.7/site-packages/numba/np/ufunc/parallel.py:
  ↵355:
NumbaWarning: The TBB threading layer requires TBB version 2019.5 or later i.e.,
TBB_INTERFACE_VERSION >= 11005. Found TBB_INTERFACE_VERSION = 11004. The TBB
  ↵threading
layer is disabled.
warnings.warn(problem)
```

22.3 The McCall Model

An unemployed agent receives in each period a job offer at wage w_t .

The wage offer is a nonnegative function of some underlying state:

$$w_t = w(s_t) \quad \text{where } s_t \in \mathbb{S}$$

Here you should think of state process $\{s_t\}$ as some underlying, unspecified random factor that impacts on wages.

(Introducing an exogenous stochastic state process is a standard way for economists to inject randomness into their models.)

In this lecture, we adopt the following simple environment:

- $\{s_t\}$ is IID, with $q(s)$ being the probability of observing state s in \mathbb{S} at each point in time, and
- the agent observes s_t at the start of t and hence knows $w_t = w(s_t)$,
- the set \mathbb{S} is finite.

(In later lectures, we will relax all of 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

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t y_t$$

The constant β lies in $(0, 1)$ and is called a **discount factor**.

The smaller is β , the more the agent discounts future utility relative to current utility.

The variable y_t is income, equal to

- his/her wage w_t when employed
- unemployment compensation c when unemployed

The agent is assumed to know that $\{s_t\}$ is IID with common distribution q and can use this when computing expectations.

22.3.1 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 optimally 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.

22.3.2 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 (in this case, either employment or unemployment)

To weigh these two aspects of the decision problem, we need to assign *values* to states.

To this end, let $v^*(s)$ be the total lifetime *value* accruing to an unemployed worker who enters the current period unemployed when the state is $s \in \mathbb{S}$.

In particular, the agent has wage offer $w(s)$ in hand.

More precisely, $v^*(s)$ denotes the value of the objective function (1) when an agent in this situation makes *optimal* decisions now and at all future points in time.

Of course $v^*(s)$ is not trivial to calculate because we don't yet know what decisions are optimal and what aren't!

But think of v^* as a function that assigns to each possible state s the maximal lifetime value that can be obtained with that offer in hand.

A crucial observation is that this function v^* must satisfy the recursion

$$v^*(s) = \max \left\{ \frac{w(s)}{1-\beta}, c + \beta \sum_{s' \in S} v^*(s') q(s') \right\} \quad (1)$$

for every possible s in S .

This important equation 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

$$\frac{w(s)}{1-\beta} = w(s) + \beta w(s) + \beta^2 w(s) + \dots$$

- 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 state s .

But this is precisely $v^*(s)$, which is the l.h.s. of (1).

22.3.3 The Optimal Policy

Suppose for now that we are able to solve (1) for the unknown function v^* .

Once we have this function in hand we can behave optimally (i.e., make the right choice between accept and reject).

All we have to do is select the maximal choice on the r.h.s. of (1).

The optimal action is best thought of as a **policy**, which is, in general, a map from states to actions.

Given *any* s , we can read off the corresponding best choice (accept or reject) by picking the max on the r.h.s. of (1).

Thus, we have a map from \mathbb{R} to $\{0, 1\}$, with 1 meaning accept and 0 meaning reject.

We can write the policy as follows

$$\sigma(s) := \mathbf{1} \left\{ \frac{w(s)}{1-\beta} \geq c + \beta \sum_{s' \in S} v^*(s') q(s') \right\}$$

Here $\mathbf{1}\{P\} = 1$ if statement P is true and equals 0 otherwise.

We can also write this as

$$\sigma(s) := \mathbf{1}\{w(s) \geq \bar{w}\}$$

where

$$\bar{w} := (1 - \beta) \left\{ c + \beta \sum_{s'} v^*(s') q(s') \right\} \quad (2)$$

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 (2), we can compute this reservation wage if we can compute the value function.

22.4 Computing the Optimal Policy: Take 1

To put the above ideas into action, we need to compute the value function at each possible state $s \in \mathbb{S}$.

Let's suppose that $\mathbb{S} = \{1, \dots, n\}$.

The value function is then represented by the vector $v^* = (v^*(i))_{i=1}^n$.

In view of (1), this vector satisfies the nonlinear system of equations

$$v^*(i) = \max \left\{ \frac{w(i)}{1 - \beta}, c + \beta \sum_{1 \leq j \leq n} v^*(j) q(j) \right\} \quad \text{for } i = 1, \dots, n \quad (3)$$

22.4.1 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

$$v'(i) = \max \left\{ \frac{w(i)}{1 - \beta}, c + \beta \sum_{1 \leq j \leq n} v(j) q(j) \right\} \quad \text{for } i = 1, \dots, n \quad (4)$$

Step 3: calculate a measure of the deviation 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 .

Let $\{v_k\}$ denote the sequence generated by this algorithm.

This sequence converges to the solution to (3) as $k \rightarrow \infty$, which is the value function v^* .

22.4.2 The Fixed Point Theory

What's the mathematics behind these ideas?

First, one defines a mapping T from \mathbb{R}^n to itself via

$$(Tv)(i) = \max \left\{ \frac{w(i)}{1-\beta}, c + \beta \sum_{1 \leq j \leq n} v(j)q(j) \right\} \quad \text{for } i = 1, \dots, n \quad (5)$$

(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 .

22.4.3 Implementation

Our default for q , the distribution of the state process, will be [Beta-binomial](#).

```
In [3]: n, a, b = 50, 200, 100          # default parameters
q_default = BetaBinomial(n, a, b).pdf()  # default choice of q
```

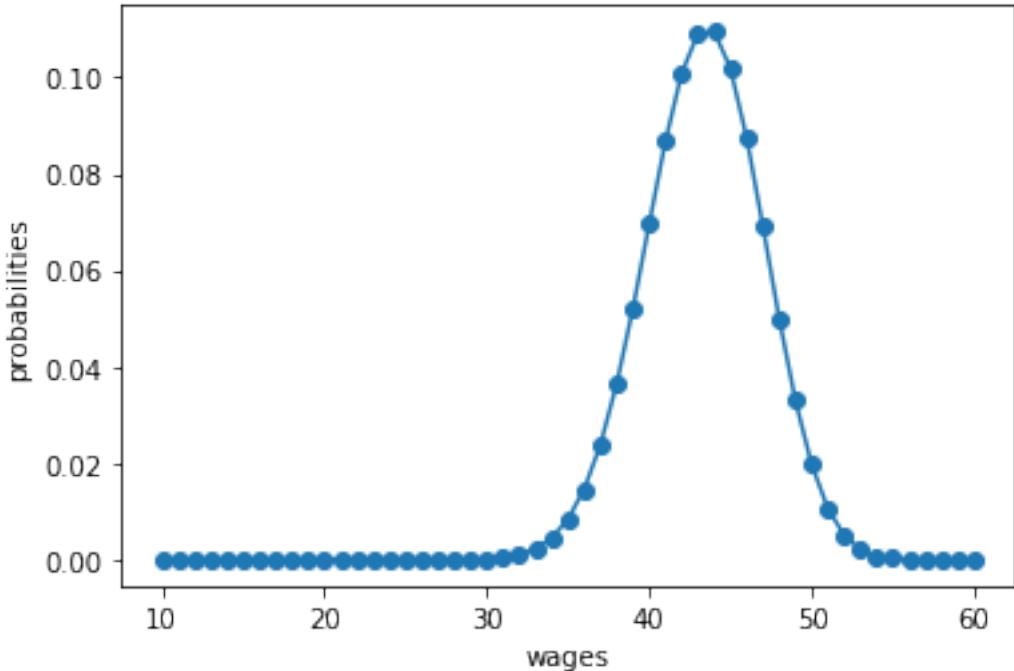
Our default set of values for wages will be

```
In [4]: w_min, w_max = 10, 60
w_default = np.linspace(w_min, w_max, n+1)
```

Here's a plot of the probabilities of different wage outcomes:

```
In [5]: 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 are going to use Numba to accelerate our code.

- See, in particular, the discussion of `@jitclass` in [our lecture on Numba](#).

The following helps Numba by providing some type

```
In [6]: mccall_data = [
    ('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
]
```

Here's a class that stores the data and computes the values of state-action pairs, i.e. the value in the maximum bracket on the right hand side of the Bellman equation (4), given the current state and an arbitrary feasible action.

Default parameter values are embedded in the class.

```
In [7]: @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_default, q_default

    def state_action_values(self, i, v):
        """
        The values of state-action pairs.
        """
        # 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 * np.sum(v * q)

return np.array([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.

Here's a function to implement this:

```
In [8]: def plot_value_function_seq(mcm, ax, num_plots=6):
    """
    Plot a sequence of value functions.

    * mcm is an instance of McCallModel
    * ax is an axes object that implements a plot method.

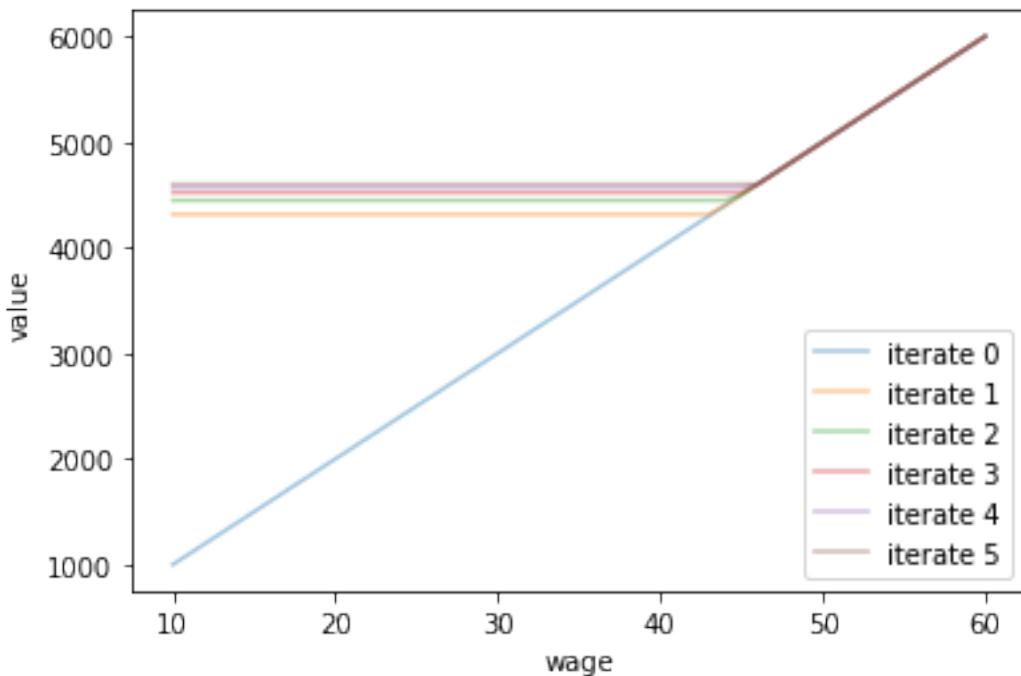
    """
    n = len(mcm.w)
    v = mcm.w / (1 - mcm.β)
    v_next = np.empty_like(v)
    for i in range(num_plots):
        ax.plot(mcm.w, v, '-', alpha=0.4, label=f"iterate {i}")
        # Update guess
        for i in range(n):
            v_next[i] = np.max(mcm.state_action_values(i, v))
        v[:] = v_next # copy contents into v

    ax.legend(loc='lower right')
```

Now let's create an instance of `McCallModel` and call the function:

```
In [9]: mcm = McCallModel()

fig, ax = plt.subplots()
ax.set_xlabel('wage')
ax.set_ylabel('value')
plot_value_function_seq(mcm, ax)
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.

We'll be using JIT compilation via Numba to turbocharge our loops.

```
In [10]: @jit(nopython=True)
def compute_reservation_wage(mcm,
                             max_iter=500,
                             tol=1e-6):

    # Simplify names
    c, β, w, q = mcm.c, mcm.β, mcm.w, mcm.q

    # == First compute the value function == #

    n = len(w)
    v = w / (1 - β)           # initial guess
    v_next = np.empty_like(v)
    i = 0
    error = tol + 1
    while i < max_iter and error > tol:

        for i in range(n):
            v_next[i] = np.max(mcm.state_action_values(i, v))

        error = np.max(np.abs(v_next - v))
        i += 1

    v[:] = v_next  # copy contents into v
```

```
# == Now compute the reservation wage == #

return (1 - β) * (c + β * np.sum(v * q))
```

The next line computes the reservation wage at the default parameters

In [11]: `compute_reservation_wage(mcm)`

Out[11]: 47.316499710024964

22.4.4 Comparative Statics

Now we know how to compute the reservation wage, let's see how it varies with parameters.

In particular, let's look at what happens when we change β and c .

```
In [12]: grid_size = 25
R = np.empty((grid_size, grid_size))

c_vals = np.linspace(10.0, 30.0, grid_size)
β_vals = np.linspace(0.9, 0.99, grid_size)

for i, c in enumerate(c_vals):
    for j, β in enumerate(β_vals):
        mcm = McCallModel(c=c, β=β)
        R[i, j] = compute_reservation_wage(mcm)
```

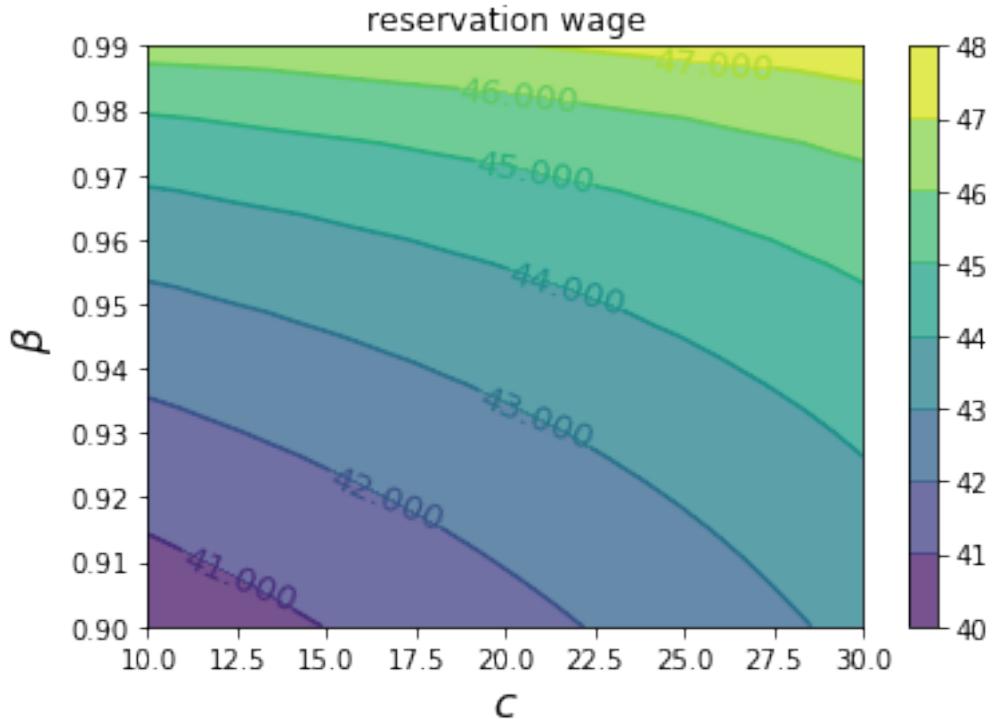
```
In [13]: fig, ax = plt.subplots()

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)

ax.set_title("reservation wage")
ax.set_xlabel("$c$", fontsize=16)
ax.set_ylabel("$\beta$", fontsize=16)

ax.ticklabel_format(useOffset=False)
plt.show()
```



As expected, the reservation wage increases both with patience and with unemployment compensation.

22.5 Computing the Optimal Policy: Take 2

The approach to dynamic programming just described is very standard and broadly applicable.

For this particular problem, there's also an easier way, which circumvents the need to compute the value function.

Let h denote the continuation value:

$$h = c + \beta \sum_{s'} v^*(s') q(s') \quad (6)$$

The Bellman equation can now be written as

$$v^*(s') = \max \left\{ \frac{w(s')}{1 - \beta}, h \right\}$$

Substituting this last equation into (6) gives

$$h = c + \beta \sum_{s' \in S} \max \left\{ \frac{w(s')}{1 - \beta}, h \right\} q(s') \quad (7)$$

This is a nonlinear equation 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

$$h' = c + \beta \sum_{s' \in \mathbb{S}} \max \left\{ \frac{w(s')}{1 - \beta}, h \right\} q(s') \quad (8)$$

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 .

Once again, one can 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 single number, rather than an n -vector.

Here's an implementation:

```
In [14]: @jit(nopython=True)
def compute_reservation_wage_two(mcm,
                                  max_iter=500,
                                  tol=1e-5):

    # Simplify names
    c, beta, w, q = mcm.c, mcm.beta, mcm.w, mcm.q

    # == First compute h == #

    h = np.sum(w * q) / (1 - beta)
    i = 0
    error = tol + 1
    while i < max_iter and error > tol:

        s = np.maximum(w / (1 - beta), h)
        h_next = c + beta * np.sum(s * q)

        error = np.abs(h_next - h)
        i += 1

        h = h_next

    # == Now compute the reservation wage == #

    return (1 - beta) * h
```

You can use this code to solve the exercise below.

22.6 Exercises

22.6.1 Exercise 1

Compute the average duration of unemployment when $\beta = 0.99$ and c takes the following values

```
c_vals = np.linspace(10, 40, 25)
```

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`.

22.6.2 Exercise 2

The purpose of this exercise is to show how to replace the discrete wage offer distribution used above with a continuous distribution.

This is a significant topic because many convenient distributions are continuous (i.e., have a density).

Fortunately, the theory changes little in our simple model.

Recall that h in (6) 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 (6) by

$$h = c + \beta \int v^*(s')q(s')ds'. \quad (9)$$

Equation (7) becomes

$$h = c + \beta \int \max \left\{ \frac{w(s')}{1 - \beta}, h \right\} q(s')ds' \quad (10)$$

The aim is to solve this nonlinear equation by iteration, and from it obtain the reservation wage.

Try to carry this out, setting

- the state sequence $\{s_t\}$ to be IID and standard normal and
- the wage function to be $w(s) = \exp(\mu + \sigma s)$.

You will need to implement a new version of the `McCallModel` class that assumes a lognormal wage distribution.

Calculate the integral by Monte Carlo, by averaging over a large number of wage draws.

For default parameters, use `c=25, beta=0.99, sigma=0.5, mu=2.5`.

Once your code is working, investigate how the reservation wage changes with c and β .

22.7 Solutions

22.7.1 Exercise 1

Here's one solution

```
In [15]: cdf = np.cumsum(q_default)

@jit(nopython=True)
def compute_stopping_time(w_bar, seed=1234):

    np.random.seed(seed)
    t = 1
    while True:
        # Generate a wage draw
        w = w_default[qe.random.draw(cdf)]
        # Stop when the draw is above the reservation wage
        if w >= w_bar:
            stopping_time = t
            break
        else:
            t += 1
    return stopping_time

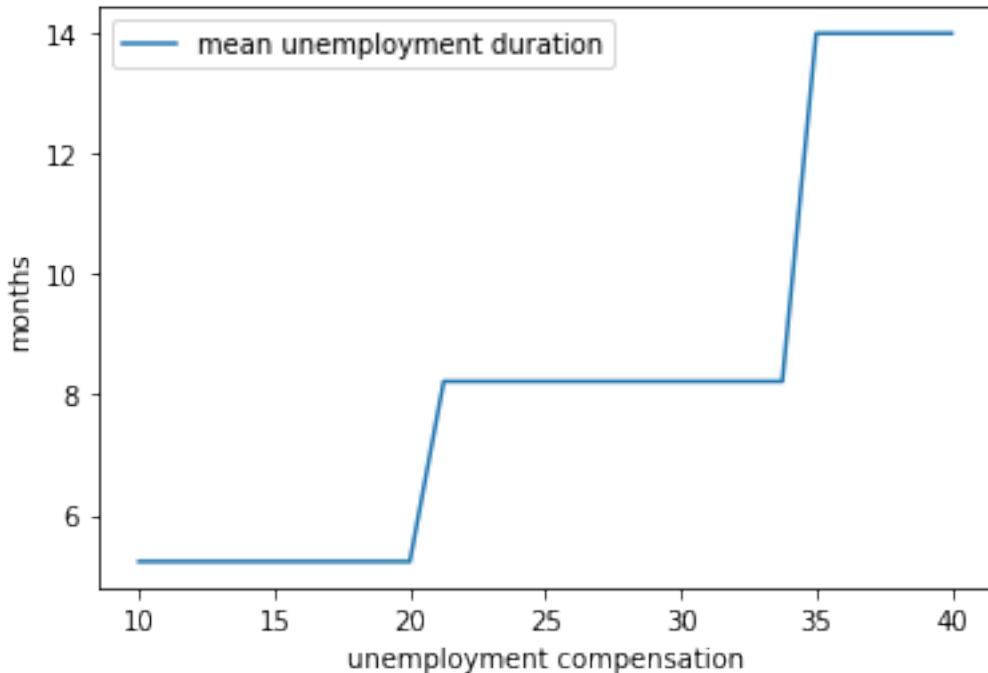
@jit(nopython=True)
def compute_mean_stopping_time(w_bar, num_reps=100000):
    obs = np.empty(num_reps)
    for i in range(num_reps):
        obs[i] = compute_stopping_time(w_bar, seed=i)
    return obs.mean()

c_vals = np.linspace(10, 40, 25)
stop_times = np.empty_like(c_vals)
for i, c in enumerate(c_vals):
    mcm = McCallModel(c=c)
    w_bar = compute_reservation_wage_two(mcm)
    stop_times[i] = compute_mean_stopping_time(w_bar)

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()
```



22.7.2 Exercise 2

```
In [16]: mcall_data_continuous = [
    ('c', float64),           # unemployment compensation
    ('β', float64),           # discount factor
    ('σ', float64),           # scale parameter in lognormal distribution
    ('μ', float64),           # location parameter in lognormal distribution
    ('w_draws', float64[:])   # draws of wages for Monte Carlo
]

@jitclass(mcall_data_continuous)
class McCallModelContinuous:

    def __init__(self, c=25, β=0.99, σ=0.5, μ=2.5, mc_size=1000):
        self.c, self.β, self.σ, self.μ = c, β, σ, μ

        # Draw and store shocks
        np.random.seed(1234)
        s = np.random.randn(mc_size)
        self.w_draws = np.exp(μ + σ * s)

    @jit(nopython=True)
    def compute_reservation_wage_continuous(mcmc, max_iter=500, tol=1e-5):

        c, β, σ, μ, w_draws = mcmc.c, mcmc.β, mcmc.σ, mcmc.μ, mcmc.w_draws

        h = np.mean(w_draws) / (1 - β) # initial guess
        i = 0
        error = tol + 1
        while i < max_iter and error > tol:
```

```

integral = np.mean(np.maximum(w_draws / (1 - β), h))
h_next = c + β * integral

error = np.abs(h_next - h)
i += 1

h = h_next

# == Now compute the reservation wage == #

return (1 - β) * h

```

Now we investigate how the reservation wage changes with c and β .

We will do this using a contour plot.

```

In [17]: grid_size = 25
R = np.empty((grid_size, grid_size))

c_vals = np.linspace(10.0, 30.0, grid_size)
β_vals = np.linspace(0.9, 0.99, grid_size)

for i, c in enumerate(c_vals):
    for j, β in enumerate(β_vals):
        mcmc = McCallModelContinuous(c=c, β=β)
        R[i, j] = compute_reservation_wage_continuous(mcmc)

```

```

In [18]: fig, ax = plt.subplots()

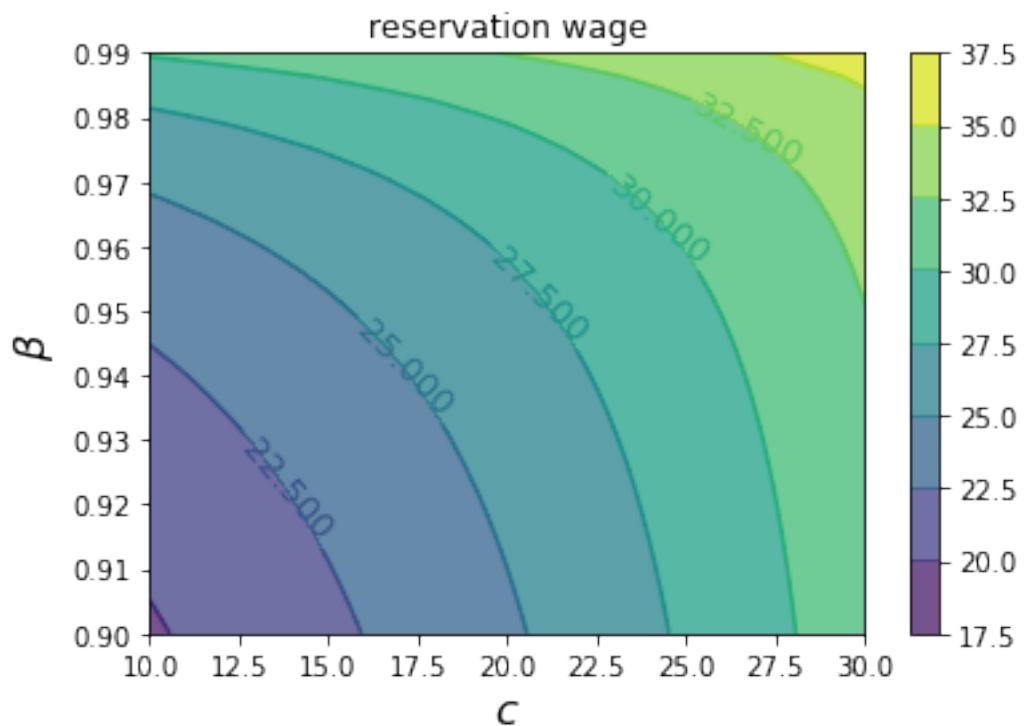
cs1 = ax.contourf(c_vals, β_vals, R.T, alpha=0.75)
ctr1 = ax.contour(c_vals, β_vals, R.T)

plt.clabel(ctr1, inline=1, fontsize=13)
plt.colorbar(cs1, ax=ax)

ax.set_title("reservation wage")
ax.set_xlabel("$c$", fontsize=16)
ax.set_ylabel("$\beta$", fontsize=16)

ax.ticklabel_format(useOffset=False)
plt.show()

```



Chapter 23

Job Search II: Search and Separation

23.1 Contents

- Overview 23.2
- The Model 23.3
- Solving the Model 23.4
- Implementation 23.5
- Impact of Parameters 23.6
- Exercises 23.7
- Solutions 23.8

In addition to what's in Anaconda, this lecture will need the following libraries:

```
In [1]: !pip install quantecon
```

23.2 Overview

Previously we looked at the McCall job search model [80] as a way of understanding unemployment and worker decisions.

One unrealistic feature of the model is that every job is permanent.

In this lecture, we extend the McCall 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

```
In [2]: import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

```

from numba import njit, float64
from numba.experimental import jitclass
from quantecon.distributions import BetaBinomial

/home/ubuntu/anaconda3/lib/python3.7/site-packages/numba/np/ufunc/parallel.py:
  ↪355:
NumbaWarning: The TBB threading layer requires TBB version 2019.5 or later i.e.,
TBB_INTERFACE_VERSION >= 11005. Found TBB_INTERFACE_VERSION = 11004. The TBB
  ↪threading
layer is disabled.
warnings.warn(problem)

```

23.3 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

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t u(y_t) \quad (1)$$

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 $u'' < 0$.

23.3.1 The Wage Process

For now we will drop the separation of state process and wage process that we maintained for the [baseline model](#).

In particular, we simply suppose that wage offers $\{w_t\}$ are IID with common distribution q .

The set of possible wage values is denoted by \mathbb{W} .

(Later we will go back to having a separate state process $\{s_t\}$ driving random outcomes, since this formulation is usually convenient in more sophisticated models.)

23.3.2 Timing and Decisions

At the start of each period, the agent can be either

- unemployed or

- employed at some existing wage level w_e .

At the start of a given period, the current wage offer w_t is observed.

If currently *employed*, the worker

1. receives utility $u(w_e)$ and
2. is fired with some (small) probability α .

If currently *unemployed*, the worker either accepts or rejects the current offer w_t .

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](#).)

23.4 Solving the Model

We drop time subscripts in what follows and primes denote next period values.

Let

- $v(w_e)$ be total lifetime value accruing to a worker who enters the current period *employed* with existing wage w_e
- $h(w)$ be total lifetime value accruing to a worker who enters the current period *unemployed* and receives wage offer w .

Here *value* means the value of the objective function (1) when the worker makes optimal decisions at all future points in time.

Our first aim is to obtain these functions.

23.4.1 The Bellman Equations

Suppose for now that the worker can calculate the functions v and h and use them in his decision making.

Then v and h should satisfy

$$v(w_e) = u(w_e) + \beta \left[(1 - \alpha)v(w_e) + \alpha \sum_{w' \in \mathbb{W}} h(w')q(w') \right] \quad (2)$$

and

$$h(w) = \max \left\{ v(w), u(c) + \beta \sum_{w' \in \mathbb{W}} h(w')q(w') \right\} \quad (3)$$

Equation (2) expresses the value of being employed at wage w_e in terms of

- current reward $u(w_e)$ plus

- discounted expected reward tomorrow, given the α probability of being fired

Equation (3) 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.

Accepting transitions the worker to employment and hence yields reward $v(w)$.

Rejecting leads to unemployment compensation and unemployment tomorrow.

Equations (2) and (3) are the Bellman equations for this model.

They provide enough information to solve for both v and h .

23.4.2 A Simplifying Transformation

Rather than jumping straight into solving these equations, let's see if we can simplify them somewhat.

(This process will be analogous to our [second pass](#) at the plain vanilla McCall model, where we simplified the Bellman equation.)

First, let

$$d := \sum_{w' \in \mathbb{W}} h(w') q(w') \quad (4)$$

be the expected value of unemployment tomorrow.

We can now write (3) as

$$h(w) = \max \{v(w), u(c) + \beta d\}$$

or, shifting time forward one period

$$\sum_{w' \in \mathbb{W}} h(w') q(w') = \sum_{w' \in \mathbb{W}} \max \{v(w'), u(c) + \beta d\} q(w')$$

Using (4) again now gives

$$d = \sum_{w' \in \mathbb{W}} \max \{v(w'), u(c) + \beta d\} q(w') \quad (5)$$

Finally, (2) can now be rewritten as

$$v(w) = u(w) + \beta [(1 - \alpha)v(w) + \alpha d] \quad (6)$$

In the last expression, we wrote w_e as w to make the notation simpler.

23.4.3 The Reservation Wage

Suppose we can use (5) and (6) to solve for d and v .

(We will do this soon.)

We can then determine optimal behavior for the worker.

From (3), we see that an unemployed agent accepts current offer w if $v(w) \geq u(c) + \beta d$.

This means precisely that the value of accepting is higher than the expected value of rejecting.

It is clear that v is (at least weakly) increasing in w , since the agent is never made worse off by a higher wage offer.

Hence, we can express the optimal choice as accepting wage offer w if and only if

$$w \geq \bar{w} \quad \text{where } \bar{w} \text{ solves } v(\bar{w}) = u(c) + \beta d$$

23.4.4 Solving the Bellman Equations

We'll use the same iterative approach to solving the Bellman equations that we adopted in the [first job search lecture](#).

Here this amounts to

1. make guesses for d and v
2. plug these guesses into the right-hand sides of (5) and (6)
3. update the left-hand sides from this rule and then repeat

In other words, we are iterating using the rules

$$d_{n+1} = \sum_{w' \in \mathbb{W}} \max \{v_n(w'), u(c) + \beta d_n\} q(w') \quad (7)$$

$$v_{n+1}(w) = u(w) + \beta [(1 - \alpha)v_n(w) + \alpha d_n] \quad (8)$$

starting from some initial conditions d_0, v_0 .

As before, the system always converges to the true solutions—in this case, the v and d that solve (5) and (6).

(A proof can be obtained via the Banach contraction mapping theorem.)

23.5 Implementation

Let's implement this iterative process.

In the code, you'll see that we use a class to store the various parameters and other objects associated with a given model.

This helps to tidy up the code and provides an object that's easy to pass to functions.

The default utility function is a CRRA utility function

```
In [3]: @njit
def u(c, σ=2.0):
    return (c**(1 - σ) - 1) / (1 - σ)
```

Also, here's a default wage distribution, based around the BetaBinomial distribution:

```
In [4]: n = 60                                # n possible outcomes for w
w_default = np.linspace(10, 20, n)          # wages between 10 and 20
a, b = 600, 400                             # shape parameters
dist = BetaBinomial(n-1, a, b)
q_default = dist.pdf()
```

Here's our jitted class for the McCall model with separation.

```
In [5]: mccall_data = [
    ('α', float64),      # job separation rate
    ('β', float64),      # discount factor
    ('c', float64),      # unemployment compensation
    ('w', float64[:]),   # list of wage values
    ('q', float64[:])    # pmf of random variable w
]

@jitclass(mccall_data)
class McCallModel:
    """
    Stores the parameters and functions associated with a given model.
    """

    def __init__(self, α=0.2, β=0.98, c=6.0, w=w_default, q=q_default):
        self.α, self.β, self.c, self.w, self.q = α, β, c, w, q

    def update(self, v, d):
        α, β, c, w, q = self.α, self.β, self.c, self.w, self.q
        v_new = np.empty_like(v)

        for i in range(len(w)):
            v_new[i] = u(w[i]) + β * ((1 - α) * v[i] + α * d)

        d_new = np.sum(np.maximum(v, u(c) + β * d) * q)

        return v_new, d_new
```

Now we iterate until successive realizations are closer together than some small tolerance level.

We then return the current iterate as an approximate solution.

```
In [6]: @njit
def solve_model(mcm, tol=1e-5, max_iter=2000):
    """
    Iterates to convergence on the Bellman equations
    * mcm is an instance of McCallModel
    """

    v = np.ones_like(mcm.w)      # Initial guess of v
```

```

d = 1                      # Initial guess of d
i = 0
error = tol + 1

while error > tol and i < max_iter:
    v_new, d_new = mcm.update(v, d)
    error_1 = np.max(np.abs(v_new - v))
    error_2 = np.abs(d_new - d)
    error = max(error_1, error_2)
    v = v_new
    d = d_new
    i += 1

return v, d

```

23.5.1 The Reservation Wage: First Pass

The optimal choice of the agent is summarized by the reservation wage.

As discussed above, the reservation wage is the \bar{w} that solves $v(\bar{w}) = h$ where $h := u(c) + \beta d$ is the continuation value.

Let's compare v and h to see what they look like.

We'll use the default parameterizations found in the code above.

```

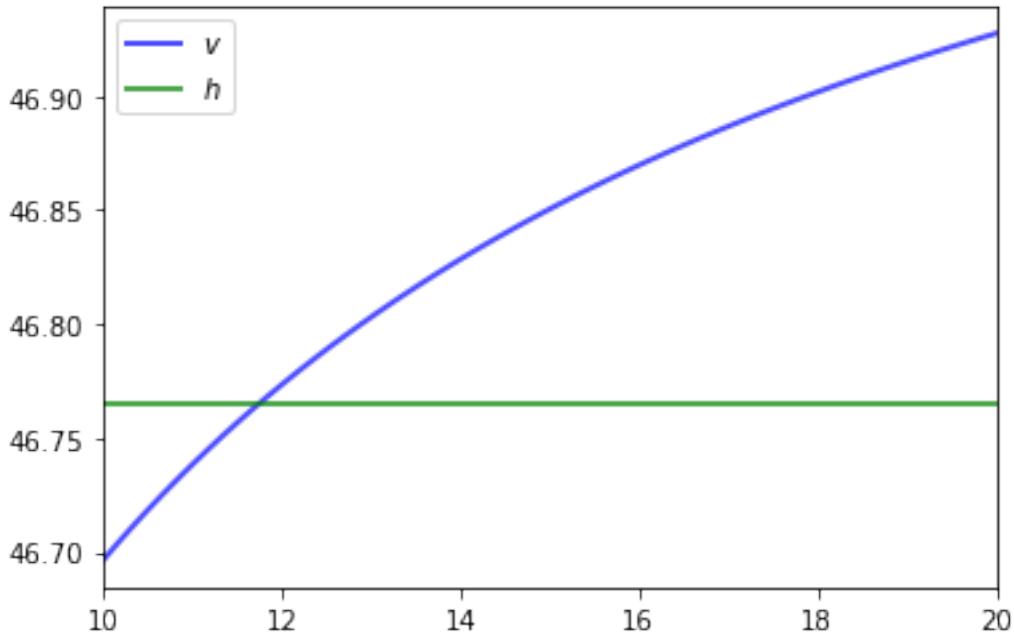
In [7]: mcm = McCallModel()
v, d = solve_model(mcm)
h = u(mcm.c) + mcm.beta * d

fig, ax = plt.subplots()

ax.plot(mcm.w, v, 'b-', lw=2, alpha=0.7, label='$v$')
ax.plot(mcm.w, [h] * len(mcm.w),
        'g-', lw=2, alpha=0.7, label='$h$')
ax.set_xlim(min(mcm.w), max(mcm.w))
ax.legend()

plt.show()

```



The value v is increasing because higher w generates a higher wage flow conditional on staying employed.

23.5.2 The Reservation Wage: Computation

Here's a function `compute_reservation_wage` that takes an instance of `McCallModel` and returns the associated reservation wage.

```
In [8]: @njit
def compute_reservation_wage(mcm):
    """
    Computes the reservation wage of an instance of the McCall model
    by finding the smallest w such that v(w) >= h.

    If no such w exists, then w_bar is set to np.inf.
    """

    v, d = solve_model(mcm)
    h = u(mcm.c) + mcm.beta * d

    w_bar = np.inf
    for i, wage in enumerate(mcm.w):
        if v[i] > h:
            w_bar = wage
            break

    return w_bar
```

Next we will investigate how the reservation wage varies with parameters.

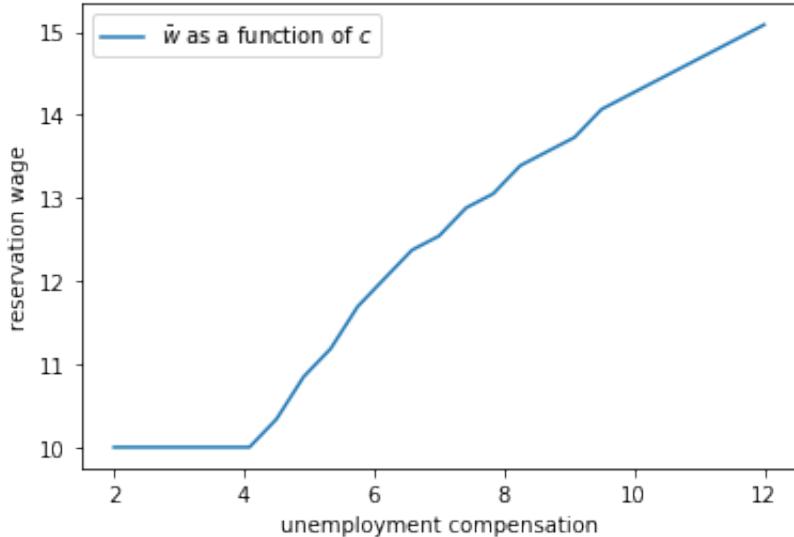
23.6 Impact of Parameters

In each instance below, we'll show you a figure and then ask you to reproduce it in the exercises.

23.6.1 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 `McCallModel` class, apart from c (which takes the values given on the horizontal axis)



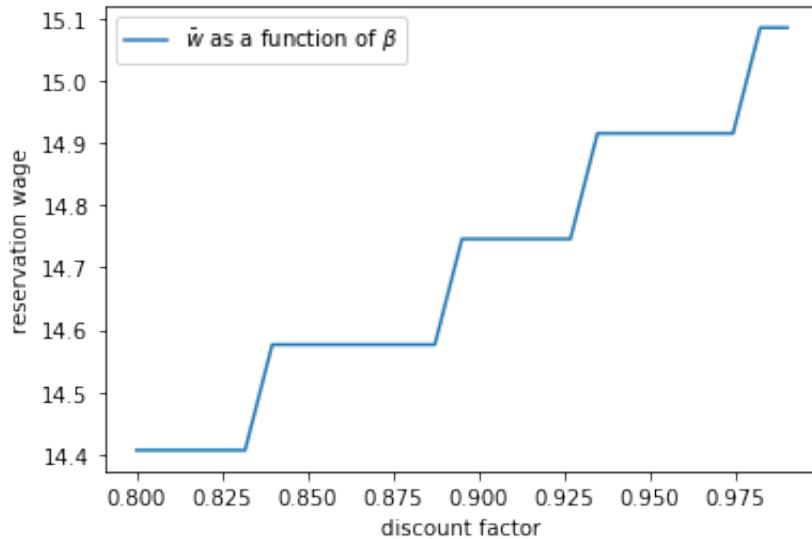
As expected, higher unemployment compensation causes the worker to hold out for higher wages.

In effect, the cost of continuing job search is reduced.

23.6.2 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

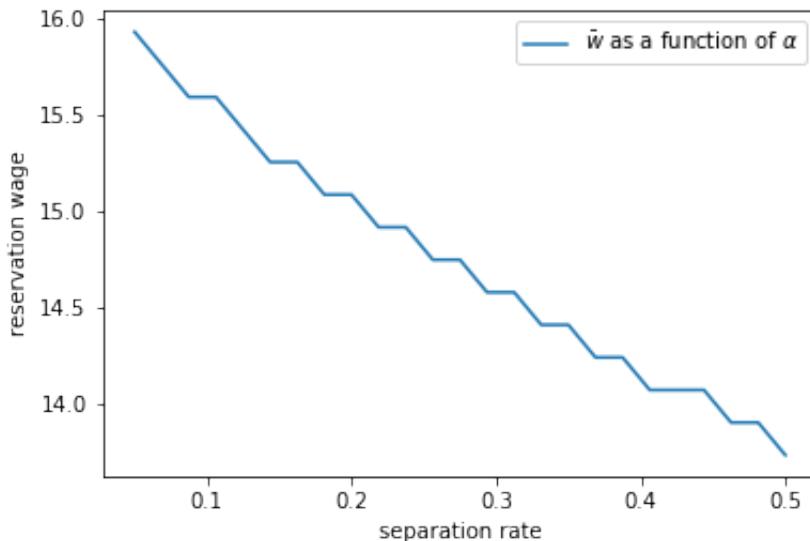


Again, the results are intuitive: More patient workers will hold out for higher wages.

23.6.3 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.

23.7 Exercises

23.7.1 Exercise 1

Reproduce all the reservation wage figures shown above.

Regarding the values on the horizontal axis, use

```
In [9]: grid_size = 25
c_vals = np.linspace(2, 12, grid_size)      # unemployment compensation
beta_vals = np.linspace(0.8, 0.99, grid_size) # discount factors
alpha_vals = np.linspace(0.05, 0.5, grid_size) # separation rate
```

23.8 Solutions

23.8.1 Exercise 1

Here's the first figure.

```
In [10]: mcm = McCallModel()

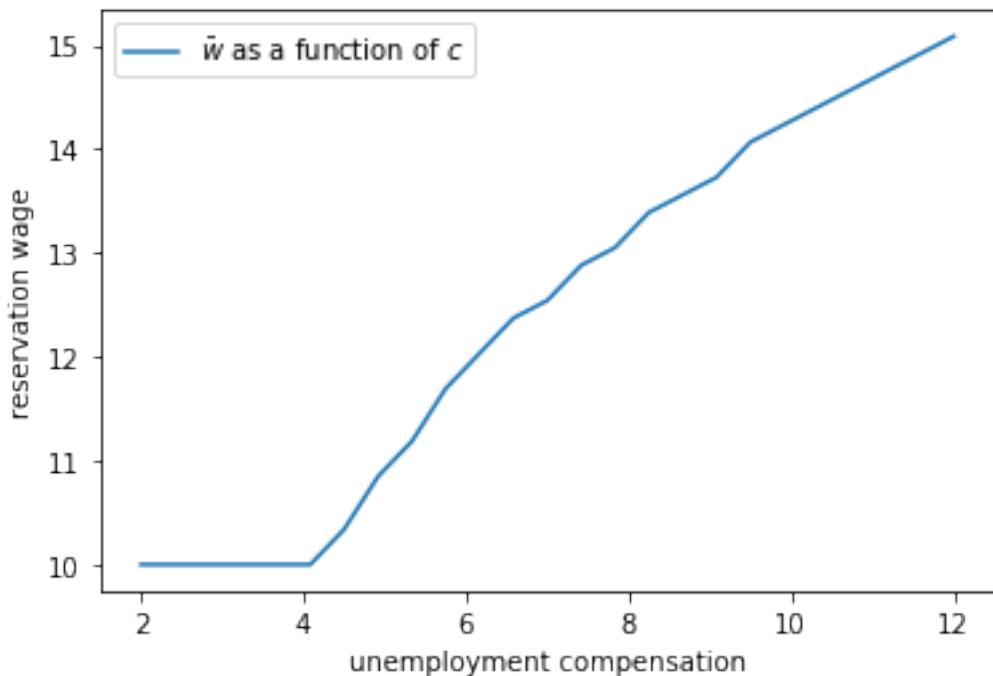
w_bar_vals = np.empty_like(c_vals)

fig, ax = plt.subplots()

for i, c in enumerate(c_vals):
    mcm.c = c
    w_bar = compute_reservation_wage(mcm)
    w_bar_vals[i] = w_bar

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()
```



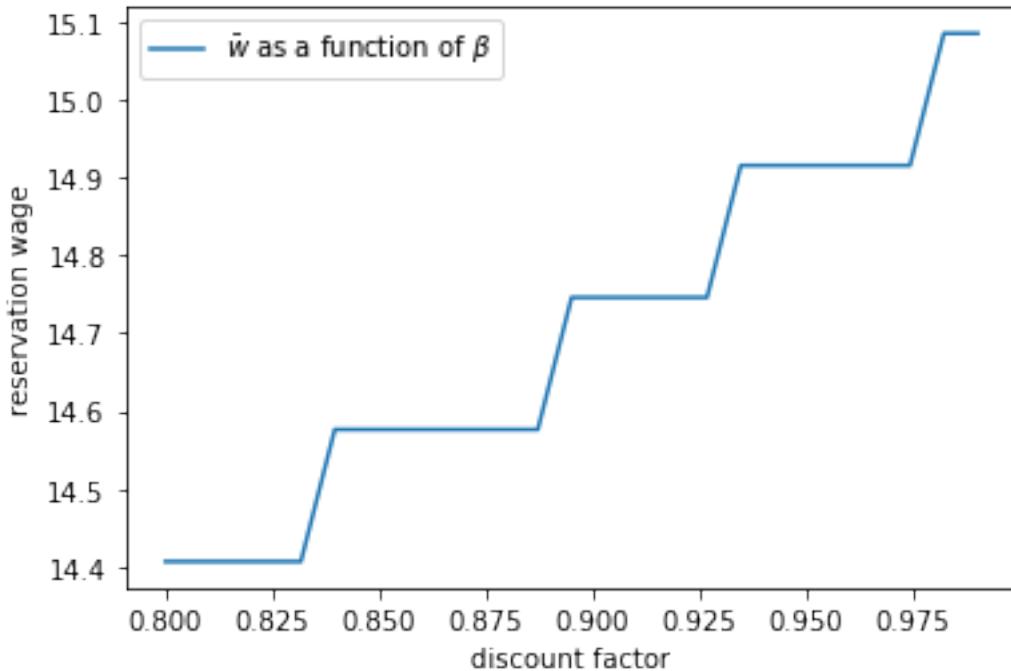
Here's the second one.

```
In [11]: fig, ax = plt.subplots()

for i, beta in enumerate(beta_vals):
    mcm.beta = beta
    w_bar = compute_reservation_wage(mcm)
    w_bar_vals[i] = w_bar

ax.set(xlabel='discount factor', ylabel='reservation wage')
ax.plot(beta_vals, w_bar_vals, label=r'$\bar{w}$ as a function of $\beta$')
ax.legend()

plt.show()
```

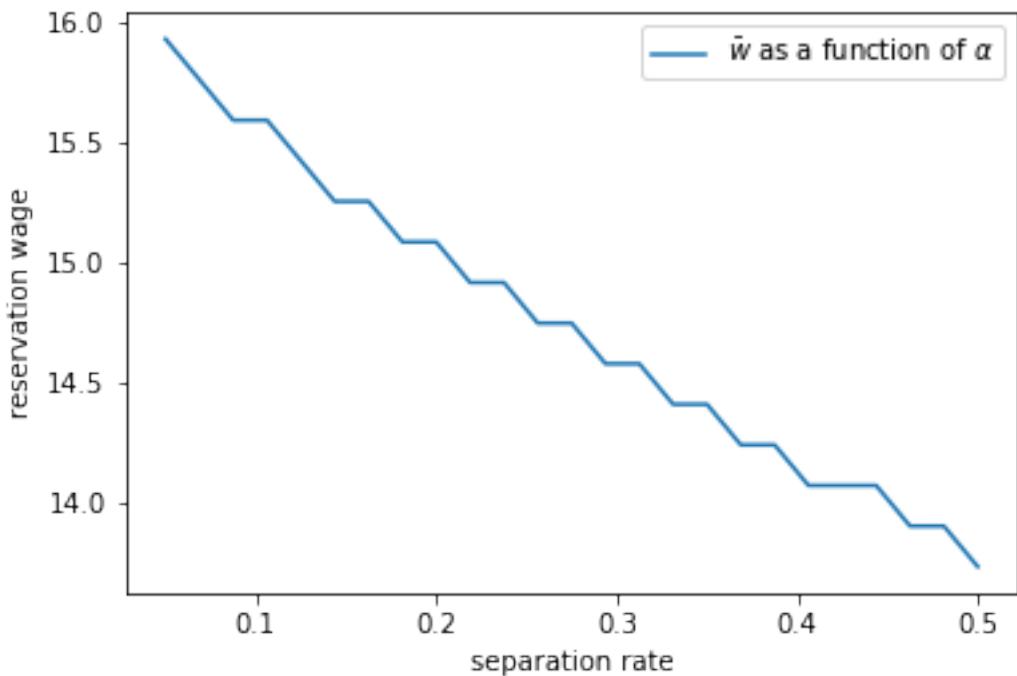


Here's the third.

```
In [12]: fig, ax = plt.subplots()

for i, alpha in enumerate(alpha_vals):
    mcm.alpha = alpha
    w_bar = compute_reservation_wage(mcm)
    w_bar_vals[i] = w_bar

ax.set(xlabel='separation rate', ylabel='reservation wage')
ax.plot(alpha_vals, w_bar_vals, label=r'$\bar{w}$ as a function of $\alpha$')
ax.legend()
plt.show()
```



Chapter 24

Job Search III: Fitted Value Function Iteration

24.1 Contents

- Overview [24.2](#)
- The Algorithm [24.3](#)
- Implementation [24.4](#)
- Exercises [24.5](#)
- Solutions [24.6](#)

In addition to what's in Anaconda, this lecture will need the following libraries:

```
In [1]: !pip install quantecon  
!pip install interpolation
```

24.2 Overview

In this lecture we again study the [McCall job search model with separation](#), but now with a continuous wage distribution.

While we already considered continuous wage distributions briefly in the exercises of the [first job search lecture](#), the change was relatively trivial in that case.

This is because we were able to reduce the problem to solving for a single scalar value (the continuation value).

Here, with separation, 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.

We will use the following imports:

```
In [2]: import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

import quantecon as qe
from interpolation import interp
from numpy.random import randn
from numba import njit, prange, float64, int32
from numba.experimental import jitclass

/home/ubuntu/anaconda3/lib/python3.7/site-packages/numba/np/ufunc/parallel.py:
 355:
NumbaWarning: The TBB threading layer requires TBB version 2019.5 or later i.e.,
TBB_INTERFACE_VERSION >= 11005. Found TBB_INTERFACE_VERSION = 11004. The TBB
  ↪threading
layer is disabled.
warnings.warn(problem)
```

24.3 The Algorithm

The model is the same as the McCall model with job separation we [studied before](#), except that the wage offer distribution is continuous.

We are going to start with the two Bellman equations we obtained for the model with job separation after [a simplifying transformation](#).

Modified to accommodate continuous wage draws, they take the following form:

$$d = \int \max \{v(w'), u(c) + \beta d\} q(w') dw' \quad (1)$$

and

$$v(w) = u(w) + \beta [(1 - \alpha)v(w) + \alpha d] \quad (2)$$

The unknowns here are the function v and the scalar d .

The difference between these and the pair of Bellman equations we previously worked on are

1. in (1), what used to be a sum over a finite number of wage values is an integral over an infinite set.
2. The function v in (2) is defined over all $w \in \mathbb{R}_+$.

The function q in (1) is the density of the wage offer distribution.

Its support is taken as equal to \mathbb{R}_+ .

24.3.1 Value Function Iteration

In theory, we should now proceed as follows:

1. Begin with a guess v, d for the solutions to (1)–(2).
2. Plug v, d into the right hand side of (1)–(2) and compute the left hand side to obtain updates v', d'
3. Unless some stopping condition is satisfied, set $(v, d) = (v', d')$ 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 (2).

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.

24.3.2 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 $w_1 < w_2 < \dots < w_I$ and then reconstruct v' from this information when required.

More precisely, the algorithm will be

1. Begin with an array \mathbf{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 \mathbf{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., [44] or [100]) and

2. preserves useful shape properties such as monotonicity and concavity/convexity.

Linear interpolation will be implemented using a JIT-aware Python interpolation library called `interpolation.py`.

The next figure illustrates piecewise linear interpolation of an arbitrary function on grid points 0, 0.2, 0.4, 0.6, 0.8, 1.

```
In [3]: def f(x):
    y1 = 2 * np.cos(6 * x) + np.sin(14 * x)
    return y1 + 2.5

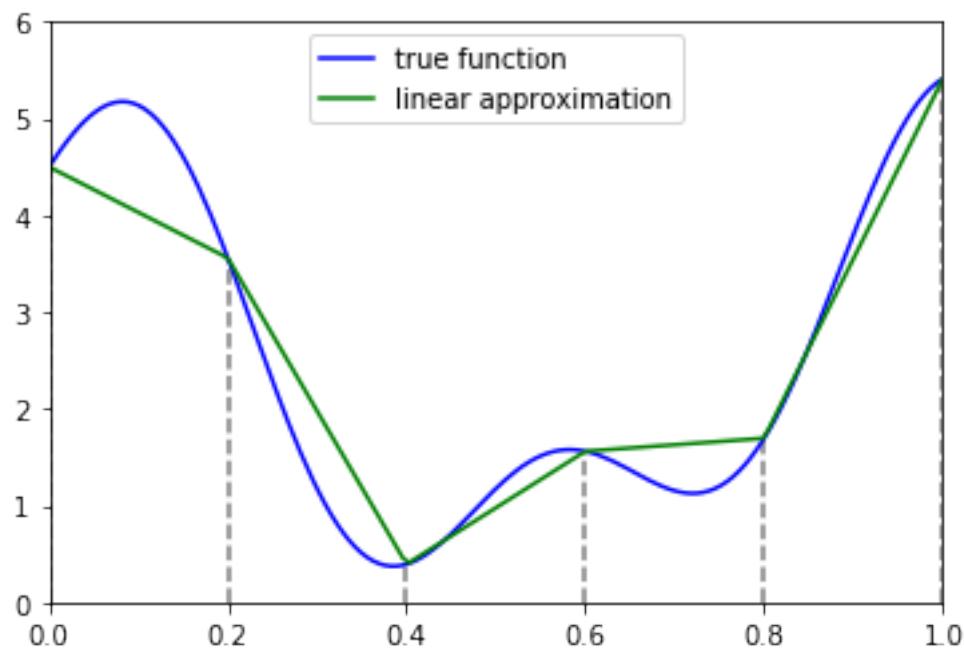
c_grid = np.linspace(0, 1, 6)
f_grid = np.linspace(0, 1, 150)

def Af(x):
    return interp(c_grid, f(c_grid), x)

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()
```



24.4 Implementation

The first step is to build a jitted class for the McCall model with separation and a continuous wage offer distribution.

We will take the utility function to be the log function for this application, with $u(c) = \ln c$.

We will adopt the lognormal distribution for wages, with $w = \exp(\mu + \sigma z)$ when z is standard normal and μ, σ are parameters.

```
In [4]: @njit
def lognormal_draws(n=1000, mu=2.5, sigma=0.5, seed=1234):
    np.random.seed(seed)
    z = np.random.randn(n)
    w_draws = np.exp(mu + sigma * z)
    return w_draws
```

Here's our class.

```
In [5]: mccall_data_continuous = [
    ('c', float64),                      # unemployment compensation
    ('alpha', float64),                   # job separation rate
    ('beta', float64),                   # discount factor
    ('sigma', float64),                  # scale parameter in lognormal distribution
    ('mu', float64),                     # location parameter in lognormal distribution
    ('w_grid', float64[:]),              # grid of points for fitted VFI
    ('w_draws', float64[:])              # draws of wages for Monte Carlo
]

@jitclass(mccall_data_continuous)
class McCallModelContinuous:

    def __init__(self,
                 c=1,
                 alpha=0.1,
                 beta=0.96,
                 grid_min=1e-10,
                 grid_max=5,
                 grid_size=100,
                 w_draws=lognormal_draws()):
        self.c, self.alpha, self.beta = c, alpha, beta
        self.w_grid = np.linspace(grid_min, grid_max, grid_size)
        self.w_draws = w_draws

    def update(self, v, d):

        # Simplify names
        c, alpha, beta, sigma, mu = self.c, self.alpha, self.beta, self.sigma, self.mu
        w = self.w_grid
        u = lambda x: np.log(x)

        # Interpolate array represented value function
        vf = lambda x: interp(w, v, x)

        # Update d using Monte Carlo to evaluate integral
```

```

d_new = np.mean(np.maximum(vf(self.w_draws), u(c) + β * d))

# Update v
v_new = u(w) + β * ((1 - α) * v + α * d)

return v_new, d_new

```

We then return the current iterate as an approximate solution.

```

In [6]: @njit
def solve_model(mcm, tol=1e-5, max_iter=2000):
    """
    Iterates to convergence on the Bellman equations

    * mcm is an instance of McCallModel
    """

    v = np.ones_like(mcm.w_grid)      # Initial guess of v
    d = 1                            # Initial guess of d
    i = 0
    error = tol + 1

    while error > tol and i < max_iter:
        v_new, d_new = mcm.update(v, d)
        error_1 = np.max(np.abs(v_new - v))
        error_2 = np.abs(d_new - d)
        error = max(error_1, error_2)
        v = v_new
        d = d_new
        i += 1

    return v, d

```

Here's a function `compute_reservation_wage` that takes an instance of `McCallModelContinuous` and returns the associated reservation wage.

If $v(w) < h$ for all w , then the function returns `np.inf`

```

In [7]: @njit
def compute_reservation_wage(mcm):
    """
    Computes the reservation wage of an instance of the McCall model
    by finding the smallest w such that v(w) >= h.

    If no such w exists, then w_bar is set to np.inf.
    """

    u = lambda x: np.log(x)

    v, d = solve_model(mcm)
    h = u(mcm.c) + mcm.β * d

    w_bar = np.inf
    for i, wage in enumerate(mcm.w_grid):
        if v[i] > h:
            w_bar = wage
            break

```

```
return w_bar
```

The exercises ask you to explore the solution and how it changes with parameters.

24.5 Exercises

24.5.1 Exercise 1

Use the code above to explore what happens to the reservation wage when the wage parameter μ changes.

Use the default parameters and μ in `mu_vals = np.linspace(0.0, 2.0, 15)`.

Is the impact on the reservation wage as you expected?

24.5.2 Exercise 2

Let us now consider how the agent responds to an increase in volatility.

To try to understand this, compute the reservation wage when the wage offer distribution is uniform on $(m - s, m + s)$ and s varies.

The idea here is that we are holding the mean constant and spreading the support.

(This is a form of *mean-preserving spread*.)

Use `s_vals = np.linspace(1.0, 2.0, 15)` and `m = 2.0`.

State how you expect the reservation wage to vary with s .

Now compute it. Is this as you expected?

24.6 Solutions

24.6.1 Exercise 1

Here is one solution.

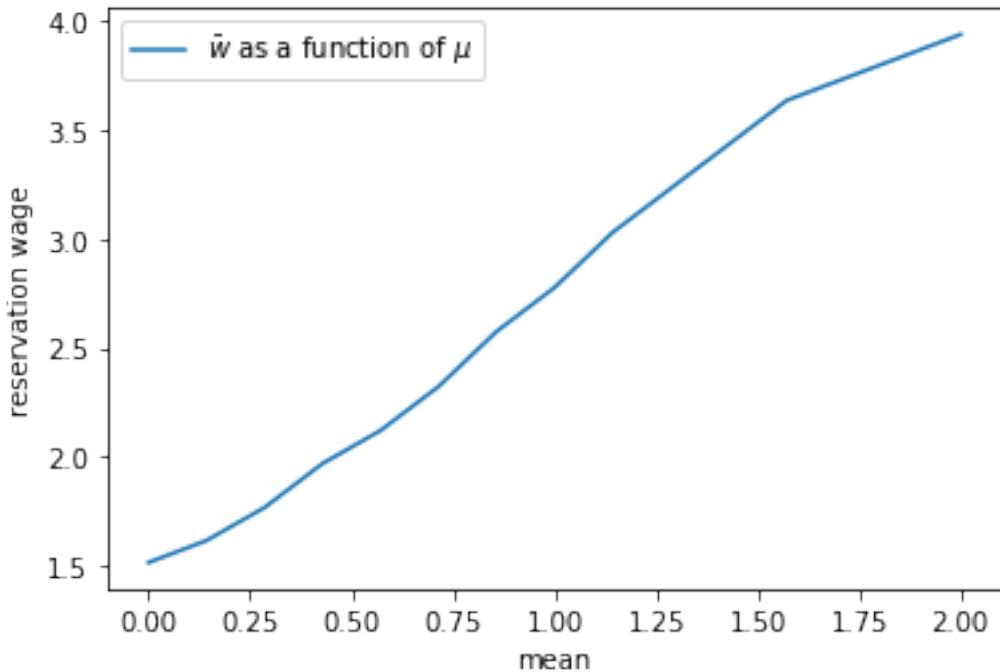
```
In [8]: mcm = McCallModelContinuous()
mu_vals = np.linspace(0.0, 2.0, 15)
w_bar_vals = np.empty_like(mu_vals)

fig, ax = plt.subplots()

for i, m in enumerate(mu_vals):
    mcm.w_draws = lognormal_draws(mu=m)
    w_bar = compute_reservation_wage(mcm)
    w_bar_vals[i] = w_bar

ax.set(xlabel='mean', ylabel='reservation wage')
ax.plot(mu_vals, w_bar_vals, label=r'$\bar{w}$ as a function of $\mu$')
ax.legend()
```

```
plt.show()
```



Not surprisingly, the agent is more inclined to wait when the distribution of offers shifts to the right.

24.6.2 Exercise 2

Here is one solution.

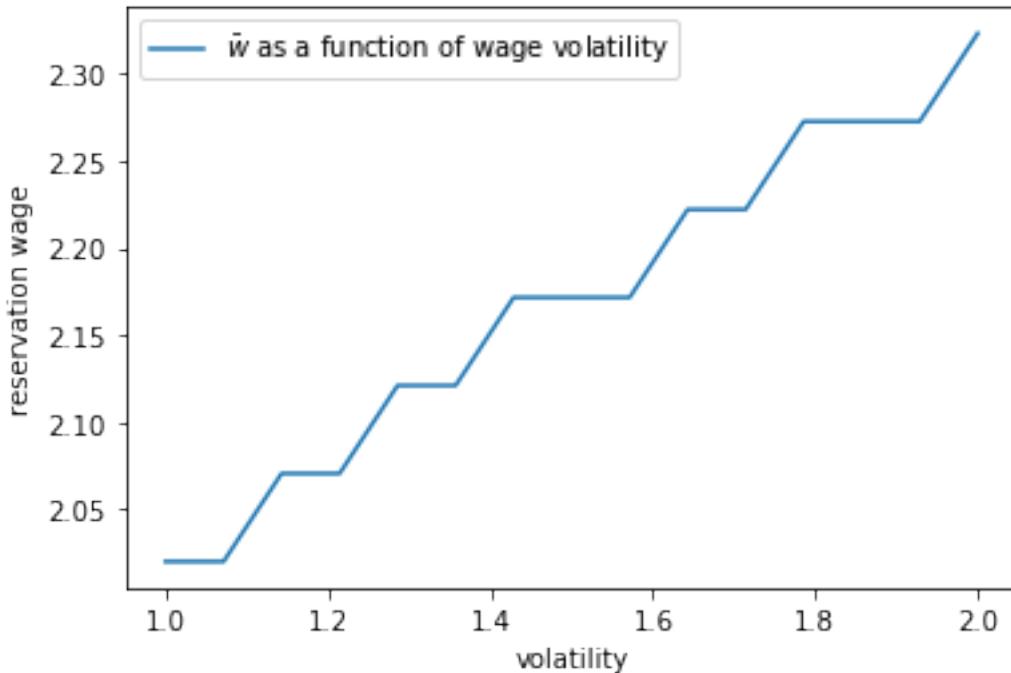
```
In [9]: mcm = McCallModelContinuous()
s_vals = np.linspace(1.0, 2.0, 15)
m = 2.0
w_bar_vals = np.empty_like(s_vals)

fig, ax = plt.subplots()

for i, s in enumerate(s_vals):
    a, b = m - s, m + s
    mcm.w_draws = np.random.uniform(low=a, high=b, size=10_000)
    w_bar = compute_reservation_wage(mcm)
    w_bar_vals[i] = w_bar

    ax.set(xlabel='volatility', ylabel='reservation wage')
    ax.plot(s_vals, w_bar_vals, label=r'$\bar{w}$ as a function of wage')
    ax.legend()

plt.show()
```



The reservation wage increases with volatility.

One might think that higher volatility would make the agent more inclined to take a given offer, since doing so represents certainty and waiting represents risk.

But job search is like holding an option: the worker is only exposed to upside risk (since, in a free market, no one can force them to take a bad offer).

More volatility means higher upside potential, which encourages the agent to wait.

Chapter 25

Job Search IV: Correlated Wage Offers

25.1 Contents

- Overview 25.2
- The Model 25.3
- Implementation 25.4
- Unemployment Duration 25.5
- Exercises 25.6
- Solutions 25.7

In addition to what's in Anaconda, this lecture will need the following libraries:

```
In [1]: !pip install quantecon  
!pip install interpolation
```

25.2 Overview

In this lecture we solve a [McCall style job search model](#) with persistent and transitory components to wages.

In other words, we relax the unrealistic assumption that randomness in wages is independent over time.

At the same time, we will go back to assuming that jobs are permanent and no separation occurs.

This is to keep the model relatively simple as we study the impact of correlation.

We will use the following imports:

```
In [2]: import numpy as np  
import matplotlib.pyplot as plt  
%matplotlib inline  
  
import quantecon as qe  
from interpolation import interp  
from numpy.random import randn
```

```

from numba import njit, prange, float64
from numba.experimental import jitclass

/home/ubuntu/anaconda3/lib/python3.7/site-packages/numba/np/ufunc/parallel.py:
  ↵355:
NumbaWarning: The TBB threading layer requires TBB version 2019.5 or later i.e.,
TBB_INTERFACE_VERSION >= 11005. Found TBB_INTERFACE_VERSION = 11004. The TBB
  ↵threading
layer is disabled.
warnings.warn(problem)

```

25.3 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 express, u is a utility function and \mathbb{E}_z is 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.

25.3.1 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 $u(c) = \ln(c)$.

The reservation wage is the wage where equality holds in the last expression.

That is,

$$\bar{w}(z) := \exp(f^*(z)(1 - \beta)) \tag{1}$$

Our main aim is to solve for the reservation rule and study its properties and implications.

25.4 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.

The following list helps Numba by providing some type information about the data we will work with.

```
In [3]: job_search_data = [
    ('μ', float64),                      # transient shock log mean
    ('σ', float64),                      # transient shock log variance
    ('d', float64),                      # shift coefficient of persistent state
    ('ρ', float64),                      # correlation coefficient of persistent state
    ('σ', float64),                      # state volatility
    ('β', float64),                      # discount factor
    ('c', float64),                      # unemployment compensation
    ('z_grid', float64[:]),              # grid over the state space
    ('e_draws', float64[:, :])           # Monte Carlo draws for integration
]
```

Here's a class that stores the data and the right hand side of the Bellman equation.

Default parameter values are embedded in the class.

```
In [4]: @jitclass(job_search_data)
class JobSearch:

    def __init__(self,
                 μ=0.0,                      # transient shock log mean
                 σ=1.0,                       # transient shock log variance
                 d=0.0,                       # shift coefficient of persistent state
                 ρ=0.9,                       # correlation coefficient of persistent state
                 σ=0.1,                       # state volatility
                 β=0.98,                      # discount factor
                 c=5,                          # unemployment compensation
                 mc_size=1000,
                 grid_size=100):

        self.μ, self.σ, self.d, = μ, σ, d,
        self.ρ, self.σ, self.β, self.c = ρ, σ, β, c

        # Set up grid
        z_mean = d / (1 - ρ)
        z_sd = np.sqrt(σ / (1 - ρ**2))
        k = 3 # std devs from mean
        a, b = z_mean - k * z_sd, z_mean + k * z_sd
        self.z_grid = np.linspace(a, b, grid_size)

        # Draw and store shocks
        np.random.seed(1234)
        self.e_draws = randn(2, mc_size)

    def parameters(self):
        """
        Return all parameters as a tuple.
        """
        return self.μ, self.σ, self.d, \
               self.ρ, self.σ, self.β, self.c
```

Next we implement the Q operator.

```
In [5]: @njit(parallel=True)
def Q(js, f_in, f_out):
    """
```

Apply the operator Q .

```
* js is an instance of JobSearch
* f_in and f_out are arrays that represent f and Qf respectively

"""

μ, s, d, ρ, σ, β, c = js.parameters()
M = js.e_draws.shape[1]

for i in prange(len(js.z_grid)):
    z = js.z_grid[i]
    expectation = 0.0
    for m in range(M):
        e1, e2 = js.e_draws[:, m]
        z_next = d + ρ * z + σ * e1
        go_val = interp(js.z_grid, f_in, z_next)      # f(z')
        y_next = np.exp(μ + s * e2)                    # y' draw
        w_next = np.exp(z_next) + y_next               # w' draw
        stop_val = np.log(w_next) / (1 - β)
        expectation += max(stop_val, go_val)
    expectation = expectation / M
    f_out[i] = np.log(c) + β * expectation
```

Here's a function to compute an approximation to the fixed point of Q .

```
In [6]: def compute_fixed_point(js,
                               use_parallel=True,
                               tol=1e-4,
                               max_iter=1000,
                               verbose=True,
                               print_skip=25):

    f_init = np.log(js.c) * np.ones(len(js.z_grid))
    f_out = np.empty_like(f_init)

    # Set up loop
    f_in = f_init
    i = 0
    error = tol + 1

    while i < max_iter and error > tol:
        Q(js, f_in, f_out)
        error = np.max(np.abs(f_in - f_out))
        i += 1
        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")
        f_in[:] = f_out

    if i == max_iter:
        print("Failed to converge!")

    if verbose and i < max_iter:
        print(f"\nConverged in {i} iterations.")

return f_out
```

Let's try generating an instance and solving the model.

In [7]: `js = JobSearch()`

```
qe.tic()
f_star = compute_fixed_point(js, verbose=True)
qe.toc()
```

```
Error at iteration 25 is 0.6540143893175809.
Error at iteration 50 is 0.12643184012381425.
Error at iteration 75 is 0.030376323858035903.
Error at iteration 100 is 0.007581959253982973.
Error at iteration 125 is 0.0019085682645538782.
Error at iteration 150 is 0.00048173786846916755.
Error at iteration 175 is 0.000121400125664195.
```

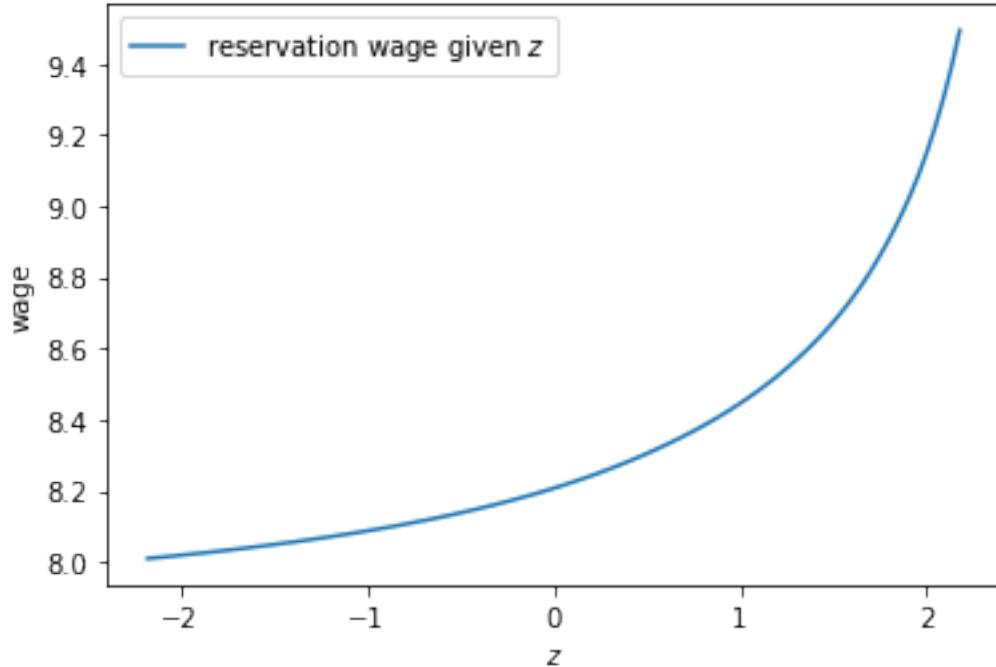
```
Converged in 179 iterations.
TOC: Elapsed: 0:00:7.48
```

Out[7]: 7.482851505279541

Next we will compute and plot the reservation wage function defined in (1).

In [8]: `res_wage_function = np.exp(f_star * (1 - js.β))`

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

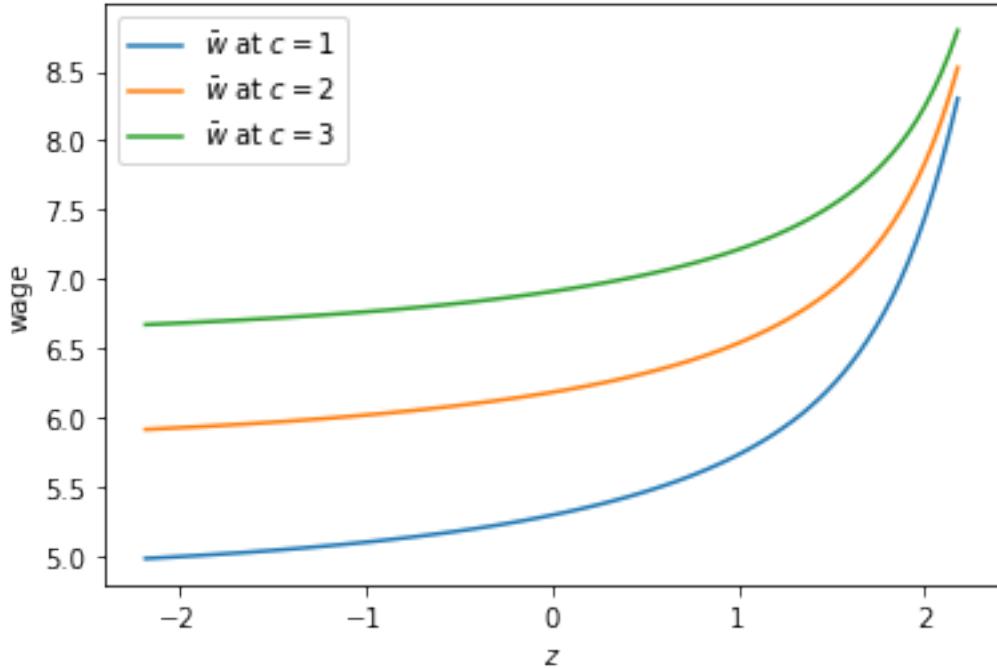
Let's try changing unemployment compensation and look at its impact on the reservation wage:

```
In [9]: c_vals = 1, 2, 3

fig, ax = plt.subplots()

for c in c_vals:
    js = JobSearch(c=c)
    f_star = compute_fixed_point(js, verbose=False)
    res_wage_function = np.exp(f_star * (1 - js.β))
    ax.plot(js.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.

25.5 Unemployment Duration

Next we study how mean unemployment duration varies with unemployment compensation.

For simplicity we'll fix the initial state at $z_t = 0$.

```
In [10]: def compute_unemployment_duration(js, seed=1234):
```

```

f_star = compute_fixed_point(js, verbose=False)
μ, s, d, ρ, σ, β, c = js.parameters()
z_grid = js.z_grid
np.random.seed(seed)

@njit
def f_star_function(z):
    return interp(z_grid, f_star, z)

@njit
def draw_tau(t_max=10_000):
    z = 0
    t = 0

    unemployed = True
    while unemployed and t < t_max:
        # draw current wage
        y = np.exp(μ + s * np.random.randn())
        w = np.exp(z) + y
        res_wage = np.exp(f_star_function(z) * (1 - β))
        # if optimal to stop, record t
        if w >= res_wage:
            unemployed = False
            τ = t
        # else increment data and state
        else:
            z = ρ * z + d + σ * np.random.randn()
            t += 1
    return τ

@njit(parallel=True)
def compute_expected_tau(num_reps=100_000):
    sum_value = 0
    for i in prange(num_reps):
        sum_value += draw_tau()
    return sum_value / num_reps

return compute_expected_tau()

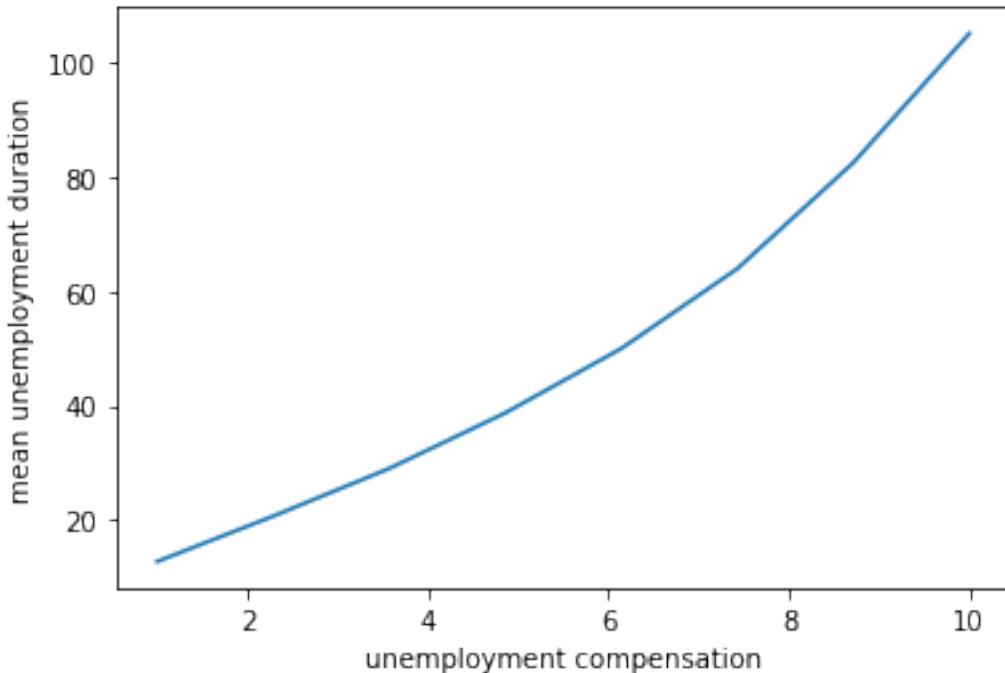
```

Let's test this out with some possible values for unemployment compensation.

```
In [11]: c_vals = np.linspace(1.0, 10.0, 8)
durations = np.empty_like(c_vals)
for i, c in enumerate(c_vals):
    js = JobSearch(c=c)
    τ = compute_unemployment_duration(js)
    durations[i] = τ
```

Here is a plot of the results.

```
In [12]: 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.

25.6 Exercises

25.6.1 Exercise 1

Investigate how mean unemployment duration varies with the discount factor β .

- What is your prior expectation?
- Do your results match up?

25.7 Solutions

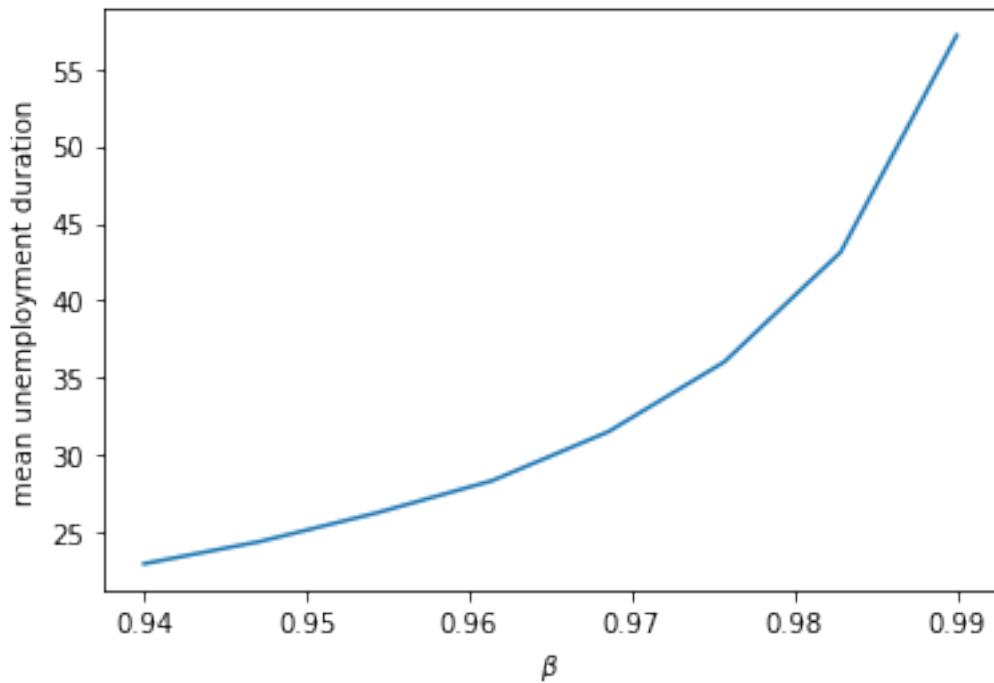
25.7.1 Exercise 1

Here is one solution.

```
In [13]: beta_vals = np.linspace(0.94, 0.99, 8)
durations = np.empty_like(beta_vals)
for i, β in enumerate(beta_vals):
    js = JobSearch(β=β)
    τ = compute_unemployment_duration(js)
    durations[i] = τ
```

```
In [14]: 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 26

Job Search V: Modeling Career Choice

26.1 Contents

- Overview 26.2
- Model 26.3
- Implementation 26.4
- Exercises 26.5
- Solutions 26.6

In addition to what's in Anaconda, this lecture will need the following libraries:

```
In [1]: !pip install quantecon
```

26.2 Overview

Next, we study a computational problem concerning career and job choices.

The model is originally due to Derek Neal [83].

This exposition draws on the presentation in [72], section 6.5.

We begin with some imports:

```
In [2]: import numpy as np
import quantecon as qe
import matplotlib.pyplot as plt
%matplotlib inline
from numba import njit, prange
from quantecon.distributions import BetaBinomial
from scipy.special import binom, beta
from mpl_toolkits.mplot3d.axes3d import Axes3D
from matplotlib import cm
```

/home/ubuntu/anaconda3/lib/python3.7/site-packages/numba/np/ufunc/parallel.py:
355:

```
NumbaWarning: The TBB threading layer requires TBB version 2019.5 or later i.e.,
TBB_INTERFACE_VERSION >= 11005. Found TBB_INTERFACE_VERSION = 11004. The TBB
threading
layer is disabled.
warnings.warn(problem)
```

26.2.1 Model Features

- Career and job within career both chosen to maximize expected discounted wage flow.
- Infinite horizon dynamic programming with two state variables.

26.3 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

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t w_t \tag{1}$$

subject to the choice restrictions specified above.

Let $v(\theta, \epsilon)$ denote the value function, which is the maximum of (1) overall feasible (career, job) policies, given the initial state (θ, ϵ) .

The value function obeys

$$v(\theta, \epsilon) = \max\{I, II, III\}$$

where

$$\begin{aligned}
 I &= \theta + \epsilon + \beta v(\theta, \epsilon) \\
 II &= \theta + \int \epsilon' G(d\epsilon') + \beta \int v(\theta, \epsilon') G(d\epsilon') \\
 III &= \int \theta' F(d\theta') + \int \epsilon' G(d\epsilon') + \beta \int \int v(\theta', \epsilon') G(d\epsilon') F(d\theta')
 \end{aligned} \tag{2}$$

Evidently I , II and III correspond to “stay put”, “new job” and “new life”, respectively.

26.3.1 Parameterization

As in [72], 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

$$p(k | n, a, b) = \binom{n}{k} \frac{B(k+a, n-k+b)}{B(a,b)}, \quad k = 0, \dots, n$$

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

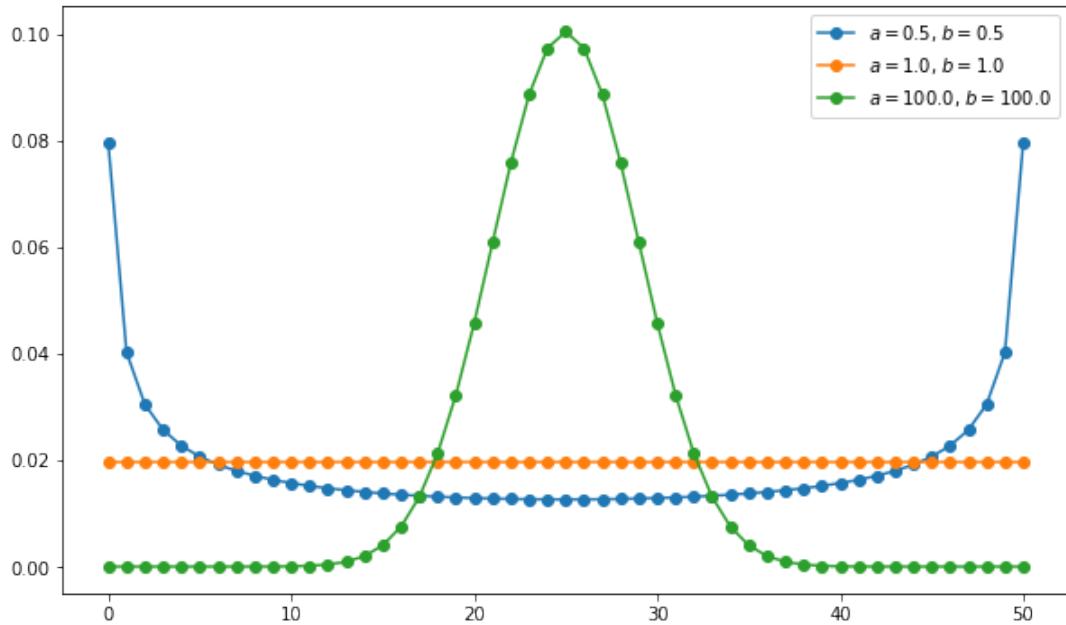
Nice properties:

- 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 $n = 50$.

```
In [3]: 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()
```



26.4 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.

In [4]: `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 = np.sum(self.θ * self.F_probs)
    self.G_mean = np.sum(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 (2).

```
In [5]: 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``
    """
    theta, epsilon, beta = cw.theta, cw.epsilon, cw.beta
    F_probs, G_probs = cw.F_probs, cw.G_probs
    F_mean, G_mean = cw.F_mean, cw.G_mean

    @njit(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 = theta[i] + epsilon[j] + beta * v[i, j]           # Stay put
                v2 = theta[i] + G_mean + beta * v[i, :] @ G_probs    # New job
                v3 = G_mean + F_mean + beta * F_probs @ v @ G_probs # New life
                v_new[i, j] = max(v1, v2, v3)

        return v_new

    @njit
    def get_greedy(v):
        "Computes the v-greedy policy"

        sigma = np.empty(v.shape)

        for i in range(len(v)):
            for j in range(len(v)):
                v1 = theta[i] + epsilon[j] + beta * v[i, j]
                v2 = theta[i] + G_mean + beta * v[i, :] @ G_probs
                v3 = G_mean + F_mean + beta * F_probs @ v @ G_probs
                if v1 > max(v2, v3):
                    action = 1
                elif v2 > max(v1, v3):
                    action = 2
                else:
                    action = 3
                sigma[i, j] = action

        return sigma

    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 value function.

```
In [6]: 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.ones((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 i == max_iter:
        print("Failed to converge!")

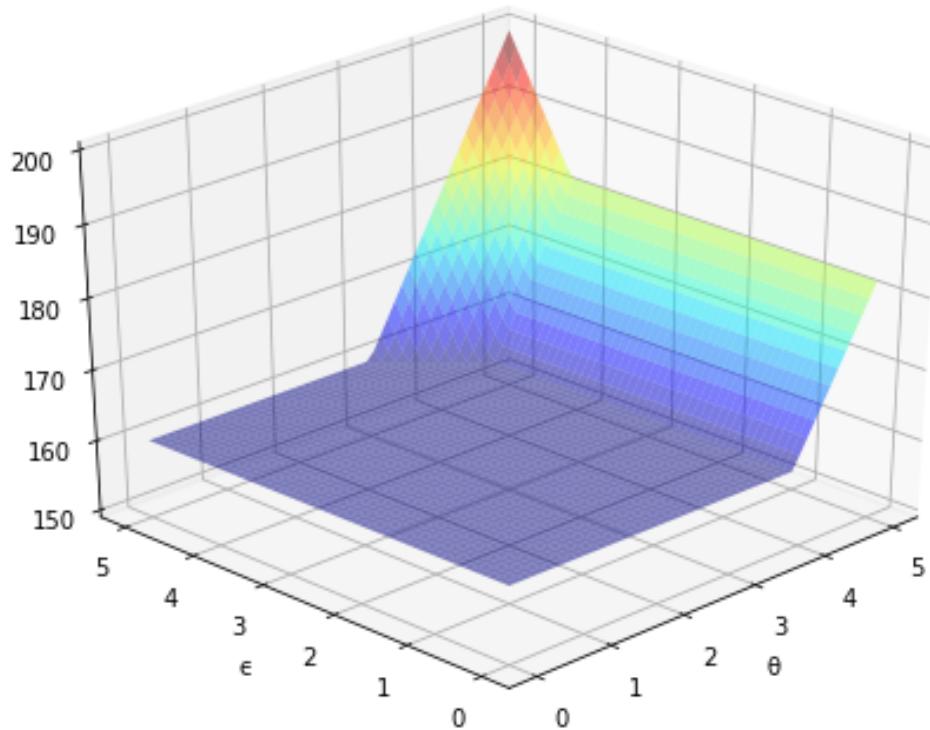
    if verbose and i < max_iter:
        print(f"\nConverged in {i} iterations.")

return v_new
```

Here's the solution to the model – an approximate value function

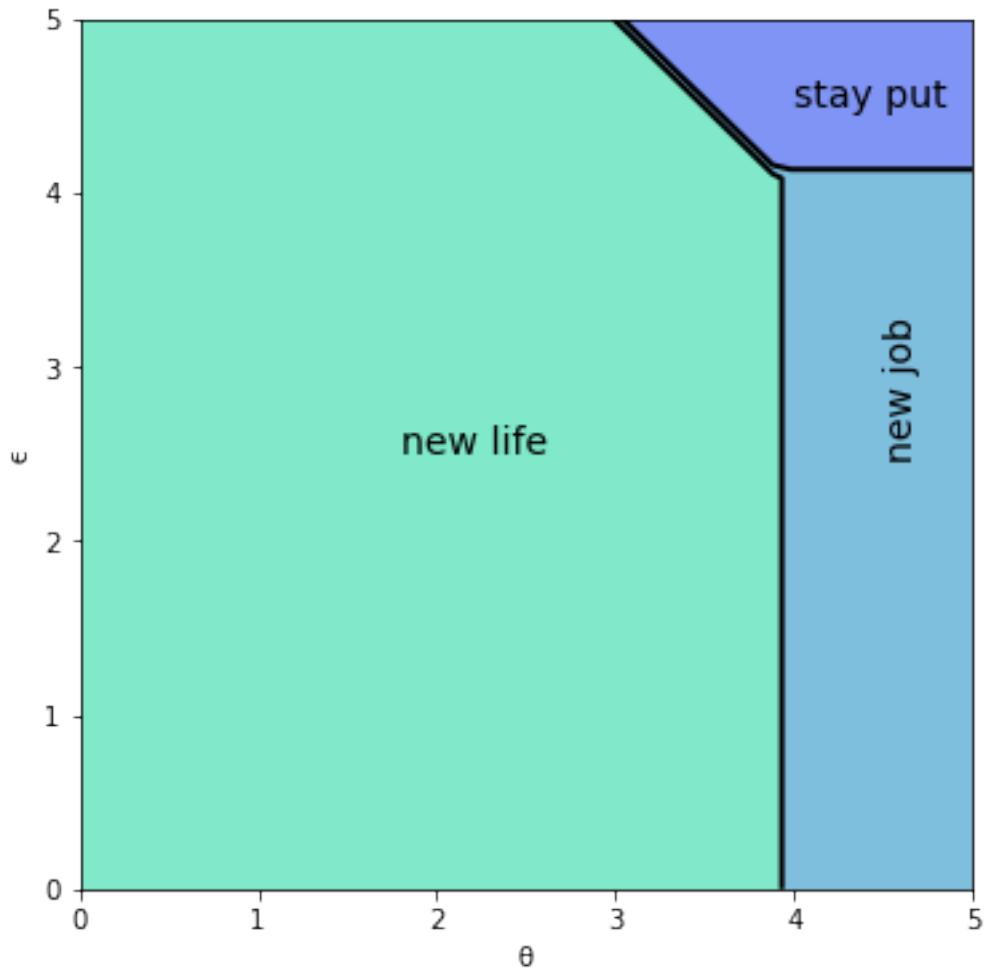
```
In [7]: cw = CareerWorkerProblem()
T, get_greedy = operator_factory(cw)
v_star = solve_model(cw, verbose=False)
greedy_star = get_greedy(v_star)

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

```
In [8]: fig, ax = plt.subplots(figsize=(6, 6))
tg, eg = np.meshgrid(cw.θ, cw.ε)
lvl = (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.

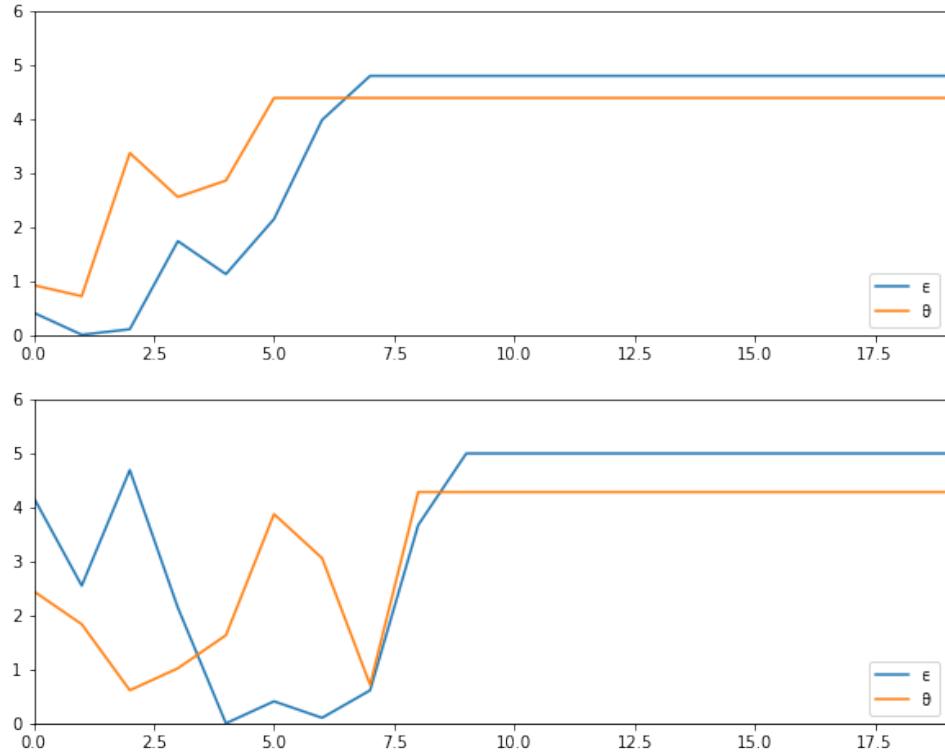
- Sometimes a good job must be sacrificed in order to change to a better career.

26.5 Exercises

26.5.1 Exercise 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()`.

26.5.2 Exercise 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:

$$T^* := \inf\{t \geq 0 \mid (\theta_t, \epsilon_t) \in S\}$$

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.

26.5.3 Exercise 3

Set the parameterization to $G_a = G_b = 100$ and generate a new optimal policy figure – interpret.

26.6 Solutions

26.6.1 Exercise 1

Simulate job/career paths.

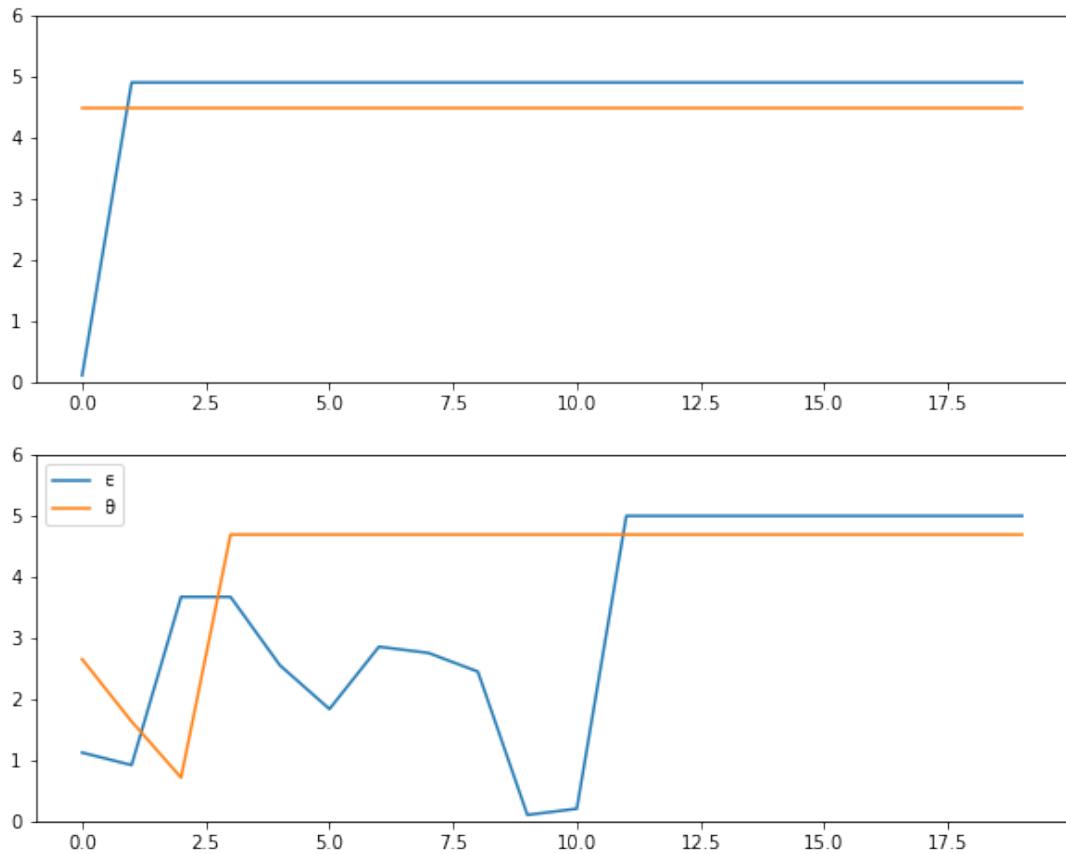
In reading the code, recall that `optimal_policy[i, j]` = policy at (θ_i, ϵ_j) = either 1, 2 or 3; meaning ‘stay put’, ‘new job’ and ‘new life’.

```
In [9]: 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 greedy_star[i, j] == 1:          # Stay put
            pass
        elif greedy_star[i, j] == 2:         # New job
            j = int(qe.random.draw(G))
        else:                            # New life
            i, j = int(qe.random.draw(F)), int(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_xlim(0, 6)

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



26.6.2 Exercise 2

The median for the original parameterization can be computed as follows

```
In [10]: 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)

@njit
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 = int(qe.random.draw(G))
        else:                          # New life
            i, j = int(qe.random.draw(F)), int(qe.random.draw(G))
        t += 1

@njit(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)

```

Out[10]: 7.0

To compute the median with $\beta = 0.99$ instead of the default value $\beta = 0.95$, replace `cw = CareerWorkerProblem()` with `cw = CareerWorkerProblem($\beta=0.99$)`.

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.

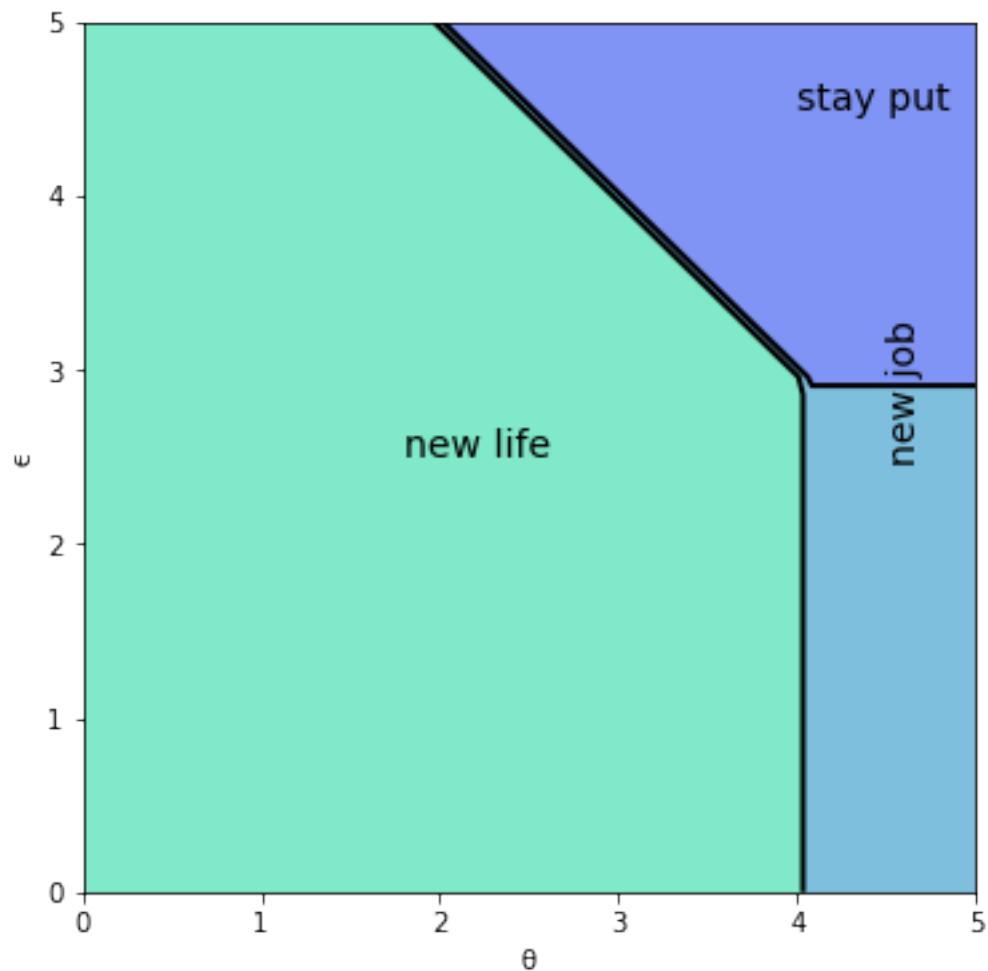
26.6.3 Exercise 3

```

In [11]: 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, 2.5, '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 27

Job Search VI: On-the-Job Search

27.1 Contents

- Overview 27.2
- Model 27.3
- Implementation 27.4
- Solving for Policies 27.5
- Exercises 27.6
- Solutions 27.7

In addition to what's in Anaconda, this lecture will need the following libraries:

```
In [1]: !pip install quantecon  
!pip install interpolation
```

27.2 Overview

In this section, we solve a simple on-the-job search model

- based on [72], exercise 6.18, and [62]

Let's start with some imports:

```
In [2]: import numpy as np  
import scipy.stats as stats  
from interpolation import interp  
from numba import njit, prange  
import matplotlib.pyplot as plt  
%matplotlib inline  
from math import gamma
```

27.2.1 Model Features

- job-specific human capital accumulation combined with on-the-job search
- infinite-horizon dynamic programming with one state variable and two controls

27.3 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

$$x_{t+1} = (1 - b_{t+1})g(x_t, \phi_t) + b_{t+1} \max\{g(x_t, \phi_t), u_{t+1}\} \quad (1)$$

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 (1), the Bellman equation for this problem can be written as

$$v(x) = \max_{s+\phi \leq 1} \left\{ x(1 - s - \phi) + \beta(1 - \pi(s))v[g(x, \phi)] + \beta\pi(s) \int v[g(x, \phi) \vee u]f(du) \right\} \quad (2)$$

Here nonnegativity of s and ϕ is understood, while $a \vee b := \max\{a, b\}$.

27.3.1 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.

27.3.2 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 (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.

27.4 Implementation

We will set up a class `JVWorker` that holds the parameters of the model described above

```
In [3]: class JVWorker:
    """
    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,
    ε=1e-4):

self.A, self.α, self.β, self.π = A, α, β, π
self.mc_size, self.ε = mc_size, ε

self.g = njit(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
# 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

$$w(s, \phi) := x(1 - s - \phi) + \beta(1 - \pi(s))v[g(x, \phi)] + \beta\pi(s) \int v[g(x, \phi) \vee u]f(du) \quad (3)$$

When we represent v , it will be with a NumPy array \mathbf{v} giving values on grid `x_grid`.

But to evaluate the right-hand side of (3), we need a function, so we replace the arrays \mathbf{v} and `x_grid` with a function `v_func` that gives linear interpolation of \mathbf{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 (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.

In [4]: `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

```

```

@njit
def state_action_values(z, x, v):
    s, φ = z
    v_func = lambda x: interp(x_grid, v, x)

    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

@njit(parallel=parallel_flag)
def T(v):
    """
    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(0, 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

@njit
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(0, 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_phi = s, phi
        s_policy[i], phi_policy[i] = max_s, max_phi
    return s_policy, phi_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.

```
In [5]: 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
    """

    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 i == max_iter:
        print("Failed to converge!")

    if verbose and i < max_iter:
        print(f"\nConverged in {i} iterations.")

return v_new
```

27.5 Solving for Policies

Let's generate the optimal policies and see what they look like.

```
In [6]: jv = JVWorker()
T, get_greedy = operator_factory(jv)
v_star = solve_model(jv)
s_star, phi_star = get_greedy(v_star)
```

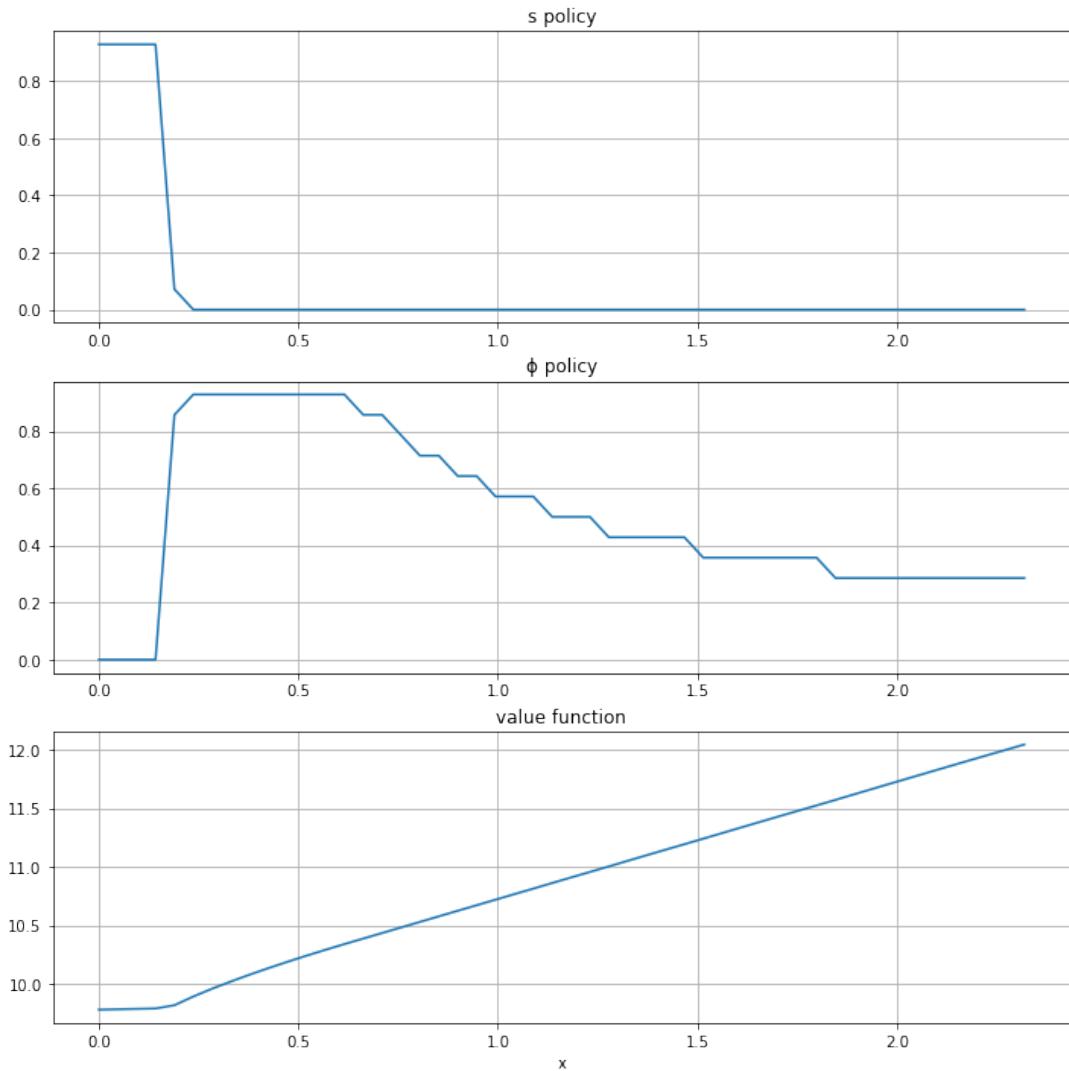
```
/home/ubuntu/anaconda3/lib/python3.7/site-packages/numba/np/ufunc/parallel.py:  
  ↵355:  
NumbaWarning: The TBB threading layer requires TBB version 2019.5 or later i.e.,  
TBB_INTERFACE_VERSION >= 11005. Found TBB_INTERFACE_VERSION = 11004. The TBB  
  ↵threading  
layer is disabled.  
    warnings.warn(problem)
```

```
Error at iteration 25 is 0.1511113094618306.  
Error at iteration 50 is 0.05446001978456394.  
Error at iteration 75 is 0.019627212330380672.  
Error at iteration 100 is 0.007073582884947527.  
Error at iteration 125 is 0.002549296048163896.  
Error at iteration 150 is 0.0009187579260583334.  
Error at iteration 175 is 0.00033111734014212857.  
Error at iteration 200 is 0.00011933360228510992.
```

Converged in 205 iterations.

Here are the plots:

```
In [7]: plots = [s_star, phi_star, v_star]  
titles = ["s policy", "phi 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)  
    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.

27.6 Exercises

27.6.1 Exercise 1

Let's look at the dynamics for the state process $\{x_t\}$ associated with these policies.

The dynamics are given by (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 $\phi_t \approx 0.6$.

27.6.2 Exercise 2

In the preceding exercise, 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?

27.7 Solutions

27.7.1 Exercise 1

Here's code to produce the 45 degree diagram

```
In [8]: 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: interp(x_grid, s_policy, y)
φ = lambda y: interp(x_grid, φ_policy, y)

def h(x, b, u):
```

```

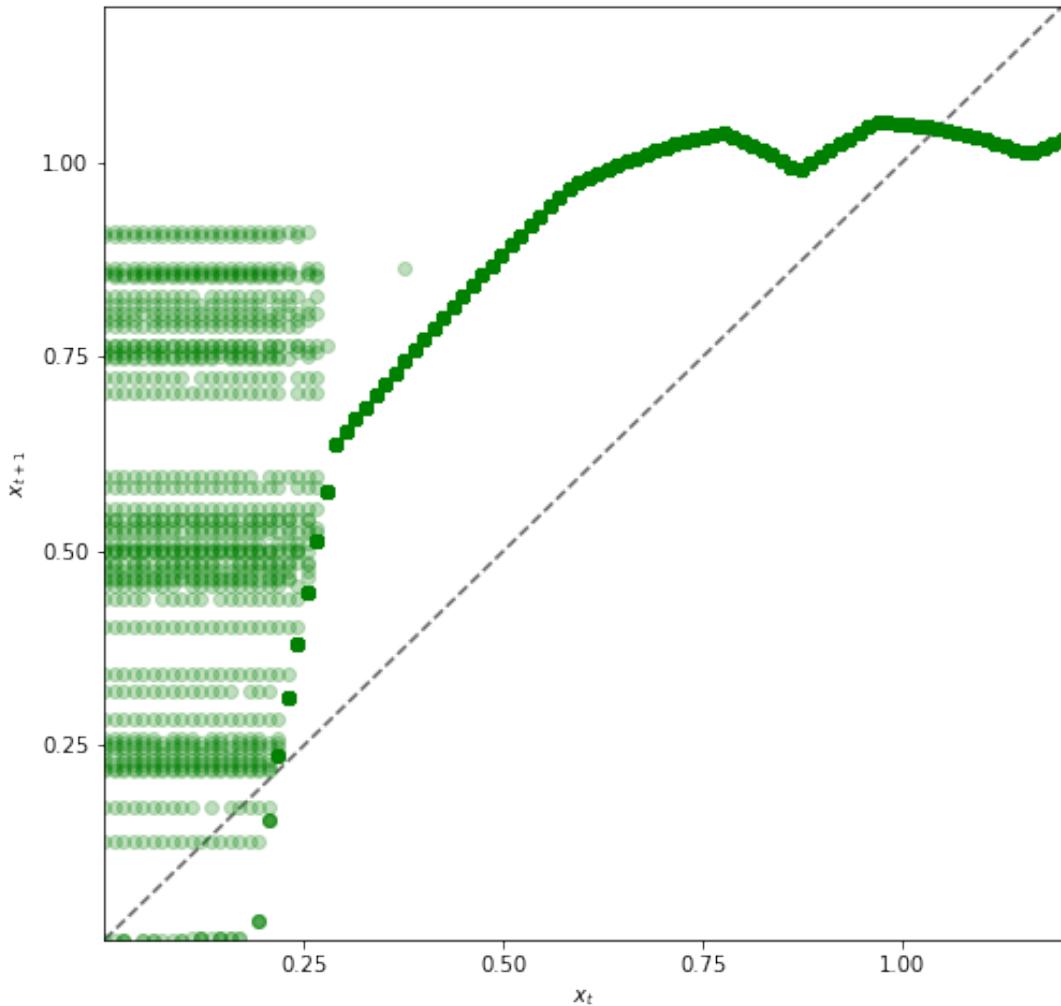
    return (1 - b) * g(x, phi(x)) + b * max(g(x, phi(x)), u)

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) < pi(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 $\phi_t = \phi(x_t) \approx 0.6$.

27.7.2 Exercise 2

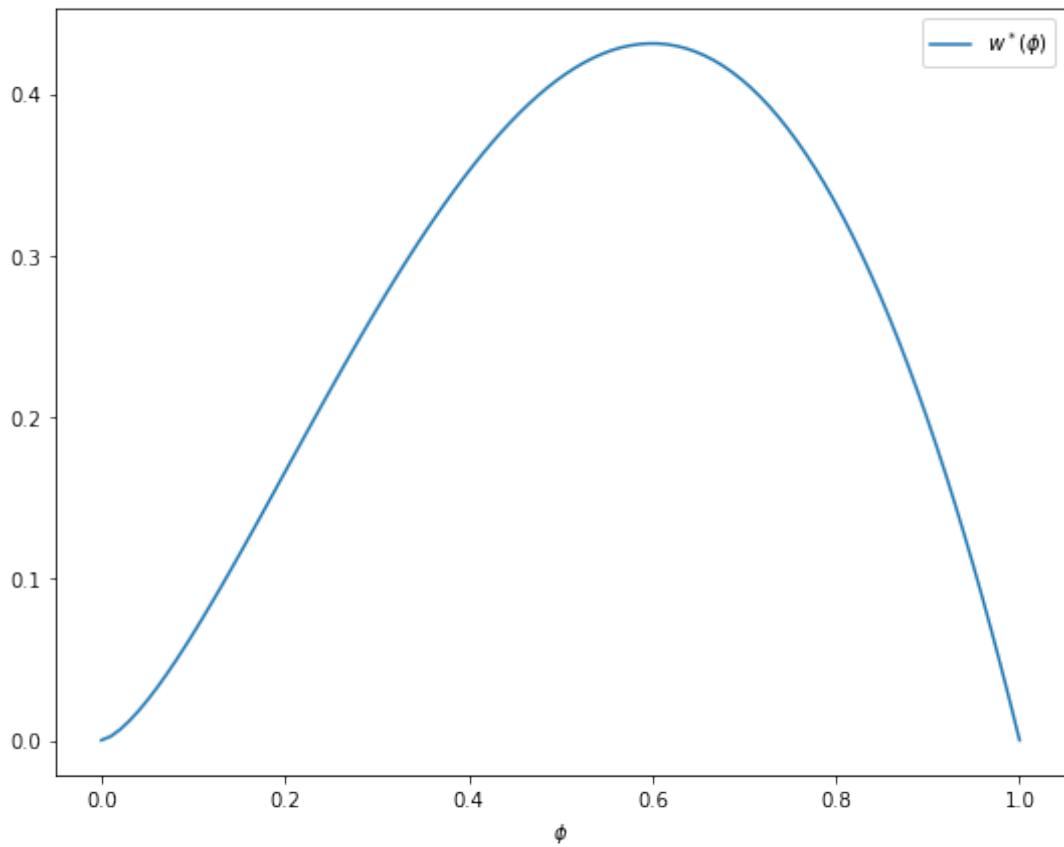
The figure can be produced as follows

```
In [9]: jv = JVWorker()

def xbar(phi):
    A, alpha = jv.A, jv.alpha
    return (A * phi**alpha)**(1 / (1 - alpha))

phi_grid = np.linspace(0, 1, 100)
fig, ax = plt.subplots(figsize=(9, 7))
ax.set(xlabel='$\phi$')
ax.plot(phi_grid, [xbar(phi) * (1 - phi) for phi in phi_grid], label='$w^*(\phi)$')
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 1.

Hence the behavior of the infinitely patient 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.

Part IV

Consumption, Savings and Growth

Chapter 28

Cake Eating I: Introduction to Optimal Saving

28.1 Contents

- Overview [28.2](#)
- The Model [28.3](#)
- The Value Function [28.4](#)
- The Optimal Policy [28.5](#)
- The Euler Equation [28.6](#)
- Exercises [28.7](#)
- Solutions [28.8](#)

28.2 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?

Although the topic sounds trivial, this kind of trade-off between current and future utility is at the heart of many savings and consumption problems.

Once we master the ideas in this simple environment, we will apply them to progressively more challenging—and useful—problems.

The main tool we will use to solve the cake eating problem is dynamic programming.

Readers might find it helpful to review the following lectures before reading this one:

- The [shortest paths lecture](#)
- The [basic McCall model](#)
- The [McCall model with separation](#)
- The [McCall model with separation and a continuous wage distribution](#)

In what follows, we require the following imports:

```
In [1]: import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

28.3 The Model

We consider an infinite time horizon $t = 0, 1, 2, 3..$

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 there is

$$x_{t+1} = x_t - c_t$$

left in period $t + 1$.

Consuming quantity c of the cake gives current utility $u(c)$.

We adopt the CRRA utility function

$$u(c) = \frac{c^{1-\gamma}}{1-\gamma} \quad (\gamma > 0, \gamma \neq 1) \quad (1)$$

In Python this is

```
In [2]: def u(c, gamma):
    return c**(1 - gamma) / (1 - gamma)
```

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

$$\max_{\{c_t\}} \sum_{t=0}^{\infty} \beta^t u(c_t) \quad (2)$$

subject to

$$x_{t+1} = x_t - c_t \quad \text{and} \quad 0 \leq c_t \leq x_t \quad (3)$$

for all t .

A consumption path $\{c_t\}$ satisfying (3) where $x_0 = \bar{x}$ 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**

28.3.1 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.

28.3.2 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.

First, higher β implies less discounting, and hence the agent is more patient, which should reduce the rate of consumption.

Second, higher γ implies that marginal utility $u'(c) = c^{-\gamma}$ falls faster with c .

This suggests more smoothing, and hence a lower rate of consumption.

In summary, we expect the rate of consumption to be *decreasing in both parameters*.

Let's see if this is true.

28.4 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.

28.4.1 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,

$$v(x) = \max \sum_{t=0}^{\infty} \beta^t u(c_t) \quad (4)$$

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

$$v(x) = \max_{0 \leq c \leq x} \{u(c) + \beta v(x - c)\} \quad \text{for any given } x \geq 0. \quad (5)$$

The intuition here is essentially the same 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 (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 (5), as claimed.

28.4.2 An Analytical Solution

It has been shown that, with u as the CRRA utility function in (1), the function

$$v^*(x_t) = (1 - \beta^{1/\gamma})^{-\gamma} u(x_t) \quad (6)$$

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.

The solution (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 when the time comes.

Here is a Python representation of the value function:

```
In [3]: 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:

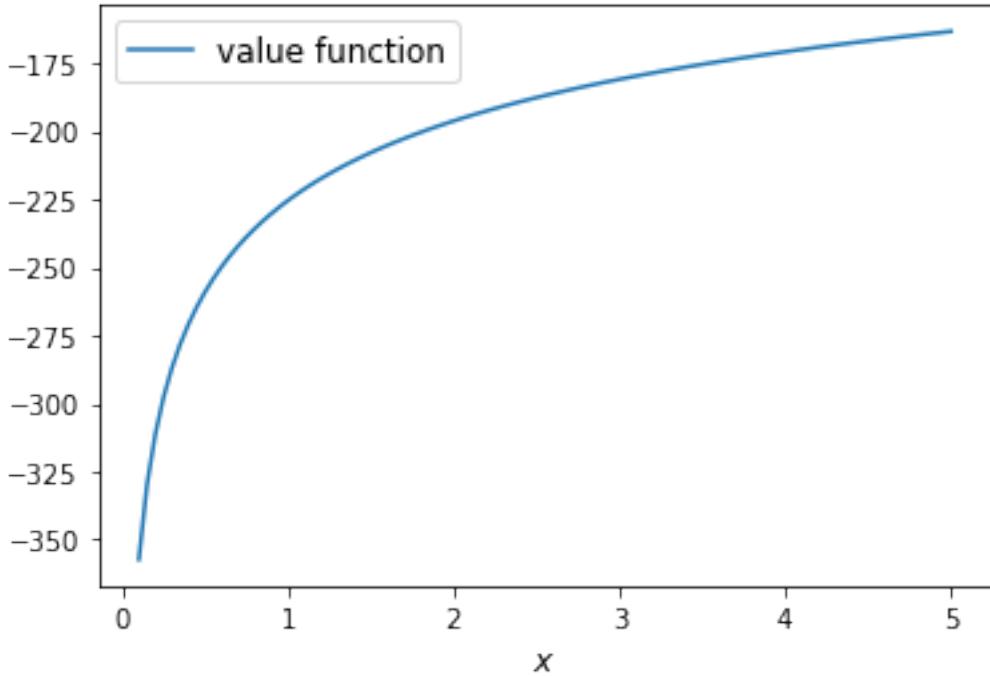
```
In [4]: β, γ = 0.95, 1.2
x_grid = np.linspace(0.1, 5, 100)

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()
```



28.5 The Optimal Policy

Now that we have the value function, 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 (5).

$$c^* = \arg \max_c \{u(c) + \beta v(x - c)\}$$

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

$$\sigma^*(x) := \arg \max_c \{u(c) + \beta v(x - c)\} \quad \text{for all } x$$

If we plug the analytical expression (6) for the value function into the right hand side and compute the optimum, we find that

$$\sigma^*(x) = (1 - \beta^{1/\gamma}) x \tag{7}$$

Now let's recall our intuition on the impact of parameters.

We guessed that the consumption rate would be decreasing in both parameters.

This is in fact the case, as can be seen from (7).

Here's some plots that illustrate.

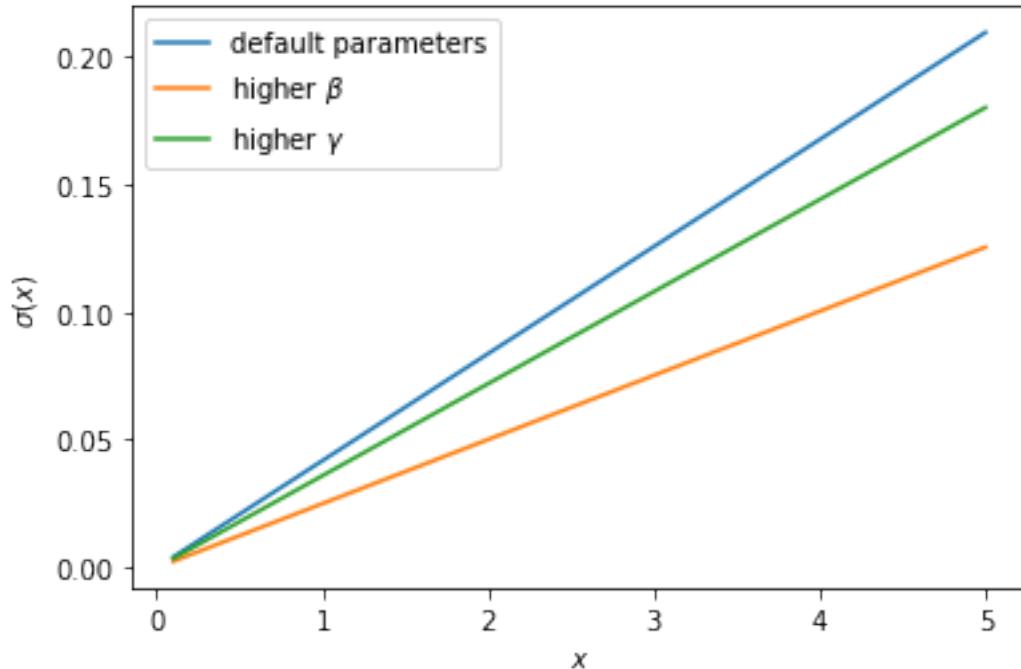
```
In [5]: def c_star(x, β, γ):
```

```
    return (1 - β ** (1/γ)) * x
```

Continuing with the values for β and γ used above, the plot is

```
In [6]: 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()
```



28.6 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.

28.6.1 Statement and Implications

The Euler equation for the present problem can be stated as

$$u'(c_t^*) = \beta u'(c_{t+1}^*) \quad (8)$$

This is necessary condition for the optimal path.

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 $x > 0$,

$$u'(\sigma(x)) = \beta u'(\sigma(x - \sigma(x))) \quad (9)$$

Evidently (9) is just the policy equivalent of (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 (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 [75].

The following arguments focus on necessity, explaining why an optimal path or policy should satisfy the Euler equation.

28.6.2 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

$$\lim_{h \rightarrow 0} \frac{U(c^h) - U(c^*)}{h} = U'(c^*) = 0$$

Recalling that consumption only changes at t and $t + 1$, this becomes

$$\lim_{h \rightarrow 0} \frac{\beta^t u(c_t^* - h) + \beta^{t+1} u(c_{t+1}^* + h) - \beta^t u(c_t^*) - \beta^{t+1} u(c_{t+1}^*)}{h} = 0$$

After rearranging, the same expression can be written as

$$\lim_{h \rightarrow 0} \frac{u(c_t^* - h) - u(c_t^*)}{h} + \lim_{h \rightarrow 0} \frac{\beta u(c_{t+1}^* + h) - u(c_{t+1}^*)}{h} = 0$$

or, taking the limit,

$$-u'(c_t^*) + \beta u'(c_{t+1}^*) = 0$$

This is just the Euler equation.

28.6.3 Derivation II: Using the Bellman Equation

Another way to derive the Euler equation is to use the Bellman equation (5).

Taking the derivative on the right hand side of the Bellman equation with respect to c and setting it to zero, we get

$$u'(c) = \beta v'(x - c) \tag{10}$$

To obtain $v'(x - c)$, we set $g(c, x) = u(c) + \beta v(x - c)$, so that, at the optimal choice of consumption,

$$v(x) = g(c, x) \tag{11}$$

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

$$v'(x) = \frac{\partial g(c, x)}{\partial x} = \frac{\partial}{\partial x} \beta v(x - c) = \beta v'(x - c) \quad (12)$$

(This argument is an example of the [Envelope Theorem](#).)

But now an application of (10) gives

$$u'(c) = v'(x) \quad (13)$$

Thus, the derivative of the value function is equal to marginal utility.

Combining this fact with (12) recovers the Euler equation.

28.7 Exercises

28.7.1 Exercise 1

How does one obtain the expressions for the value function and optimal policy given in (6) and (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 (6) and (7).

In doing so, you will need to use the definition of the value function and the Bellman equation.

28.8 Solutions

28.8.1 Exercise 1

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,

$$\begin{aligned}
v(x) &= \max_{0 \leq c \leq x} \left\{ u(c) + \beta \frac{\theta^{1-\gamma}}{1 - \beta(1-\theta)^{1-\gamma}} \cdot u(x-c) \right\} \\
&= \max_{0 \leq c \leq x} \left\{ \frac{c^{1-\gamma}}{1-\gamma} + \beta \frac{\theta^{1-\gamma}}{1 - \beta(1-\theta)^{1-\gamma}} \cdot \frac{(x-c)^{1-\gamma}}{1-\gamma} \right\}
\end{aligned}$$

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.

Chapter 29

Cake Eating II: Numerical Methods

29.1 Contents

- Overview 29.2
- Reviewing the Model 29.3
- Value Function Iteration 29.4
- Time Iteration 29.5
- Exercises 29.6
- Solutions 29.7

In addition to what's in Anaconda, this lecture will require the following library:

```
In [1]: !pip install interpolation
```

29.2 Overview

In this lecture we continue the study of [the cake eating problem](#).

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.

We will use the following imports:

```
In [2]: import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

from interpolation import interp
from scipy.optimize import minimize_scalar, bisect
```

29.3 Reviewing the Model

You might like to [review the details](#) before we start.

Recall in particular that the Bellman equation is

$$v(x) = \max_{0 \leq c \leq x} \{u(c) + \beta v(x - c)\} \quad \text{for all } x \geq 0. \quad (1)$$

where u is the CRRA utility function.

The analytical solutions for the value function and optimal policy were found to be as follows.

In [3]: `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.

29.4 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 w defined by

$$w(x) = \max_{0 \leq c \leq x} \{u(c) + \beta v(x - c)\}$$

1. Stop if w is approximately equal to v , otherwise set $v = w$ and go back to step 2.

Let's write this a bit more mathematically.

29.4.1 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 $T^2v := T(Tv)$ 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.

29.4.2 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 $\{v_0, \dots, v_I\}$ representing the values of some initial function v on the grid points $\{x_0, \dots, x_I\}$.
2. Build a function \hat{v} on the state space \mathbb{R}_+ by linear interpolation, based on these data points.
3. Obtain and record the value $T\hat{v}(x_i)$ on each grid point x_i by repeatedly solving the maximization problem in the Bellman equation.
4. Unless some stopping condition is satisfied, set $\{v_0, \dots, v_I\} = \{T\hat{v}(x_0), \dots, T\hat{v}(x_I)\}$ and go to step 2.

In step 2 we'll use continuous piecewise linear interpolation.

29.4.3 Implementation

The `maximize` function below is a small helper function that converts a SciPy minimization routine into a maximization routine.

```
In [4]: def maximize(g, a, b, args):
    """
    Maximize the function g over the interval [a, b].
    """
```

We use the fact that the maximizer of g on any interval is also the minimizer of $-g$. The tuple args collects any extra arguments to g .

Returns the maximal value and the maximizer.

```
objective = lambda x: -g(x, *args)
result = minimize_scalar(objective, bounds=(a, b), method='bounded')
maximizer, maximum = result.x, -result.fun
return maximizer, maximum
```

We'll store the parameters β and γ in a class called `CakeEating`.

The same class will also provide a method called `state_action_value` that returns the value of a consumption choice given a particular state and guess of v .

In [5]: `class CakeEating:`

```

def __init__(self,
            β=0.96,           # discount factor
            γ=1.5,            # degree of relative risk aversion
            x_grid_min=1e-3,  # exclude zero for numerical stability
            x_grid_max=2.5,   # size of cake
            x_grid_size=120):

    self.β, self.γ = β, γ

    # Set up grid
    self.x_grid = np.linspace(x_grid_min, x_grid_max, x_grid_size)

# Utility function
def u(self, c):

    γ = self.γ

    if γ == 1:
        return np.log(c)
    else:
        return (c ** (1 - γ)) / (1 - γ)

# first derivative of utility function
def u_prime(self, c):

    return c ** (-self.γ)

def state_action_value(self, c, x, v_array):
    """
    Right hand side of the Bellman equation given x and c.
    """

    u, β = self.u, self.β
    v = lambda x: interp(self.x_grid, v_array, x)

    return u(c) + β * v(x - c)

```

We now define the Bellman operation:

In [6]: `def T(v, ce):`

```

"""
The Bellman operator. Updates the guess of the value function.

* ce is an instance of CakeEating
* v is an array representing a guess of the value function

"""

v_new = np.empty_like(v)

for i, x in enumerate(ce.x_grid):

```

```

# Maximize RHS of Bellman equation at state x
v_new[i] = maximize(ce.state_action_value, 1e-10, x, (x, v))[1]

return v_new

```

After defining the Bellman operator, we are ready to solve the model.

Let's start by creating a `CakeEating` instance using the default parameterization.

In [7]: `ce = CakeEating()`

Now let's see the iteration of the value function in action.

We start from guess v given by $v(x) = u(x)$ for every x grid point.

```

In [8]: x_grid = ce.x_grid
v = ce.u(x_grid)           # Initial guess
n = 12                      # Number of iterations

fig, ax = plt.subplots()

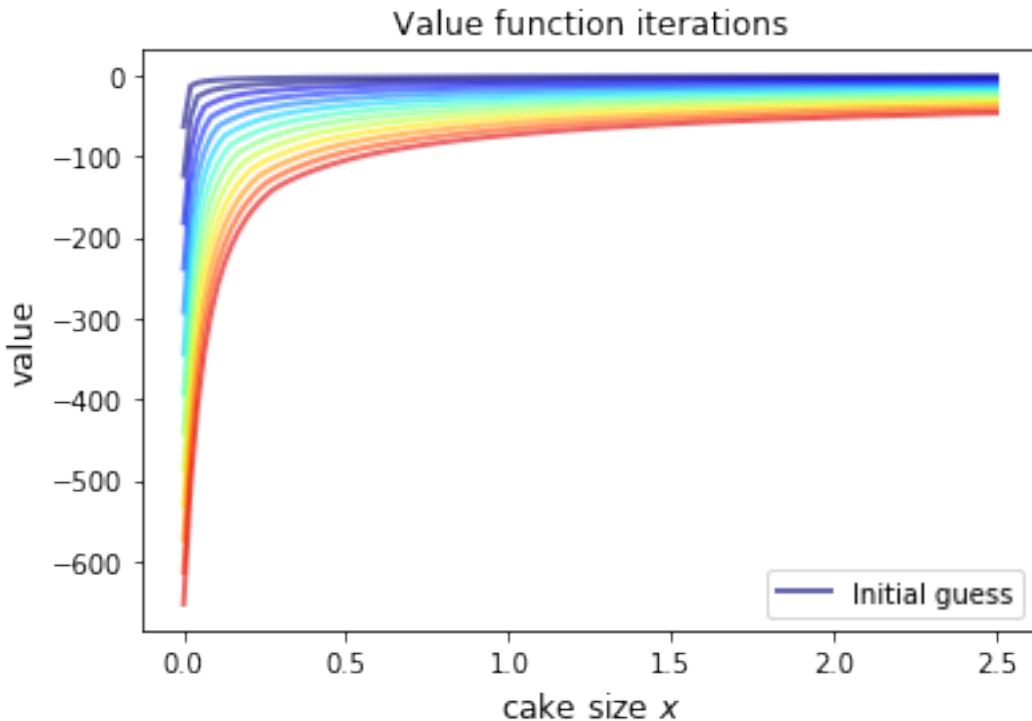
ax.plot(x_grid, v, color=plt.cm.jet(0),
        lw=2, alpha=0.6, label='Initial guess')

for i in range(n):
    v = T(v, ce)      # Apply the Bellman operator
    ax.plot(x_grid, v, color=plt.cm.jet(i / n), lw=2, alpha=0.6)

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 do this more systematically, we introduce a wrapper function called `compute_value_function` that iterates until some convergence conditions are satisfied.

```
In [9]: def compute_value_function(ce,
                               tol=1e-4,
                               max_iter=1000,
                               verbose=True,
                               print_skip=25):

    # Set up loop
    v = np.zeros(len(ce.x_grid)) # Initial guess
    i = 0
    error = tol + 1

    while i < max_iter and error > tol:
        v_new = T(v, ce)

        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 i == max_iter:
        print("Failed to converge!")

    if verbose and i < max_iter:
        print(f"\nConverged in {i} iterations.)
```

```
return v_new
```

Now let's call it, noting that it takes a little while to run.

In [10]: `v = compute_value_function(ce)`

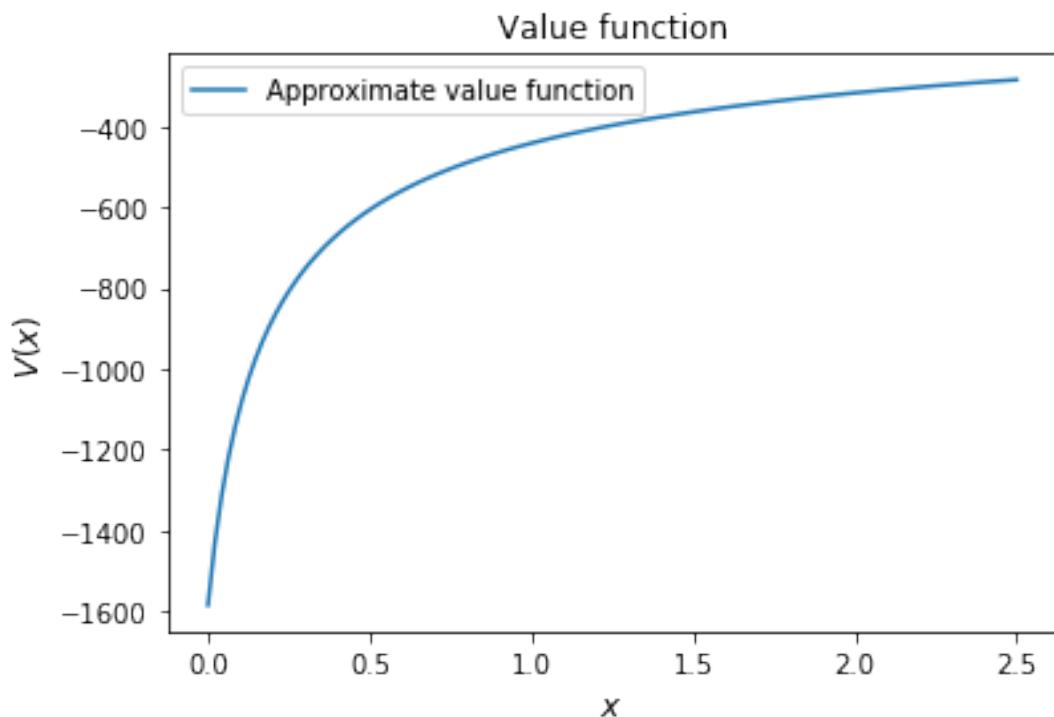
```
Error at iteration 25 is 23.8003755134813.
Error at iteration 50 is 8.577577195046615.
Error at iteration 75 is 3.091330659691039.
Error at iteration 100 is 1.1141054204751981.
Error at iteration 125 is 0.4015199357729671.
Error at iteration 150 is 0.14470646660561215.
Error at iteration 175 is 0.052151735472762084.
Error at iteration 200 is 0.018795314242879613.
Error at iteration 225 is 0.006773769545588948.
Error at iteration 250 is 0.0024412443051460286.
Error at iteration 275 is 0.000879816432870939.
Error at iteration 300 is 0.00031708295398402697.
Error at iteration 325 is 0.00011427565573285392.
```

Converged in 329 iterations.

Now we can plot and see what the converged value function looks like.

In [11]: `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()
```

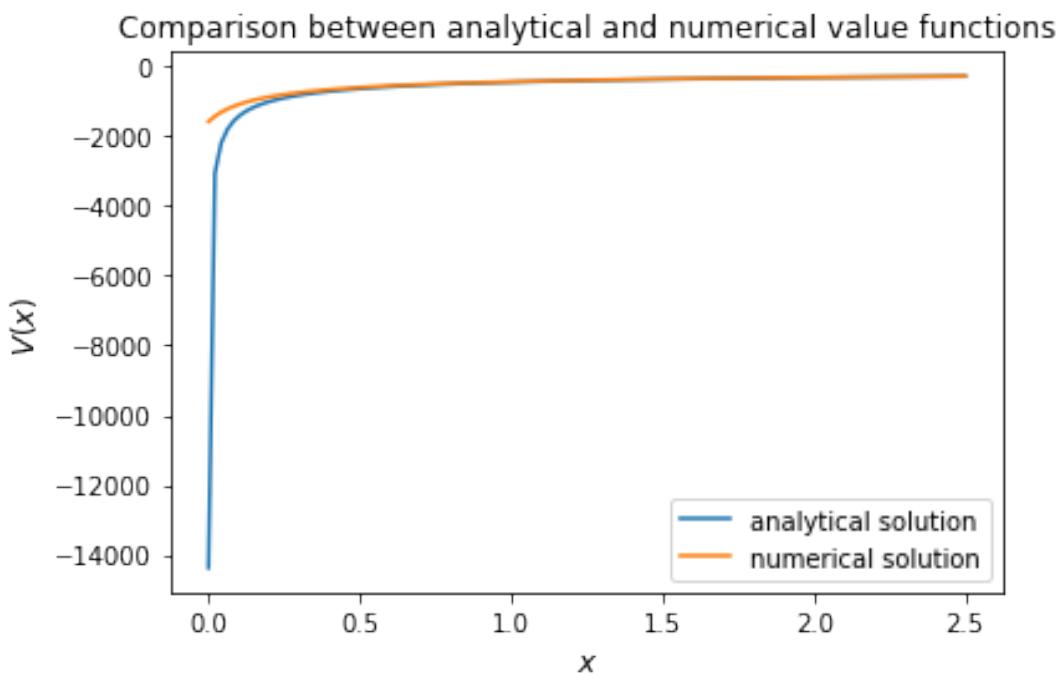


Next let's compare it to the analytical solution.

```
In [12]: v_analytical = v_star(ce.x_grid, ce.β, ce.γ)
```

```
In [13]: 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.

29.4.4 Policy Function

Let's see how this plays out in terms of computing the optimal policy.

In the [first lecture on cake eating](#), the optimal consumption policy was shown to be

$$\sigma^*(x) = (1 - \beta^{1/\gamma}) x$$

Let's see if our numerical results lead to something similar.

Our numerical strategy will be to compute

$$\sigma(x) = \arg \max_{0 \leq c \leq x} \{u(c) + \beta v(x - c)\}$$

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:

```
In [14]: def sigma(ce, v):
    """
        The optimal policy function. Given the value function,
        it finds optimal consumption in each state.

        * ce is an instance of CakeEating
        * v is a value function array

    """
    c = np.empty_like(v)

    for i in range(len(ce.x_grid)):
        x = ce.x_grid[i]
        # Maximize RHS of Bellman equation at state x
        c[i] = maximize(ce.state_action_value, 1e-10, x, (x, v))[0]

    return c
```

Now let's pass the approximate value function and compute optimal consumption:

```
In [15]: c = sigma(ce, v)
```

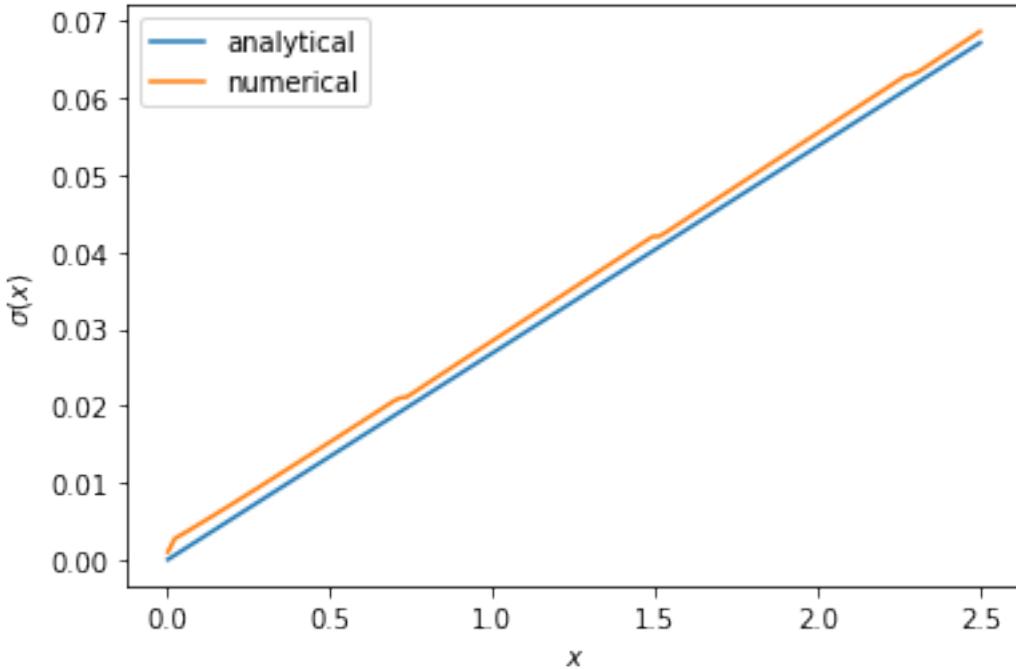
Let's plot this next to the true analytical solution

```
In [16]: c_analytical = c_star(ce.x_grid, ce.beta, ce.gamma)

fig, ax = plt.subplots()

ax.plot(ce.x_grid, c_analytical, label='analytical')
ax.plot(ce.x_grid, c, 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 next.

29.5 Time Iteration

Now let's look at a different strategy to compute the optimal policy.

Recall that the optimal policy satisfies the Euler equation

$$u'(\sigma(x)) = \beta u'(\sigma(x - \sigma(x))) \quad \text{for all } x > 0 \quad (2)$$

Computationally, we can start with any initial guess of σ_0 and now choose c to solve

$$u'(c) = \beta u'(\sigma_0(x - c))$$

Choosing c to satisfy this equation at all $x > 0$ produces a function of x .

Call this new function σ_1 , treat it as the new guess and repeat.

This is called **time iteration**.

As with value function iteration, we can view the update step as action of an operator, this time denoted by K .

- In particular, $K\sigma$ is the policy updated from σ using the procedure just described.
- We will use this terminology in the exercises below.

The main advantage of time iteration relative to value function iteration is that it operates in policy space rather than value function space.

This is helpful because the policy function has less curvature, and hence is easier to approximate.

In the exercises you are asked to implement time iteration and compare it to value function iteration.

You should find that the method is faster and more accurate.

This is due to

1. the curvature issue mentioned just above and
2. the fact that we are using more information — in this case, the first order conditions.

29.6 Exercises

29.6.1 Exercise 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.

29.6.2 Exercise 2

Implement time iteration, returning to the original case (i.e., dropping the modification in the exercise above).

29.7 Solutions

29.7.1 Exercise 1

We need to create a class to hold our primitives and return the right hand side of the Bellman equation.

We will use [inheritance](#) to maximize code reuse.

```
In [17]: class OptimalGrowth(CakeEating):
    """
    A subclass of CakeEating that adds the parameter  $\alpha$  and overrides
    the state_action_value method.
    """

    def __init__(self,
                  $\beta=0.96$ ,          # discount factor
                  $\gamma=1.5$ ,           # degree of relative risk aversion
                  $\alpha=0.4$ ,          # productivity parameter
                 x_grid_min=1e-3,      # exclude zero for numerical stability
                 x_grid_max=2.5,       # size of cake
                 x_grid_size=120):

        self. $\alpha$  =  $\alpha$ 
        CakeEating.__init__(self,  $\beta$ ,  $\gamma$ , x_grid_min, x_grid_max, x_grid_size)

    def state_action_value(self, c, x, v_array):
        """
        Right hand side of the Bellman equation given  $x$  and  $c$ .
        """

        u,  $\beta$ ,  $\alpha$  = self.u, self. $\beta$ , self. $\alpha$ 
        v = lambda x: interp(self.x_grid, v_array, x)

        return u(c) +  $\beta$  * v((x - c) $^{**\alpha}$ )
```

```
In [18]: og = OptimalGrowth()
```

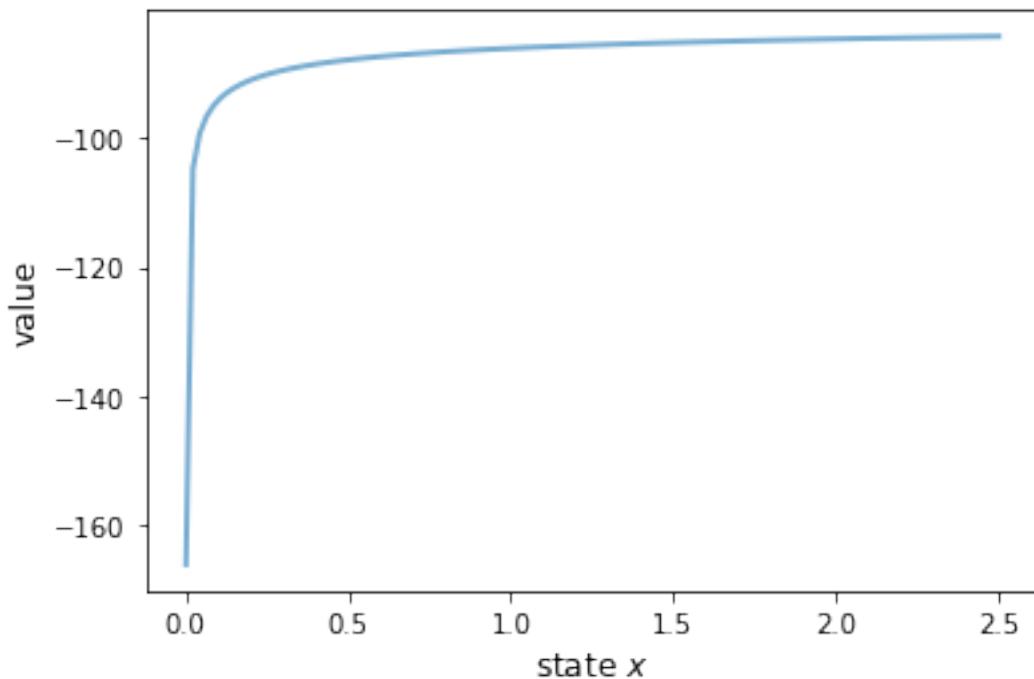
Here's the computed value function.

```
In [19]: v = compute_value_function(og, verbose=False)

fig, ax = plt.subplots()

ax.plot(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 $\alpha = 1$.

```
In [20]: c_new = σ(og, v)

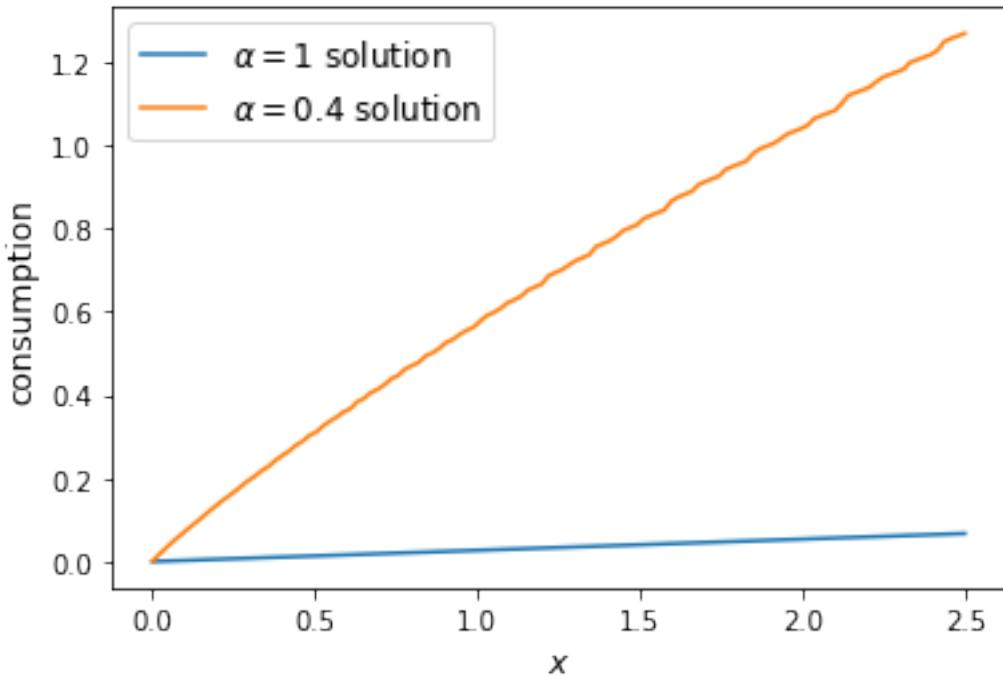
fig, ax = plt.subplots()

ax.plot(ce.x_grid, c_analytical, label=r'$\alpha=1$ solution')
ax.plot(ce.x_grid, c_new, label=fr'$\alpha={og.α}$ 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.

29.7.2 Exercise 2

Here's one way to implement time iteration.

```
In [21]: def K(sigma_array, ce):
    """
    The policy function operator. Given the policy function,
    it updates the optimal consumption using Euler equation.

    * sigma_array is an array of policy function values on the grid
    * ce is an instance of CakeEating
    """

    u_prime, beta, x_grid = ce.u_prime, ce.beta, ce.x_grid
    sigma_new = np.empty_like(sigma_array)

    sigma = lambda x: interp(x_grid, sigma_array, x)

    def euler_diff(c, x):
        return u_prime(c) - beta * u_prime(sigma(x - c))

    for i, x in enumerate(x_grid):
        # handle small x separately --- helps numerical stability
        if x < 1e-12:
            sigma_new[i] = 0.0
        else:
            sigma_new[i] = K(euler_diff, ce)(sigma(x))
```

```

# handle other x
else:
    σ_new[i] = bisect(euler_diff, 1e-10, x - 1e-10, x)

return σ_new

```

In [22]:

```

def iterate_euler_equation(ce,
                           max_iter=500,
                           tol=1e-5,
                           verbose=True,
                           print_skip=25):

    x_grid = ce.x_grid

    σ = np.copy(x_grid)           # initial guess

    i = 0
    error = tol + 1
    while i < max_iter and error > tol:

        σ_new = K(σ, ce)

        error = np.max(np.abs(σ_new - σ))
        i += 1

        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")

        σ = σ_new

    if i == max_iter:
        print("Failed to converge!")

    if verbose and i < max_iter:
        print(f"\nConverged in {i} iterations.")

return σ

```

In [23]:

```

ce = CakeEating(x_grid_min=0.0)
c_euler = iterate_euler_equation(ce)

```

```

Error at iteration 25 is 0.0036456675931543225.
Error at iteration 50 is 0.0008283185047067848.
Error at iteration 75 is 0.00030791132300957147.
Error at iteration 100 is 0.00013555502390599772.
Error at iteration 125 is 6.417740905302616e-05.
Error at iteration 150 is 3.1438019047758115e-05.
Error at iteration 175 is 1.5658492883291464e-05.

```

Converged in 192 iterations.

In [24]:

```

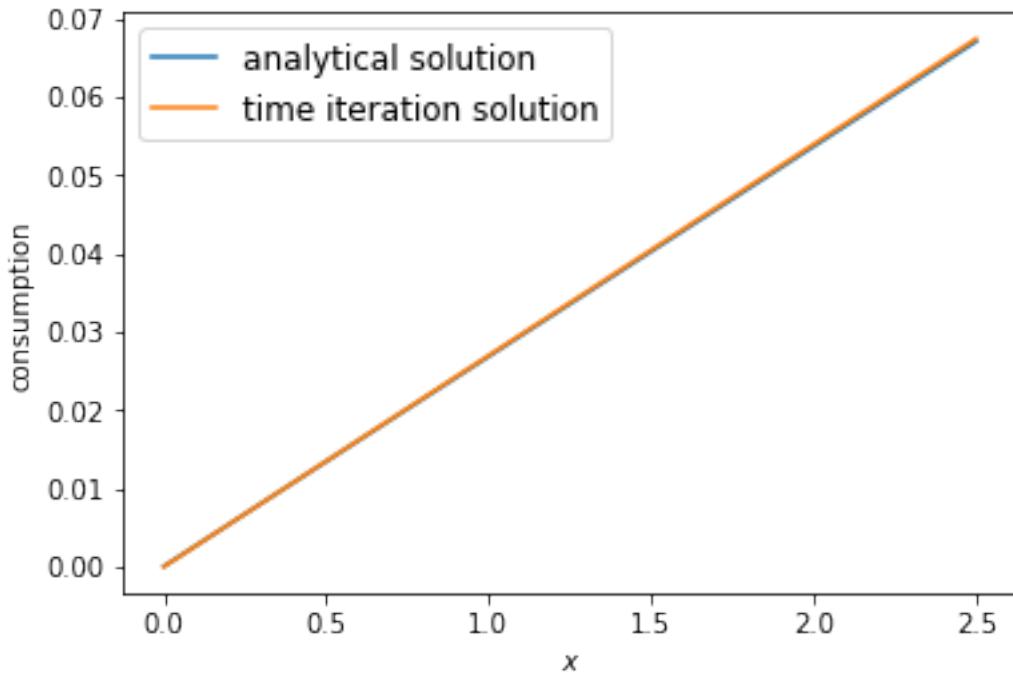
fig, ax = plt.subplots()

ax.plot(ce.x_grid, c_analytical, label='analytical solution')
ax.plot(ce.x_grid, c_euler, label='time iteration solution')

```

```
ax.set_ylabel('consumption')
ax.set_xlabel('$x$')
ax.legend(fontsize=12)

plt.show()
```



Chapter 30

Optimal Growth I: The Stochastic Optimal Growth Model

30.1 Contents

- Overview [30.2](#)
- The Model [30.3](#)
- Computation [30.4](#)
- Exercises [30.5](#)
- Solutions [30.6](#)

30.2 Overview

In this lecture, we're going to study a simple optimal growth model with one agent.

The model is a version of the standard one sector infinite horizon growth model studied in

- [\[102\]](#), chapter 2
- [\[72\]](#), section 3.1
- [EDTC](#), chapter 1
- [\[104\]](#), chapter 12

It is an extension of the simple [cake eating problem](#) we looked at earlier.

The extension involves

- nonlinear returns to saving, through a production function, and
- stochastic returns, due to shocks to production.

Despite these additions, the model is still relatively simple.

We regard it as a stepping stone to more sophisticated models.

We solve the model using dynamic programming and a range of numerical techniques.

In this first lecture on optimal growth, the solution method will be value function iteration (VFI).

While the code in this first lecture runs slowly, we will use a variety of techniques to drastically improve execution time over the next few lectures.

Let's start with some imports:

```
In [1]: import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d
from scipy.optimize import minimize_scalar

%matplotlib inline
```

30.3 The Model

Consider an agent who owns an amount $y_t \in \mathbb{R}_+ := [0, \infty)$ of a consumption good at time t .

This output can either be consumed or invested.

When the good is invested, it is transformed one-for-one into capital.

The resulting capital stock, denoted here by k_{t+1} , will then be 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

$$y_{t+1} := f(k_{t+1})\xi_{t+1}$$

where $f: \mathbb{R}_+ \rightarrow \mathbb{R}_+$ is called the production function.

The resource constraint is

$$k_{t+1} + c_t \leq y_t \tag{1}$$

and all variables are required to be nonnegative.

30.3.1 Assumptions and Comments

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.
- Depreciation of capital is not made explicit but can be incorporated into the production function.

While many other treatments of the stochastic growth model use k_t as the state variable, we will use y_t .

This will allow us to treat a stochastic model while maintaining only one state variable.

We consider alternative states and timing specifications in some of our other lectures.

30.3.2 Optimization

Taking y_0 as given, the agent wishes to maximize

$$\mathbb{E} \left[\sum_{t=0}^{\infty} \beta^t u(c_t) \right] \quad (2)$$

subject to

$$y_{t+1} = f(y_t - c_t) \xi_{t+1} \quad \text{and} \quad 0 \leq c_t \leq y_t \quad \text{for all } t \quad (3)$$

where

- u is a bounded, continuous and strictly increasing utility function and
- $\beta \in (0, 1)$ is a discount factor.

In (3) we are assuming that the resource constraint (1) holds with equality — which is reasonable because u is strictly increasing and no output will be wasted at the optimum.

In summary, the agent's aim is to select a path c_0, c_1, c_2, \dots for consumption that is

1. nonnegative,
2. feasible in the sense of (1),
3. optimal, in the sense that it maximizes (2) relative to all other feasible consumption sequences, and
4. *adapted*, in the sense that the action c_t depends only on observable outcomes, not on future outcomes such as ξ_{t+1} .

In the present context

- y_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.

30.3.3 The Policy Function Approach

One way to think about solving this problem is to look for the best **policy function**.

A policy function is a map from past and present observables into current action.

We'll be particularly interested in **Markov policies**, which are maps from the current state y_t into a current action c_t .

For dynamic programming problems such as this one (in fact for any **Markov decision process**), the optimal policy is always a Markov policy.

In other words, the current state y_t provides a **sufficient statistic** for the history in terms of making an optimal decision today.

This is quite intuitive, but if you wish you can find proofs in texts such as [102] (section 4.1).

Hereafter we focus on finding the best Markov policy.

In our context, a Markov policy is a function $\sigma: \mathbb{R}_+ \rightarrow \mathbb{R}_+$, with the understanding that states are mapped to actions via

$$c_t = \sigma(y_t) \quad \text{for all } t$$

In what follows, we will call σ a *feasible consumption policy* if it satisfies

$$0 \leq \sigma(y) \leq y \quad \text{for all } y \in \mathbb{R}_+ \quad (4)$$

In other words, a feasible consumption policy is a Markov policy that respects the resource constraint.

The set of all feasible consumption policies will be denoted by Σ .

Each $\sigma \in \Sigma$ determines a *continuous state Markov process* $\{y_t\}$ for output via

$$y_{t+1} = f(y_t - \sigma(y_t))\xi_{t+1}, \quad y_0 \text{ given} \quad (5)$$

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

$$\mathbb{E} \left[\sum_{t=0}^{\infty} \beta^t u(c_t) \right] = \mathbb{E} \left[\sum_{t=0}^{\infty} \beta^t u(\sigma(y_t)) \right] \quad (6)$$

This is the total expected present value of following policy σ forever, given initial income y_0 .

The aim is to select a policy that makes this number as large as possible.

The next section covers these ideas more formally.

30.3.4 Optimality

The σ associated with a given policy σ is the mapping defined by

$$v_\sigma(y) = \mathbb{E} \left[\sum_{t=0}^{\infty} \beta^t u(\sigma(y_t)) \right] \quad (7)$$

when $\{y_t\}$ is given by (5) with $y_0 = y$.

In other words, it is the lifetime value of following policy σ starting at initial condition y .

The **value function** is then defined as

$$v^*(y) := \sup_{\sigma \in \Sigma} v_\sigma(y) \quad (8)$$

The value function gives the maximal value that can be obtained from state y , after considering all feasible policies.

A policy $\sigma \in \Sigma$ is called **optimal** if it attains the supremum in (8) for all $y \in \mathbb{R}_+$.

30.3.5 The Bellman Equation

With our assumptions on utility and production functions, the value function as defined in (8) also satisfies a **Bellman equation**.

For this problem, the Bellman equation takes the form

$$v(y) = \max_{0 \leq c \leq y} \left\{ u(c) + \beta \int v(f(y - c)z) \phi(dz) \right\} \quad (y \in \mathbb{R}_+) \quad (9)$$

This is a *functional equation in v* .

The term $\int v(f(y - c)z) \phi(dz)$ can be understood as the expected next period value when

- v is used to measure value
- the state is y
- consumption is set to c

As shown in [EDTC](#), theorem 10.1.11 and a range of other texts

The value function v^ satisfies the Bellman equation*

In other words, (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 gives us more information about the value function.

It also suggests a way of computing the value function, which we discuss below.

30.3.6 Greedy Policies

The primary importance of the value function is that we can use it to compute optimal policies.

The details are as follows.

Given a continuous function v on \mathbb{R}_+ , we say that $\sigma \in \Sigma$ is **v -greedy** if $\sigma(y)$ is a solution to

$$\max_{0 \leq c \leq y} \left\{ u(c) + \beta \int v(f(y - c)z) \phi(dz) \right\} \quad (10)$$

for every $y \in \mathbb{R}_+$.

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

- A feasible consumption policy is optimal if and only if it is v^* -greedy.

The intuition is similar to the intuition for the Bellman equation, which was provided after (9).

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.

30.3.7 The Bellman Operator

How, then, should we compute the value function?

One way is to use the so-called **Bellman operator**.

(An operator is a map that sends functions into functions.)

The Bellman operator is denoted by T and defined by

$$Tv(y) := \max_{0 \leq c \leq y} \left\{ u(c) + \beta \int v(f(y - c)z) \phi(dz) \right\} \quad (y \in \mathbb{R}_+) \quad (11)$$

In other words, T sends the function v into the new function Tv defined by (11).

By construction, the set of solutions to the Bellman equation (9) *exactly coincides with* the set of fixed points of T .

For example, if $Tv = v$, then, for any $y \geq 0$,

$$v(y) = Tv(y) = \max_{0 \leq c \leq y} \left\{ u(c) + \beta \int v^*(f(y - c)z) \phi(dz) \right\}$$

which says precisely that v is a solution to the Bellman equation.

It follows that v^* is a fixed point of T .

30.3.8 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

$$\rho(g, h) = \sup_{y \geq 0} |g(y) - h(y)|$$

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, \dots 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 (see [EDTC](#), theorem 10.1.11 if you get stuck).

Hence, at least one optimal policy exists.

Our problem now is how to compute it.

30.3.9 Unbounded Utility

The results stated above assume that the utility function is bounded.

In practice economists often work with unbounded utility functions — and so will we.

In the unbounded setting, various optimality theories exist.

Unfortunately, they tend to be case-specific, as opposed to valid for a large range of applications.

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”).

Consult, for example, section 12.2 of [EDTC](#), [64] or [78].

30.4 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.

Both of these things are helpful, but they do cost us some speed — as you will see when you run the code.

[Later](#) we will sacrifice some of this clarity and flexibility in order to accelerate our code with just-in-time (JIT) compilation.

The algorithm we will use is fitted value function iteration, which was described in earlier lectures [the McCall model](#) and [cake eating](#).

The algorithm will be

1. Begin with an array of values $\{v_1, \dots, v_I\}$ representing the values of some initial function v on the grid points $\{y_1, \dots, y_I\}$.
2. Build a function \hat{v} on the state space \mathbb{R}_+ by linear interpolation, based on these data points.
3. Obtain and record the value $T\hat{v}(y_i)$ on each grid point y_i by repeatedly solving (11).
4. Unless some stopping condition is satisfied, set $\{v_1, \dots, v_I\} = \{T\hat{v}(y_1), \dots, T\hat{v}(y_I)\}$ and go to step 2.

30.4.1 Scalar Maximization

To maximize the right hand side of the Bellman equation (9), we are going to use the `minimize_scalar` routine from SciPy.

Since we are maximizing rather than minimizing, we will use the fact that the maximizer of g on the interval $[a, b]$ is the minimizer of $-g$ on the same interval.

To this end, and to keep the interface tidy, we will wrap `minimize_scalar` in an outer function as follows:

```
In [2]: def maximize(g, a, b, args):
    """
    Maximize the function g over the interval [a, b].
    We use the fact that the maximizer of g on any interval is
    also the minimizer of -g. The tuple args collects any extra
    arguments to g.

    Returns the maximal value and the maximizer.
    """
    objective = lambda x: -g(x, *args)
    result = minimize_scalar(objective, bounds=(a, b), method='bounded')
    maximizer, maximum = result.x, -result.fun
    return maximizer, maximum
```

30.4.2 Optimal Growth Model

We will assume for now that ϕ is the distribution of $\xi := \exp(\mu + s\zeta)$ where

- ζ is standard normal,
- μ is a shock location parameter and
- s is a shock scale parameter.

We will store this and other primitives of the optimal growth model in a class.

The class, defined below, combines both parameters and a method that realizes the right hand side of the Bellman equation (9).

```
In [3]: class OptimalGrowthModel:

    def __init__(self,
                 u,                      # utility function
                 f,                      # production function
                 β=0.96,                 # discount factor
                 μ=0,                     # shock location parameter
                 s=0.1,                   # shock scale parameter
                 grid_max=4,
                 grid_size=120,
                 shock_size=250,
                 seed=1234):

        self.u, self.f, self.β, self.μ, self.s = u, f, β, μ, s

        # Set up grid
        self.grid = np.linspace(1e-4, grid_max, grid_size)

        # Store shocks (with a seed, so results are reproducible)
        np.random.seed(seed)
        self.shocks = np.exp(μ + s * np.random.randn(shock_size))

    def state_action_value(self, c, y, v_array):
        """
        Right hand side of the Bellman equation.
```

```

"""
u, f, beta, shocks = self.u, self.f, self.beta, self.shocks
v = interp1d(self.grid, v_array)
return u(c) + beta * np.mean(v(f(y - c) * shocks))

```

In the second last line we are using linear interpolation.

In the last line, the expectation in (11) is computed via [Monte Carlo](#), using the approximation

$$\int v(f(y - c)z)\phi(dz) \approx \frac{1}{n} \sum_{i=1}^n v(f(y - c)\xi_i)$$

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., [85].)

30.4.3 The Bellman Operator

The next function implements the Bellman operator.

(We could have added it as a method to the `OptimalGrowthModel` class, but we prefer small classes rather than monolithic ones for this kind of numerical work.)

```

In [4]: def T(v, og):
"""
The Bellman operator. Updates the guess of the value function
and also computes a v-greedy policy.

* og is an instance of OptimalGrowthModel
* v is an array representing a guess of the value function

"""
v_new = np.empty_like(v)
v_greedy = np.empty_like(v)

for i in range(len(grid)):
    y = grid[i]

    # Maximize RHS of Bellman equation at state y
    c_star, v_max = maximize(og.state_action_value, 1e-10, y, (y, v))
    v_new[i] = v_max
    v_greedy[i] = c_star

return v_greedy, v_new

```

30.4.4 An Example

Let's suppose now that

$$f(k) = k^\alpha \quad \text{and} \quad u(c) = \ln c$$

For this particular problem, an exact analytical solution is available (see [72], section 3.1.2), with

$$v^*(y) = \frac{\ln(1 - \alpha\beta)}{1 - \beta} + \frac{(\mu + \alpha \ln(\alpha\beta))}{1 - \alpha} \left[\frac{1}{1 - \beta} - \frac{1}{1 - \alpha\beta} \right] + \frac{1}{1 - \alpha\beta} \ln y \quad (12)$$

and optimal consumption policy

$$\sigma^*(y) = (1 - \alpha\beta)y$$

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:

```
In [5]: def v_star(y, 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(y)

def σ_star(y, α, β):
    """
    True optimal policy
    """
    return (1 - α * β) * y
```

Next let's create an instance of the model with the above primitives and assign it to the variable `og`.

```
In [6]: α = 0.4
def fcd(k):
    return k**α

og = OptimalGrowthModel(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.

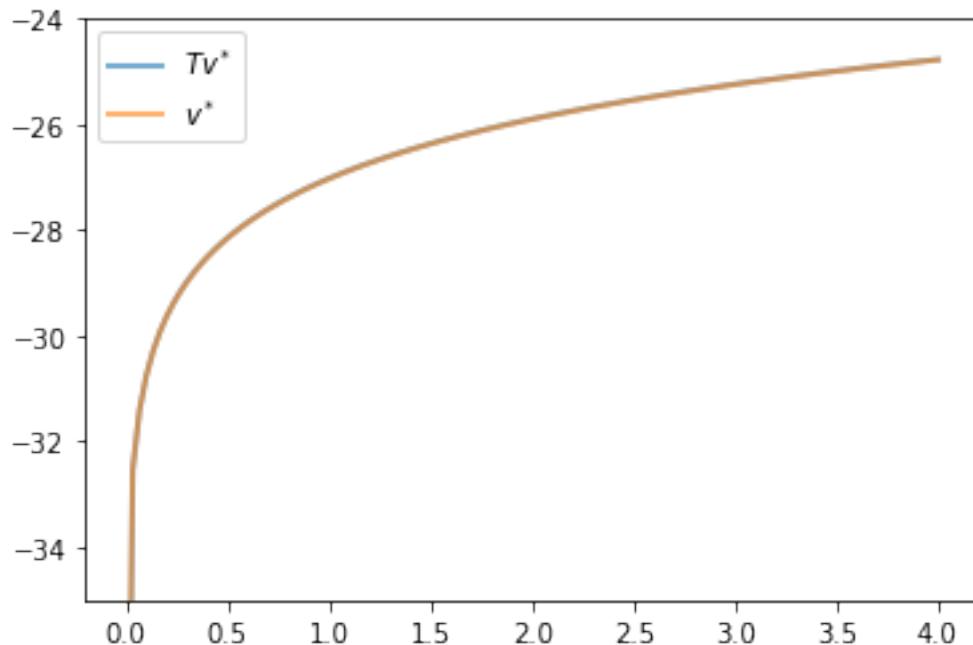
In theory, since v^* is a fixed point, the resulting function should again be v^* .

In practice, we expect some small numerical error.

In [7]: `grid = og.grid`

```
v_init = v_star(grid, α, og.β, og.μ)      # Start at the solution
v_greedy, v = T(v_init, og)                  # Apply T once

fig, ax = plt.subplots()
ax.set_ylim(-35, -24)
ax.plot(grid, v, lw=2, alpha=0.6, label='$Tv^*$')
ax.plot(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, $v(y) = 5 \ln(y)$.

In [8]: `v = 5 * np.log(grid) # An initial condition`
`n = 35`

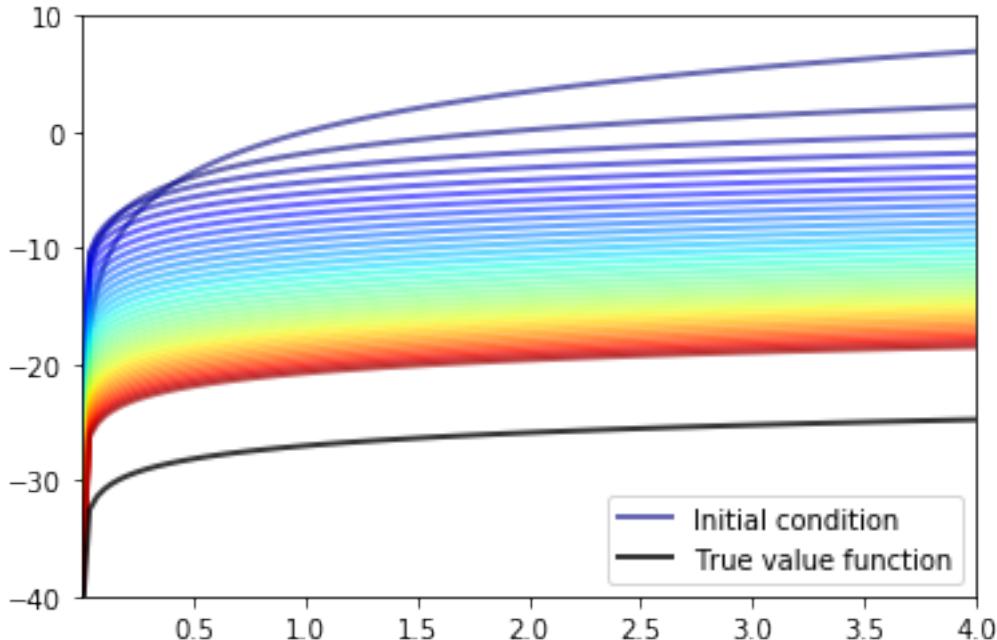
```
fig, ax = plt.subplots()

ax.plot(grid, v, color=plt.cm.jet(0),
         lw=2, alpha=0.6, label='Initial condition')

for i in range(n):
    v_greedy, v = T(v, og) # Apply the Bellman operator
    ax.plot(grid, v, color=plt.cm.jet(i / n), lw=2, alpha=0.6)

ax.plot(grid, v_star(grid, α, og.β, og.μ), 'k-', lw=2,
        alpha=0.8, label='True value function')
```

```
ax.legend()
ax.set(ylim=(-40, 10), xlim=(np.min(grid), np.max(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.

30.4.5 Iterating to Convergence

We can write a function that iterates until the difference is below a particular tolerance level.

```
In [9]: def solve_model(og,
                      tol=1e-4,
                      max_iter=1000,
                      verbose=True,
                      print_skip=25):
    """
    Solve model by iterating with the Bellman operator.

    """
    # Set up loop
    v = og.u(og.grid) # Initial condition
```

```

i = 0
error = tol + 1

while i < max_iter and error > tol:
    v_greedy, v_new = T(v, og)
    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 i == max_iter:
    print("Failed to converge!")

if verbose and i < max_iter:
    print(f"\nConverged in {i} iterations.")

return v_greedy, v_new

```

Let's use this function to compute an approximate solution at the defaults.

In [10]: `v_greedy, v_solution = solve_model(og)`

```

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 0.006912744396029069.
Error at iteration 150 is 0.002491330384817303.
Error at iteration 175 is 0.000897867291303811.
Error at iteration 200 is 0.00032358842396718046.
Error at iteration 225 is 0.00011662020561331587.

```

Converged in 229 iterations.

Now we check our result by plotting it against the true value:

In [11]: `fig, ax = plt.subplots()`

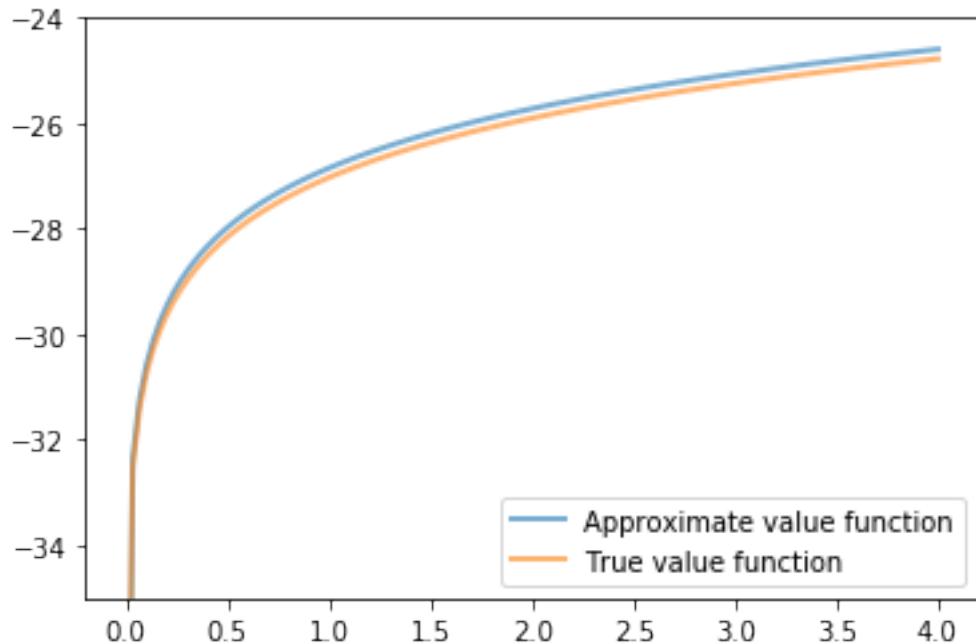
```

ax.plot(grid, v_solution, lw=2, alpha=0.6,
        label='Approximate value function')

ax.plot(grid, v_star(grid, α, og.β, og.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.

30.4.6 The Policy Function

The policy `v_greedy` computed above corresponds to an approximate optimal policy.

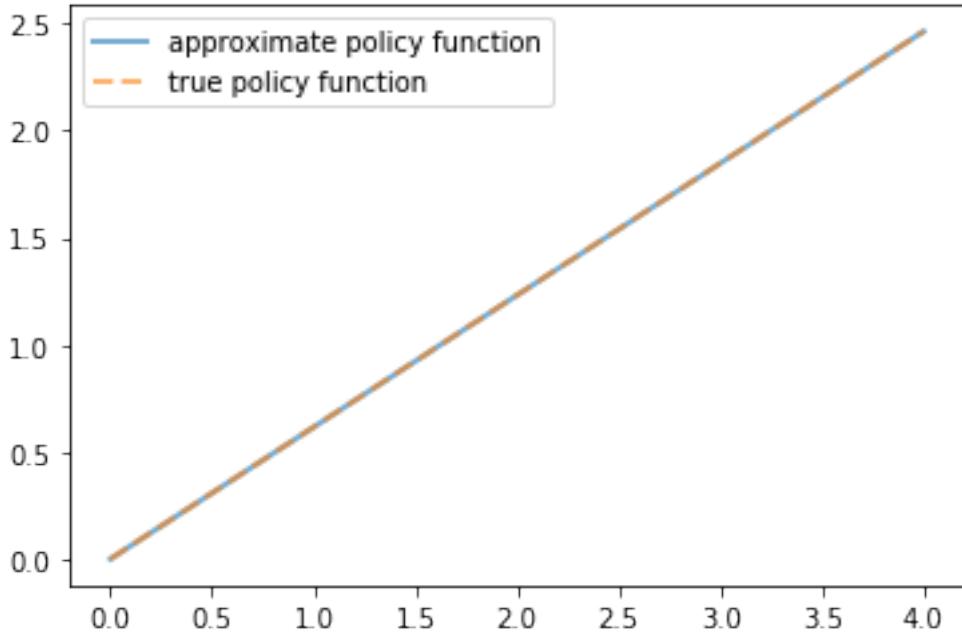
The next figure compares it to the exact solution, which, as mentioned above, is $\sigma(y) = (1 - \alpha\beta)y$

```
In [12]: fig, ax = plt.subplots()

ax.plot(grid, v_greedy, lw=2,
         alpha=0.6, label='approximate policy function')

ax.plot(grid, sigma_star(grid, alpha, omega_beta), '--',
         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.

30.5 Exercises

30.5.1 Exercise 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 growth model with this utility specification.

Setting $\gamma = 1.5$, compute and plot an estimate of the optimal policy.

Time how long this function takes to run, so you can compare it to faster code developed in the [next lecture](#).

30.5.2 Exercise 2

Time how long it takes to iterate with the Bellman operator 20 times, starting from initial condition $v(y) = u(y)$.

Use the model specification in the previous exercise.

(As before, we will compare this number with that for the faster code developed in the [next lecture](#).)

30.6 Solutions

30.6.1 Exercise 1

Here we set up the model.

```
In [13]: γ = 1.5 # Preference parameter

def u_crra(c):
    return (c**(1 - γ) - 1) / (1 - γ)

og = OptimalGrowthModel(u=u_crra, f=fcd)
```

Now let's run it, with a timer.

```
In [14]: %time
v_greedy, v_solution = solve_model(og)
```

```
Error at iteration 25 is 0.5528151810417512.
Error at iteration 50 is 0.19923228425590978.
Error at iteration 75 is 0.07180266113800826.
Error at iteration 100 is 0.025877443335843964.
Error at iteration 125 is 0.009326145618970827.
Error at iteration 150 is 0.003361112262005861.
Error at iteration 175 is 0.0012113338243295857.
Error at iteration 200 is 0.0004365607333056687.
Error at iteration 225 is 0.00015733505506432266.

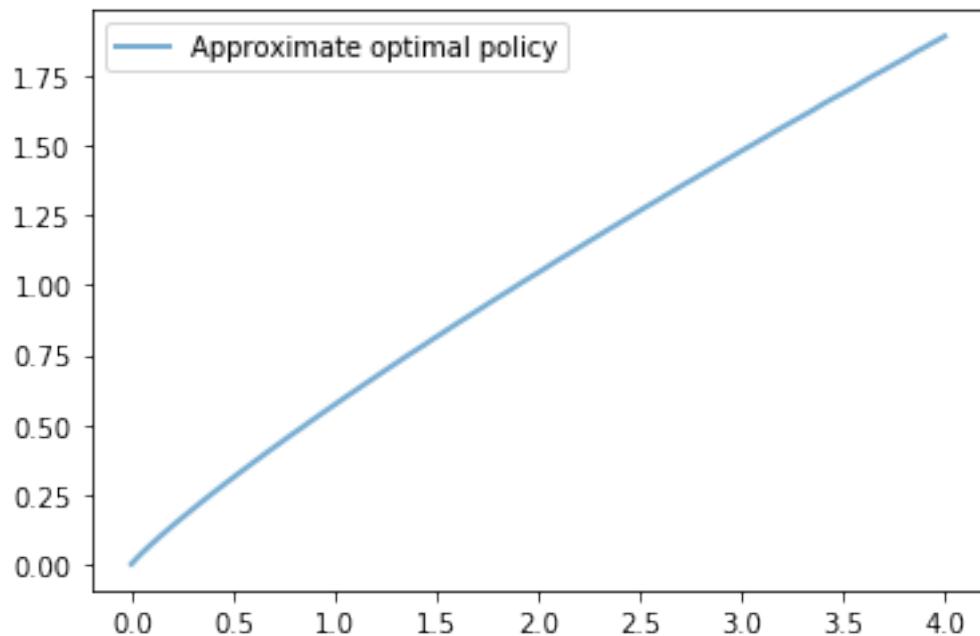
Converged in 237 iterations.
CPU times: user 45.9 s, sys: 3.99 ms, total: 45.9 s
Wall time: 45.9 s
```

Let's plot the policy function just to see what it looks like:

```
In [15]: fig, ax = plt.subplots()

ax.plot(grid, v_greedy, lw=2,
         alpha=0.6, label='Approximate optimal policy')

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



30.6.2 Exercise 2

Let's set up:

```
In [16]: og = OptimalGrowthModel(u=u_crra, f=fcd)
v = og.u(og.grid)
```

Here's the timing:

```
In [17]: %time
```

```
for i in range(20):
    v_greedy, v_new = T(v, og)
    v = v_new
```

```
CPU times: user 3.47 s, sys: 4 µs, total: 3.47 s
Wall time: 3.47 s
```


Chapter 31

Optimal Growth II: Accelerating the Code with Numba

31.1 Contents

- Overview 31.2
- The Model 31.3
- Computation 31.4
- Exercises 31.5
- Solutions 31.6

In addition to what's in Anaconda, this lecture will need the following libraries:

```
In [1]: !pip install quantecon  
!pip install interpolation
```

31.2 Overview

Previously, we studied a stochastic optimal growth model with one representative agent.

We solved the model using dynamic programming.

In writing our code, we focused on clarity and flexibility.

These are important, but there's often a trade-off between flexibility and speed.

The reason is that, when code is less flexible, we can exploit structure more easily.

(This is true about algorithms and mathematical problems more generally: more specific problems have more structure, which, with some thought, can be exploited for better results.)

So, in this lecture, we are going to accept less flexibility while gaining speed, using just-in-time (JIT) compilation to accelerate our code.

Let's start with some imports:

```
In [2]: import numpy as np  
import matplotlib.pyplot as plt  
from interpolation import interp  
from numba import jit, njit, prange, float64, int32
```

```

from numba.experimental import jitclass
from quantecon.optimize.scalar_maximization import brent_max

%matplotlib inline

/home/ubuntu/anaconda3/lib/python3.7/site-packages/numba/np/ufunc/parallel.py:
 355:
NumbaWarning: The TBB threading layer requires TBB version 2019.5 or later i.e.,
TBB_INTERFACE_VERSION >= 11005. Found TBB_INTERFACE_VERSION = 11004. The TBB
  threading
layer is disabled.
    warnings.warn(problem)

```

We are using an interpolation function from `interpolation.py` because it helps us JIT-compile our code.

The function `brent_max` is also designed for embedding in JIT-compiled code.

These are alternatives to similar functions in SciPy (which, unfortunately, are not JIT-aware).

31.3 The Model

The model is the same as discussed in our [previous lecture](#) on optimal growth.

We will start with log utility:

$$u(c) = \ln(c)$$

We continue to assume that

- $f(k) = k^\alpha$
- ϕ is the distribution of $\xi := \exp(\mu + s\zeta)$ when ζ is standard normal

We will once again use value function iteration to solve the model.

In particular, the algorithm is unchanged, and the only difference is in the implementation itself.

As before, we will be able to compare with the true solutions

```

In [3]: def v_star(y, alpha, beta, mu):
    """
    True value function
    """
    c1 = np.log(1 - alpha * beta) / (1 - beta)
    c2 = (mu + alpha * np.log(alpha * beta)) / (1 - alpha)
    c3 = 1 / (1 - beta)
    c4 = 1 / (1 - alpha * beta)
    return c1 + c2 * (c3 - c4) + c4 * np.log(y)

def o_star(y, alpha, beta):
    """
    True optimal policy
    """
    return (1 - alpha * beta) * y

```

31.4 Computation

We will again store the primitives of the optimal growth model in a class.

But now we are going to use Numba's `@jitclass` decorator to target our class for JIT compilation.

Because we are going to use Numba to compile our class, we need to specify the data types.

You will see this as a list called `opt_growth_data` above our class.

Unlike in the [previous lecture](#), we hardwire the production and utility specifications into the class.

This is where we sacrifice flexibility in order to gain more speed.

```
In [4]: opt_growth_data = [
    ('α', float64),           # Production parameter
    ('β', float64),           # Discount factor
    ('μ', float64),           # Shock location parameter
    ('s', float64),           # Shock scale parameter
    ('grid', float64[:]),     # Grid (array)
    ('shocks', float64[:])    # Shock draws (array)
]

@jitclass(opt_growth_data)
class OptimalGrowthModel:

    def __init__(self,
                 α=0.4,
                 β=0.96,
                 μ=0,
                 s=0.1,
                 grid_max=4,
                 grid_size=120,
                 shock_size=250,
                 seed=1234):

        self.α, self.β, self.μ, self.s = α, β, μ, s

        # Set up grid
        self.grid = np.linspace(1e-5, grid_max, grid_size)

        # Store shocks (with a seed, so results are reproducible)
        np.random.seed(seed)
        self.shocks = np.exp(μ + s * np.random.randn(shock_size))

    def f(self, k):
        "The production function"
        return k**self.α

    def u(self, c):
        "The utility function"
        return np.log(c)

    def f_prime(self, k):
        "Derivative of f"
```

```

    return self.α * (k**(self.α - 1))

def u_prime(self, c):
    "Derivative of u"
    return 1/c

def u_prime_inv(self, c):
    "Inverse of u'"
    return 1/c

```

The class includes some methods such as `u_prime` that we do not need now but will use in later lectures.

31.4.1 The Bellman Operator

We will use JIT compilation to accelerate the Bellman operator.

First, here's a function that returns the value of a particular consumption choice `c`, given state `y`, as per the Bellman equation (9).

```
In [5]: @njit
def state_action_value(c, y, v_array, og):
    """
    Right hand side of the Bellman equation.

    * c is consumption
    * y is income
    * og is an instance of OptimalGrowthModel
    * v_array represents a guess of the value function on the grid

    """
    u, f, β, shocks = og.u, og.f, og.β, og.shocks
    v = lambda x: interp(og.grid, v_array, x)
    return u(c) + β * np.mean(v(f(y - c) * shocks))
```

Now we can implement the Bellman operator, which maximizes the right hand side of the Bellman equation:

```
In [6]: @jit(nopython=True)
def T(v, og):
    """
    The Bellman operator.

    * og is an instance of OptimalGrowthModel
    * v is an array representing a guess of the value function

    """
    v_new = np.empty_like(v)
    v_greedy = np.empty_like(v)
```

```

for i in range(len(og.grid)):
    y = og.grid[i]

    # Maximize RHS of Bellman equation at state y
    result = brent_max(state_action_value, 1e-10, y, args=(y, v, og))
    v_greedy[i], v_new[i] = result[0], result[1]

return v_greedy, v_new

```

We use the `solve_model` function to perform iteration until convergence.

```

In [7]: def solve_model(og,
                      tol=1e-4,
                      max_iter=1000,
                      verbose=True,
                      print_skip=25):
    """
    Solve model by iterating with the Bellman operator.

    """

    # Set up loop
    v = og.u(og.grid) # Initial condition
    i = 0
    error = tol + 1

    while i < max_iter and error > tol:
        v_greedy, v_new = T(v, og)
        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 i == max_iter:
        print("Failed to converge!")

    if verbose and i < max_iter:
        print(f"\nConverged in {i} iterations.")

    return v_greedy, v_new

```

Let's compute the approximate solution at the default parameters.

First we create an instance:

```
In [8]: og = OptimalGrowthModel()
```

Now we call `solve_model`, using the `%%time` magic to check how long it takes.

```
In [9]: %%time
v_greedy, v_solution = solve_model(og)
```

```
Error at iteration 25 is 0.41372668361362486.
Error at iteration 50 is 0.14767653072604503.
```

```
Error at iteration 75 is 0.053221715530327174.
Error at iteration 100 is 0.019180931418532055.
Error at iteration 125 is 0.006912744709513419.
Error at iteration 150 is 0.002491330497818467.
Error at iteration 175 is 0.0008978673320534369.
Error at iteration 200 is 0.0003235884386789678.
Error at iteration 225 is 0.00011662021094238639.
```

```
Converged in 229 iterations.
CPU times: user 14.5 s, sys: 47.5 ms, total: 14.6 s
Wall time: 14.9 s
```

You will notice that this is *much* faster than our [original implementation](#).

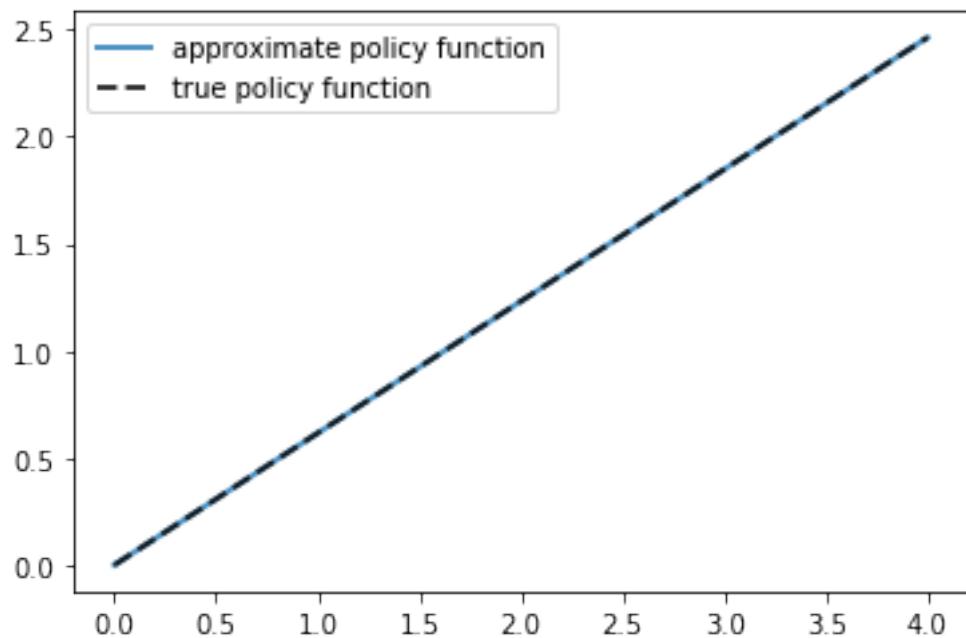
Here is a plot of the resulting policy, compared with the true policy:

```
In [10]: fig, ax = plt.subplots()

ax.plot(og.grid, v_greedy, lw=2,
        alpha=0.8, label='approximate policy function')

ax.plot(og.grid, σ_star(og.grid, og.α, og.β), 'k--',
        lw=2, alpha=0.8, label='true policy function')

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



Again, the fit is excellent — this is as expected since we have not changed the algorithm.

The maximal absolute deviation between the two policies is

```
In [11]: np.max(np.abs(v_greedy - σ_star(og.grid, og.α, og.β)))
```

```
Out[11]: 0.0010480511607799947
```

31.5 Exercises

31.5.1 Exercise 1

Time how long it takes to iterate with the Bellman operator 20 times, starting from initial condition $v(y) = u(y)$.

Use the default parameterization.

31.5.2 Exercise 2

Modify the optimal growth model to use the CRRA utility specification.

$$u(c) = \frac{c^{1-\gamma}}{1-\gamma}$$

Set $\gamma = 1.5$ as the default value and maintaining other specifications.

(Note that `jitclass` currently does not support inheritance, so you will have to copy the class and change the relevant parameters and methods.)

Compute an estimate of the optimal policy, plot it and compare visually with the same plot from the [analogous exercise](#) in the first optimal growth lecture.

Compare execution time as well.

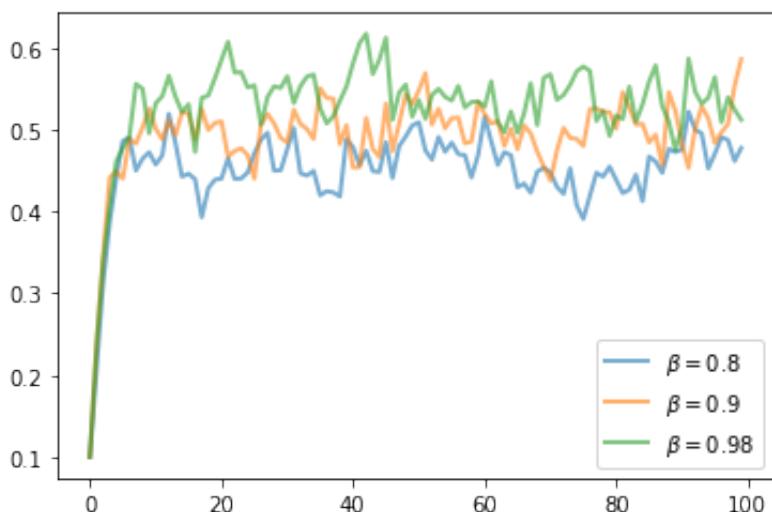
31.5.3 Exercise 3

In this exercise we return to the original log utility specification.

Once an optimal consumption policy σ is given, income follows

$$y_{t+1} = f(y_t - \sigma(y_t))\xi_{t+1}$$

The next figure shows a simulation of 100 elements of this sequence for three different discount factors (and hence three different policies).



In each sequence, the initial condition is $y_0 = 0.1$.

The discount factors are `discount_factors = (0.8, 0.9, 0.98)`.

We have also dialed down the shocks a bit with `s = 0.05`.

Otherwise, the parameters and primitives are the same as the log-linear model discussed earlier in the lecture.

Notice that more patient agents typically have higher wealth.

Replicate the figure modulo randomness.

31.6 Solutions

31.6.1 Exercise 1

Let's set up the initial condition.

```
In [12]: v = og.u(og.grid)
```

Here's the timing:

```
In [13]: %time
```

```
for i in range(20):
    v_greedy, v_new = T(v, og)
    v = v_new

CPU times: user 898 ms, sys: 4.01 ms, total: 902 ms
Wall time: 945 ms
```

Compared with our [timing](#) for the non-compiled version of value function iteration, the JIT-compiled code is usually an order of magnitude faster.

31.6.2 Exercise 2

Here's our CRRA version of `OptimalGrowthModel`:

```
In [14]: opt_growth_data = [
    ('alpha', float64),           # Production parameter
    ('beta', float64),            # Discount factor
    ('mu', float64),              # Shock location parameter
    ('gamma', float64),            # Preference parameter
    ('sigma', float64),            # Shock scale parameter
    ('grid', float64[:]),          # Grid (array)
    ('shocks', float64[:])         # Shock draws (array)
]

@jitclass(opt_growth_data)
class OptimalGrowthModel_CRRA:
```

```

def __init__(self,
            α=0.4,
            β=0.96,
            μ=0,
            S=0.1,
            γ=1.5,
            grid_max=4,
            grid_size=120,
            shock_size=250,
            seed=1234):

    self.α, self.β, self.γ, self.μ, self.S = α, β, γ, μ, S

    # Set up grid
    self.grid = np.linspace(1e-5, grid_max, grid_size)

    # Store shocks (with a seed, so results are reproducible)
    np.random.seed(seed)
    self.shocks = np.exp(μ + S * np.random.randn(shock_size))

def f(self, k):
    "The production function."
    return k**self.α

def u(self, c):
    "The utility function."
    return c**(1 - self.γ) / (1 - self.γ)

def f_prime(self, k):
    "Derivative of f."
    return self.α * (k**(self.α - 1))

def u_prime(self, c):
    "Derivative of u."
    return c**(-self.γ)

def u_prime_inv(c):
    return c**(-1 / self.γ)

```

Let's create an instance:

In [15]: `og_crqa = OptimalGrowthModel_CRRA()`

Now we call `solve_model`, using the `%time` magic to check how long it takes.

In [16]: `%time`
`v_greedy, v_solution = solve_model(og_crqa)`

```

Error at iteration 25 is 1.6201897527216715.
Error at iteration 50 is 0.4591060470565935.
Error at iteration 75 is 0.1654235221617455.
Error at iteration 100 is 0.05961808343499797.
Error at iteration 125 is 0.021486161531640846.
Error at iteration 150 is 0.007743542074422294.
Error at iteration 175 is 0.0027907471408923357.

```

```
Error at iteration 200 is 0.0010057761070925153.
Error at iteration 225 is 0.0003624784085332067.
Error at iteration 250 is 0.00013063602803242702.
```

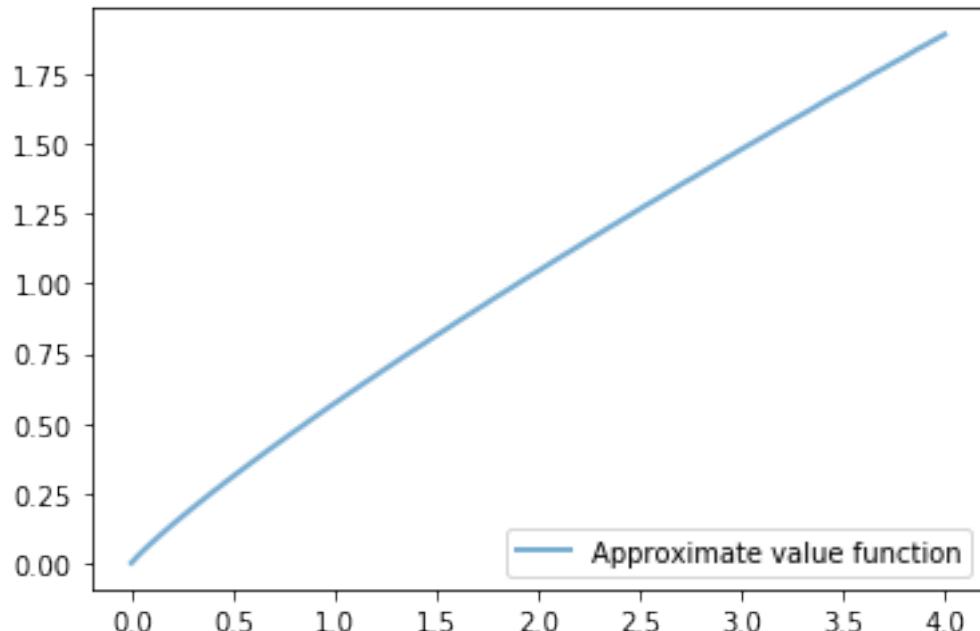
```
Converged in 257 iterations.
CPU times: user 13.7 s, sys: 7.95 ms, total: 13.7 s
Wall time: 13.9 s
```

Here is a plot of the resulting policy:

```
In [17]: fig, ax = plt.subplots()

ax.plot(og.grid, v_greedy, lw=2,
        alpha=0.6, label='Approximate value function')

ax.legend(loc='lower right')
plt.show()
```



This matches the solution that we obtained in our non-jitted code, [in the exercises](#).

Execution time is an order of magnitude faster.

31.6.3 Exercise 3

Here's one solution:

```
In [18]: def simulate_og(sigma_func, og, y0=0.1, ts_length=100):
    """
    Compute a time series given consumption policy sigma.
    """
    y = np.empty(ts_length)
```

```

ξ = np.random.randn(ts_length-1)
y[0] = y0
for t in range(ts_length-1):
    y[t+1] = (y[t] - σ_func(y[t]))**og.α * np.exp(og.μ + og.s * ξ[t])
return y

```

```

In [19]: fig, ax = plt.subplots()

for β in (0.8, 0.9, 0.98):

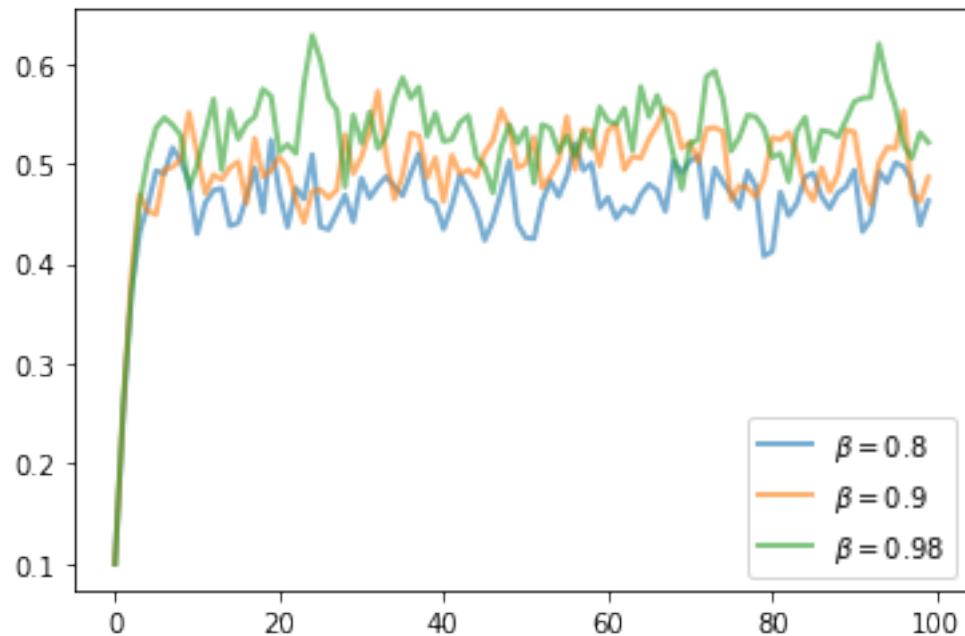
    og = OptimalGrowthModel(β=β, s=0.05)

    v_greedy, v_solution = solve_model(og, verbose=False)

    # Define an optimal policy function
    σ_func = lambda x: interp(og.grid, v_greedy, x)
    y = simulate_og(σ_func, og)
    ax.plot(y, lw=2, alpha=0.6, label=r'$\beta = \{\beta\}$')

ax.legend(loc='lower right')
plt.show()

```



Chapter 32

Optimal Growth III: Time Iteration

32.1 Contents

- Overview 32.2
- The Euler Equation 32.3
- Implementation 32.4
- Exercises 32.5
- Solutions 32.6

In addition to what's in Anaconda, this lecture will need the following libraries:

```
In [1]: !pip install quantecon  
!pip install interpolation
```

32.2 Overview

In this lecture, we'll continue our [earlier](#) study of the stochastic optimal growth model.

In that lecture, we solved the associated dynamic programming problem using value function iteration.

The beauty of this technique is its broad applicability.

With numerical problems, however, we can often attain higher efficiency in specific applications by deriving methods that are carefully tailored to the application at hand.

The stochastic optimal growth model has plenty of structure to exploit for this purpose, especially when we adopt some concavity and smoothness assumptions over primitives.

We'll use this structure to obtain an Euler equation based method.

This will be an extension of the time iteration method considered in our elementary lecture on [cake eating](#).

In a [subsequent lecture](#), we'll see that time iteration can be further adjusted to obtain even more efficiency.

Let's start with some imports:

```
In [2]: import numpy as np  
import quantecon as qe
```

```

import matplotlib.pyplot as plt
%matplotlib inline

from interpolation import interp
from quantecon.optimize import brentq
from numba import njit, float64
from numba.experimental import jitclass

/home/ubuntu/anaconda3/lib/python3.7/site-packages/numba/np/ufunc/parallel.py:
  ↵355:
NumbaWarning: The TBB threading layer requires TBB version 2019.5 or later i.e.,
TBB_INTERFACE_VERSION >= 11005. Found TBB_INTERFACE_VERSION = 11004. The TBB
  ↵threading
layer is disabled.
    warnings.warn(problem)

```

32.3 The Euler Equation

Our first step is to derive the Euler equation, which is a generalization of the Euler equation we obtained in the [lecture on cake eating](#).

We take the model set out in [the stochastic growth model lecture](#) and add the following assumptions:

1. u and f are continuously differentiable and strictly concave
2. $f(0) = 0$
3. $\lim_{c \rightarrow 0} u'(c) = \infty$ and $\lim_{c \rightarrow \infty} u'(c) = 0$
4. $\lim_{k \rightarrow 0} f'(k) = \infty$ and $\lim_{k \rightarrow \infty} f'(k) = 0$

The last two conditions are usually called **Inada conditions**.

Recall the Bellman equation

$$v^*(y) = \max_{0 \leq c \leq y} \left\{ u(c) + \beta \int v^*(f(y - c)z) \phi(dz) \right\} \quad \text{for all } y \in \mathbb{R}_+ \quad (1)$$

Let the optimal consumption policy be denoted by σ^* .

We know that σ^* is a v^* -greedy policy so that $\sigma^*(y)$ is the maximizer in (1).

The conditions above imply that

- σ^* is the unique optimal policy for the stochastic optimal growth model
- the optimal policy is continuous, strictly increasing and also **interior**, in the sense that $0 < \sigma^*(y) < y$ for all strictly positive y , and
- the value function is strictly concave and continuously differentiable, with

$$(v^*)'(y) = u'(\sigma^*(y)) := (u' \circ \sigma^*)(y) \quad (2)$$

The last result is called the **envelope condition** due to its relationship with the [envelope theorem](#).

To see why (2) holds, write the Bellman equation in the equivalent form

$$v^*(y) = \max_{0 \leq k \leq y} \left\{ u(y - k) + \beta \int v^*(f(k)z) \phi(dz) \right\},$$

Differentiating with respect to y , and then evaluating at the optimum yields (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 (1), which is

$$u'(\sigma^*(y)) = \beta \int (v^*)'(f(y - \sigma^*(y))z) f'(y - \sigma^*(y)) z \phi(dz) \quad (3)$$

Combining (2) and the first-order condition (3) gives the **Euler equation**

$$(u' \circ \sigma^*)(y) = \beta \int (u' \circ \sigma^*)(f(y - \sigma^*(y))z) f'(y - \sigma^*(y)) z \phi(dz) \quad (4)$$

We can think of the Euler equation as a functional equation

$$(u' \circ \sigma)(y) = \beta \int (u' \circ \sigma)(f(y - \sigma(y))z) f'(y - \sigma(y)) z \phi(dz) \quad (5)$$

over interior consumption policies σ , one solution of which is the optimal policy σ^* .

Our aim is to solve the functional equation (5) and hence obtain σ^* .

32.3.1 The Coleman-Reffett Operator

Recall the Bellman operator

$$Tv(y) := \max_{0 \leq c \leq y} \left\{ u(c) + \beta \int v(f(y - c)z) \phi(dz) \right\} \quad (6)$$

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 \mathcal{P}

1. The operator K takes as its argument a $\sigma \in \mathcal{P}$ and
2. returns a new function $K\sigma$, where $K\sigma(y)$ is the $c \in (0, y)$ that solves.

$$u'(c) = \beta \int (u' \circ \sigma)(f(y - c)z) f'(y - c) z \phi(dz) \quad (7)$$

We call this operator the **Coleman-Reffett operator** to acknowledge the work of [23] and [89].

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 (5).

In particular, the optimal policy σ^* is a fixed point.

Indeed, for fixed y , the value $K\sigma^*(y)$ is the c that solves

$$u'(c) = \beta \int (u' \circ \sigma^*)(f(y - c)z)f'(y - c)z\phi(dz)$$

In view of the Euler equation, this is exactly $\sigma^*(y)$.

32.3.2 Is the Coleman-Reffett Operator Well Defined?

In particular, is there always a unique $c \in (0, y)$ that solves (7)?

The answer is yes, under our assumptions.

For any $\sigma \in \mathcal{P}$, the right side of (7)

- is continuous and strictly increasing in c on $(0, y)$
- diverges to $+\infty$ as $c \uparrow y$

The left side of (7)

- is continuous and strictly decreasing in c on $(0, y)$
- 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 $(0, y)$.

With a bit more analysis, one can show in addition that $K\sigma \in \mathcal{P}$ whenever $\sigma \in \mathcal{P}$.

32.3.3 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, the two operators are *topologically conjugate*.

Loosely speaking, this means 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.

Examples are given below.

32.4 Implementation

As in our [previous study](#), we continue to assume that

- $u(c) = \ln c$
- $f(k) = k^\alpha$
- ϕ is the distribution of $\xi := \exp(\mu + s\zeta)$ when ζ is standard normal

This will allow us to compare our results to the analytical solutions

```
In [3]: def v_star(y, 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(y)

def σ_star(y, α, β):
    """
    True optimal policy
    """
    return (1 - α * β) * y
```

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' .

These are available in a class called `OptimalGrowthModel` that we constructed in an [earlier lecture](#).

```
In [4]: opt_growth_data = [
    ('α', float64),           # Production parameter
    ('β', float64),           # Discount factor
    ('μ', float64),           # Shock location parameter
    ('s', float64),           # Shock scale parameter
    ('grid', float64[:]),     # Grid (array)
    ('shocks', float64[:])    # Shock draws (array)
]

@jitclass(opt_growth_data)
class OptimalGrowthModel:

    def __init__(self,
                 α=0.4,
                 β=0.96,
                 μ=0,
                 s=0.1,
                 grid_max=4,
                 grid_size=120,
                 shock_size=250,
                 seed=1234):

        self.α, self.β, self.μ, self.s = α, β, μ, s
```

```

# Set up grid
self.grid = np.linspace(1e-5, grid_max, grid_size)

# Store shocks (with a seed, so results are reproducible)
np.random.seed(seed)
self.shocks = np.exp(mu + s * np.random.randn(shock_size))

def f(self, k):
    "The production function"
    return k**self.alpha

def u(self, c):
    "The utility function"
    return np.log(c)

def f_prime(self, k):
    "Derivative of f"
    return self.alpha * (k**(self.alpha - 1))

def u_prime(self, c):
    "Derivative of u"
    return 1/c

def u_prime_inv(self, c):
    "Inverse of u"
    return 1/c

```

Now we implement a method called `euler_diff`, which returns

$$u'(c) - \beta \int (u' \circ \sigma)(f(y - c)z)f'(y - c)z\phi(dz) \quad (8)$$

```
In [5]: @njit
def euler_diff(c, sigma, y, og):
    """
    Set up a function such that the root with respect to c,
    given y and sigma, is equal to Ksigma(y).

    """
    beta, shocks, grid = og.beta, og.shocks, og.grid
    f, f_prime, u_prime = og.f, og.f_prime, og.u_prime

    # First turn sigma into a function via interpolation
    sigma_func = lambda x: interp(grid, sigma, x)

    # Now set up the function we need to find the root of.
    vals = u_prime(sigma_func(f(y - c) * shocks)) * f_prime(y - c) * shocks
    return u_prime(c) - beta * 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 (8) for c given state y and σ , the current guess of the policy.

Here's the operator K , that implements the root-finding step.

```
In [6]: @njit
def K(σ, og):
    """
        The Coleman-Reffett operator

        Here og is an instance of OptimalGrowthModel.
    """

    β = og.β
    f, f_prime, u_prime = og.f, og.f_prime, og.u_prime
    grid, shocks = og.grid, og.shocks

    σ_new = np.empty_like(σ)
    for i, y in enumerate(grid):
        # Solve for optimal c at y
        c_star = brentq(euler_diff, 1e-10, y-1e-10, args=(σ, y, og))[0]
        σ_new[i] = c_star

    return σ_new
```

32.4.1 Testing

Let's generate an instance and plot some iterates of K , starting from $\sigma(y) = y$.

```
In [7]: og = OptimalGrowthModel()
grid = og.grid

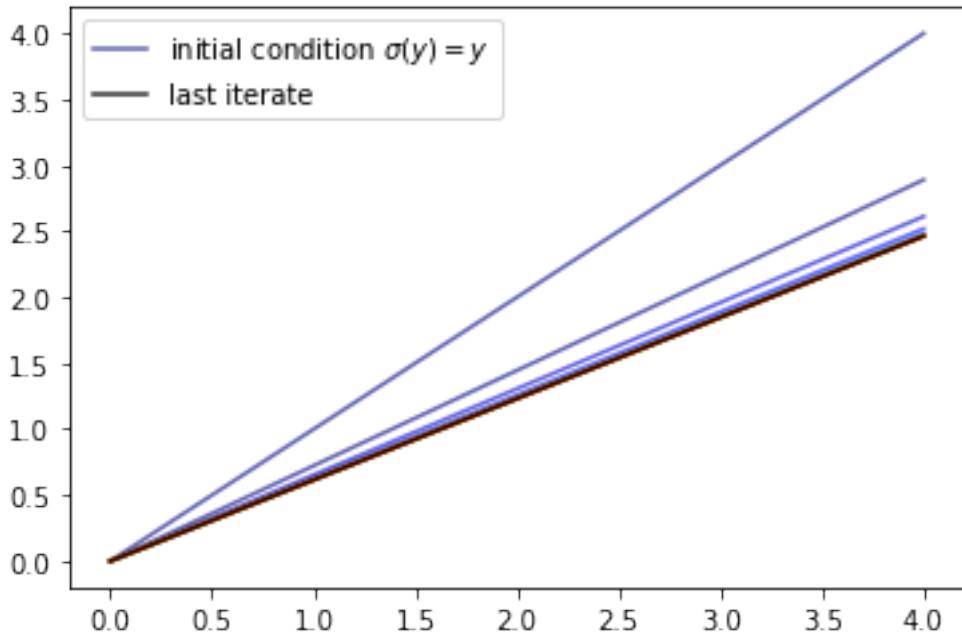
n = 15
σ = grid.copy() # Set initial condition

fig, ax = plt.subplots()
lb = 'initial condition $\sigma(y) = y$'
ax.plot(grid, σ, color=plt.cm.jet(0), alpha=0.6, label=lb)

for i in range(n):
    σ = K(σ, og)
    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, og)
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 [the previous lecture](#).

Here is a function called `solve_model_time_iter` that takes an instance of `OptimalGrowthModel` and returns an approximation to the optimal policy, using time iteration.

```
In [8]: def solve_model_time_iter(model,      # Class with model information
                               σ,          # Initial condition
                               tol=1e-4,
                               max_iter=1000,
                               verbose=True,
                               print_skip=25):

    # Set up loop
    i = 0
    error = tol + 1

    while i < max_iter and error > tol:
        σ_new = K(σ, model)
        error = np.max(np.abs(σ - σ_new))
        i += 1
        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")
        σ = σ_new

    if i == max_iter:
        print("Failed to converge!")

    if verbose and i < max_iter:
        print(f"\nConverged in {i} iterations.")

return σ_new
```

Let's call it:

```
In [9]: σ_init = np.copy(og.grid)
σ = solve_model_time_iter(og, σ_init)
```

Converged in 11 iterations.

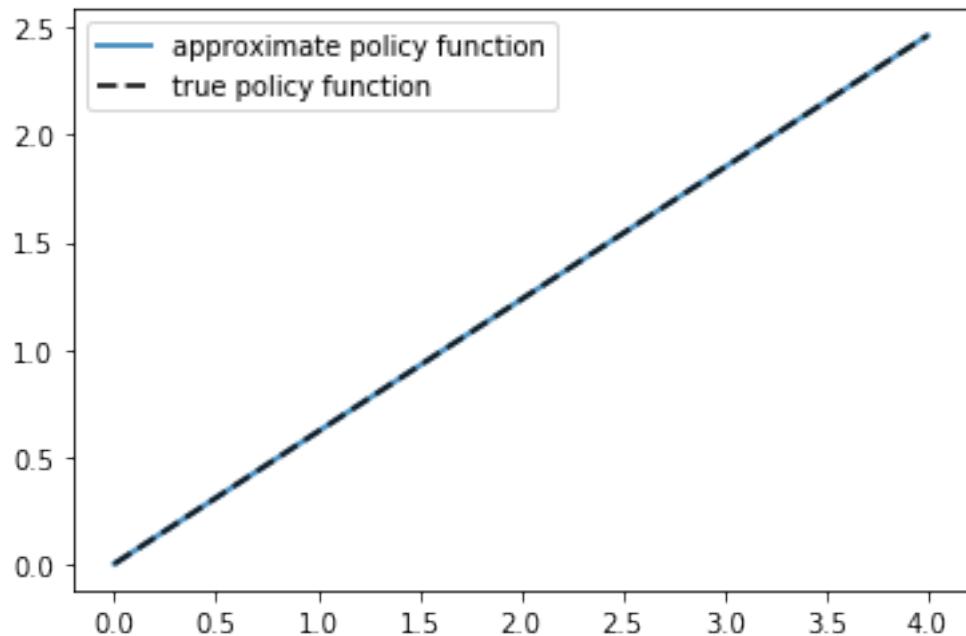
Here is a plot of the resulting policy, compared with the true policy:

```
In [10]: fig, ax = plt.subplots()

ax.plot(og.grid, σ, lw=2,
        alpha=0.8, label='approximate policy function')

ax.plot(og.grid, σ_star(og.grid, og.α, og.β), 'k--',
        lw=2, alpha=0.8, label='true policy function')

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



Again, the fit is excellent.

The maximal absolute deviation between the two policies is

```
In [11]: np.max(np.abs(σ - σ_star(og.grid, og.α, og.β)))
```

```
Out[11]: 2.5329106212446106e-05
```

How long does it take to converge?

```
In [12]: %%timeit -n 3 -r 1
σ = solve_model_time_iter(og, σ_init, verbose=False)
```

394 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 3 loops each)

Convergence is very fast, even compared to our JIT-compiled value function iteration.

Overall, we find that time iteration provides a very high degree of efficiency and accuracy, at least for this model.

32.5 Exercises

32.5.1 Exercise 1

Solve the model with CRRA utility

$$u(c) = \frac{c^{1-\gamma}}{1-\gamma}$$

Set $\gamma = 1.5$.

Compute and plot the optimal policy.

32.6 Solutions

32.6.1 Exercise 1

We use the class `OptimalGrowthModel_CRRA` from our [VFI](#) lecture.

```
In [13]: opt_growth_data = [
    ('α', float64),                      # Production parameter
    ('β', float64),                      # Discount factor
    ('μ', float64),                      # Shock location parameter
    ('γ', float64),                      # Preference parameter
    ('σ', float64),                      # Shock scale parameter
    ('grid', float64[:]),                # Grid (array)
    ('shocks', float64[:])               # Shock draws (array)
]
@jitclass(opt_growth_data)
class OptimalGrowthModel_CRRA:

    def __init__(self,
                 α=0.4,
                 β=0.96,
                 μ=0,
                 σ=0.1,
                 γ=1.5,
                 grid_max=4,
                 grid_size=120,
                 shock_size=250,
                 seed=1234):

        self.α, self.β, self.γ, self.μ, self.σ = α, β, γ, μ, σ
```

```

# Set up grid
self.grid = np.linspace(1e-5, grid_max, grid_size)

# Store shocks (with a seed, so results are reproducible)
np.random.seed(seed)
self.shocks = np.exp(mu + s * np.random.randn(shock_size))

def f(self, k):
    "The production function."
    return k**self.alpha

def u(self, c):
    "The utility function."
    return c**(1 - self.gamma) / (1 - self.gamma)

def f_prime(self, k):
    "Derivative of f."
    return self.alpha * (k**(self.alpha - 1))

def u_prime(self, c):
    "Derivative of u."
    return c**(-self.gamma)

def u_prime_inv(c):
    return c**(-1 / self.gamma)

```

Let's create an instance:

```
In [14]: og_crra = OptimalGrowthModel_CRRRA()
```

Now we solve and plot the policy:

```

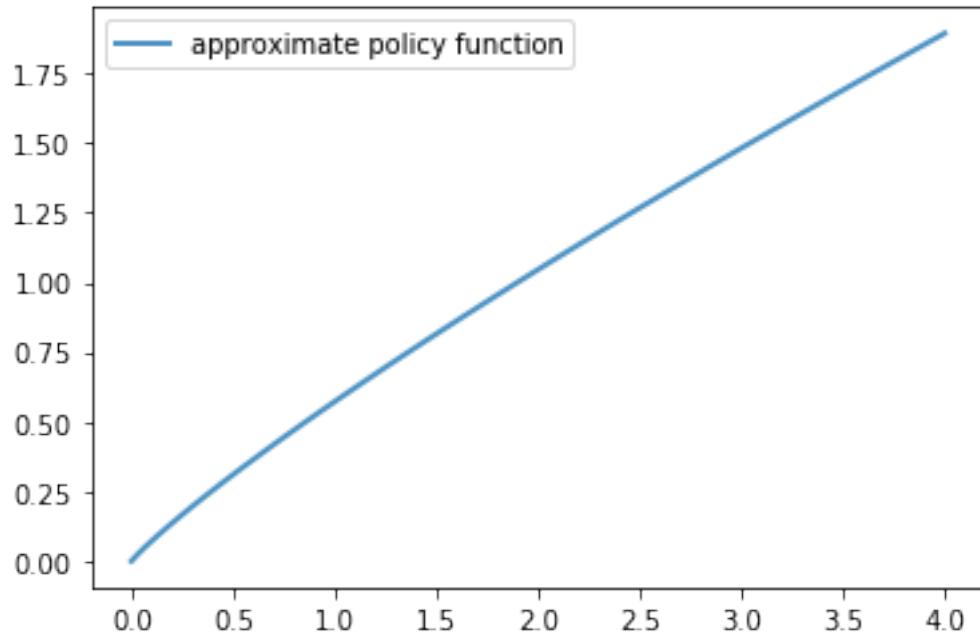
In [15]: %%time
sigma = solve_model_time_iter(og_crra, sigma_init)

fig, ax = plt.subplots()
ax.plot(og.grid, sigma, lw=2,
        alpha=0.8, label='approximate policy function')

ax.legend()
plt.show()

```

Converged in 13 iterations.



```
CPU times: user 3.81 s, sys: 15.9 ms, total: 3.82 s
Wall time: 3.89 s
```

Chapter 33

Optimal Growth IV: The Endogenous Grid Method

33.1 Contents

- Overview 33.2
- Key Idea 33.3
- Implementation 33.4

In addition to what's in Anaconda, this lecture will need the following libraries:

```
In [1]: !pip install quantecon  
!pip install interpolation
```

33.2 Overview

Previously, we solved the stochastic optimal growth model 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 [\[21\]](#).

Let's start with some standard imports:

```
In [2]: import numpy as np  
import quantecon as qe  
from interpolation import interp  
from numba import njit, float64  
from numba.experimental import jitclass
```

```

from quantecon.optimize import brentq
import matplotlib.pyplot as plt
%matplotlib inline

/home/ubuntu/anaconda3/lib/python3.7/site-packages/numba/np/ufunc/parallel.py:
 355:
NumbaWarning: The TBB threading layer requires TBB version 2019.5 or later i.e.,
TBB_INTERFACE_VERSION >= 11005. Found TBB_INTERFACE_VERSION = 11004. The TBB
threading
layer is disabled.
warnings.warn(problem)

```

33.3 Key Idea

Let's start by reminding ourselves of the theory and then see how the numerics fit in.

33.3.1 Theory

Take the model set out in [the time iteration lecture](#), following the same terminology and notation.

The Euler equation is

$$(u' \circ \sigma^*)(y) = \beta \int (u' \circ \sigma^*)(f(y - \sigma^*(y))z) f'(y - \sigma^*(y)) z \phi(dz) \quad (1)$$

As we saw, the Coleman-Reffett operator is a nonlinear operator K engineered so that 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 $(K\sigma)(y)$ is the $c \in (0, \infty)$ that solves

$$u'(c) = \beta \int (u' \circ \sigma)(f(y - c)z) f'(y - c) z \phi(dz) \quad (2)$$

33.3.2 Exogenous Grid

As discussed in [the lecture on time iteration](#), to implement the method on a computer, we need a numerical approximation.

In particular, 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.

[Previously](#), to obtain a finite representation of an updated consumption policy, we

- fixed a grid of income points $\{y_i\}$
- calculated the consumption value c_i corresponding to each y_i using (2) and a root-finding routine

Each c_i is then interpreted as the value of the function $K\sigma$ at y_i .

Thus, with the points $\{y_i, c_i\}$ in hand, we can reconstruct $K\sigma$ via approximation.

Iteration then continues...

33.3.3 Endogenous Grid

The method discussed above requires a root-finding routine to find the c_i corresponding to a given income value y_i .

Root-finding is costly because it typically involves a significant number of function evaluations.

As pointed out by Carroll [21], we can avoid this if y_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 $\{k_i\}$ for capital ($k = y - c$).
- Then we obtain c_i via

$$c_i = (u')^{-1} \left\{ \beta \int (u' \circ \sigma)(f(k_i)z) f'(k_i) z \phi(dz) \right\} \quad (3)$$

- Finally, for each c_i we set $y_i = c_i + k_i$.

It is clear that each (y_i, c_i) pair constructed in this manner satisfies (2).

With the points $\{y_i, c_i\}$ in hand, we can reconstruct $K\sigma$ via approximation as before.

The name EGM comes from the fact that the grid $\{y_i\}$ is determined **endogenously**.

33.4 Implementation

As before, we will start with a simple setting where

- $u(c) = \ln c$,
- production is Cobb-Douglas, and
- the shocks are lognormal.

This will allow us to make comparisons with the analytical solutions

```
In [3]: def v_star(y, alpha, beta, mu):
    """
    True value function
    """
    c1 = np.log(1 - alpha * beta) / (1 - beta)
    c2 = (mu + alpha * np.log(alpha * beta)) / (1 - alpha)
    c3 = 1 / (1 - beta)
    c4 = 1 / (1 - alpha * beta)
    return c1 + c2 * (c3 - c4) + c4 * np.log(y)

def sigma_star(y, alpha, beta):
    """
```

True optimal policy

"""

```
return (1 - α * β) * y
```

We reuse the `OptimalGrowthModel` class

```
In [4]: opt_growth_data = [
    ('α', float64),           # Production parameter
    ('β', float64),           # Discount factor
    ('μ', float64),           # Shock location parameter
    ('s', float64),           # Shock scale parameter
    ('grid', float64[:]),     # Grid (array)
    ('shocks', float64[:])    # Shock draws (array)
]

@jitclass(opt_growth_data)
class OptimalGrowthModel:

    def __init__(self,
                 α=0.4,
                 β=0.96,
                 μ=0,
                 s=0.1,
                 grid_max=4,
                 grid_size=120,
                 shock_size=250,
                 seed=1234):

        self.α, self.β, self.μ, self.s = α, β, μ, s

        # Set up grid
        self.grid = np.linspace(1e-5, grid_max, grid_size)

        # Store shocks (with a seed, so results are reproducible)
        np.random.seed(seed)
        self.shocks = np.exp(μ + s * np.random.randn(shock_size))

    def f(self, k):
        "The production function"
        return k**self.α

    def u(self, c):
        "The utility function"
        return np.log(c)

    def f_prime(self, k):
        "Derivative of f"
        return self.α * (k**(self.α - 1))

    def u_prime(self, c):
        "Derivative of u"
        return 1/c

    def u_prime_inv(self, c):
```

```
"Inverse of u"
return 1/c
```

33.4.1 The Operator

Here's an implementation of K using EGM as described above.

```
In [5]: @njit
def K(σ_array, og):
    """
    The Coleman-Reffett operator using EGM

    """

    # Simplify names
    f, β = og.f, og.β
    f_prime, u_prime = og.f_prime, og.u_prime
    u_prime_inv = og.u_prime_inv
    grid, shocks = og.grid, og.shocks

    # Determine endogenous grid
    y = grid + σ_array #  $y_i = k_i + c_i$ 

    # Linear interpolation of policy using endogenous grid
    σ = lambda x: interp(y, σ_array, x)

    # Allocate memory for new consumption array
    c = np.empty_like(grid)

    # Solve for updated consumption value
    for i, k in enumerate(grid):
        vals = u_prime(σ(f(k) * shocks)) * f_prime(k) * shocks
        c[i] = u_prime_inv(β * np.mean(vals))

    return c
```

Note the lack of any root-finding algorithm.

33.4.2 Testing

First we create an instance.

```
In [6]: og = OptimalGrowthModel()
grid = og.grid
```

Here's our solver routine:

```
In [7]: def solve_model_time_iter(model,      # Class with model information
                               σ,          # Initial condition
                               tol=1e-4,
                               max_iter=1000,
                               verbose=True,
                               print_skip=25):
```

```

# Set up loop
i = 0
error = tol + 1

while i < max_iter and error > tol:
    sigma_new = K(sigma, model)
    error = np.max(np.abs(sigma - sigma_new))
    i += 1
    if verbose and i % print_skip == 0:
        print(f"Error at iteration {i} is {error}.")
    sigma = sigma_new

if i == max_iter:
    print("Failed to converge!")

if verbose and i < max_iter:
    print(f"\nConverged in {i} iterations.")

return sigma_new

```

Let's call it:

```
In [8]: sigma_init = np.copy(grid)
sigma = solve_model_time_iter(og, sigma_init)
```

Converged in 12 iterations.

Here is a plot of the resulting policy, compared with the true policy:

```

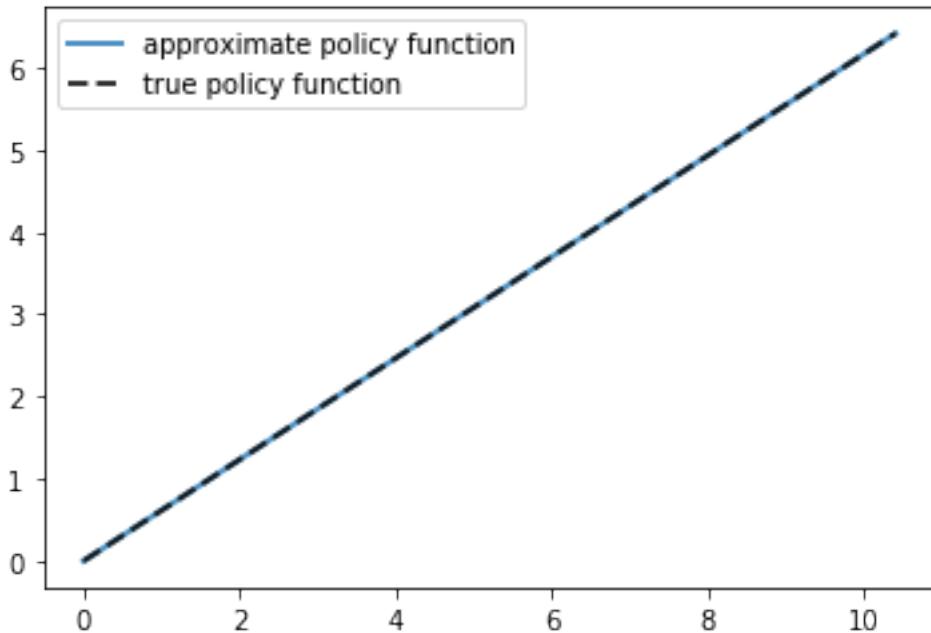
In [9]: y = grid + sigma # y_i = k_i + c_i

fig, ax = plt.subplots()

ax.plot(y, sigma, lw=2,
         alpha=0.8, label='approximate policy function')

ax.plot(y, sigma_star(y, og.alpha, og.beta), 'k--',
         lw=2, alpha=0.8, label='true policy function')

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



The maximal absolute deviation between the two policies is

```
In [10]: np.max(np.abs(σ - σ_star(y, og.α, og.β)))
```

```
Out[10]: 1.530274914252061e-05
```

How long does it take to converge?

```
In [11]: %timeit -n 3 -r 1
σ = solve_model_time_iter(og, σ_init, verbose=False)
```

```
54.5 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 3 loops each)
```

Relative to time iteration, which as already found to be highly efficient, EGM has managed to shave off still more run time without compromising accuracy.

This is due to the lack of a numerical root-finding step.

We can now solve the optimal growth model at given parameters extremely fast.

Chapter 34

The Income Fluctuation Problem I: Basic Model

34.1 Contents

- Overview 34.2
- The Optimal Savings Problem 34.3
- Computation 34.4
- Implementation 34.5
- Exercises 34.6
- Solutions 34.7

In addition to what's in Anaconda, this lecture will need the following libraries:

```
In [1]: !pip install quantecon  
!pip install interpolation
```

34.2 Overview

In this lecture, we study an optimal savings problem for an infinitely lived consumer—the “common ancestor” described in [72], section 1.3.

This is an essential sub-problem for many representative macroeconomic models

- [4]
- [59]
- etc.

It is related to the decision problem in the [stochastic optimal growth model](#) and yet differs in important ways.

For example, the choice problem for the agent includes an additive income term that leads to an occasionally binding constraint.

Moreover, in this and the following lectures, we will inject more realistic features such as correlated shocks.

To solve the model we will use Euler equation based time iteration, which proved to be [fast](#) and [accurate](#) in our investigation of the [stochastic optimal growth model](#).

Time iteration is globally convergent under mild assumptions, even when utility is unbounded (both above and below).

We'll need the following imports:

```
In [2]: import numpy as np
from quantecon.optimize import brent_max, brentq
from interpolation import interp
from numba import njit, float64
from numba.experimental import jitclass
import matplotlib.pyplot as plt
%matplotlib inline
from quantecon import MarkovChain

/home/ubuntu/anaconda3/lib/python3.7/site-packages/numba/np/ufunc/parallel.py:
 355:
NumbaWarning: The TBB threading layer requires TBB version 2019.5 or later i.e.,
TBB_INTERFACE_VERSION >= 11005. Found TBB_INTERFACE_VERSION = 11004. The TBB
  ↪threading
layer is disabled.
warnings.warn(problem)
```

34.2.1 References

Our presentation is a simplified version of [75].

Other references include [26], [28], [68], [87], [90] and [96].

34.3 The Optimal Savings Problem

Let's write down the model and then discuss how to solve it.

34.3.1 Set-Up

Consider 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

$$a_{t+1} \leq R(a_t - c_t) + Y_{t+1}, \quad c_t \geq 0, \quad a_t \geq 0 \quad t = 0, 1, \dots \quad (1)$$

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 chooses consumption c_t .
2. Labor is supplied by the household throughout the period and labor income Y_{t+1} is received at the end of period t .
3. Financial income $R(a_t - c_t)$ is received at the end of period t .
4. Time shifts to $t + 1$ and the process repeats.

Non-capital income Y_t is given by $Y_t = y(Z_t)$, where $\{Z_t\}$ is an exogenous state process.

As is common in the literature, we take $\{Z_t\}$ to be a finite state Markov chain taking values in Z with Markov matrix P .

We further assume that

1. $\beta R < 1$
2. u is smooth, strictly increasing and strictly concave with $\lim_{c \rightarrow 0} u'(c) = \infty$ and $\lim_{c \rightarrow \infty} u'(c) = 0$

The asset space is \mathbb{R}_+ and the state is the pair $(a, z) \in S := \mathbb{R}_+ \times Z$.

A *feasible consumption path* from $(a, z) \in S$ 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 (1), and
3. measurability, 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 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.

34.3.2 Value Function and Euler Equation

The *value function* $V: S \rightarrow \mathbb{R}$ is defined by

$$V(a, z) := \max \mathbb{E} \left\{ \sum_{t=0}^{\infty} \beta^t u(c_t) \right\} \quad (2)$$

where the maximization is over all feasible consumption paths from (a, z) .

An *optimal consumption path* from (a, z) is a feasible consumption path from (a, z) that attains the supremum in (2).

To pin down such paths we can use a version of the Euler equation, which in the present setting is

$$u'(c_t) \geq \beta R \mathbb{E}_t u'(c_{t+1}) \quad (3)$$

and

$$c_t < a_t \implies u'(c_t) = \beta R \mathbb{E}_t u'(c_{t+1}) \quad (4)$$

When $c_t = a_t$ we obviously have $u'(c_t) = u'(a_t)$,

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 lower boundary case $c_t = 0$ never arises at the optimum because $u'(0) = \infty$.)

With some thought, one can show that (3) and (4) are equivalent to

$$u'(c_t) = \max \{\beta R \mathbb{E}_t u'(c_{t+1}), u'(a_t)\} \quad (5)$$

34.3.3 Optimality Results

As shown in [75],

1. For each $(a, z) \in S$, a unique optimal consumption path from (a, z) exists
2. This path is the unique feasible path from (a, z) satisfying the Euler equality (5) and the transversality condition

$$\lim_{t \rightarrow \infty} \beta^t \mathbb{E}[u'(c_t)a_{t+1}] = 0 \quad (6)$$

Moreover, there exists an *optimal consumption function* $\sigma^*: S \rightarrow \mathbb{R}_+$ 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 (5) and (6), and hence is the unique optimal path from (a, z) .

Thus, to solve the optimization problem, we need to compute the policy σ^* .

34.4 Computation

There are two standard ways to solve for σ^*

1. time iteration using the Euler equality and

2. value function iteration.

Our investigation of the cake eating problem and stochastic optimal growth model suggests that time iteration will be faster and more accurate.

This is the approach that we apply below.

34.4.1 Time Iteration

We can rewrite (5) to make it a statement about functions rather than random variables.

In particular, consider the functional equation

$$(u' \circ \sigma)(a, z) = \max \left\{ \beta R \mathbb{E}_z(u' \circ \sigma)[R(a - \sigma(a, z)) + \hat{Y}, \hat{Z}], u'(a) \right\} \quad (7)$$

where

- $(u' \circ \sigma)(s) := u'(\sigma(s))$.
- \mathbb{E}_z conditions on current state z and \hat{X} indicates next period value of random variable X and
- σ is the unknown function.

We need a suitable class of candidate solutions for the optimal consumption policy.

The right way to pick such a class is to consider what properties the solution is likely to have, in order to restrict the search space and ensure that iteration is well behaved.

To this end, let \mathcal{C} be the space of continuous functions $\sigma: \mathbf{S} \rightarrow \mathbb{R}$ such that σ is increasing in the first argument, $0 < \sigma(a, z) \leq a$ for all $(a, z) \in \mathbf{S}$, and

$$\sup_{(a,z) \in \mathbf{S}} |(u' \circ \sigma)(a, z) - u'(a)| < \infty \quad (8)$$

This will be our candidate class.

In addition, let $K: \mathcal{C} \rightarrow \mathcal{C}$ be defined as follows.

For given $\sigma \in \mathcal{C}$, the value $K\sigma(a, z)$ is the unique $c \in [0, a]$ that solves

$$u'(c) = \max \left\{ \beta R \mathbb{E}_z(u' \circ \sigma)[R(a - c) + \hat{Y}, \hat{Z}], u'(a) \right\} \quad (9)$$

We refer to K as the Coleman–Reffett operator.

The operator K is constructed so that fixed points of K coincide with solutions to the functional equation (7).

It is shown in [75] that the unique optimal policy can be computed by picking any $\sigma \in \mathcal{C}$ and iterating with the operator K defined in (9).

34.4.2 Some Technical Details

The proof of the last statement is somewhat technical but here is a quick summary:

It is shown in [75] that K is a contraction mapping on \mathcal{C} under the metric

$$\rho(c, d) := \| u' \circ \sigma_1 - u' \circ \sigma_2 \| := \sup_{s \in S} | u'(\sigma_1(s)) - u'(\sigma_2(s)) | \quad (\sigma_1, \sigma_2 \in \mathcal{C})$$

which evaluates the maximal difference in terms of marginal utility.

(The benefit of this measure of distance is that, while elements of \mathcal{C} are not generally bounded, ρ is always finite under our assumptions.)

It is also shown that the metric ρ is complete on \mathcal{C} .

In consequence, K has a unique fixed point $\sigma^* \in \mathcal{C}$ and $K^n c \rightarrow \sigma^*$ as $n \rightarrow \infty$ for any $\sigma \in \mathcal{C}$.

By the definition of K , the fixed points of K in \mathcal{C} coincide with the solutions to (7) in \mathcal{C} .

As a consequence, the path $\{c_t\}$ generated from $(a_0, z_0) \in S$ using policy function σ^* is the unique optimal path from $(a_0, z_0) \in S$.

34.5 Implementation

We use the CRRA utility specification

$$u(c) = \frac{c^{1-\gamma}}{1-\gamma}$$

The exogenous state process $\{Z_t\}$ defaults to a two-state Markov chain with state space $\{0, 1\}$ and transition matrix P .

Here we build a class called `IFP` that stores the model primitives.

```
In [3]: ifp_data = [
    ('R', float64),                      # Interest rate 1 + r
    ('beta', float64),                    # Discount factor
    ('gamma', float64),                  # Preference parameter
    ('P', float64[:, :]),                # Markov matrix for binary Z_t
    ('y', float64[:]),                  # Income is Y_t = y[Z_t]
    ('asset_grid', float64[:])           # Grid (array)
]

@jitclass(ifp_data)
class IFP:

    def __init__(self,
                 r=0.01,
                 beta=0.96,
                 gamma=1.5,
                 P=((0.6, 0.4),
                     (0.05, 0.95)),
                 y=(0.0, 2.0),
                 grid_max=16,
                 grid_size=50):

        self.R = 1 + r
        self.beta, self.gamma = beta, gamma
        self.P, self.y = np.array(P), np.array(y)
        self.asset_grid = np.linspace(0, grid_max, grid_size)
```

```

# Recall that we need R β < 1 for convergence.
assert self.R * self.β < 1, "Stability condition violated."

def u_prime(self, c):
    return c**(-self.γ)

```

Next we provide a function to compute the difference

$$u'(c) - \max \left\{ \beta R \mathbb{E}_z(u' \circ \sigma) [R(a - c) + \hat{Y}, \hat{Z}], u'(a) \right\} \quad (10)$$

```

In [4]: @njit
def euler_diff(c, a, z, σ_vals, ifp):
    """
    The difference between the left- and right-hand side
    of the Euler Equation, given current policy σ.

    * c is the consumption choice
    * (a, z) is the state, with z in {0, 1}
    * σ_vals is a policy represented as a matrix.
    * ifp is an instance of IFP
    """

    # Simplify names
    R, P, y, β, γ = ifp.R, ifp.P, ifp.y, ifp.β, ifp.γ
    asset_grid, u_prime = ifp.asset_grid, ifp.u_prime
    n = len(P)

    # Convert policy into a function by linear interpolation
    def σ(a, z):
        return interp(asset_grid, σ_vals[:, z], a)

    # Calculate the expectation conditional on current z
    expect = 0.0
    for z_hat in range(n):
        expect += u_prime(σ(R * (a - c) + y[z_hat], z_hat)) * P[z, z_hat]

    return u_prime(c) - max(β * R * expect, u_prime(a))

```

Note that we use linear interpolation along the asset grid to approximate the policy function.

The next step is to obtain the root of the Euler difference.

```

In [5]: @njit
def K(σ, ifp):
    """
    The operator K.

    """

    σ_new = np.empty_like(σ)
    for i, a in enumerate(ifp.asset_grid):
        for z in (0, 1):
            result = brentq(euler_diff, 1e-8, a, args=(a, z, σ, ifp))
            σ_new[i, z] = result.root

    return σ_new

```

With the operator K in hand, we can choose an initial condition and start to iterate.

The following function iterates to convergence and returns the approximate optimal policy.

```
In [6]: def solve_model_time_iter(model,      # Class with model information
                                σ,          # Initial condition
                                tol=1e-4,
                                max_iter=1000,
                                verbose=True,
                                print_skip=25):

    # Set up loop
    i = 0
    error = tol + 1

    while i < max_iter and error > tol:
        σ_new = K(σ, model)
        error = np.max(np.abs(σ - σ_new))
        i += 1
        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")
        σ = σ_new

    if i == max_iter:
        print("Failed to converge!")

    if verbose and i < max_iter:
        print(f"\nConverged in {i} iterations.")

return σ_new
```

Let's carry this out using the default parameters of the `IFP` class:

```
In [7]: ifp = IFP()

# Set up initial consumption policy of consuming all assets at all z
z_size = len(ifp.P)
a_grid = ifp.asset_grid
a_size = len(a_grid)
σ_init = np.repeat(a_grid.reshape(a_size, 1), z_size, axis=1)

σ_star = solve_model_time_iter(ifp, σ_init)

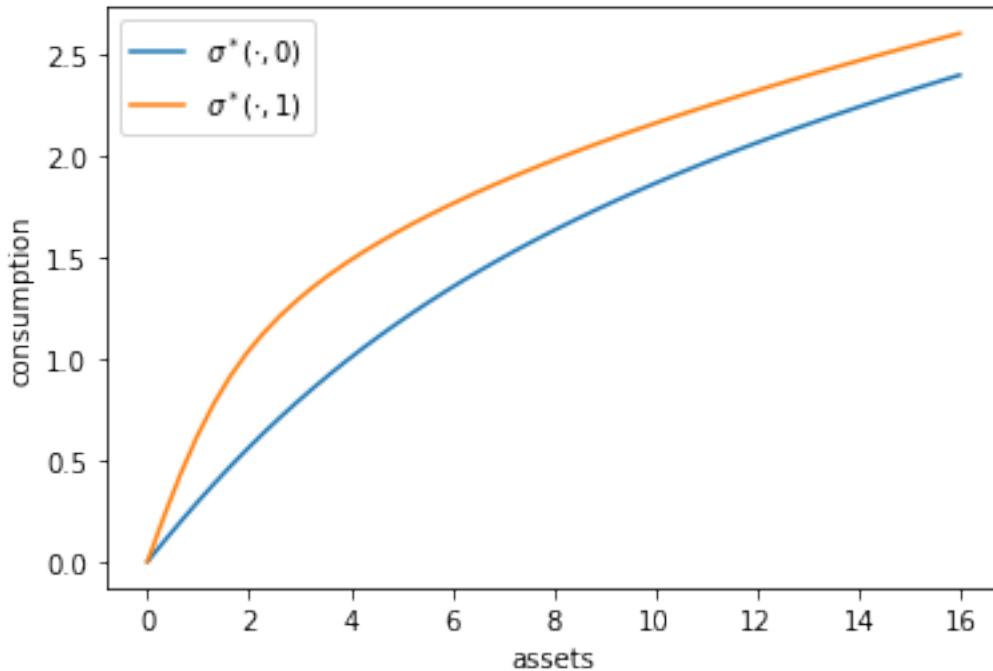
Error at iteration 25 is 0.011629589188246303.
Error at iteration 50 is 0.0003857183099467143.

Converged in 60 iterations.
```

Here's a plot of the resulting policy for each exogenous state z .

```
In [8]: fig, ax = plt.subplots()
for z in range(z_size):
    label = rf'$\sigma^*(\cdot, \{z\})$'
    ax.plot(a_grid, σ_star[:, z], label=label)
ax.set(xlabel='assets', ylabel='consumption')
```

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



The following exercises walk you through several applications where policy functions are computed.

34.5.1 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 cake eating problem.

We know that, in this case, the value function and optimal consumption policy are given by

```
In [9]: 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:

```
In [10]: ifp_cake_eating = IFP(r=0.0, y=(0.0, 0.0))
σ_star = solve_model_time_iter(ifp_cake_eating, σ_init)
```

```

fig, ax = plt.subplots()
ax.plot(a_grid, o_star[:, 0], label='numerical')
ax.plot(a_grid, c_star(a_grid, ifp.β, ifp.γ), '--', label='analytical')

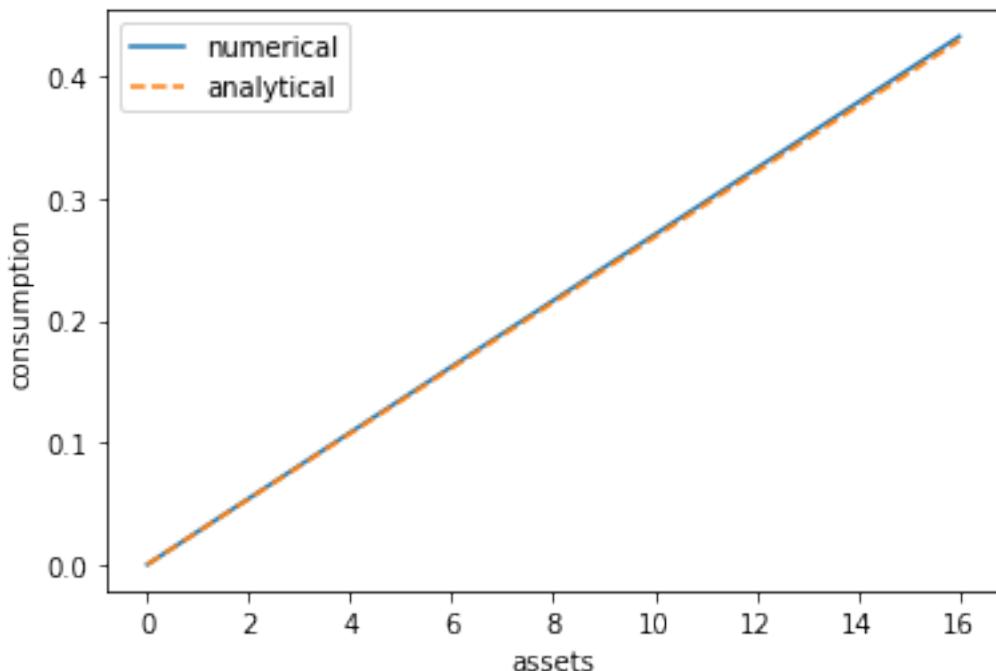
ax.set(xlabel='assets', ylabel='consumption')
ax.legend()

plt.show()

Error at iteration 25 is 0.023332272630545492.
Error at iteration 50 is 0.005301238424249566.
Error at iteration 75 is 0.0019706324625650695.
Error at iteration 100 is 0.0008675521337956349.
Error at iteration 125 is 0.00041073542212255454.
Error at iteration 150 is 0.00020120334010526042.
Error at iteration 175 is 0.00010021430795065234.

Converged in 176 iterations.

```



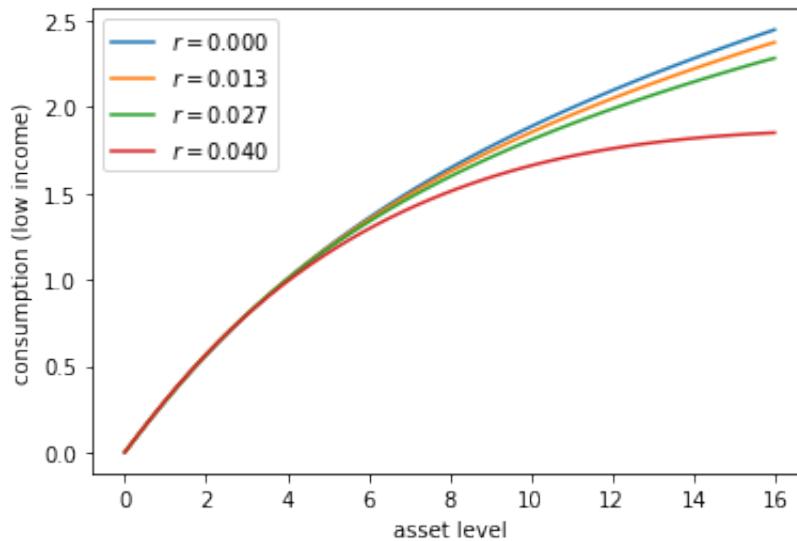
Success!

34.6 Exercises

34.6.1 Exercise 1

Let's consider how the interest rate affects consumption.

Reproduce the following figure, which shows (approximately) optimal consumption policies for different interest rates



- Other than r , all parameters are at their default values.
- r steps through `np.linspace(0, 0.04, 4)`.
- Consumption is plotted against assets for income shock fixed at the smallest value.

The figure shows that higher interest rates boost savings and hence suppress consumption.

34.6.2 Exercise 2

Now let's consider the long run asset levels held by households under the default parameters.

The following figure is a 45 degree diagram showing the law of motion for assets when consumption is optimal

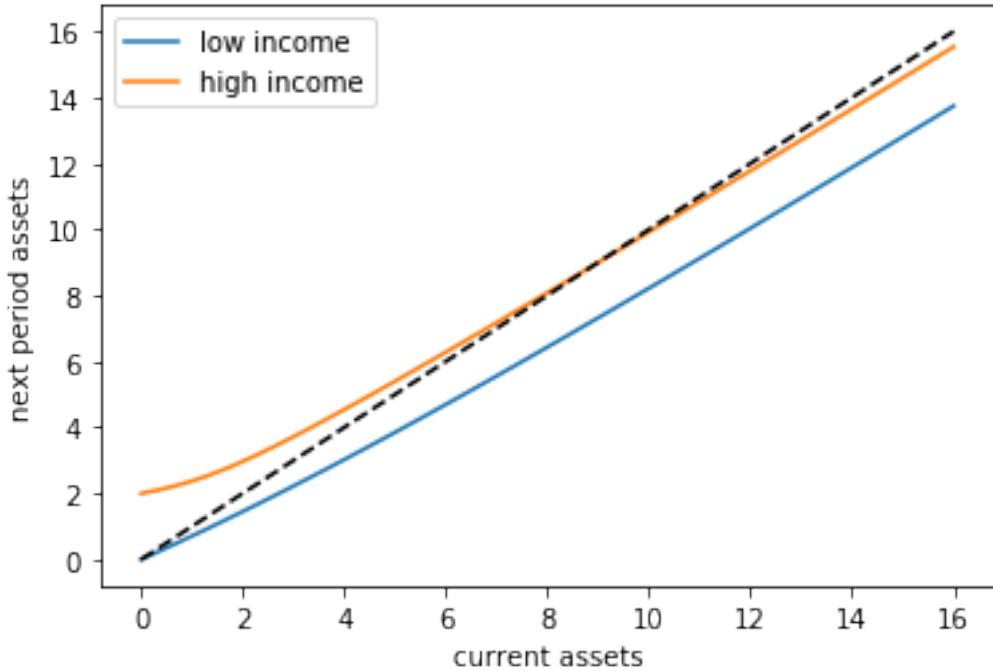
```
In [11]: ifp = IFP()

σ_star = solve_model_time_iter(ifp, σ_init, verbose=False)
a = ifp.asset_grid
R, y = ifp.R, ifp.y

fig, ax = plt.subplots()
for z, lb in zip((0, 1), ('low income', 'high income')):
    ax.plot(a, R * (a - σ_star[:, z]) + y[z], label=lb)

ax.plot(a, a, '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

$$a \mapsto R(a - \sigma^*(a, z)) + y(z)$$

The dashed line is the 45 degree line.

We can see from the figure that the dynamics will be stable — assets do not diverge even in the highest state.

In fact there is a unique stationary distribution of assets that we can calculate by simulation

- Can be proved via theorem 2 of [58].
- It represents the long run dispersion of assets across households when households have idiosyncratic shocks.

Ergodicity is valid here, so stationary probabilities can be calculated by averaging over a single long time series.

Hence to approximate the stationary distribution we can simulate a long time series for assets and histogram it.

Your task is to generate such a histogram.

- Use a single time series $\{a_t\}$ of length 500,000.
- Given the length of this time series, the initial condition (a_0, z_0) will not matter.
- You might find it helpful to use the `MarkovChain` class from `quantecon`.

34.6.3 Exercise 3

Following on from exercises 1 and 2, let's look at how savings and aggregate asset holdings vary with the interest rate

- Note: [72] 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.

Following tradition, put the price (i.e., interest rate) on the vertical axis.

On the horizontal axis put aggregate capital, computed as the mean of the stationary distribution given the interest rate.

34.7 Solutions

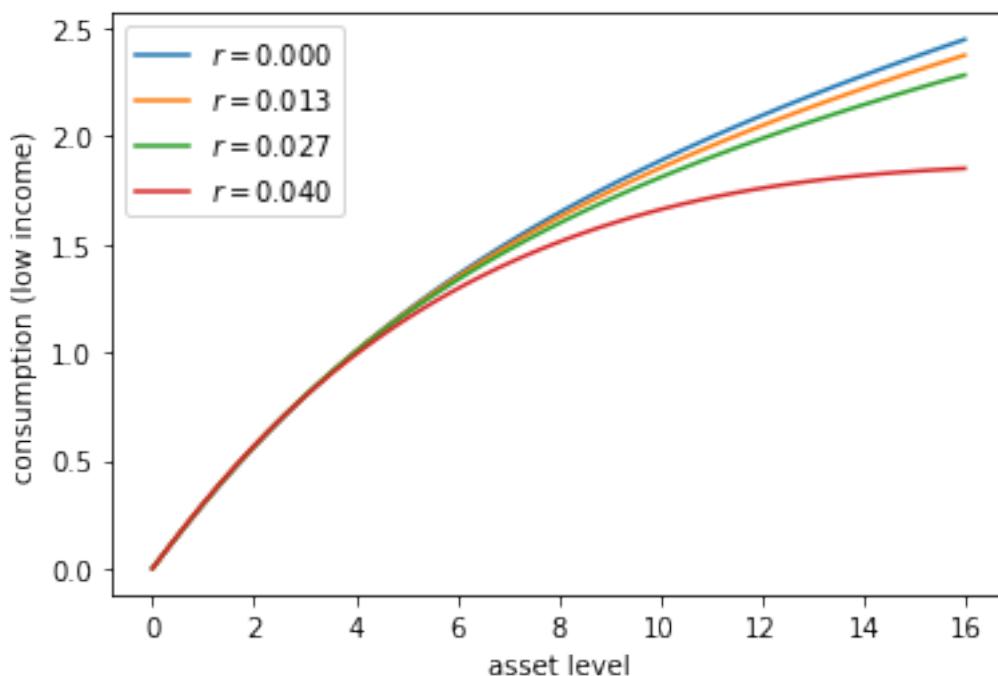
34.7.1 Exercise 1

Here's one solution:

```
In [12]: r_vals = np.linspace(0, 0.04, 4)

fig, ax = plt.subplots()
for r_val in r_vals:
    ifp = IFP(r=r_val)
    sigma_star = solve_model_time_iter(ifp, sigma_init, verbose=False)
    ax.plot(ifp.asset_grid, sigma_star[:, 0], label=f'r = {r_val:.3f}')

ax.set(xlabel='asset level', ylabel='consumption (low income)')
ax.legend()
plt.show()
```



34.7.2 Exercise 2

First we write a function to compute a long asset series.

```
In [13]: def compute_asset_series(ifp, T=500_000, seed=1234):
    """
    Simulates a time series of length T for assets, given optimal
    savings behavior.

    ifp is an instance of IFP
    """
    P, y, R = ifp.P, ifp.y, ifp.R # Simplify names

    # Solve for the optimal policy
    sigma_star = solve_model_time_iter(ifp, sigma_init, verbose=False)
    sigma = lambda a, z: interp(ifp.asset_grid, sigma_star[:, z], a)

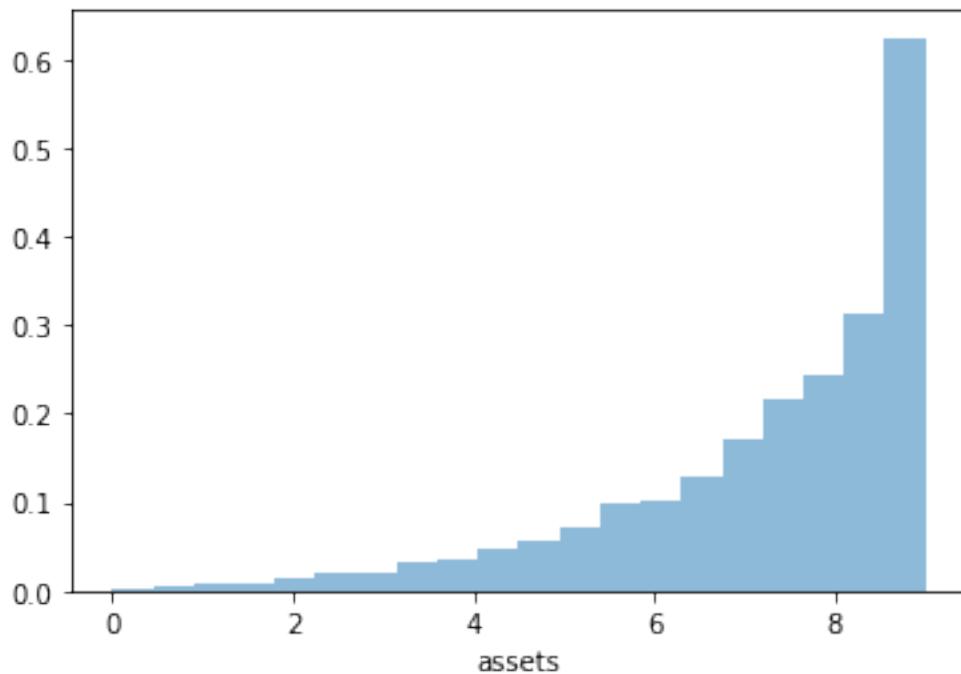
    # Simulate the exogenous state process
    mc = MarkovChain(P)
    z_seq = mc.simulate(T, random_state=seed)

    # Simulate the asset path
    a = np.zeros(T+1)
    for t in range(T):
        z = z_seq[t]
        a[t+1] = R * (a[t] - sigma(a[t], z)) + y[z]
    return a
```

Now we call the function, generate the series and then histogram it:

```
In [14]: ifp = IFP()
a = compute_asset_series(ifp)

fig, ax = plt.subplots()
ax.hist(a, bins=20, alpha=0.5, density=True)
ax.set(xlabel='assets')
plt.show()
```



The shape of the asset distribution is unrealistic.

Here it is left skewed when in reality it has a long right tail.

In a [subsequent lecture](#) we will rectify this by adding more realistic features to the model.

34.7.3 Exercise 3

Here's one solution

```
In [15]: M = 25
r_vals = np.linspace(0, 0.02, M)
fig, ax = plt.subplots()

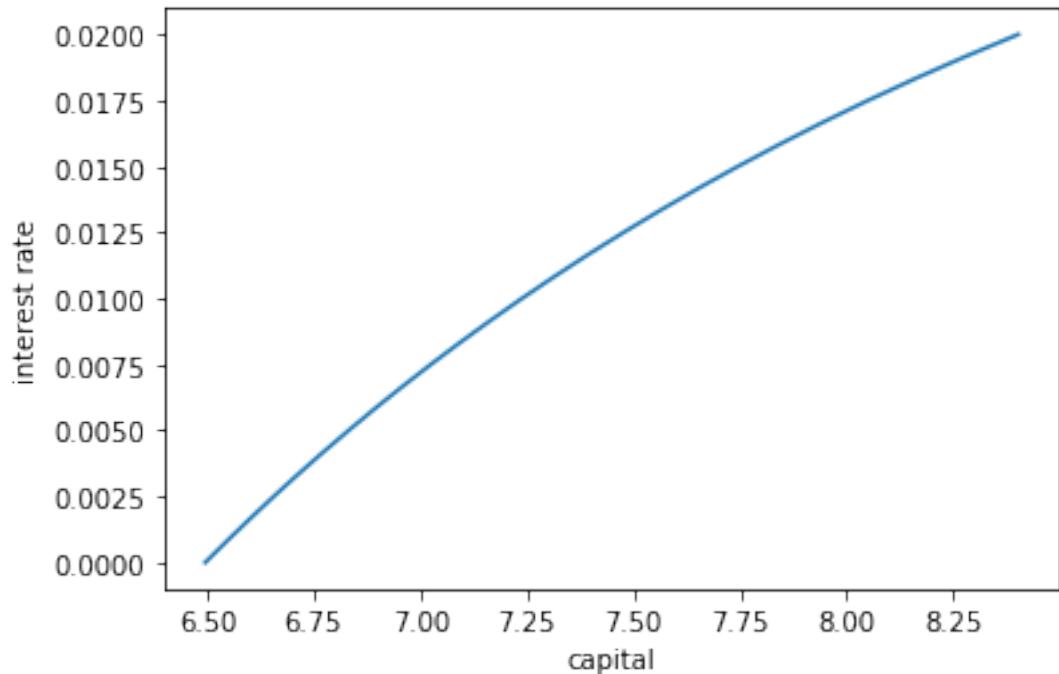
asset_mean = []
for r in r_vals:
    print(f'Solving model at r = {r}')
    ifp = IFP(r=r)
    mean = np.mean(compute_asset_series(ifp, T=250_000))
    asset_mean.append(mean)
ax.plot(asset_mean, r_vals)

ax.set(xlabel='capital', ylabel='interest rate')

plt.show()

Solving model at r = 0.0
Solving model at r = 0.000833333333333334
Solving model at r = 0.0016666666666666668
Solving model at r = 0.0025
Solving model at r = 0.003333333333333335
Solving model at r = 0.004166666666666667
```

```
Solving model at r = 0.005
Solving model at r = 0.005833333333333334
Solving model at r = 0.006666666666666667
Solving model at r = 0.007500000000000001
Solving model at r = 0.008333333333333333
Solving model at r = 0.009166666666666667
Solving model at r = 0.01
Solving model at r = 0.01083333333333334
Solving model at r = 0.011666666666666667
Solving model at r = 0.0125
Solving model at r = 0.01333333333333334
Solving model at r = 0.01416666666666668
Solving model at r = 0.015000000000000001
Solving model at r = 0.01583333333333335
Solving model at r = 0.01666666666666666
Solving model at r = 0.0175
Solving model at r = 0.01833333333333333
Solving model at r = 0.01916666666666667
Solving model at r = 0.02
```



As expected, aggregate savings increases with the interest rate.

Chapter 35

The Income Fluctuation Problem II: Stochastic Returns on Assets

35.1 Contents

- Overview 35.2
- The Savings Problem 35.3
- Solution Algorithm 35.4
- Implementation 35.5
- Exercises 35.6
- Solutions 35.7

In addition to what's in Anaconda, this lecture will need the following libraries:

```
In [1]: !pip install quantecon  
!pip install interpolation
```

35.2 Overview

In this lecture, we continue our study of the [income fluctuation problem](#).

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., [\[10\]](#) or [\[101\]](#)).

Theoretical properties of the household savings model presented here are analyzed in detail in [\[75\]](#).

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:

```
In [2]: import numpy as np  
from quantecon.optimize import brent_max, brentq
```

```

from interpolation import interp
from numba import njit, float64
from numba.experimental import jitclass
import matplotlib.pyplot as plt
%matplotlib inline
from quantecon import MarkovChain

/home/ubuntu/anaconda3/lib/python3.7/site-packages/numba/np/ufunc/parallel.py:
  ↵355:
NumbaWarning: The TBB threading layer requires TBB version 2019.5 or later i.e.,
TBB_INTERFACE_VERSION >= 11005. Found TBB_INTERFACE_VERSION = 11004. The TBB
  ↵threading
layer is disabled.
warnings.warn(problem)

```

35.3 The Savings Problem

In this section we review the household problem and optimality results.

35.3.1 Set Up

A household chooses a consumption-asset path $\{(c_t, a_t)\}$ to maximize

$$\mathbb{E} \left\{ \sum_{t=0}^{\infty} \beta^t u(c_t) \right\} \quad (1)$$

subject to

$$a_{t+1} = R_{t+1}(a_t - c_t) + Y_{t+1} \quad \text{and} \quad 0 \leq c_t \leq a_t, \quad (2)$$

with initial condition $(a_0, Z_0) = (a, z)$ treated as given.

Note that $\{R_t\}_{t \geq 1}$, the gross rate of return on wealth, is allowed to be stochastic.

The sequence $\{Y_t\}_{t \geq 1}$ is non-financial income.

The stochastic components of the problem obey

$$R_t = R(Z_t, \zeta_t) \quad \text{and} \quad Y_t = Y(Z_t, \eta_t), \quad (3)$$

where

- the maps 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 an irreducible time-homogeneous Markov chain on a finite set Z

Let P represent the Markov matrix for the chain $\{Z_t\}_{t \geq 0}$.

Our assumptions on preferences are the same as our [previous lecture](#) on the income fluctuation problem.

As before, $\mathbb{E}_z \hat{X}$ means expectation of next period value \hat{X} given current value $Z = z$.

35.3.2 Assumptions

We need restrictions to ensure that the objective (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 $\beta R < 1$.

Now it is stochastic, we require that

$$\beta G_R < 1, \quad \text{where} \quad G_R := \lim_{n \rightarrow \infty} \left(\mathbb{E} \prod_{t=1}^n R_t \right)^{1/n} \quad (4)$$

Notice that, when $\{R_t\}$ takes some constant value R , this reduces to the previous restriction $\beta R < 1$

The value G_R can be thought of as the long run (geometric) average gross rate of return.

More intuition behind (4) is provided in [75].

Discussion on how to check it is given 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 [10].

35.3.3 Optimality

Let the class of candidate consumption policies \mathcal{C} be defined [as before](#).

In [75] it is shown that, under the stated assumptions,

- any $\sigma \in \mathcal{C}$ satisfying the Euler equation is an optimal policy and
- exactly one such policy exists in \mathcal{C} .

In the present setting, the Euler equation takes the form

$$(u' \circ \sigma)(a, z) = \max \left\{ \beta \mathbb{E}_z \hat{R}(u' \circ \sigma)[\hat{R}(a - \sigma(a, z)) + \hat{Y}, \hat{Z}], u'(a) \right\} \quad (5)$$

(Intuition and derivation are similar to our [earlier lecture](#) on the income fluctuation problem.)

We again solve the Euler equation using time iteration, iterating with a Coleman–Reffett operator K defined to match the Euler equation (5).

35.4 Solution Algorithm

35.4.1 A Time Iteration Operator

Our definition of the candidate class $\sigma \in \mathcal{C}$ of consumption policies is the same as in our [earlier lecture](#) on the income fluctuation problem.

For fixed $\sigma \in \mathcal{C}$ 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

$$u'(\xi) = \max \left\{ \beta \mathbb{E}_z \hat{R} (u' \circ \sigma)[\hat{R}(a - \xi) + \hat{Y}, \hat{Z}], u'(a) \right\} \quad (6)$$

The idea behind K is that, as can be seen from the definitions, $\sigma \in \mathcal{C}$ 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 \mathcal{C} and optimal consumption policies exactly coincide (see [75] for more details).

35.4.2 Convergence Properties

As before, we pair \mathcal{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. (\mathcal{C}, ρ) is a complete metric space,
2. there exists an integer n such that K^n is a contraction mapping on (\mathcal{C}, ρ) , and
3. The unique fixed point of K in \mathcal{C} is the unique optimal policy in \mathcal{C} .

We now have a clear path to successfully approximating the optimal policy: choose some $\sigma \in \mathcal{C}$ and then iterate with K until convergence (as measured by the distance ρ).

35.4.3 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 (5) drops out, and we solve for

$$c_i = (u')^{-1} \left\{ \beta \mathbb{E}_z \hat{R}(u' \circ \sigma) [\hat{R}s_i + \hat{Y}, \hat{Z}] \right\} \quad (7)$$

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 $z \in Z$ 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.

35.4.4 Testing the Assumptions

Convergence of time iteration is dependent on the condition $\beta G_R < 1$ being satisfied.

One can check this using the fact that G_R is equal to the spectral radius of the matrix L defined by

$$L(z, \hat{z}) := P(z, \hat{z}) \int R(\hat{z}, x) \phi(x) dx$$

This identity is proved in [75], where ϕ is the density of the innovation ζ_t to returns on assets.

(Remember that Z is a finite set, so this expression defines a matrix.)

Checking the condition is even easier when $\{R_t\}$ 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.

35.5 Implementation

We will assume that $R_t = \exp(a_r \zeta_t + b_r)$ where a_r, b_r are constants and $\{\zeta_t\}$ is IID standard normal.

We allow labor income to be correlated, with

$$Y_t = \exp(a_y \eta_t + Z_t b_y)$$

where $\{\eta_t\}$ is also IID standard normal and $\{Z_t\}$ is a Markov chain taking values in $\{0, 1\}$.

```
In [3]: ifp_data = [
    ('γ', float64),                      # utility parameter
    ('β', float64),                      # discount factor
    ('P', float64[:, :]),                 # transition probs for z_t
    ('a_r', float64),                    # scale parameter for R_t
    ('b_r', float64),                    # additive parameter for R_t
    ('a_y', float64),                    # scale parameter for Y_t
    ('b_y', float64),                    # additive parameter for Y_t
    ('s_grid', float64[:]),              # Grid over savings
    ('η_draws', float64[:]),             # Draws of innovation η for MC
    ('ζ_draws', float64[:])              # Draws of innovation ζ for MC
]
```

```
In [4]: @jitclass(ifp_data)
class IFP:
    """
    A class that stores primitives for the income fluctuation
    problem.
    """

    def __init__(self,
                 γ=1.5,
                 β=0.96,
                 P=np.array([(0.9, 0.1),
                             (0.1, 0.9)]),
                 a_r=0.1,
                 b_r=0.0,
                 a_y=0.2,
                 b_y=0.5,
                 shock_draw_size=50,
                 grid_max=10,
                 grid_size=100,
                 seed=1234):
        np.random.seed(seed) # arbitrary seed

        self.P, self.γ, self.β = P, γ, β
        self.a_r, self.b_r, self.a_y, self.b_y = a_r, b_r, a_y, b_y
        self.η_draws = np.random.randn(shock_draw_size)
        self.ζ_draws = np.random.randn(shock_draw_size)
        self.s_grid = np.linspace(0, grid_max, grid_size)

        # Test stability assuming {R_t} is IID and adopts the lognormal
        # specification given below. The test is then β E R_t < 1.
        ER = np.exp(b_r + a_r**2 / 2)
        assert β * ER < 1, "Stability condition failed."

    # Marginal utility
    def u_prime(self, c):
        return c**(-self.γ)

    # Inverse of marginal utility
    def u_prime_inv(self, c):
        return c**(-1/self.γ)

    def R(self, z, ζ):
        return np.exp(self.a_r * ζ + self.b_r)
```

```
def Y(self, z, η):
    return np.exp(self.a_y * η + (z * self.b_y))
```

Here's the Coleman-Reffett operator based on EGM:

```
In [5]: @njit
def K(a_in, σ_in, ifp):
    """
    The Coleman--Reffett operator for the income fluctuation problem,
    using the endogenous grid method.

    * ifp is an instance of IFP
    * a_in[i, z] is an asset grid
    * σ_in[i, z] is consumption at a_in[i, z]
    """

    # Simplify names
    u_prime, u_prime_inv = ifp.u_prime, ifp.u_prime_inv
    R, Y, P, β = ifp.R, ifp.Y, ifp.P, ifp.β
    s_grid, η_draws, ζ_draws = ifp.s_grid, ifp.η_draws, ifp.ζ_draws
    n = len(P)

    # Create consumption function by linear interpolation
    σ = lambda a, z: interp(a_in[:, z], σ_in[:, z], a)

    # Allocate memory
    σ_out = np.empty_like(σ_in)

    # Obtain c_i at each s_i, z, store in σ_out[i, z], computing
    # the expectation term by Monte Carlo
    for i, s in enumerate(s_grid):
        for z in range(n):
            # Compute expectation
            Ez = 0.0
            for z_hat in range(n):
                for η in ifp.η_draws:
                    for ζ in ifp.ζ_draws:
                        R_hat = R(z_hat, ζ)
                        Y_hat = Y(z_hat, η)
                        U = u_prime(σ(R_hat * s + Y_hat, z_hat))
                        Ez += R_hat * U * P[z, z_hat]
            Ez = Ez / (len(η_draws) * len(ζ_draws))
            σ_out[i, z] = u_prime_inv(β * Ez)

    # Calculate endogenous asset grid
    a_out = np.empty_like(σ_out)
    for z in range(n):
        a_out[:, z] = s_grid + σ_out[:, z]

    # Fixing a consumption-asset pair at (0, 0) improves interpolation
    σ_out[0, :] = 0
    a_out[0, :] = 0

    return a_out, σ_out
```

The next function solves for an approximation of the optimal consumption policy via time iteration.

```
In [6]: def solve_model_time_iter(model,          # Class with model information
                                a_vec,           # Initial condition for assets
                                σ_vec,           # Initial condition for consumption
                                tol=1e-4,
                                max_iter=1000,
                                verbose=True,
                                print_skip=25):

    # Set up loop
    i = 0
    error = tol + 1

    while i < max_iter and error > tol:
        a_new, σ_new = K(a_vec, σ_vec, model)
        error = np.max(np.abs(σ_vec - σ_new))
        i += 1
        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")
        a_vec, σ_vec = np.copy(a_new), np.copy(σ_new)

    if i == max_iter:
        print("Failed to converge!")

    if verbose and i < max_iter:
        print(f"\nConverged in {i} iterations.")

return a_new, σ_new
```

Now we are ready to create an instance at the default parameters.

```
In [7]: ifp = IFP()
```

Next we set up an initial condition, which corresponds to consuming all assets.

```
In [8]: # Initial guess of σ = consume all assets
k = len(ifp.s_grid)
n = len(ifp.P)
σ_init = np.empty((k, n))
for z in range(n):
    σ_init[:, z] = ifp.s_grid
a_init = np.copy(σ_init)
```

Let's generate an approximation solution.

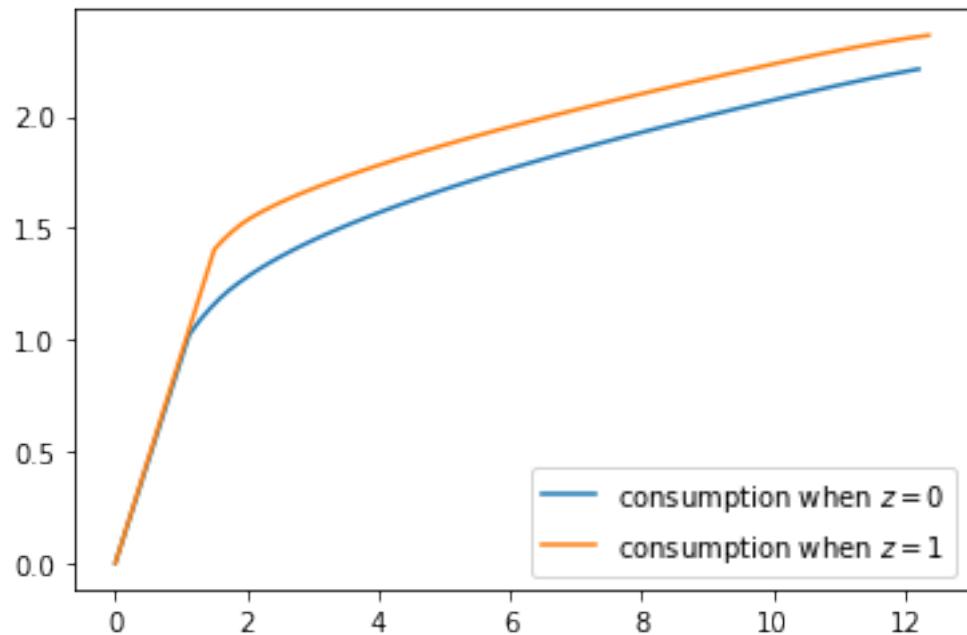
```
In [9]: a_star, σ_star = solve_model_time_iter(ifp, a_init, σ_init, print_skip=5)
```

```
Error at iteration 5 is 0.5081944529506561.
Error at iteration 10 is 0.1057246950930697.
Error at iteration 15 is 0.03658262202883744.
Error at iteration 20 is 0.013936729965906114.
Error at iteration 25 is 0.005292165269711546.
Error at iteration 30 is 0.0019748126990770665.
Error at iteration 35 is 0.0007219210463285108.
Error at iteration 40 is 0.0002590544496094971.
Error at iteration 45 is 9.163966595426842e-05.
```

Converged in 45 iterations.

Here's a plot of the resulting consumption policy.

```
In [10]: fig, ax = plt.subplots()
for z in range(len(ifp.P)):
    ax.plot(a_star[:, z], sigma_star[:, z], label=f"consumption when $z={z}$")
plt.legend()
plt.show()
```



Notice that we consume all assets in the lower range of the asset space.

This is because we anticipate income Y_{t+1} tomorrow, which makes the need to save less urgent.

Can you explain why consuming all assets ends earlier (for lower values of assets) when $z = 0$?

35.5.1 Law of Motion

Let's try to get some idea of what will happen to assets over the long run under this consumption policy.

As with our [earlier lecture](#) on the income fluctuation problem, we begin by producing a 45 degree diagram showing the law of motion for assets

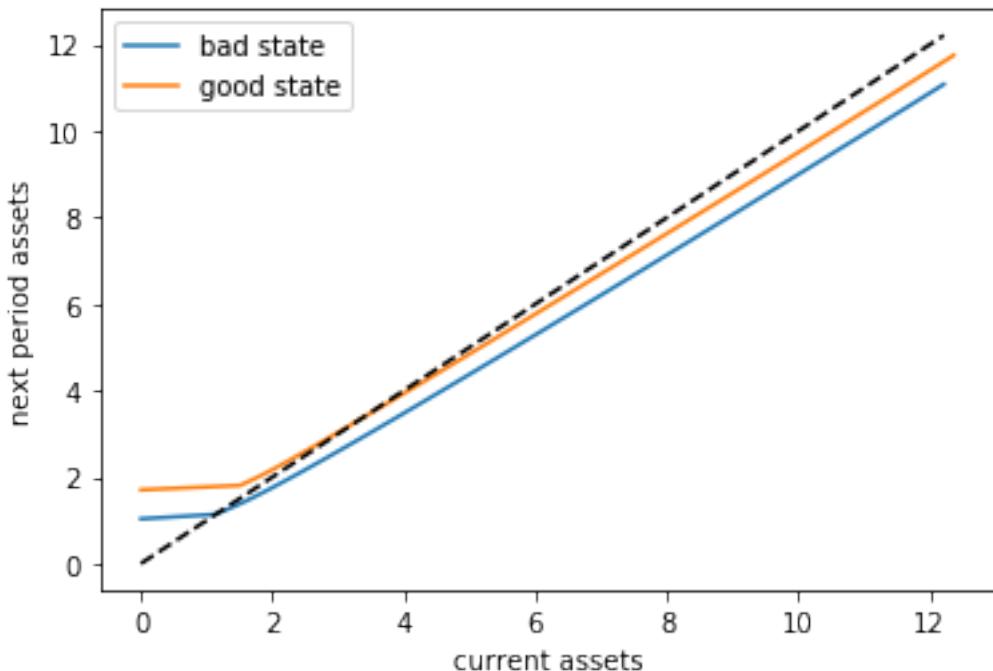
```
In [11]: # Good and bad state mean labor income
Y_mean = [np.mean(ifp.Y(z, ifp.n_draws)) for z in (0, 1)]
# Mean returns
```

```
R_mean = np.mean(ifp.R(z, ifp.z_draws))

a = a_star
fig, ax = plt.subplots()
for z, lb in zip((0, 1), ('bad state', 'good state')):
    ax.plot(a[:, z], R_mean * (a[:, z] - sigma_star[:, z]) + Y_mean[z], label=lb)

ax.plot(a[:, 0], a[:, 0], 'k--')
ax.set(xlabel='current assets', ylabel='next period assets')

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



The unbroken lines represent, for each z , an average update function for assets, given by

$$a \mapsto \bar{R}(a - \sigma^*(a, z)) + \bar{Y}(z)$$

Here

- $\bar{R} = \mathbb{E}R_t$, which is mean returns and
- $\bar{Y}(z) = \mathbb{E}_z Y(z, \eta_t)$, which is mean labor income in state z .

The dashed line is the 45 degree line.

We can see from the figure that the dynamics will be stable — assets do not diverge even in the highest state.

35.6 Exercises

35.6.1 Exercise 1

Let's repeat our [earlier exercise](#) on the long-run cross sectional distribution of assets.

In that exercise, we used a relatively simple income fluctuation model.

In the solution, we found the shape of the asset distribution to be unrealistic.

In particular, we failed to match the long right tail of the wealth distribution.

Your task is to try again, repeating the exercise, but now with our more sophisticated model.

Use the default parameters.

35.7 Solutions

35.7.1 Exercise 1

First we write a function to compute a long asset series.

Because we want to JIT-compile the function, we code the solution in a way that breaks some rules on good programming style.

For example, we will pass in the solutions `a_star`, `σ_star` along with `ifp`, even though it would be more natural to just pass in `ifp` and then solve inside the function.

The reason we do this is that `solve_model_time_iter` is not JIT-compiled.

```
In [12]: @njit
def compute_asset_series(ifp, a_star, σ_star, z_seq, T=500_000):
    """
        Simulates a time series of length T for assets, given optimal
        savings behavior.

        * ifp is an instance of IFP
        * a_star is the endogenous grid solution
        * σ_star is optimal consumption on the grid
        * z_seq is a time path for {Z_t}
    """

    # Create consumption function by linear interpolation
    σ = lambda a, z: interp(a_star[:, z], σ_star[:, z], a)

    # Simulate the asset path
    a = np.zeros(T+1)
    for t in range(T):
        z = z_seq[t]
        ζ, η = np.random.randn(), np.random.randn()
        R = ifp.R(z, ζ)
        Y = ifp.Y(z, η)
        a[t+1] = R * (a[t] - σ(a[t], z)) + Y
    return a
```

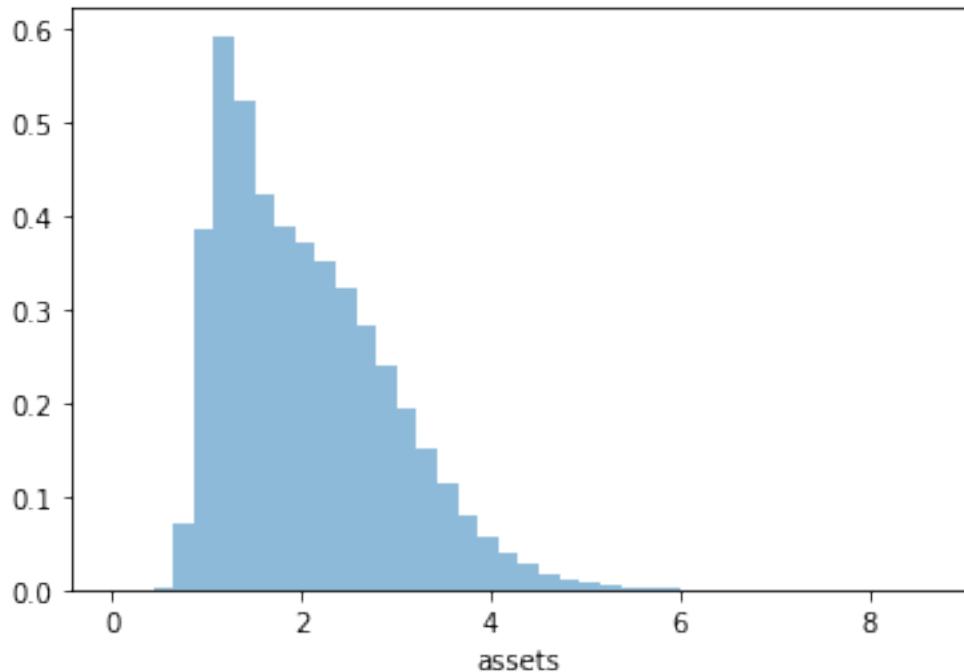
Now we call the function, generate the series and then histogram it, using the solutions com-

puted above.

```
In [13]: T = 1_000_000
mc = MarkovChain(ifp.P)
z_seq = mc.simulate(T, random_state=1234)

a = compute_asset_series(ifp, a_star, sigma_star, z_seq, T=T)

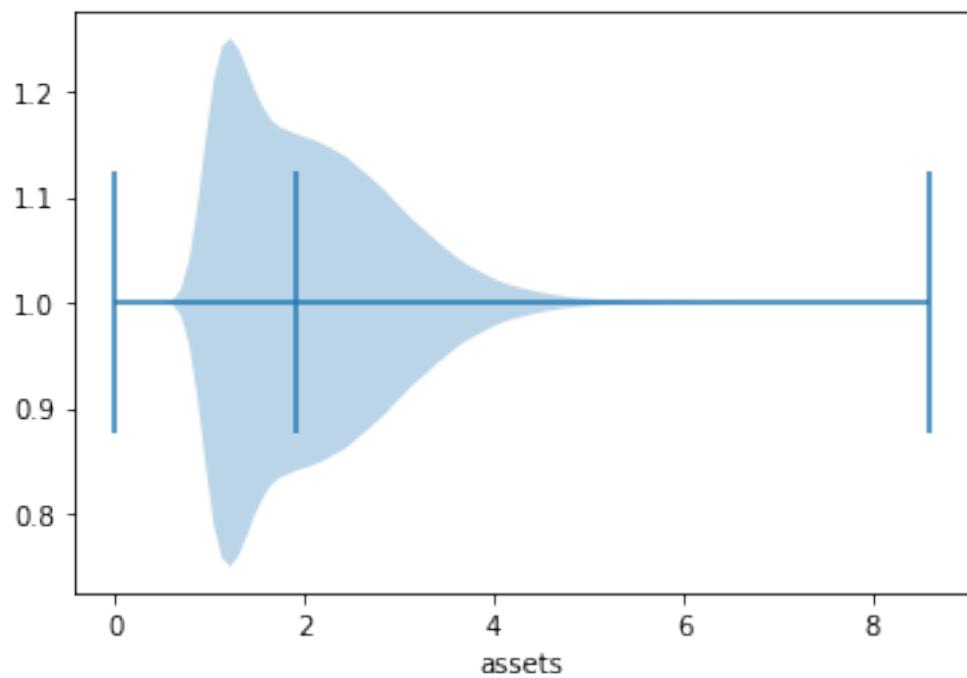
fig, ax = plt.subplots()
ax.hist(a, bins=40, alpha=0.5, density=True)
ax.set(xlabel='assets')
plt.show()
```



Now we have managed to successfully replicate the long right tail of the wealth distribution.

Here's another view of this using a horizontal violin plot.

```
In [14]: fig, ax = plt.subplots()
ax.violinplot(a, vert=False, showmedians=True)
ax.set(xlabel='assets')
plt.show()
```



Part V

Information

Chapter 36

Job Search VII: Search with Learning

36.1 Contents

- Overview 36.2
- Model 36.3
- Take 1: Solution by VFI 36.4
- Take 2: A More Efficient Method 36.5
- Another Functional Equation 36.6
- Solving the RWFE 36.7
- Implementation 36.8
- Exercises 36.9
- Solutions 36.10
- Appendix A 36.11
- Appendix B 36.12
- Examples 36.13

In addition to what's in Anaconda, this lecture deploys the libraries:

```
In [1]: !pip install quantecon  
!pip install interpolation
```

36.2 Overview

In this lecture, we consider an extension of the [previously studied](#) job search model of McCall [80].

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 [72], section 6.6.

Let's start with some imports

```
In [2]: from numba import njit, prange, vectorize
from interpolation import mlininterp, interp
from math import gamma
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from matplotlib import cm
import scipy.optimize as op
from scipy.stats import cumfreq, beta
```

36.2.1 Model Features

- Infinite horizon dynamic programming with two states and one binary control.
- Bayesian updating to learn the unknown distribution.

36.3 Model

Let's first review the basic McCall model [80] and then add the variation we want to consider.

36.3.1 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 $\mathbb{E} \sum_{t=0}^{\infty} \beta^t y_t$. The function V satisfies the recursion

$$v(w) = \max \left\{ \frac{w}{1-\beta}, c + \beta \int v(w') q(w') dw' \right\} \quad (1)$$

The optimal policy has the form $\mathbf{1}\{w \geq \bar{w}\}$, where \bar{w} is a constant called the *reservation wage*.

36.3.2 Offer Distribution Unknown

Now let's extend the model by considering the variation presented in [72], 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 — with densities f and g .

At the start of time, “nature” selects q to be either f or g — the wage distribution from which the entire sequence W_t will be drawn.

This choice is not observed by the worker, who puts prior probability π_0 on f being chosen.

Update rule: worker’s time t estimate of the distribution is $\pi_t f + (1 - \pi_t)g$, where π_t updates via

$$\pi_{t+1} = \frac{\pi_t f(w_{t+1})}{\pi_t f(w_{t+1}) + (1 - \pi_t)g(w_{t+1})} \quad (2)$$

This last expression follows from Bayes’ rule, which tells us that

$$\mathbb{P}\{q = f \mid W = w\} = \frac{\mathbb{P}\{W = w \mid q = f\}\mathbb{P}\{q = f\}}{\mathbb{P}\{W = w\}} \quad \text{and} \quad \mathbb{P}\{W = w\} = \sum_{\omega \in \{f,g\}} \mathbb{P}\{W = w \mid q = \omega\}\mathbb{P}\{q = \omega\}$$

The fact that (2) is recursive allows us to progress to a recursive solution method.

Letting

$$q_\pi(w) := \pi f(w) + (1 - \pi)g(w) \quad \text{and} \quad \kappa(w, \pi) := \frac{\pi f(w)}{\pi f(w) + (1 - \pi)g(w)}$$

we can express the value function for the unemployed worker recursively as follows

$$v(w, \pi) = \max \left\{ \frac{w}{1 - \beta}, c + \beta \int v(w', \pi') q_\pi(w') dw' \right\} \quad \text{where} \quad \pi' = \kappa(w', \pi) \quad (3)$$

Notice that the current guess π is a state variable, since it affects the worker’s perception of probabilities for future rewards.

36.3.3 Parameterization

Following section 6.6 of [72], our baseline parameterization will be

- f is Beta(1, 1)
- g is Beta(3, 1.2)
- $\beta = 0.95$ and $c = 0.3$

The densities f and g have the following shape

```
In [3]: @vectorize
def p(x, a, b):
    r = gamma(a + b) / (gamma(a) * gamma(b))
    return r * x**(a-1) * (1 - x)**(b-1)
```

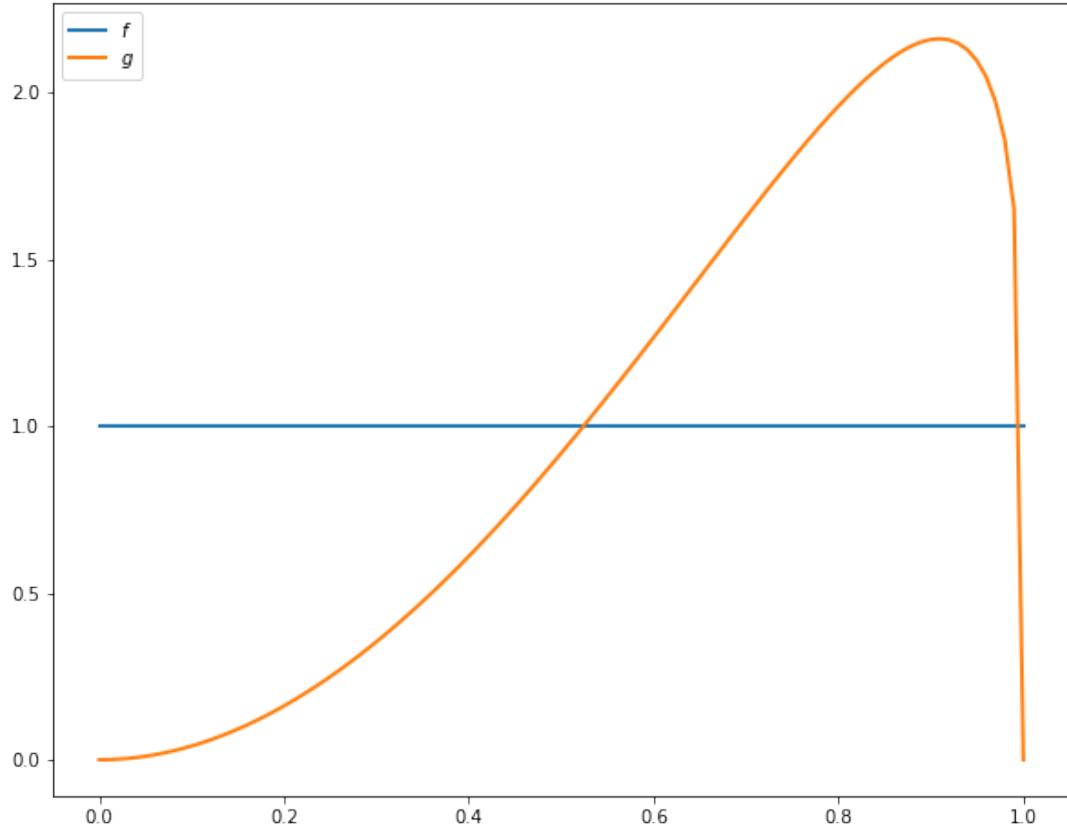
```

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()

```



36.3.4 Looking Forward

What kind of optimal policy might result from (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, and relatively low current offers become more attractive.

Summary: We conjecture that the optimal policy is of the form $\mathbf{1}w \geq \bar{w}(\pi)$ for some decreasing function \bar{w} .

36.4 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.

```
In [4]: 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,                         # Offer distribution
                 F_b=1,                         # Offer distribution
                 G_a=3,                         # Offer distribution
                 G_b=1.2,                        # Offer distribution
                 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 = njit(lambda x: p(x, F_a, F_b))
        self.g = njit(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

```
In [5]: 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

@njit
def v_func(x, y, v):
    return mlinterp((w_grid, π_grid), v, (x, y))

@njit
def x(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

@njit(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], x(w_f[m], π), v)
                integral_g += v_func(w_g[m], x(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

@njit(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], x(w_f[m], π), v)
            integral_g += v_func(w_g[m], x(w_g[m], π), v)
        integral = (π * integral_f + (1 - π) * integral_g) / mc_size

        v_2 = c + β * integral

        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

```

In [6]: 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 i == max_iter:
        print("Failed to converge!")

    if verbose and i < max_iter:
        print(f"\nConverged in {i} iterations.")

```

```
return v_new
```

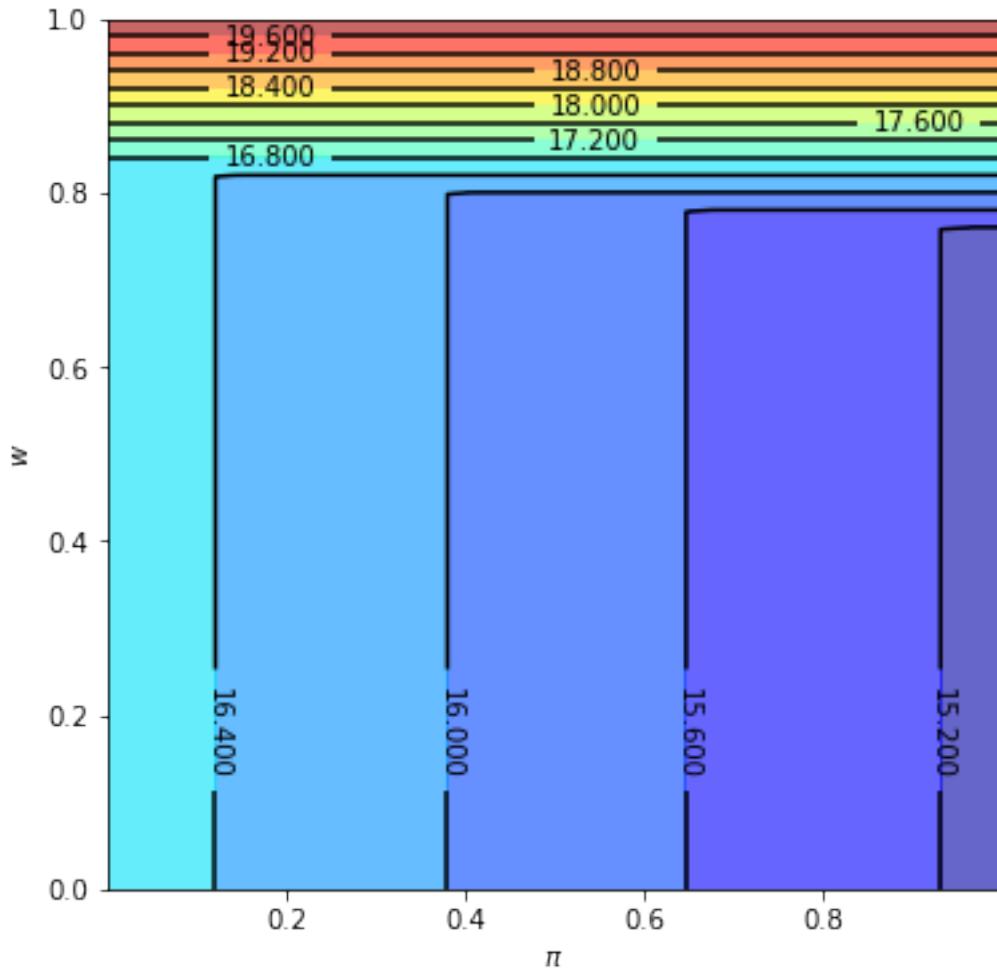
Let's look at solutions computed from value function iteration

```
In [7]: sp = SearchProblem()
v_star = solve_model(sp)
fig, ax = plt.subplots(figsize=(6, 6))
ax.contourf(sp.pi_grid, sp.w_grid, v_star, 12, alpha=0.6, cmap=cm.jet)
cs = ax.contour(sp.pi_grid, sp.w_grid, v_star, 12, colors="black")
ax.clabel(cs, inline=1, fontsize=10)
ax.set(xlabel='$\pi$', ylabel='$w$')
plt.show()
```

```
/home/ubuntu/anaconda3/lib/python3.7/site-packages/numba/np/ufunc/parallel.py:
 355:
NumbaWarning: The TBB threading layer requires TBB version 2019.5 or later i.e.,
TBB_INTERFACE_VERSION >= 11005. Found TBB_INTERFACE_VERSION = 11004. The TBB
  threading
layer is disabled.
  warnings.warn(problem)
```

```
Error at iteration 5 is 0.6052209317599999.
Error at iteration 10 is 0.10498092704095896.
Error at iteration 15 is 0.025540330996948413.
Error at iteration 20 is 0.006556117688202079.
Error at iteration 25 is 0.0016835434969930674.
Error at iteration 30 is 0.0004322350635970196.
Error at iteration 35 is 0.00011096514681341318.
```

Converged in 36 iterations.



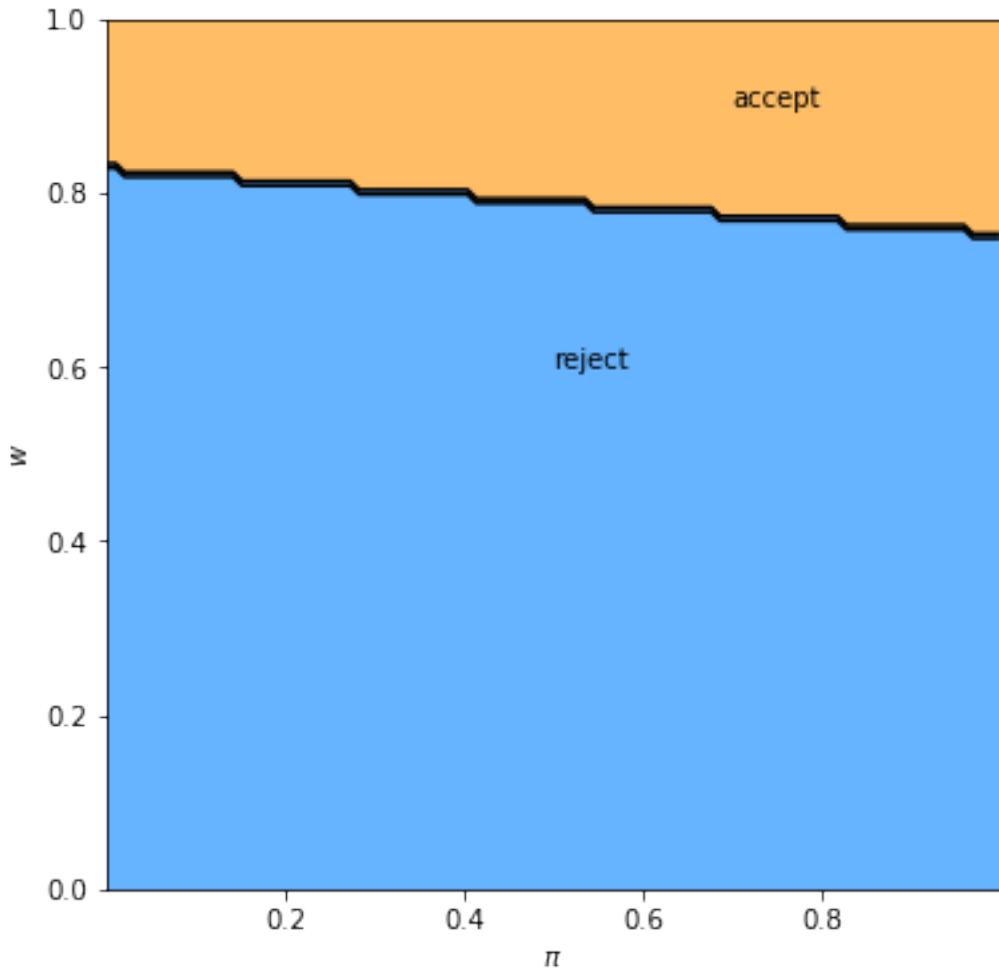
We will also plot the optimal policy

```
In [8]: 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")
ax.set(xlabel='$\pi$', 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]{looking forward}.

- The black line in the figure above corresponds to the function $\bar{w}(\pi)$ introduced there.
- It is decreasing as expected.

36.5 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.

36.6 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 (3) have equal value:

$$\frac{\bar{w}(\pi)}{1-\beta} = c + \beta \int v(w', \pi') q_\pi(w') dw' \quad (4)$$

Together, (3) and (4) give

$$v(w, \pi) = \max \left\{ \frac{w}{1-\beta}, \frac{\bar{w}(\pi)}{1-\beta} \right\} \quad (5)$$

Combining (4) and (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

$$\bar{w}(\pi) = (1 - \beta)c + \beta \int \max \{ w', \bar{w} \circ \kappa(w', \pi) \} q_\pi(w') dw' \quad (6)$$

Equation (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.

36.7 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

$$(Q\omega)(\pi) = (1 - \beta)c + \beta \int \max \{ w', \omega \circ \kappa(w', \pi) \} q_\pi(w') dw' \quad (7)$$

Comparing (6) and (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 (6) and vice versa.

Moreover, for any $\omega, \omega' \in b[0, 1]$, basic algebra and the triangle inequality for integrals tells us that

$$|(Q\omega)(\pi) - (Q\omega')(\pi)| \leq \beta \int |\max\{w', \omega \circ \kappa(w', \pi)\} - \max\{w', \omega' \circ \kappa(w', \pi)\}| q_\pi(w') dw' \quad (8)$$

Working case by case, it is easy to check that for real numbers a, b, c we always have

$$|\max\{a, b\} - \max\{a, c\}| \leq |b - c| \quad (9)$$

Combining (8) and (9) yields

$$|(Q\omega)(\pi) - (Q\omega')(\pi)| \leq \beta \int |\omega \circ \kappa(w', \pi) - \omega' \circ \kappa(w', \pi)| q_\pi(w') dw' \leq \beta \|\omega - \omega'\| \quad (10)$$

Taking the supremum over π now gives us

$$\|Q\omega - Q\omega'\| \leq \beta \|\omega - \omega'\| \quad (11)$$

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 $b[0, 1]$.
- $Q^k \omega \rightarrow \bar{w}$ uniformly as $k \rightarrow \infty$, for any $\omega \in b[0, 1]$.

36.8 Implementation

The following function takes an instance of `SearchProblem` and returns the operator `Q`

In [9]: `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

@njit
def ω_func(p, ω):
    return interp(π_grid, ω, p)

@njit
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

@njit(parallel=parallel_flag)
def Q(ω):
    """
    """
    # Implementation of the operator Q based on the given logic.
```

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(x(w_f[m], π), ω))
        integral_g += max(w_g[m], ω_func(x(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} .

36.9 Exercises

36.9.1 Exercise 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 [Take-1:-Solution-by-VFI]{shown above}.

Try experimenting with different parameters, and confirm that the change in the optimal policy coincides with your intuition.

36.10 Solutions

36.10.1 Exercise 1

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

```
In [10]: 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.pi_grid)

# Initialize w
w = np.ones_like(sp.pi_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 i == max_iter:
    print("Failed to converge!")

if verbose and i < max_iter:
    print(f"\nConverged in {i} iterations.")

return w_new

```

The solution can be plotted as follows

```

In [11]: sp = SearchProblem()
w_bar = solve_wbar(sp)

fig, ax = plt.subplots(figsize=(9, 7))

ax.plot(sp.pi_grid, w_bar, color='k')
ax.fill_between(sp.pi_grid, 0, w_bar, color='blue', alpha=0.15)
ax.fill_between(sp.pi_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='$\pi$', ylabel='$w$')
ax.grid()
plt.show()

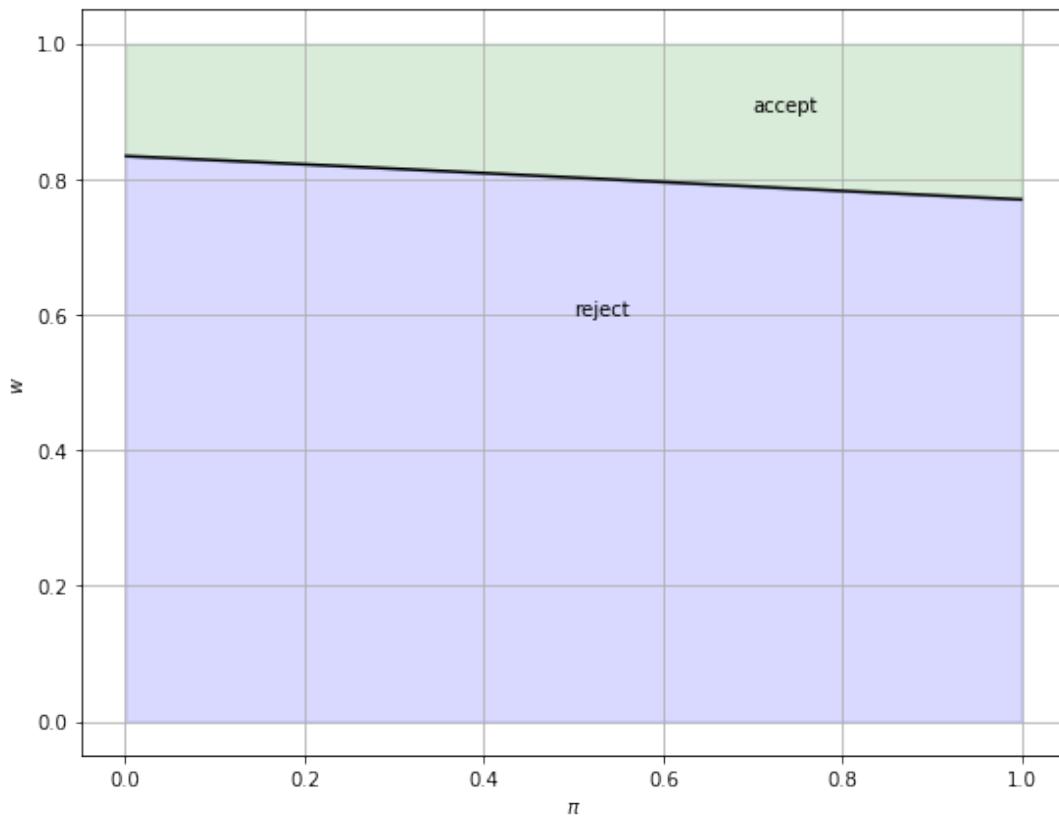
```

```

Error at iteration 5 is 0.022472695692739575.
Error at iteration 10 is 0.0072369880915718054.
Error at iteration 15 is 0.0016332155226363998.
Error at iteration 20 is 0.0003422178261048847.

```

Converged in 24 iterations.



36.11 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

```
In [12]: 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
pi_grid = sp.pi_grid
w_func = njit(lambda x: interp(pi_grid, w_bar, x))

@njit
def update(a, b, e, pi):
```

```

"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

@njit
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.ones((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)      # 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

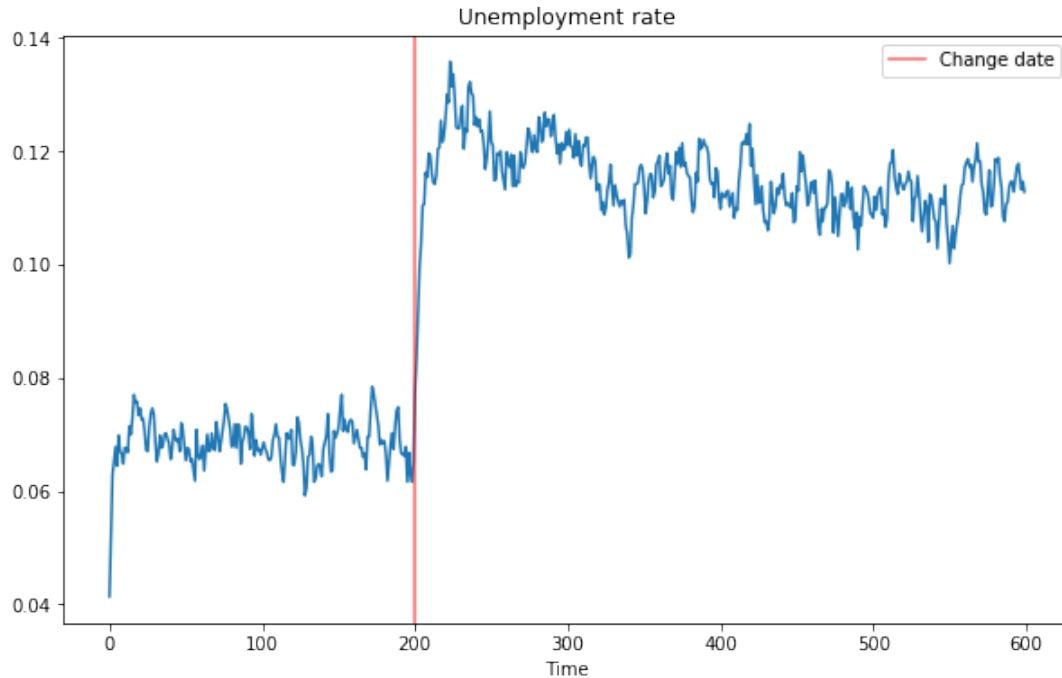
    return e[:, 1:]

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')

```

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



36.12 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.

In [13]:

```
@njit
def empirical_dist(F_a, F_b, G_a, G_b, w_bar, pi_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
    -----
```

```

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
-----
accept_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 >= interp(π_grid, w_bar, π):
                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 differ-

ent 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.

```
In [14]: 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  $\pi$ , and the probability of accepting
        an offer at different  $\pi$  level, and simulate and calculate
        the empirical cumulative distribution of the duration of
        unemployment and  $\pi$  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
    pi_grid = sp.pi_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.pi_grid, w_bar, color='k')
    axs[0, 1].fill_between(sp.pi_grid, 0, w_bar, color='blue', alpha=0.15)
    axs[0, 1].fill_between(sp.pi_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)))
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='$\pi$', 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_ylimits([0., 1.])
axs[1, 1].grid()
axs[1, 1].legend(loc=4)
axs[1, 1].set(xlabel='$\pi$', ylabel='$\mathbb{P}\{w > \overline{w}\} \rightarrow (\pi)\}$')

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")
axs[0].plot(cumfreq_x(cfq_t_G), cfq_t_G.cumcount/N, label="g")
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")
axs[1].plot(cumfreq_x(cfq_π_G), cfq_π_G.cumcount/N, label="g")
axs[1].grid(linestyle='--')

```

```

axs[1].legend(loc=4)
axs[1].title.set_text('CDF of  $\pi$  at time worker accepts wage and leaves
unemployment')
axs[1].set(xlabel=' $\pi$ ', ylabel='Prob( $\pi$ )')

plt.show()

```

We now provide some examples that provide insights about how the model works.

36.13 Examples

36.13.1 Example 1 (Baseline)

$F \sim \text{Beta}(1, 1)$, $G \sim \text{Beta}(3, 1.2)$, $c=0.3$.

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](#)

$$\frac{\pi_{t+1}}{\pi_t} = \frac{l(w_{t+1})}{\pi_t l(w_{t+1}) + (1 - \pi_t)} \begin{cases} > 1 & \text{if } l(w_{t+1}) > 1 \\ \leq 1 & \text{if } l(w_{t+1}) \leq 1 \end{cases}$$

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 prior 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.

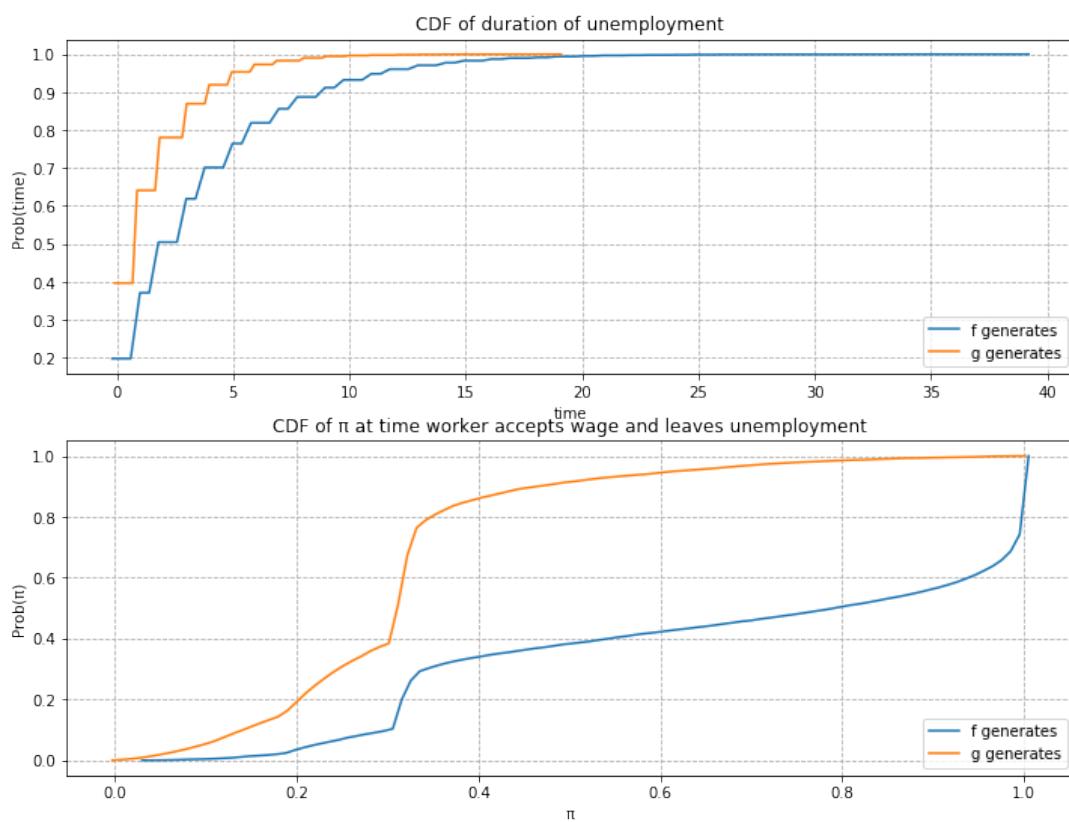
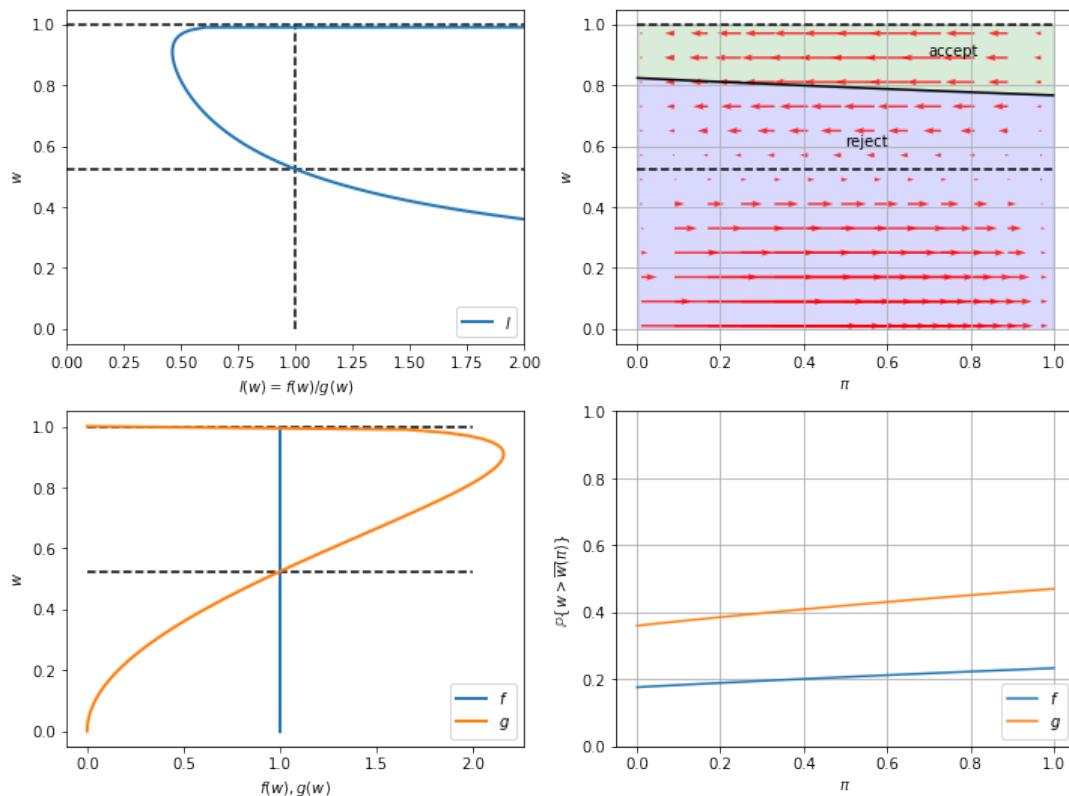
Quantitatively, the lower right figure sheds light on which effect dominates in this example.

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.

In [15]: `job_search_example()`



36.13.2 Example 2

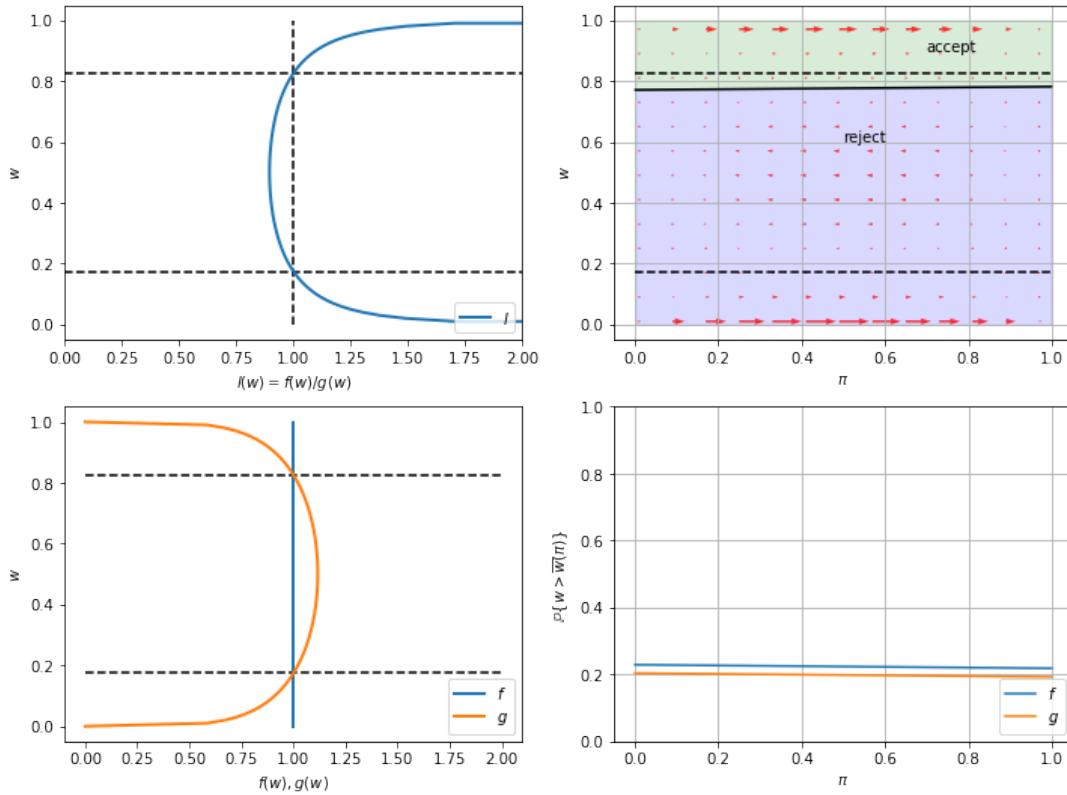
$F \sim \text{Beta}(1, 1)$, $G \sim \text{Beta}(1.2, 1.2)$, $c=0.3$.

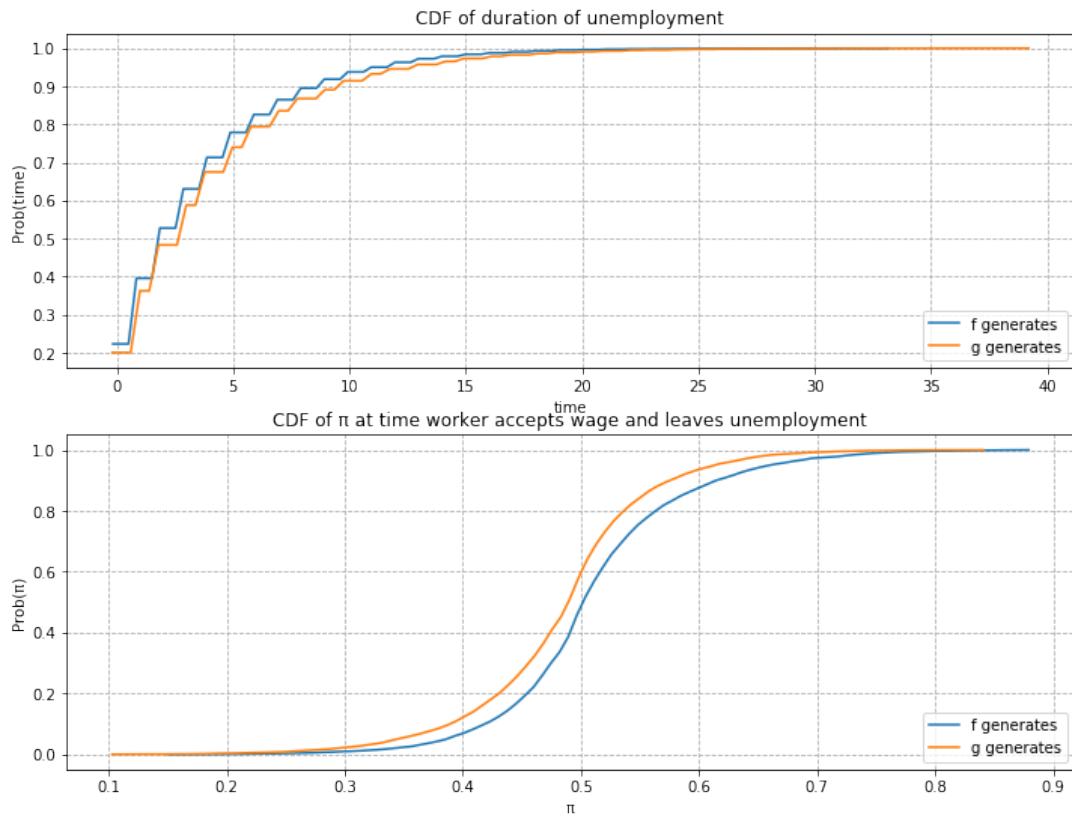
Now G has the same mean as F with a smaller variance.

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 $\bar{w}(\pi)$ is increasing in π .

```
In [16]: job_search_example(1, 1, 1.2, 1.2, 0.3)
```



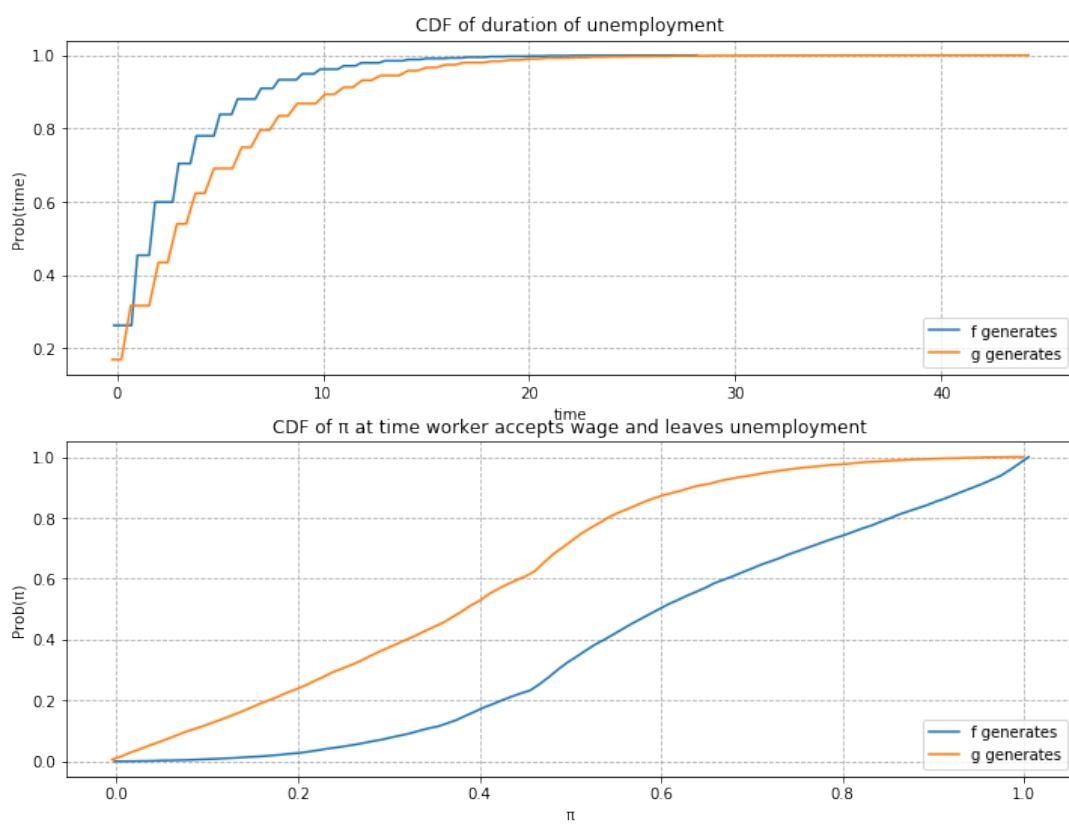
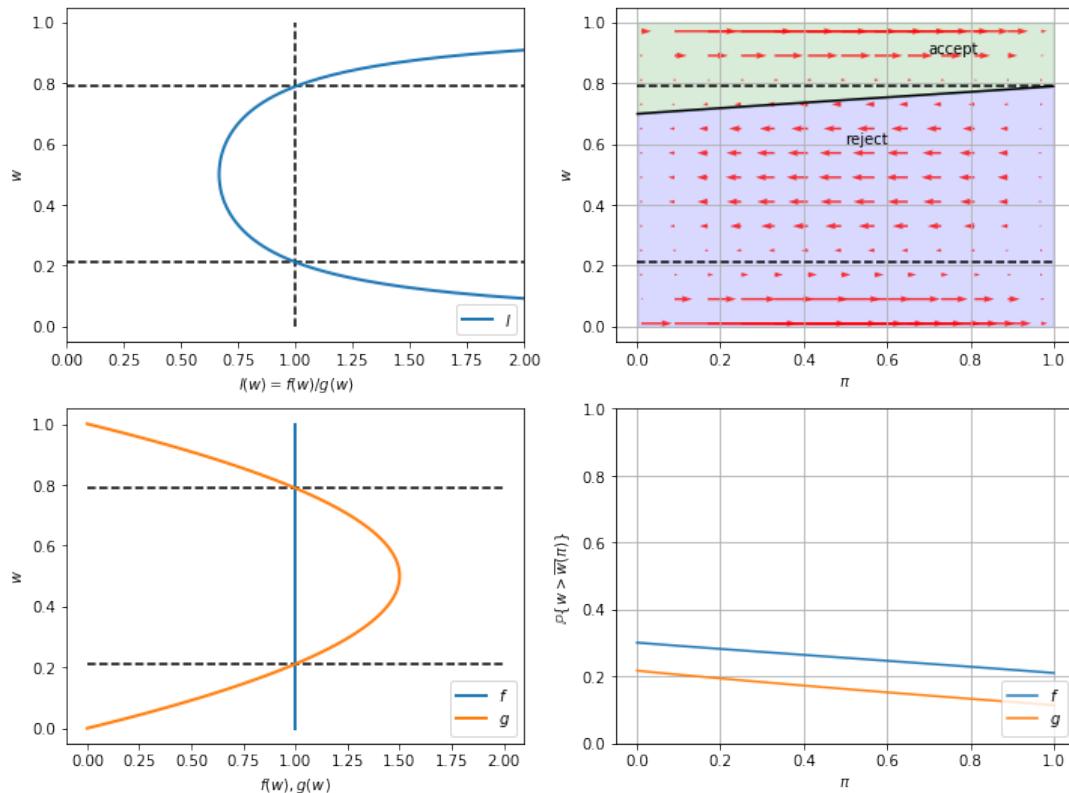


36.13.3 Example 3

$F \sim \text{Beta}(1, 1)$, $G \sim \text{Beta}(2, 2)$, $c=0.3$.

If the variance of G is smaller, we observe in the result that G is even more “inferior” and the slope of $\bar{w}(\pi)$ is larger.

In [17]: `job_search_example(1, 1, 2, 2, 0.3)`



36.13.4 Example 4

$F \sim \text{Beta}(1, 1)$, $G \sim \text{Beta}(3, 1.2)$, 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 .

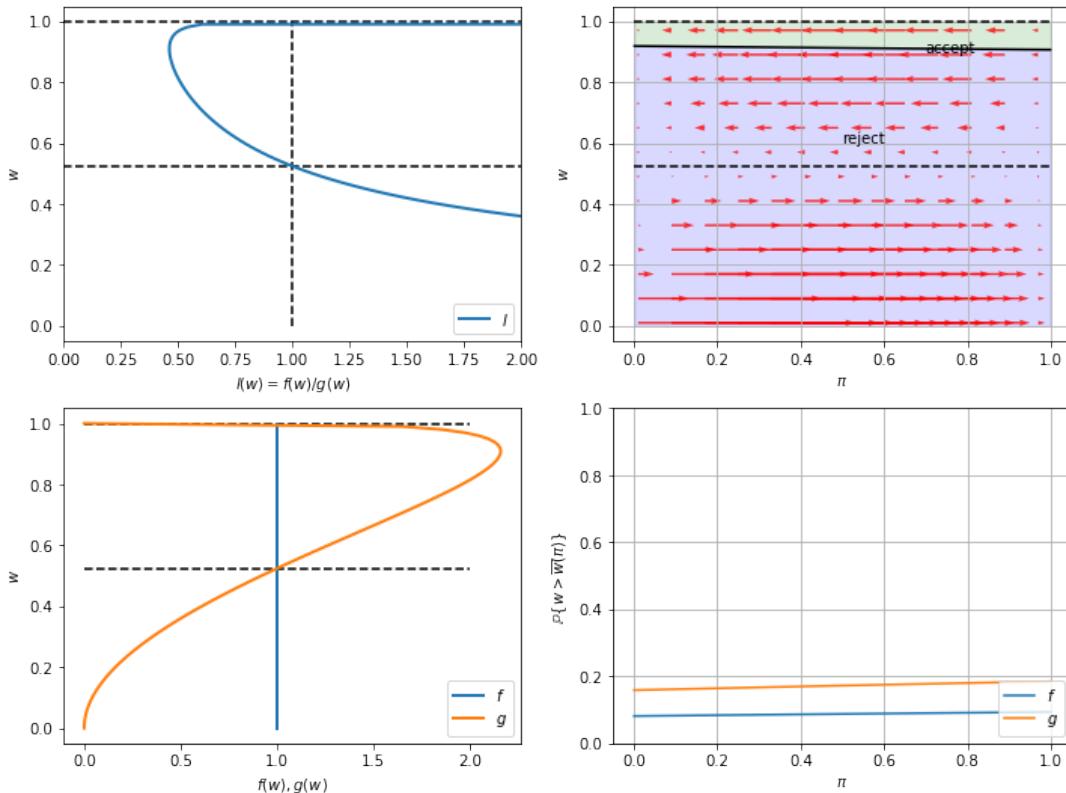
Comparing outcomes to the baseline case (example 1) in which unemployment compensation if low ($c=0.3$), now the worker can afford a longer learning period.

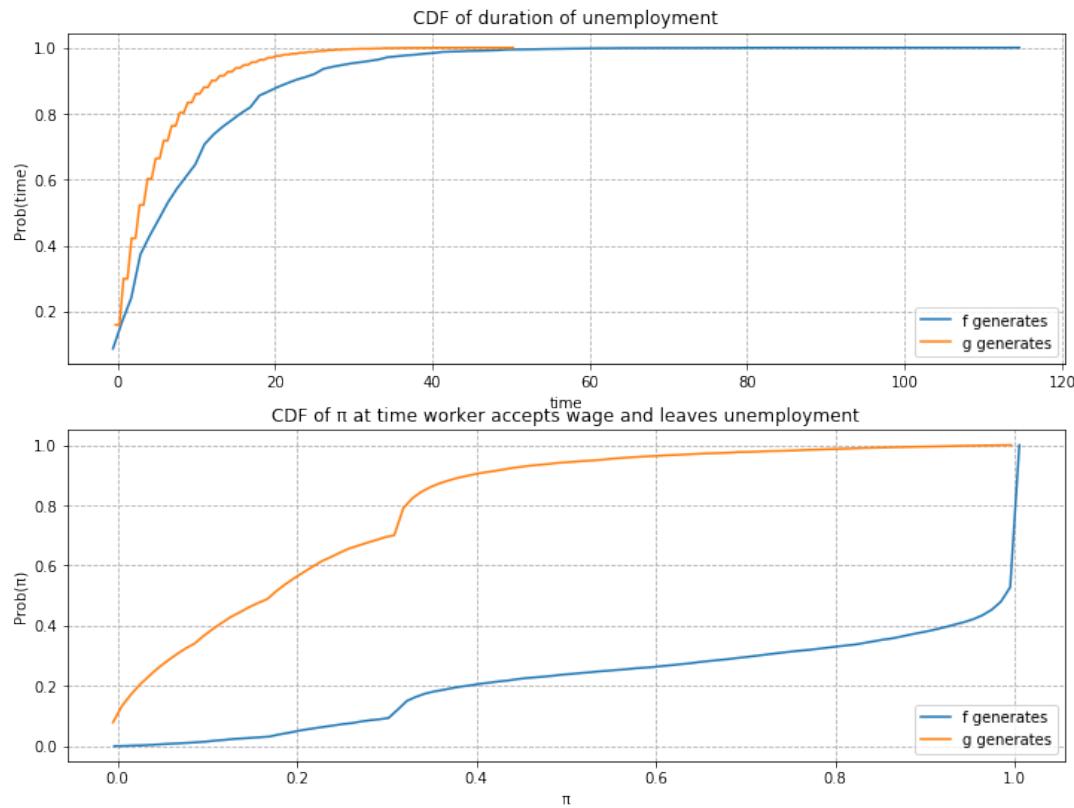
As a result, the worker tends to accept wage offers much later.

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.

In [18]: `job_search_example(1, 1, 3, 1.2, c=0.8)`



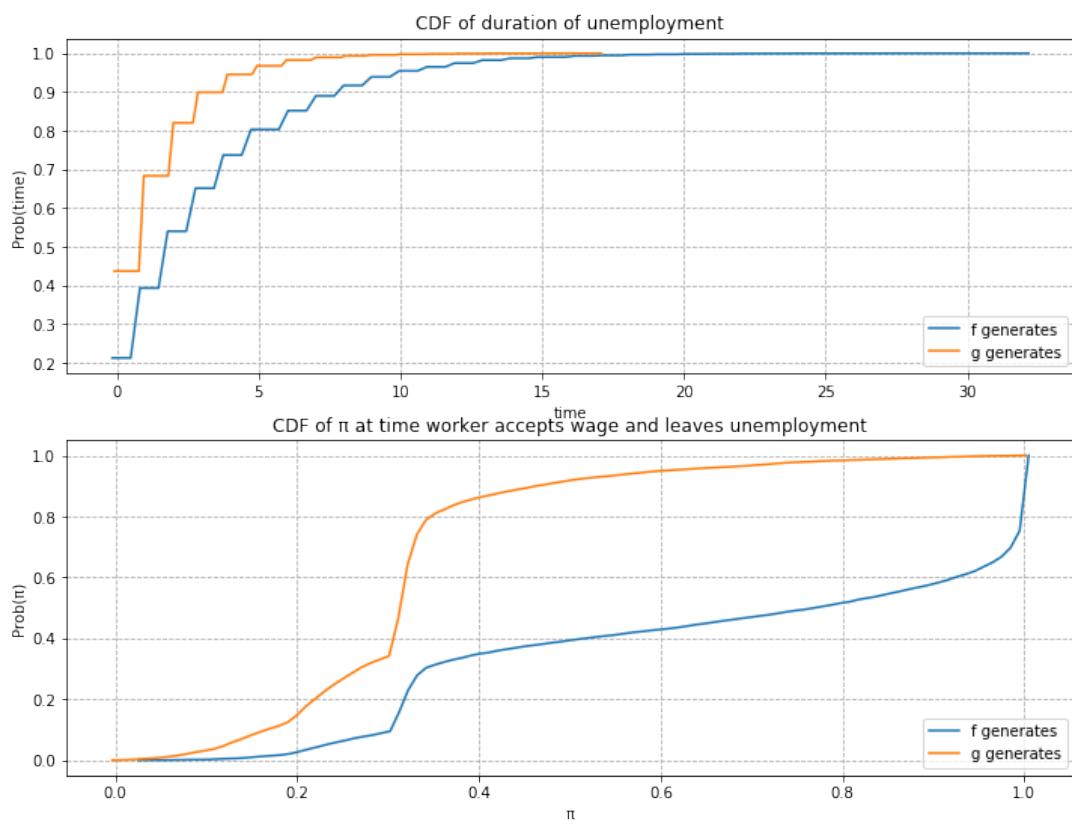
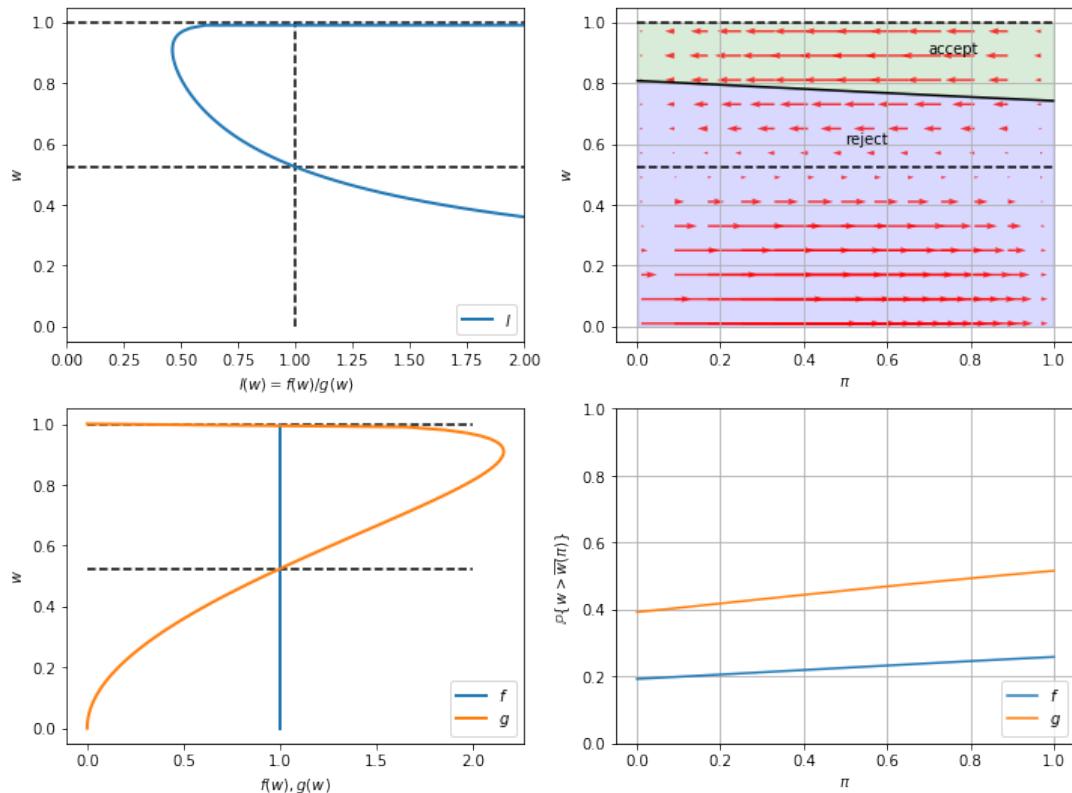


36.13.5 Example 5

$F \sim \text{Beta}(1, 1)$, $G \sim \text{Beta}(3, 1.2)$, 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.

```
In [19]: job_search_example(1, 1, 3, 1.2, c=0.1)
```



Chapter 37

Likelihood Ratio Processes

37.1 Contents

- Overview 37.2
- Likelihood Ratio Process 37.3
- Nature Permanently Draws from Density g 37.4
- Nature Permanently Draws from Density f 37.5
- Likelihood Ratio Test 37.6
- Kullback–Leibler divergence 37.7
- Sequels 37.8

```
In [1]: import numpy as np
import matplotlib.pyplot as plt
from numba import vectorize, njit
from math import gamma
%matplotlib inline
from scipy.integrate import quad
```

37.2 Overview

This lecture describes likelihood ratio processes and some of their uses.

We'll use a setting described in [this lecture](#).

Among things that we'll learn are

- A peculiar property of likelihood ratio processes
- 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 during World War II the United States Navy devised a decision rule that Captain Garret L. Schyler challenged and asked Milton Friedman to justify to him, a topic to be studied in [this lecture](#)

37.3 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 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 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 $w^t = \{w_1, \dots, w_t\}$ 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* 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 [?].

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 distributionss, for example, a sequence of IID draws from g .

In [2]: # 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 = njit(lambda x: p(x, F_a, F_b))
g = njit(lambda x: p(x, G_a, G_b))
```

```
In [3]: @njit
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
```

37.4 Nature Permanently Draws from Density g

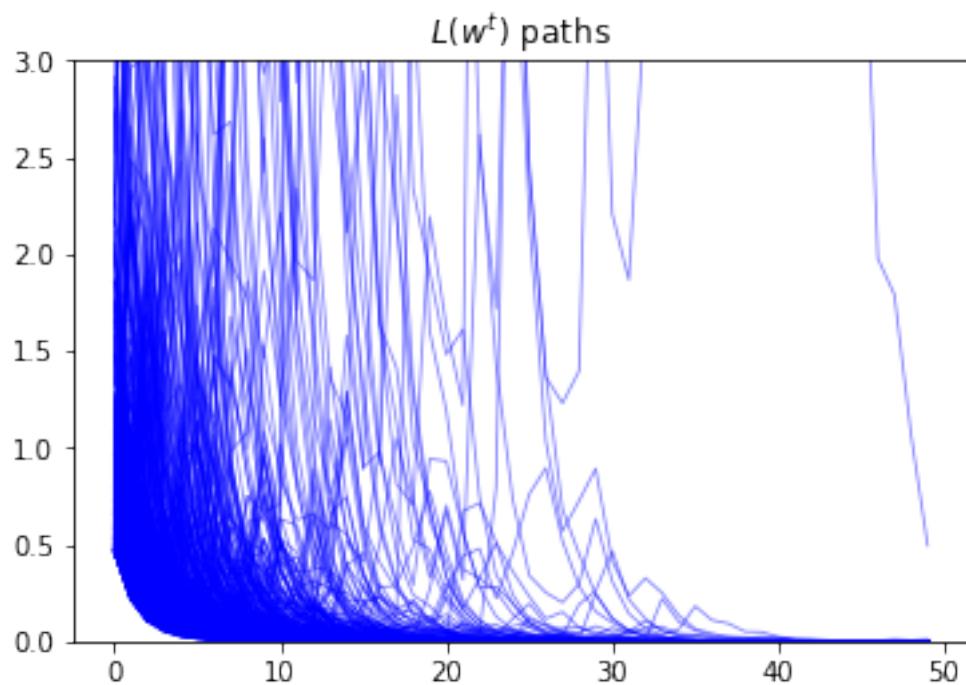
We first simulate the likelihood ratio process when nature permanently draws from g .

```
In [4]: l_arr_g = simulate(G_a, G_b)
l_seq_g = np.cumprod(l_arr_g, axis=1)
```

```
In [5]: N, T = l_arr_g.shape

for i in range(N):
    plt.plot(range(T), l_seq_g[i, :], color='b', lw=0.8, alpha=0.5)

plt.ylim([0, 3])
plt.title("$L(w^t)$ paths");
```

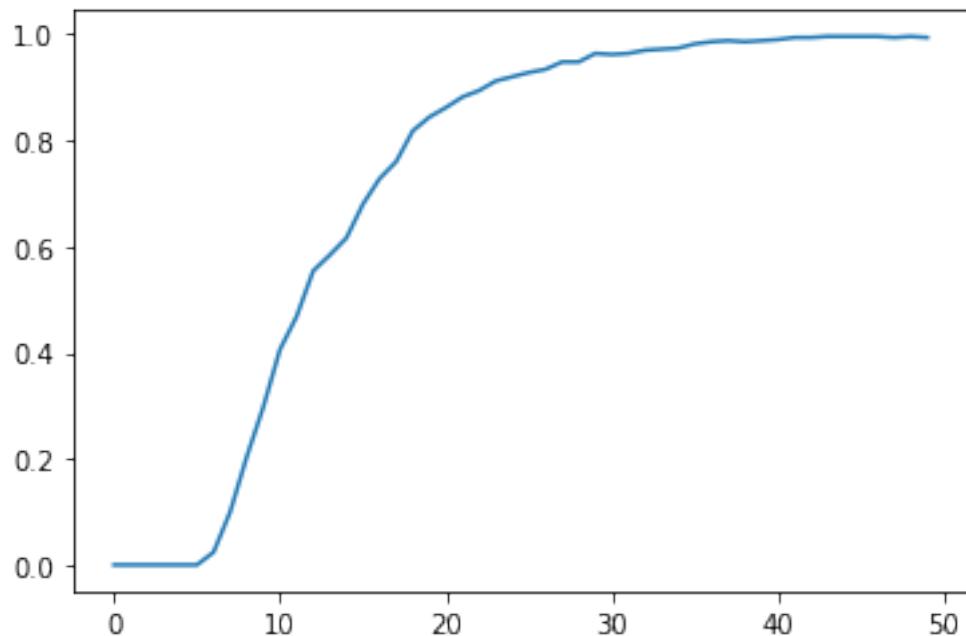


Evidently, as sample length T grows, most probability mass shifts toward zero

To see it this more clearly clearly, we plot over time the fraction of paths $L(w^t)$ that fall in the interval $[0, 0.01]$.

```
In [6]: plt.plot(range(T), np.sum(l_seq_g <= 0.01, axis=0) / N)
```

```
Out[6]: [<matplotlib.lines.Line2D at 0x7f65c828e3c8>]
```



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 $E_0[\ell(w_t) | q = g]$ is 1 for all t :

$$\begin{aligned} E_0[\ell(w_t) | q = g] &= \int \frac{f(w_t)}{g(w_t)} g(w_t) dw_t \\ &= \int f(w_t) dw_t \\ &= 1, \end{aligned}$$

which immediately implies

$$\begin{aligned} E_0[L(w^1) | q = g] &= E_0[\ell(w_1) | q = g] \\ &= 1. \end{aligned}$$

Because $L(w^t) = \ell(w_t)L(w^{t-1})$ and $\{w_t\}_{t=1}^t$ is an IID sequence, we have

$$\begin{aligned} E_0[L(w^t) | q = g] &= E_0[L(w^{t-1})\ell(w_t) | q = g] \\ &= E_0[L(w^{t-1})E[\ell(w_t) | q = g, w^{t-1}] | q = g] \\ &= E_0[L(w^{t-1})E[\ell(w_t) | q = g] | q = g] \\ &= E_0[L(w^{t-1}) | q = g] \end{aligned}$$

for any $t \geq 1$.

Mathematical induction implies $E_0[L(w^t) | q = g] = 1$ for all $t \geq 1$.

37.4.1 Peculiar Property of Likelihood Ratio Process

How can $E_0[L(w^t) | q = g] = 1$ possibly be true when most probability mass of the likelihood ratio process is piling up near 0 as $t \rightarrow +\infty$?

The answer has to be 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 $L(w^t)$ by averaging across these many paths at each t .

```
In [7]: l_arr_g = simulate(G_a, G_b, N=50000)
l_seq_g = np.cumprod(l_arr_g, axis=1)
```

It would be useful to use simulations to verify that unconditional means $E_0[L(w^t)]$ equal unity by averaging across sample paths.

But it would be too challenging for us to do that here simply by applying a standard Monte Carlo simulation approach.

The reason is that the distribution of $L(w^t)$ is extremely skewed for large values of t .

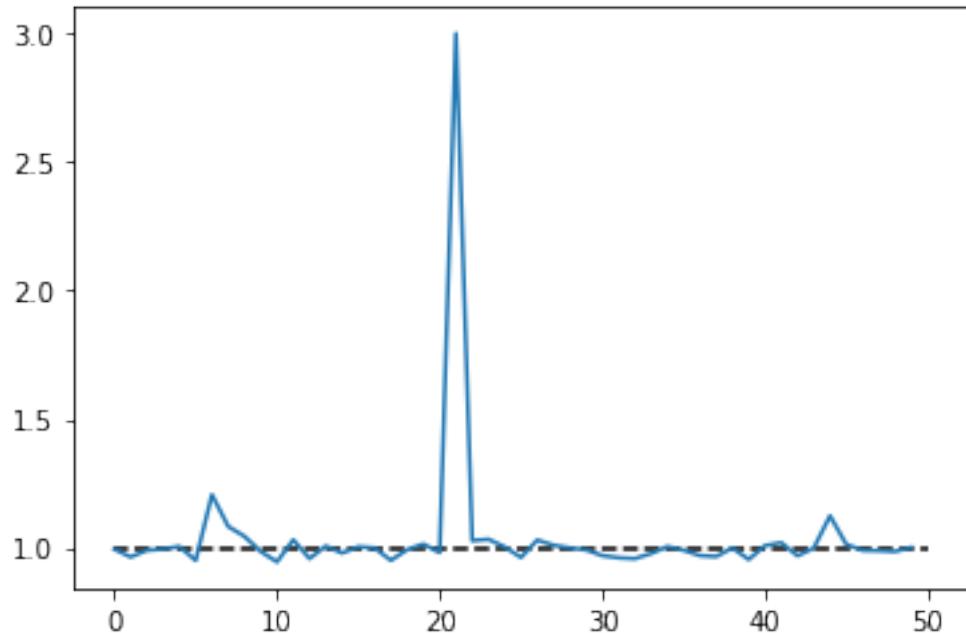
Because the probability density in the right tail is close to 0, it just takes too much computer time to sample enough points from the right tail.

Instead, the following code just illustrates that the unconditional means of $l(w_t)$ are 1.

While sample averages hover around their population means of 1, there is evidently quite a bit of variability.

```
In [8]: N, T = l_arr_g.shape
plt.plot(range(T), np.mean(l_arr_g, axis=0))
plt.hlines(1, 0, T, linestyle='--')
```

```
Out[8]: <matplotlib.collections.LineCollection at 0x7f65c8f65668>
```



37.5 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 $\ell(w_t)$ under density g is 1, its mean under the density f exceeds one.

To see this, we compute

$$\begin{aligned}
E_0 [\ell(w_t) \mid q = f] &= \int \frac{f(w_t)}{g(w_t)} f(w_t) dw_t \\
&= \int \frac{f(w_t)}{g(w_t)} \frac{f(w_t)}{g(w_t)} g(w_t) dw_t \\
&= \int \ell(w_t)^2 g(w_t) dw_t \\
&= E_0 [\ell(w_t)^2 \mid q = g] \\
&= E_0 [\ell(w_t) \mid q = g]^2 + \text{Var}(\ell(w_t) \mid q = g) \\
&> E_0 [\ell(w_t) \mid q = g]^2 = 1
\end{aligned}$$

This in turn implies that the unconditional mean of the likelihood ratio process $L(w^t)$ diverges toward $+\infty$.

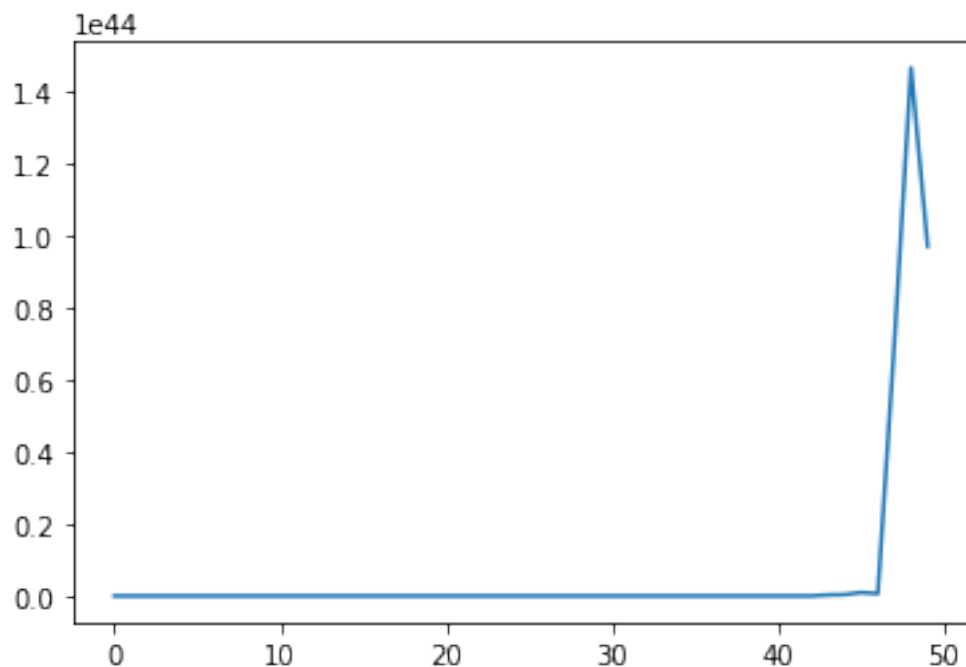
Simulations below confirm this conclusion.

Please note the scale of the y axis.

```
In [9]: l_arr_f = simulate(F_a, F_b, N=50000)
l_seq_f = np.cumprod(l_arr_f, axis=1)
```

```
In [10]: N, T = l_arr_f.shape
plt.plot(range(T), np.mean(l_seq_f, axis=0))
```

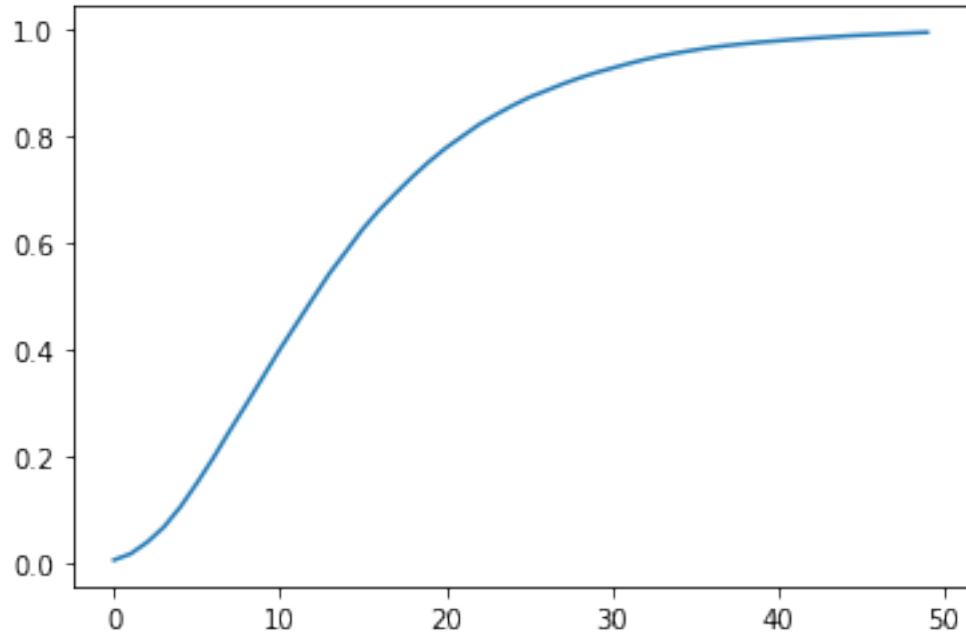
```
Out[10]: [<matplotlib.lines.Line2D at 0x7f65c8dfbeb8>]
```



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 $+\infty$.

In [11]: `plt.plot(range(T), np.sum(l_seq_f > 10000, axis=0) / N)`

Out[11]: [`<matplotlib.lines.Line2D at 0x7f65c8f087b8>`]



37.6 Likelihood Ratio Test

We now describe how to employ the machinery of Neyman and Pearson [?] to test the hypothesis that history w^t is generated by repeated IID draws from density g .

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:

- reject H_0 if $L(W^t) < c$,
- accept H_0 otherwise.

where c is a given discrimination threshold, to be chosen in a way we'll soon describe.

This test is *best* in the sense that it is a **uniformly most powerful** test.

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 detection (= power = 1 minus probability of Type II error):

$$1 - \beta \equiv \Pr \{L(w^t) < c \mid q = g\}$$

- Probability of false alarm (= significance level = probability of Type I error):

$$\alpha \equiv \Pr \{L(w^t) < c \mid q = f\}$$

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**.

To have made a good inference, 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.

In [12]: `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 to distinguish $q = f$ from $q = g$.

```
In [13]: fig, axs = plt.subplots(2, 2, figsize=(12, 8))
fig.suptitle('distribution of $\log(L(w^t))$ under f or g', fontsize=15)

for i, t in enumerate([1, 7, 14, 21]):
    nr = i // 2
    nc = 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)

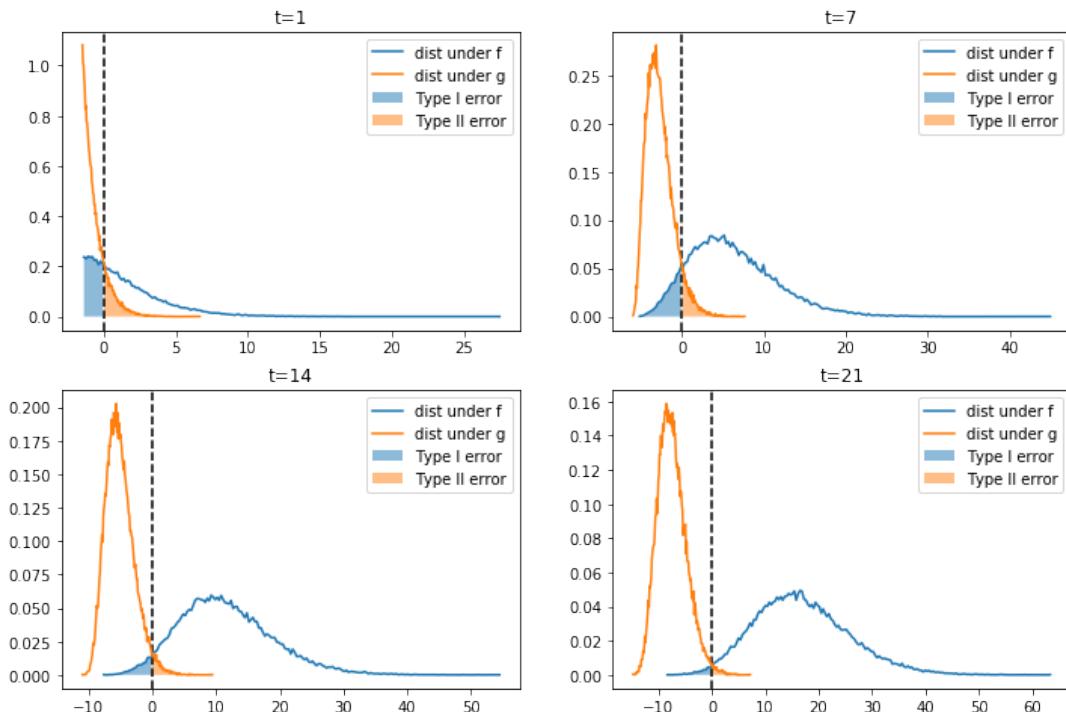
axs[nr, nc].plot(x_f[1:], hist_f, label="dist under f")
axs[nr, nc].plot(x_g[1:], hist_g, label="dist under g")

for i, (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 i == 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()

```

distribution of $\log(L(w^t))$ under f or g 

The graph below shows more clearly that, when we hold the threshold c fixed, the probability of detection monotonically increases with increases in t and that the probability of a false alarm monotonically decreases.

```

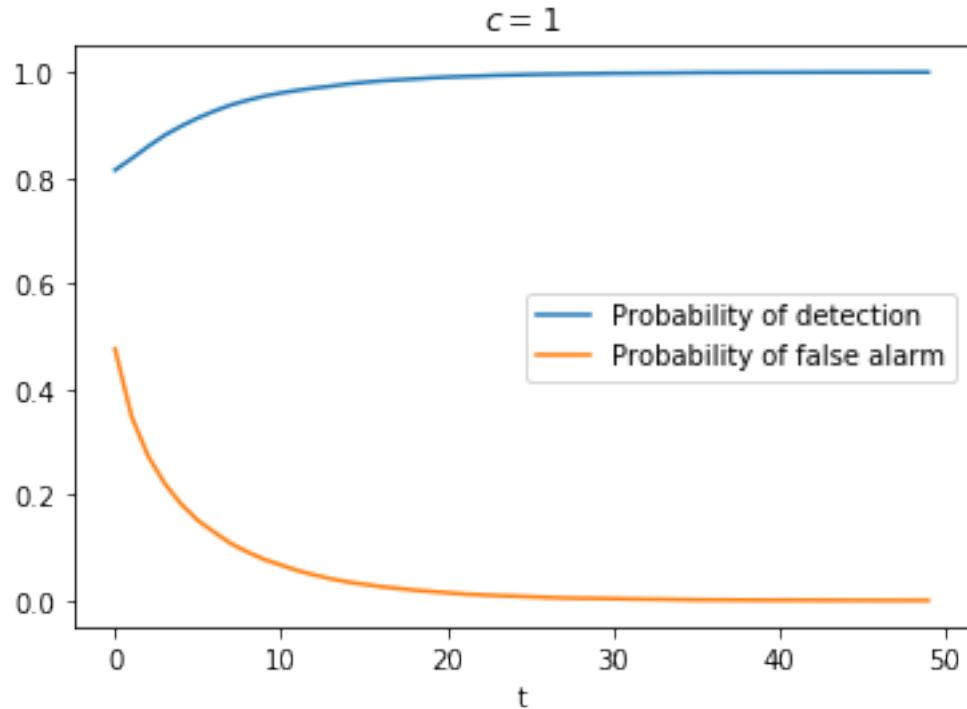
In [14]: PD = np.empty(T)
PFA = np.empty(T)

for t in range(T):
    PD[t] = np.sum(l_seq_g[:, t] < c) / N
    PFA[t] = np.sum(l_seq_f[:, t] < c) / N

plt.plot(range(T), PD, label="Probability of detection")

```

```
plt.plot(range(T), PFA, label="Probability of false alarm")
plt.xlabel("t")
plt.title("$c=1$")
plt.legend()
plt.show()
```



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 what is called a [receiver operating characteristic curve](#).

Below, we plot receiver operating characteristic curves for different sample sizes t .

```
In [15]: PFA = np.arange(0, 100, 1)

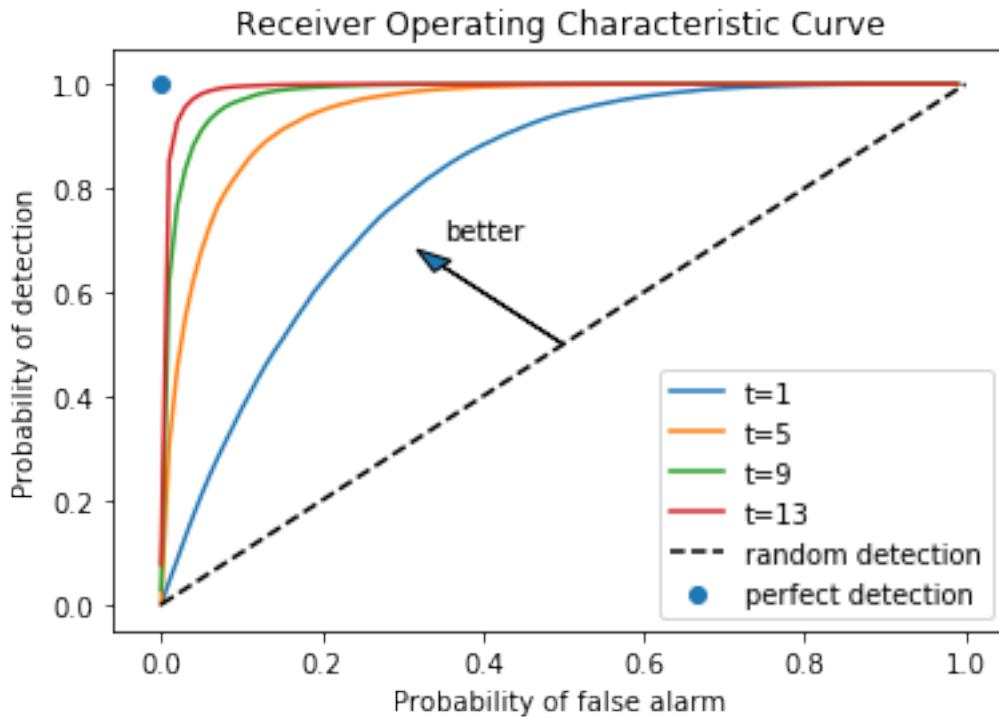
for t in range(1, 15, 4):
    percentile = np.percentile(l_seq_f[:, t], PFA)
    PD = [np.sum(l_seq_g[:, t] < p) / N for p in percentile]

    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("Receiver Operating Characteristic Curve")
plt.show()
```



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 .

As $t \rightarrow +\infty$, 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.

```
In [16]: PFA = 0.05
PD = np.empty(T)

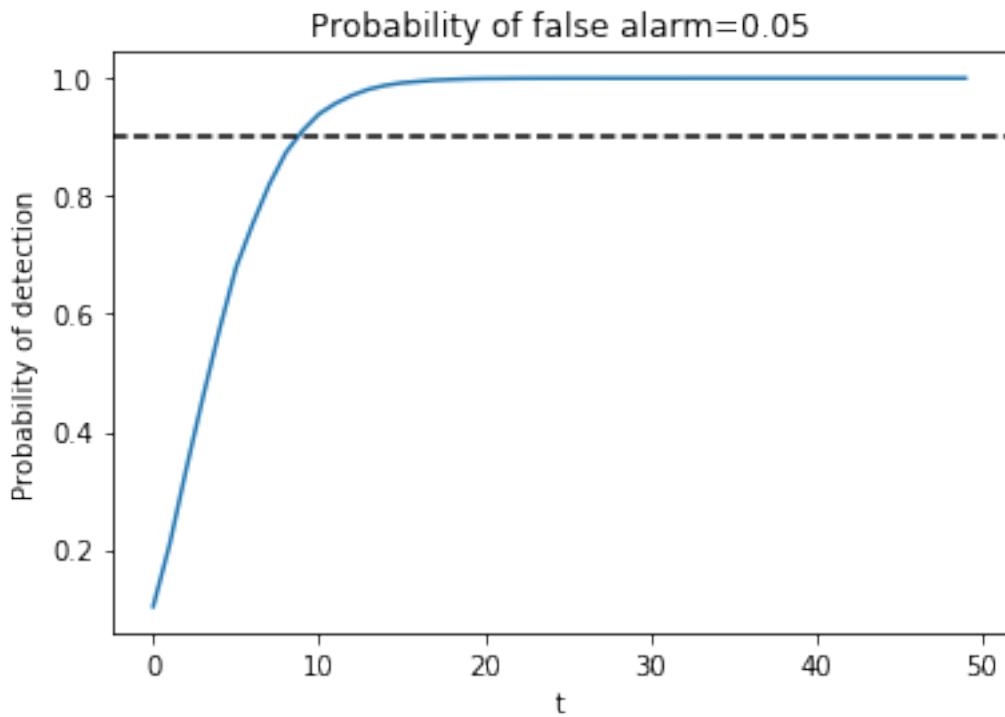
for t in range(T):
    c = np.percentile(l_seq_f[:, t], PFA * 100)
    PD[t] = np.sum(l_seq_g[:, t] < c) / N
```

```

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](#).

37.7 Kullback–Leibler divergence

Now let's consider a case in which neither g nor f generates the data.

Instead, a third distribution h does.

Let's watch how the cumulated likelihood ratios f/g behave when h governs the data.

A key tool here is called **Kullback–Leibler divergence**.

It is also called **relative entropy**.

It measures how one probability distribution differs from another.

In our application, we want to measure how f or g diverges from h

The two Kullback–Leibler divergences pertinent for us are K_f and K_g defined as

$$\begin{aligned}
 K_f &= E_h \left[\log \left(\frac{f(w)}{h(w)} \right) \frac{f(w)}{h(w)} \right] \\
 &= \int \log \left(\frac{f(w)}{h(w)} \right) \frac{f(w)}{h(w)} h(w) dw \\
 &= \int \log \left(\frac{f(w)}{h(w)} \right) f(w) dw
 \end{aligned}$$

$$\begin{aligned}
 K_g &= E_h \left[\log \left(\frac{g(w)}{h(w)} \right) \frac{g(w)}{h(w)} \right] \\
 &= \int \log \left(\frac{g(w)}{h(w)} \right) \frac{g(w)}{h(w)} h(w) dw \\
 &= \int \log \left(\frac{g(w)}{h(w)} \right) g(w) dw
 \end{aligned}$$

When $K_g < K_f$, g is closer to h than f is.

- In that case we'll find that $L(w^t) \rightarrow 0$.

When $K_g > K_f$, f is closer to h than g is.

- In that case we'll find that $L(w^t) \rightarrow +\infty$

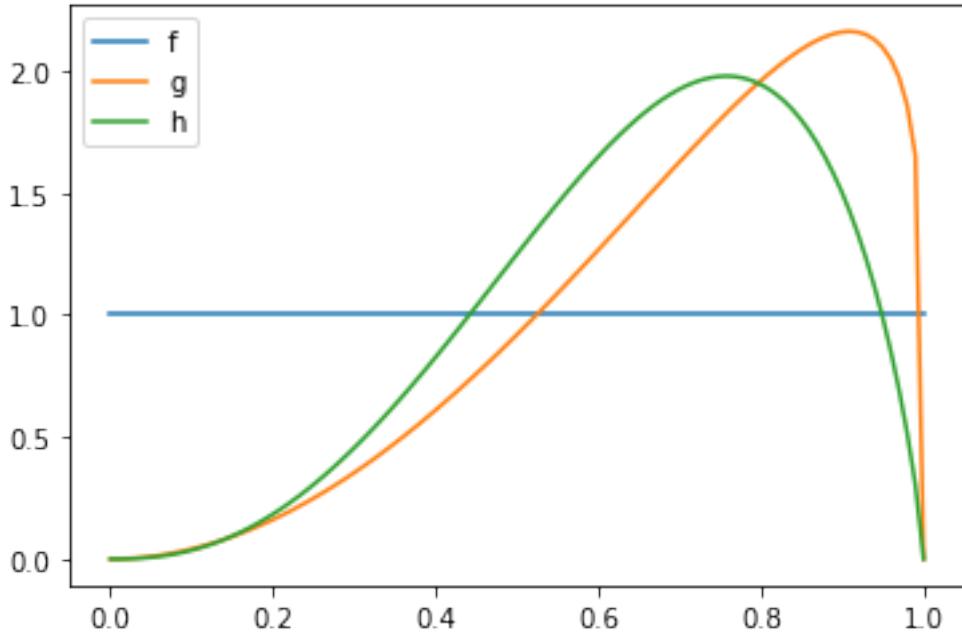
We'll now experiment with an h is also a beta distribution

We'll start by setting parameters G_a and G_b so that h is closer to g

In [17]: `H_a, H_b = 3.5, 1.8`

```
h = njit(lambda x: p(x, H_a, H_b))
```

In [18]: `x_range = np.linspace(0, 1, 100)`
`plt.plot(x_range, f(x_range), label='f')`
`plt.plot(x_range, g(x_range), label='g')`
`plt.plot(x_range, h(x_range), label='h')`
`plt.legend()`
`plt.show()`



Let's compute the Kullback–Leibler discrepancies by quadrature integration.

In [19]: `def KL_integrand(w, q, h):`

```
m = q(w) / h(w)
return np.log(m) * q(w)
```

In [20]: `def compute_KL(h, f, g):`

```
Kf, _ = quad(KL_integrand, 0, 1, args=(f, h))
Kg, _ = quad(KL_integrand, 0, 1, args=(g, h))

return Kf, Kg
```

In [21]: `Kf, Kg = compute_KL(h, f, g)`
`Kf, Kg`

Out[21]: `(0.7902536603660161, 0.08554075759988769)`

We have $K_g < K_f$.

Next, we can verify our conjecture about $L(w^t)$ by simulation.

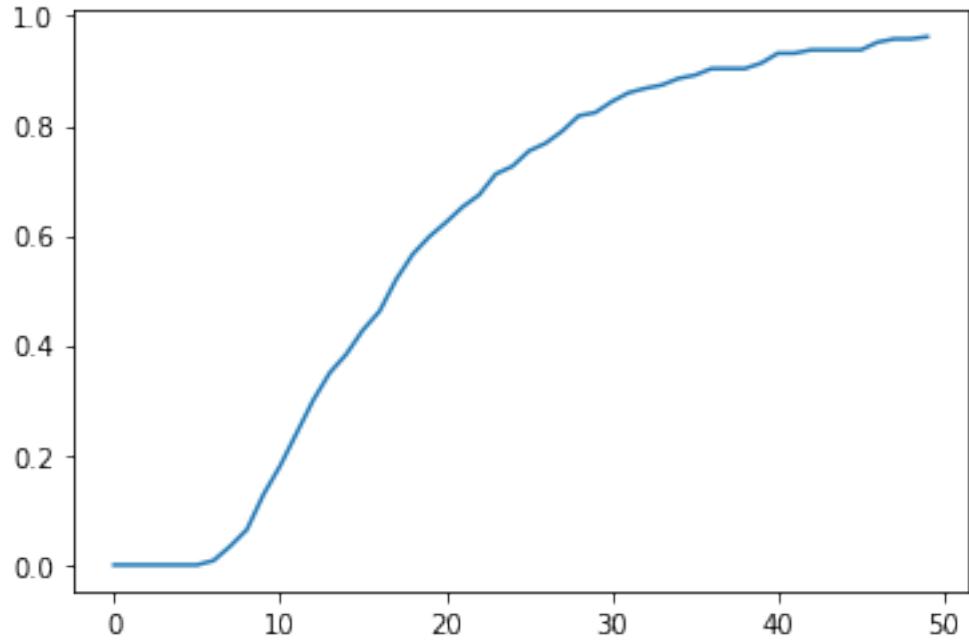
In [22]: `l_arr_h = simulate(H_a, H_b)`
`l_seq_h = np.cumprod(l_arr_h, axis=1)`

The figure below plots over time the fraction of paths $L(w^t)$ that fall in the interval $[0, 0.01]$.

Notice that it converges to 1 as expected when g is closer to h than f is.

In [23]: `N, T = l_arr_h.shape`
`plt.plot(range(T), np.sum(l_seq_h <= 0.01, axis=0) / N)`

Out[23]: [`<matplotlib.lines.Line2D at 0x7f65c8225438>`]



We can also try an h that is closer to f than is g so that now K_g is larger than K_f .

In [24]: `H_a, H_b = 1.2, 1.2`
`h = njit(lambda x: p(x, H_a, H_b))`

In [25]: `Kf, Kg = compute_KL(h, f, g)`
`Kf, Kg`

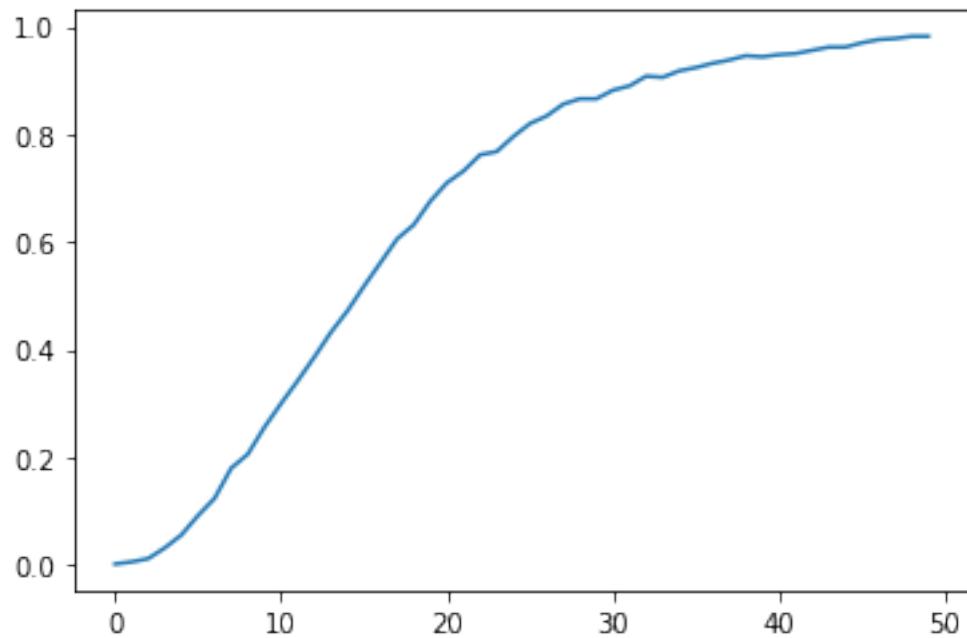
Out[25]: (`0.01239249754452668, 0.35377684280997646`)

In [26]: `l_arr_h = simulate(H_a, H_b)`
`l_seq_h = np.cumprod(l_arr_h, axis=1)`

Now probability mass of $L(w^t)$ falling above 10000 diverges to $+\infty$.

In [27]: `N, T = l_arr_h.shape`
`plt.plot(range(T), np.sum(l_seq_h > 10000, axis=0) / N)`

Out[27]: [`<matplotlib.lines.Line2D at 0x7f65c86367b8>`]



37.8 Sequels

Likelihood processes play an important role in Bayesian learning, as described in [this lecture](#) and as applied in [this lecture](#).

Likelihood ratio processes appear again in [this lecture](#), which contains another illustration of the **peculiar property** of likelihood ratio processes described above.

Chapter 38

A Problem that Stumped Milton Friedman

(and that Abraham Wald solved by inventing sequential analysis)

38.1 Contents

- Overview [38.2](#)
- Origin of the Problem [38.3](#)
- A Dynamic Programming Approach [38.4](#)
- Implementation [38.5](#)
- Analysis [38.6](#)
- Comparison with Neyman-Pearson Formulation [38.7](#)
- Sequels [38.8](#)

In addition to what's in Anaconda, this lecture will need the following libraries:

```
In [1]: !pip install quantecon  
!pip install interpolation
```

38.2 Overview

This lecture describes a statistical decision problem encountered by 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 [\[109\]](#) to formulate **sequential analysis**, an approach to statistical decision problems intimately related to dynamic programming.

In this lecture, we apply dynamic programming algorithms to Friedman and Wallis and Wald's problem.

Key ideas in play will be:

- Bayes' Law
- Dynamic programming
- 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 is 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**

We'll begin with some imports:

```
In [2]: import numpy as np
import matplotlib.pyplot as plt
from numba import jit, prange, float64, int64
from numba.experimental import jitclass
from interpolation import interp
from math import gamma
```

This lecture uses ideas studied in [this lecture](#), [this lecture](#). and [this lecture](#).

38.3 Origin of the Problem

On pages 137-139 of his 1998 book *Two Lucky People* with Rose Friedman [39], 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.

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. Schyler, the captain objected that such a test, to quote from Allen's account, may prove wasteful. If a wise and seasoned ordnance officer like Schyler 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 struggled with the problem but, after realizing that they were not able to solve it, described the problem to Abraham Wald.

That started Wald on the path that led him to *Sequential Analysis* [109].

We'll formulate the problem using dynamic programming.

38.4 A Dynamic Programming Approach

The following presentation of the problem closely follows Dmitri Berskekas's treatment in **Dynamic Programming and Stochastic Control** [12].

A decision-maker observes 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 and that the observer has something to learn, even though he knows both f_0 and f_1 .

The decision maker chooses a number of draws (i.e., random samples from the unknown distribution) and uses them to decide which of the two distributions is generating outcomes.

He starts with prior

$$\pi_{-1} = \mathbb{P}\{f = f_0 \mid \text{no observations}\} \in (0, 1)$$

After observing $k + 1$ observations z_k, z_{k-1}, \dots, z_0 , he updates this value to

$$\pi_k = \mathbb{P}\{f = f_0 \mid z_k, z_{k-1}, \dots, z_0\}$$

which is calculated recursively by applying Bayes' law:

$$\pi_{k+1} = \frac{\pi_k f_0(z_{k+1})}{\pi_k f_0(z_{k+1}) + (1 - \pi_k) f_1(z_{k+1})}, \quad k = -1, 0, 1, \dots$$

After observing z_k, z_{k-1}, \dots, z_0 , the decision-maker believes that z_{k+1} has probability distribution

$$f_{\pi_k}(v) = \pi_k f_0(v) + (1 - \pi_k) f_1(v),$$

which is a mixture of distributions f_0 and f_1 , with the weight on f_0 being the posterior probability that $f = f_0$ Section ??.

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

$$f(z; a, b) = \frac{\Gamma(a+b)z^{a-1}(1-z)^{b-1}}{\Gamma(a)\Gamma(b)} \quad \text{where } \Gamma(t) := \int_0^\infty x^{t-1} e^{-x} dx$$

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

```
In [3]: @jit(nopython=True)
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)

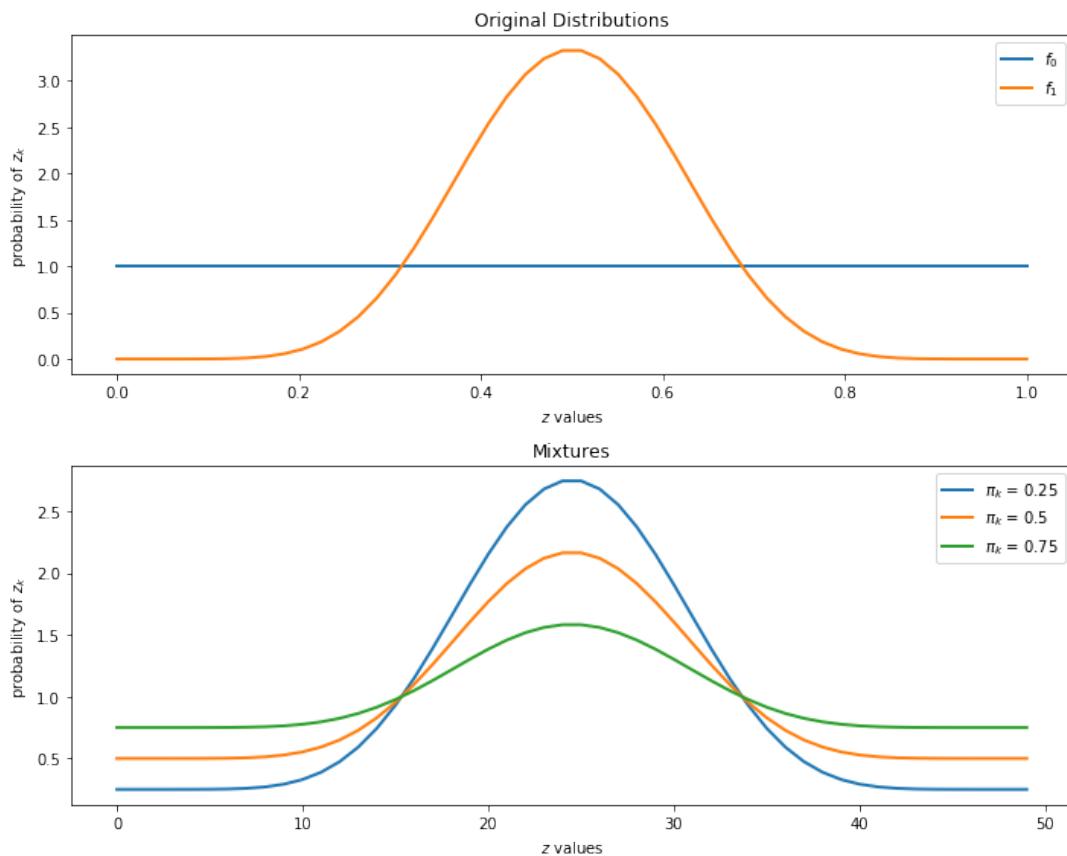
fig, axes = plt.subplots(2, figsize=(10, 8))

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 pi in 0.25, 0.5, 0.75:
    y = pi * f0(grid) + (1 - pi) * f1(grid)
    axes[1].plot(y, lw=2, label=f"$\pi_k = \{\pi\}$")

for ax in axes:
    ax.legend()
    ax.set(xlabel="z values", ylabel="probability of z_k")

plt.tight_layout()
plt.show()
```



38.4.1 Losses and Costs

After observing z_k, z_{k-1}, \dots, 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

38.4.2 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.

38.4.3 Intuition

Let's try to guess what an optimal decision rule might look like before we go further.

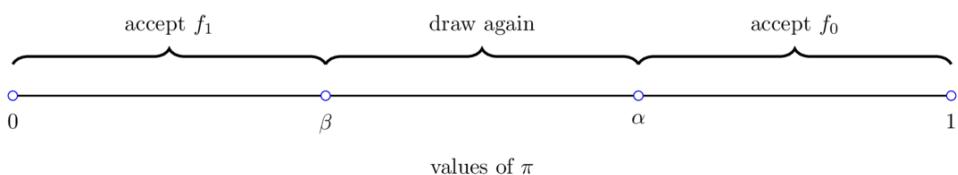
Suppose at some given point in time that π is close to 1.

Then our prior beliefs and the evidence so far point strongly to $f = f_0$.

If, on the other hand, π is close to 0, then $f = f_1$ is strongly favored.

Finally, if π is in the middle of the interval $[0, 1]$, then we have little information in either direction.

This reasoning suggests a decision rule such as the one shown in the figure



As we'll see, this is indeed the correct form of the decision rule.

The key problem is to determine the threshold values α, β , which will depend on the parameters listed 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 α or β .

38.4.4 A Bellman Equation

Let $J(\pi)$ be the total loss for a decision-maker with current belief π who chooses optimally.

With some thought, you will agree that J should satisfy the Bellman equation

$$J(\pi) = \min \{(1 - \pi)L_0, \pi L_1, c + \mathbb{E}[J(\pi')]\} \quad (1)$$

where π' is the random variable defined by Bayes' Law

$$\pi' = \kappa(z', \pi) = \frac{\pi f_0(z')}{\pi f_0(z') + (1 - \pi)f_1(z')}$$

when π is fixed and z' is drawn from the current best guess, which is the distribution f defined by

$$f_\pi(v) = \pi f_0(v) + (1 - \pi)f_1(v)$$

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

$$J(\pi) = \min \{(1 - \pi)L_0, \pi L_1, h(\pi)\} \quad (2)$$

where $\pi \in [0, 1]$ and

- $(1 - \pi)L_0$ is the expected loss associated with accepting f_0 (i.e., the cost of making a type II error).
- πL_1 is the expected loss associated with accepting f_1 (i.e., the cost of making a type I error).
- $h(\pi) := c + \mathbb{E}[J(\pi')]$ the continuation value; i.e., the expected cost associated with drawing one more z .

The optimal decision rule is characterized by two numbers $\alpha, \beta \in (0, 1) \times (0, 1)$ that satisfy

$$(1 - \pi)L_0 < \min\{\pi L_1, c + \mathbb{E}[J(\pi')]\} \text{ if } \pi \geq \alpha$$

and

$$\pi L_1 < \min\{(1 - \pi)L_0, c + \mathbb{E}[J(\pi')]\} \text{ if } \pi \leq \beta$$

The optimal decision rule is then

accept $f = f_0$ if $\pi \geq \alpha$
 accept $f = f_1$ if $\pi \leq \beta$
 draw another z if $\beta \leq \pi \leq \alpha$

Our aim is to compute the value function J , and from it the associated cutoffs α and β .

To make our computations simpler, using (2), we can write the continuation value $h(\pi)$ as

$$\begin{aligned} h(\pi) &= c + \mathbb{E}[J(\pi')] \\ &= c + \mathbb{E}_{\pi'} \min\{(1 - \pi')L_0, \pi'L_1, h(\pi')\} \\ &= c + \int \min\{(1 - \kappa(z', \pi))L_0, \kappa(z', \pi)L_1, h(\kappa(z', \pi))\} f_\pi(z') dz' \end{aligned} \quad (3)$$

The equality

$$h(\pi) = c + \int \min\{(1 - \kappa(z', \pi))L_0, \kappa(z', \pi)L_1, h(\kappa(z', \pi))\} f_\pi(z') dz' \quad (4)$$

can be understood as a functional equation, where h is the unknown.

Using the functional equation, (4), for the continuation value, we can back out optimal choices using the right side of (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

$$Qh(\pi) = c + \int \min\{(1 - \kappa(z', \pi))L_0, \kappa(z', \pi)L_1, h(\kappa(z', \pi))\} f_\pi(z') dz'$$

38.5 Implementation

First, we will construct a `jitclass` to store the parameters of the model

```
In [4]: 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[:])]
```

```
In [5]: @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 x(self, z, π):
    """
    Updates π using Bayes' rule and the current observation z
    """
    f0, f1 = self.f0, self.f1
    π_f0, π_f1 = π * f0(z), (1 - π) * f1(z)
    π_new = π_f0 / (π_f0 + π_f1)

    return π_new

```

As in the [optimal growth lecture](#), 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.

```
In [6]: @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

    x = wf.x

    h_new = np.empty_like(π_grid)
```

```

h_func = lambda p: interp(pi_grid, h, p)

for i in prange(len(pi_grid)):
    pi = pi_grid[i]

        # Find the expected value of J by integrating over z
    integral_f0, integral_f1 = 0, 0
    for m in range(mc_size):
        pi_0 = x(z0[m], pi) # Draw z from f0 and update pi
        integral_f0 += min((1 - pi_0) * L0, pi_0 * L1, h_func(pi_0))

        pi_1 = x(z1[m], pi) # Draw z from f1 and update pi
        integral_f1 += min((1 - pi_1) * L0, pi_1 * L1, h_func(pi_1))

    integral = (pi * integral_f0 + (1 - pi) * integral_f1) / mc_size

    h_new[i] = c + integral

return h_new

```

To solve the model, we will iterate using Q to find the fixed point

```

In [7]: @jit(nopython=True)
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.pi_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 i == max_iter:
        print("Failed to converge!")

    return h_new

```

38.6 Analysis

Let's inspect outcomes.

We will be using the default parameterization with distributions like so

```

In [8]: wf = WaldFriedman()

fig, ax = plt.subplots(figsize=(10, 6))

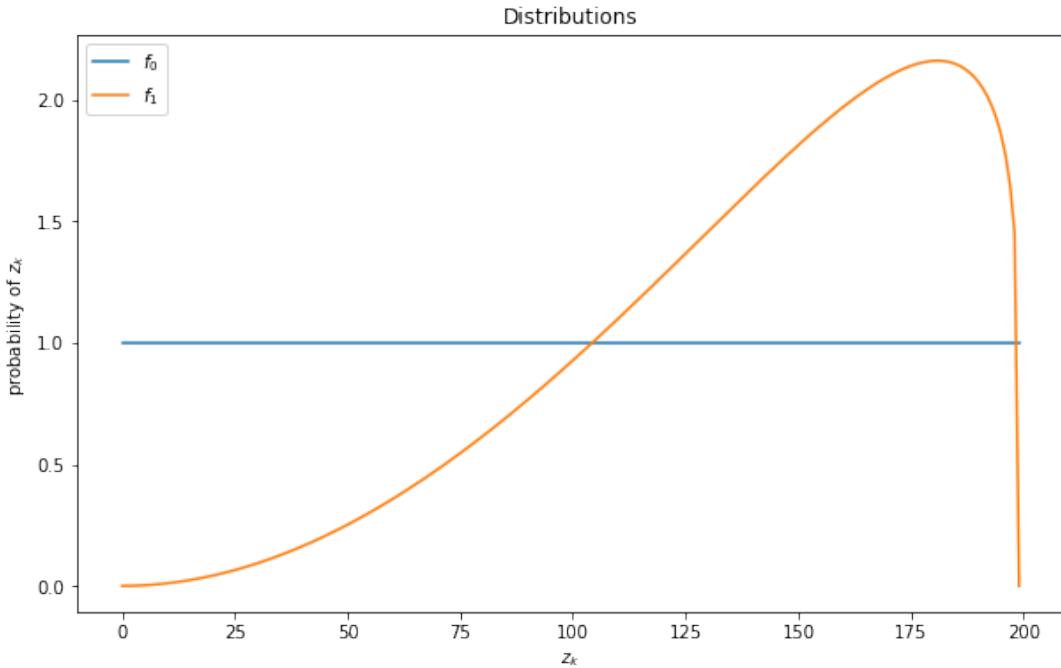
```

```

ax.plot(wf.f0(wf.pi_grid), label="$f_0$")
ax.plot(wf.f1(wf.pi_grid), label="$f_1$")
ax.set(ylabel="probability of $z_k$", xlabel="$z_k$", title="Distributions")
ax.legend()

plt.show()

```



38.6.1 Value Function

To solve the model, we will call our `solve_model` function

```
In [9]: h_star = solve_model(wf)      # Solve the model

<ipython-input-7-0c1f615d23a7>:15: NumbaWarning: The TBB threading layer
requires TBB version 2019.5 or later i.e., TBB_INTERFACE_VERSION >= 11005. Found
TBB_INTERFACE_VERSION = 11004. The TBB threading layer is disabled.
h_new = Q(h, wf)
```

We will also set up a function to compute the cutoffs α and β and plot these on our value function plot

```
In [10]: @jit(nopython=True)
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
    stopping.
    """
    alpha = None
    beta = None
```

```

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)
β = π_grid[np.searchsorted(
    payoff_f1 - np.minimum(h, payoff_f0),
    1e-10)
- 1]
α = π_grid[np.searchsorted(
    np.minimum(h, payoff_f1) - payoff_f0,
    1e-10)
- 1]

return (β, α)

β, α = 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')

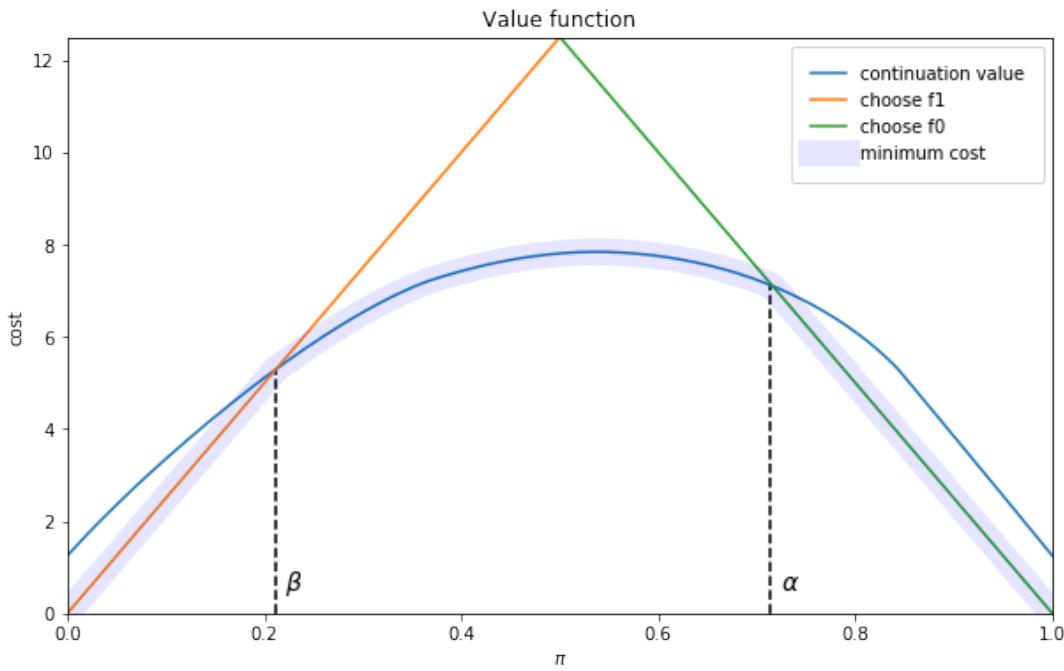
ax.annotate(r"$\beta$",
            xy=(β + 0.01, 0.5), fontsize=14)
ax.annotate(r"$\alpha$",
            xy=(α + 0.01, 0.5), fontsize=14)

plt.vlines(β, 0, β * wf.L0, linestyle="--")
plt.vlines(α, 0, (1 - α) * wf.L1, linestyle="--")

ax.set(xlim=(0, 1), ylim=(0, 0.5 * max(wf.L0, wf.L1)), ylabel="cost",
       xlabel="$\pi$",
       title="Value function")

plt.legend(borderpad=1.1)
plt.show()

```



The value function equals πL_1 for $\pi \leq \beta$, and $(1 - \pi)L_0$ for $\pi \geq \alpha$.

The slopes of the two linear pieces of the value function are determined by L_1 and $-L_0$.

The value function is smooth in the interior region, where the posterior probability assigned to f_0 is in the indecisive region $\pi \in (\beta, \alpha)$.

The decision-maker continues to sample until the probability that he attaches to model f_0 falls below β or above α .

38.6.2 Simulations

The next figure shows the outcomes of 500 simulations of the decision process.

On the left is a histogram of the stopping times, which equal 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

```
In [11]: 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
    x = wf.x
```

```

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
β, α = 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:
    # Maybe should specify which distribution is correct one so that
    # the draws come from the "right" distribution
    z = f_rvs()
    t = t + 1
    π = ς(z, π)
    if π < β:
        decision_made = True
        decision = 1
    elif π > α:
        decision_made = True
        decision = 0

    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):
    h_star = solve_model(wf)
    ndraws = 500
    cdist, tdist = stopping_dist(wf, h_star, ndraws)

    fig, ax = plt.subplots(1, 2, figsize=(16, 5))

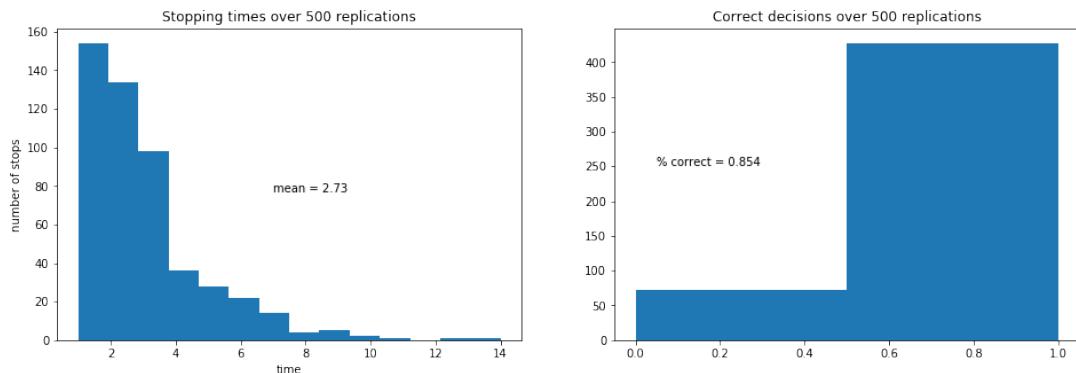
    ax[0].hist(tdist, bins=np.max(tdist))
    ax[0].set_title(f"Stopping times over {ndraws} replications")
    ax[0].set(xlabel="time", ylabel="number of stops")
    ax[0].annotate(f"mean = {np.mean(tdist)}", xy=(max(tdist) / 2,
                                                   max(np.histogram(tdist, 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)

```



38.6.3 Comparative Statics

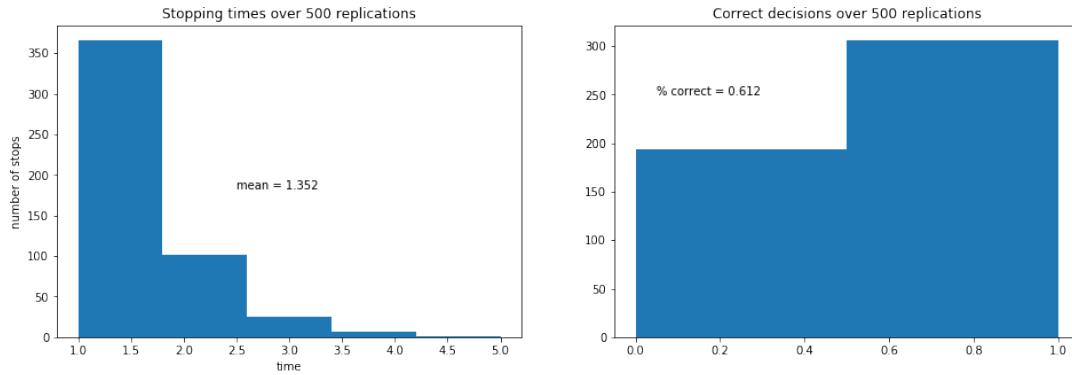
Now let's consider the following exercise.

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?

```
In [12]: 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.

38.6.4 A Notebook Implementation

To facilitate comparative statics, we provide a [Jupyter notebook](#) that generates the same plots, but with sliders.

With these sliders, you can adjust parameters and immediately observe

- 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 piece-wise 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.

38.7 Comparison with Neyman-Pearson Formulation

For several reasons, it is useful to describe the theory underlying the test that Navy Captain G. S. Schuyler had been told to use and that led him to approach Milton Friedman and Allan Wallis to convey his conjecture that superior practical procedures existed.

Evidently, the Navy had told Captain Schuyler to use what it knew to be a state-of-the-art Neyman-Pearson test.

We'll rely on Abraham Wald's [109] elegant summary of Neyman-Pearson theory.

For our purposes, 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 the probabilities α and β defined in the Neyman-Pearson theory

Recall that in the sequential analytic formulation above, that

- The sample size n is not fixed but rather an object to be chosen; technically n is a random variable.
- The parameters β and α characterize cut-off rules used to determine n as a random variable.
- Laws of large numbers make no appearances in the sequential construction.

In chapter 1 of **Sequential Analysis** [109] 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 $M \rightarrow \infty$ (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 possible 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_0 , (2) to reject the hypothesis H_0 , (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.

Footnotes

[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 [67] chapter 11, for discussions of exchangeability.

38.8 Sequels

We'll dig deeper into some of the ideas used here in the following lectures:

- [this lecture](#) discusses the key concept of **exchangeability** that rationalizes statistical learning
- [this lecture](#) describes **likelihood ratio processes** and their role in frequentist and Bayesian statistical theories
- [this lecture](#) discusses the role of likelihood ratio processes in **Bayesian learning**
- [this lecture](#) returns to the subject of this lecture and studies whether the Captain's hunch that the (frequentist) decision rule that the Navy had ordered him to use can be expected to be better or worse than the rule sequential rule that Abraham Wald designed

Chapter 39

Exchangeability and Bayesian Updating

39.1 Contents

- Overview 39.2
- Independently and Identically Distributed 39.3
- A Setting in Which Past Observations Are Informative 39.4
- Relationship Between IID and Exchangeable 39.5
- Exchangeability 39.6
- Bayes' Law 39.7
- More Details about Bayesian Updating 39.8
- Appendix 39.9
- Sequels 39.10

39.2 Overview

This lecture studies an example of learning via Bayes' Law.

We touch on foundations of Bayesian statistical inference invented by Bruno DeFinetti [25].

The relevance of DeFinetti's work for economists is presented forcefully in chapter 11 of [67] by David Kreps.

The example that we study in this lecture is a key component of [this lecture](#) that augments the [classic](#) job search model of McCall [80] 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 the 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

Understanding the distinction between these concepts is essential for appreciating how

Bayesian updating works in our example.

You can read about exchangeability [here](#).

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:

```
In [1]: from numba import njit, vectorize
from math import gamma
import scipy.optimize as op
from scipy.integrate import quad
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

39.3 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, **independently** and **identically** distributed.

A sequence W_0, W_1, \dots 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, \dots is **independently and identically distributed** if in addition the marginal density of W_t is the same for all $t = 0, 1, \dots$

For example, let $p(W_0, W_1, \dots)$ be the **joint density** of the sequence and let $p(W_t)$ be the **marginal density** for a particular W_t for all $t = 0, 1, \dots$

Then the joint density of the sequence W_0, W_1, \dots is IID if

$$p(W_0, W_1, \dots) = p(W_0)p(W_1) \cdots$$

so that the joint density is the product of a sequence of identical marginal densities.

39.3.1 IID Means Past Observations Don't Tell Us Anything About Future Observations

If a sequence of random variables is IID, past information provides no information about future realizations.

In this sense, there is **nothing to learn** about the future from the past.

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

$$p(W_T, W_{T-1}, \dots, W_1, W_0)$$

Using the laws of probability, we can always factor such a joint density into a product of conditional densities:

$$\begin{aligned} p(W_T, W_{T-1}, \dots, W_1, W_0) &= p(W_T|W_{T-1}, \dots, W_0)p(W_{T-1}|W_{T-2}, \dots, W_0) \cdots \\ &\quad \cdots p(W_1|W_0)p(W_0) \end{aligned}$$

In general,

$$p(W_t|W_{t-1}, \dots, W_0) \neq p(W_t)$$

which states that the **conditional density** on the left side does not equal the **marginal density** on the right side.

In the special IID case,

$$p(W_t|W_{t-1}, \dots, W_0) = p(W_t)$$

and partial history W_{t-1}, \dots, 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 data.

In the general case, there is something to learn from past data.

We turn next to an instance of this general case.

Please keep your eye out for **what** there is to learn from past data.

39.4 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 — with densities f and g 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** — and thereafter at each time $t \geq 0$ draws a random W 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* the probability that the data are generated as draws from F is either 0 or 1.

We now drop into this setting a decision maker who knows F and G and that nature picked one of them once and for all and then drew an IID sequence of draws from that distribution.

But our decision maker does not know which of the two distributions nature selected.

The decision maker summarizes 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
- summarizing 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 $t \geq 0$ has observed the partial history w_t, w_{t-1}, \dots, w_0 of draws from the appropriate joint density of the partial history

But what do we mean by the *appropriate joint distribution*?

We'll discuss that next and in the process describe the concept of **exchangeability**.

39.5 Relationship Between IID and Exchangeable

Conditional on nature selecting F , the joint density of the sequence W_0, W_1, \dots is

$$f(W_0)f(W_1)\dots$$

Conditional on nature selecting G , the joint density of the sequence W_0, W_1, \dots is

$$g(W_0)g(W_1)\dots$$

Notice that **conditional on nature having selected F** , the sequence W_0, W_1, \dots is independently and identically distributed.

Furthermore, **conditional on nature having selected G** , the sequence W_0, W_1, \dots is also independently and identically distributed.

But what about the unconditional distribution?

The unconditional distribution of W_0, W_1, \dots is evidently

$$h(W_0, W_1, \dots) \equiv \tilde{\pi}[f(W_0)f(W_1)\dots] + (1 - \tilde{\pi})[g(W_0)g(W_1)\dots] \quad (1)$$

Under the unconditional distribution $h(W_0, W_1, \dots)$, the sequence W_0, W_1, \dots 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 the realization w_0 contains information about w_1 .

So there is something to learn.

But what and how?

39.6 Exchangeability

While the sequence W_0, W_1, \dots is not IID, it can be verified that it is **exchangeable**, which means that

$$h(w_0, w_1) = h(w_1, w_0)$$

and so on.

More generally, a sequence of random variables is said to be **exchangeable** if the joint probability distribution for the sequence does not change when the positions in the sequence in which finitely many of the random variables appear are altered.

Equation (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.

For a Bayesian statistician, the mixing parameter $\tilde{\pi} \in (0, 1)$ has a special interpretation as a **prior probability** that nature selected probability distribution F .

DeFinetti [25] established a related representation of an exchangeable process created by mixing sequences of IID Bernoulli random variables with parameters θ and mixing probability $\pi(\theta)$ for a density $\pi(\theta)$ that a Bayesian statistician would interpret as a prior over the unknown Bernoulli parameter θ .

39.7 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 about $\tilde{\pi}$.

The answer to the *how* question is to use Bayes' Law.

Another way to say *use Bayes' Law* is to say *compute an appropriate conditional distribution*.

Let's dive into Bayes' Law in this context.

Let q represent the distribution that nature actually draws from w from and let

$$\pi = \mathbb{P}\{q = f\}$$

where we regard π as the decision maker's **subjective probability** (also called a **personal probability**).

Suppose that at $t \geq 0$, the decision maker has observed a history $w^t \equiv [w_t, w_{t-1}, \dots, w_0]$.

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

$$\pi_{t+1} = \frac{\pi_t f(w_{t+1})}{\pi_t f(w_{t+1}) + (1 - \pi_t) g(w_{t+1})} \quad (2)$$

The last expression follows from Bayes' rule, which tells us that

$$\mathbb{P}\{q = f \mid W = w\} = \frac{\mathbb{P}\{W = w \mid q = f\} \mathbb{P}\{q = f\}}{\mathbb{P}\{W = w\}} \quad \text{and} \quad \mathbb{P}\{W = w\} = \sum_{\omega \in \{f, g\}} \mathbb{P}\{W = w \mid q = \omega\} \mathbb{P}\{q = \omega\}$$

39.8 More Details about Bayesian Updating

Let's stare at and rearrange Bayes' Law as represented in equation (2) with the aim of understanding how the **posterior** π_{t+1} is influenced by the **prior** π_t and the **likelihood ratio**

$$l(w) = \frac{f(w)}{g(w)}$$

It is convenient for us to rewrite the updating rule (2) as

$$\pi_{t+1} = \frac{\pi_t f(w_{t+1})}{\pi_t f(w_{t+1}) + (1 - \pi_t) g(w_{t+1})} = \frac{\pi_t \frac{f(w_{t+1})}{g(w_{t+1})}}{\pi_t \frac{f(w_{t+1})}{g(w_{t+1})} + (1 - \pi_t)} = \frac{\pi_t l(w_{t+1})}{\pi_t l(w_{t+1}) + (1 - \pi_t)}$$

This implies that

$$\frac{\pi_{t+1}}{\pi_t} = \frac{l(w_{t+1})}{\pi_t l(w_{t+1}) + (1 - \pi_t)} \begin{cases} > 1 & \text{if } l(w_{t+1}) > 1 \\ \leq 1 & \text{if } l(w_{t+1}) \leq 1 \end{cases} \quad (3)$$

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 than one, the observation w_{t+1} nudges π downward.

Representation (3) is the foundation of the graphs that we'll use to display the dynamics of $\{\pi_t\}_{t=0}^\infty$ that are induced by Bayes' Law.

We'll plot $l(w)$ 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 .

```
In [2]: @vectorize
def p(x, a, b):
    "The general beta distribution function."
    r = gamma(a + b) / (gamma(a) * gamma(b))
    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  $\pi$ ,
    given the parameters which specify  $F$  and  $G$  distributions.
    """

    f = njit(lambda x: p(x, F_a, F_b))
    g = njit(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)
    pi_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)

w = np.arange(0.01, 0.99, 0.08)
Pi = np.arange(0.01, 0.99, 0.08)

Delta_W = np.zeros((len(w), len(Pi)))
Delta_Pi = np.empty((len(w), len(Pi)))
for i, w in enumerate(w):
    for j, pi in enumerate(Pi):
        l_w = l(w)
        Delta_Pi[i, j] = pi * (l_w / (pi * l_w + 1 - pi) - 1)

q = ax3.quiver(Pi, w, Delta_Pi, Delta_W, scale=2, color='r', alpha=0.8)

ax3.fill_between(pi_grid, 0, roots[0], color='blue', alpha=0.15)
ax3.fill_between(pi_grid, roots[0], roots[1], color='green', alpha=0.15)
ax3.fill_between(pi_grid, roots[1], w_max, color='blue', alpha=0.15)
ax3.hlines(roots, 0., 1., linestyle="--")
ax3.set(xlabel='$\pi$', ylabel='$w$')
ax3.grid()

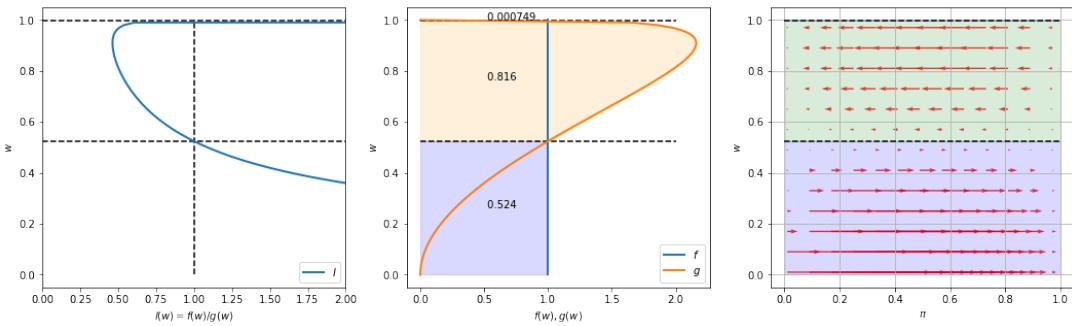
plt.show()

```

Now we'll create a group of graphs designed to illustrate the dynamics induced by Bayes' Law.

We'll begin with the default values of various objects, then change them in a subsequent example.

In [3]: `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 $l(w)$ on the coordinate axis against w on the ordinate axis.

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.

Notice how the length of the arrows, which show the magnitude of the force from Bayes' Law impelling π to change, depends on both the prior probability π on the ordinate axis and the evidence in the form of the current draw of w on the coordinate axis.

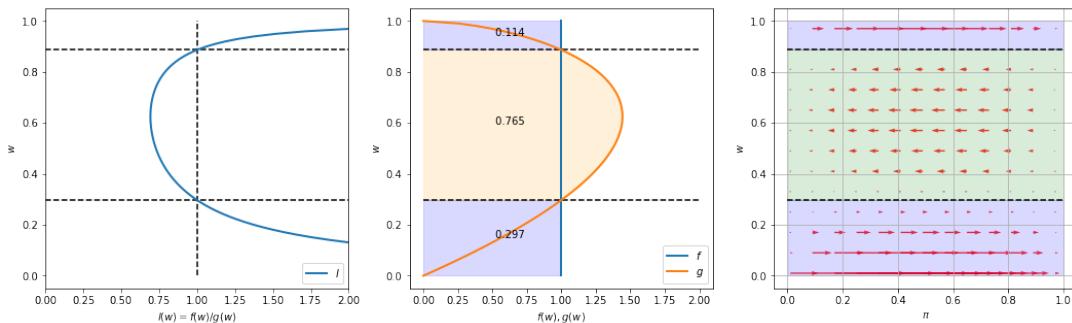
The fractions in the colored areas of the middle graphs are probabilities under F and G , respectively, that realizations of w fall into the interval that updates the belief π in a correct direction (i.e., toward 0 when G is the true distribution, and towards 1 when F is the true distribution).

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 $G_a = 2, G_b = 1.6$.

```
In [4]: 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.

39.9 Appendix

39.9.1 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:

- that nature permanently draws from F
- that nature permanently draws from G

Outcomes depend on a peculiar property of likelihood ratio processes that are discussed in [this lecture](#).

To do this, we create some Python code.

In [5]: `def function_factory(F_a=1, F_b=1, G_a=3, G_b=1.2):`

```

# define f and g
f = njit(lambda x: p(x, F_a, F_b))
g = njit(lambda x: p(x, G_a, G_b))

@njit
def update(a, b, π):
    "Update π by drawing from beta distribution with parameters a and b"

    # Draw
    w = np.random.beta(a, b)

    # Update belief
    π = 1 / (1 + ((1 - π) * g(w)) / (π * f(w)))

    return π

@njit
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

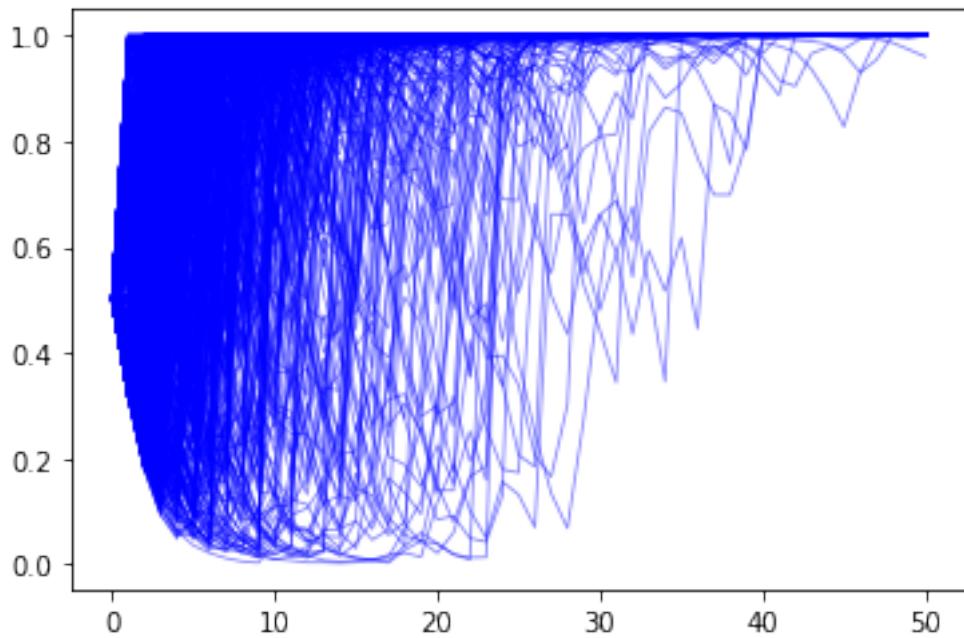
```

In [6]: `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 the initial prior $\pi_{-1} = .5$.

In [7]: `T = 50`

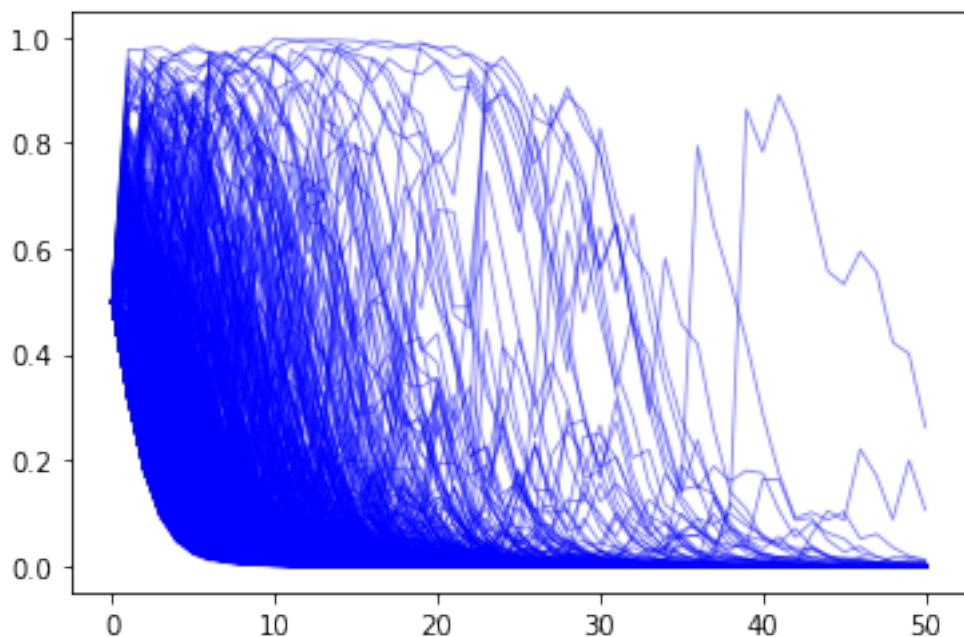
```
In [8]: # when nature selects F
pi_paths_F = simulate(a=1, b=1, T=T, N=1000)
```



In the above graph we observe that for most paths $\pi_t \rightarrow 1$. 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 $\pi_{-1} = .5$.

```
In [9]: # when nature selects G
pi_paths_G = simulate(a=3, b=1.2, T=T, N=1000)
```



In the above graph we observe that now most paths $\pi_t \rightarrow 0$.

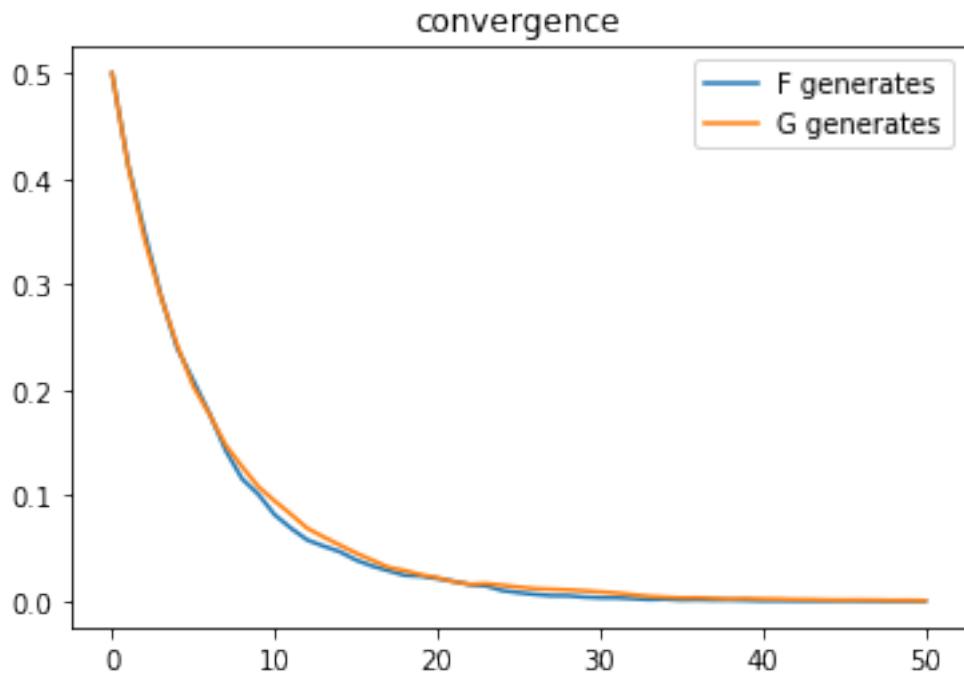
39.9.2 Rates of convergence

We study rates of convergence of π_t to 1 when nature generates the data as IID draws from F and of π_t to 0 when nature generates the data as 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 .

```
In [10]: plt.plot(range(T+1), 1 - np.mean(pi_paths_F, 0), label='F generates')
plt.plot(range(T+1), np.mean(pi_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.

39.9.3 Another Graph of Population Dynamics of π_t

More insights about the dynamics of $\{\pi_t\}$ can be gleaned by computing the following conditional expectations of $\frac{\pi_{t+1}}{\pi_t}$ as functions of π_t via integration with respect to the pertinent probability distribution:

$$\begin{aligned} E\left[\frac{\pi_{t+1}}{\pi_t} \mid q = \omega, \pi_t\right] &= E\left[\frac{l(w_{t+1})}{\pi_t l(w_{t+1}) + (1 - \pi_t)} \mid q = \omega, \pi_t\right], \\ &= \int_0^1 \frac{l(w_{t+1})}{\pi_t l(w_{t+1}) + (1 - \pi_t)} \omega(w_{t+1}) dw_{t+1} \end{aligned}$$

where $\omega = f, g$.

The following code approximates the integral above:

```
In [11]: def expected_ratio(F_a=1, F_b=1, G_a=3, G_b=1.2):

    # define f and g
    f = njit(lambda x: p(x, F_a, F_b))
    g = njit(lambda x: p(x, G_a, G_b))

    l = lambda w: f(w) / g(w)
    integrand_f = lambda w, pi: f(w) * l(w) / (pi * l(w) + 1 - pi)
    integrand_g = lambda w, pi: g(w) * l(w) / (pi * l(w) + 1 - pi)

    pi_grid = np.linspace(0.02, 0.98, 100)

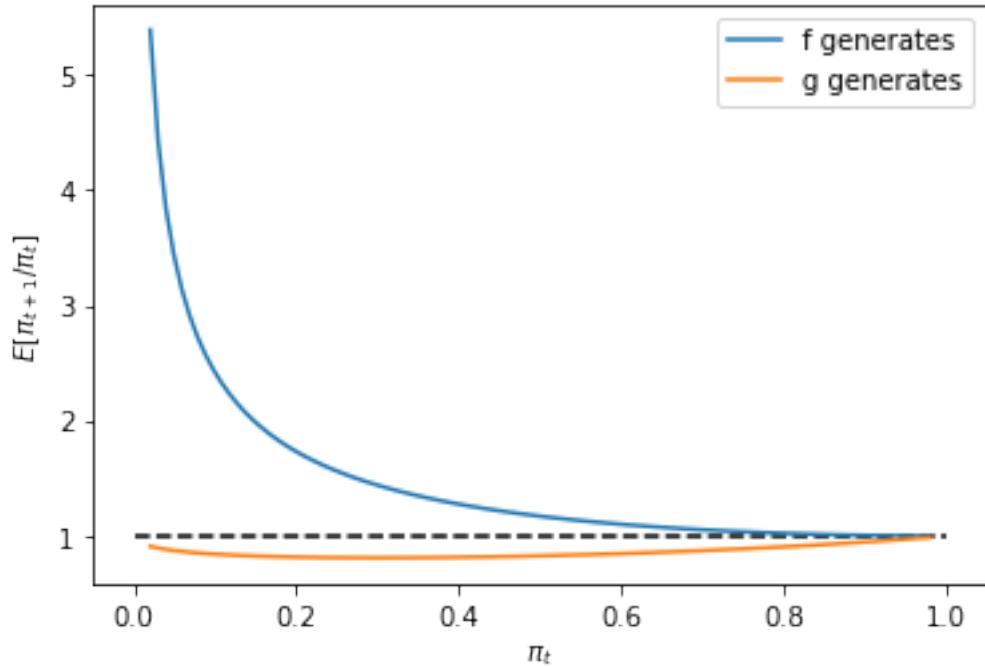
    expected_ratio = np.empty(len(pi_grid))
    for q, inte in zip(["f", "g"], [integrand_f, integrand_g]):
        for i, pi in enumerate(pi_grid):
            expected_ratio[i] = quad(inte, 0, 1, args=(pi,))[0]
    plt.plot(pi_grid, expected_ratio, label=f"{q} generates")

    plt.hlines(1, 0, 1, linestyle="--")
    plt.xlabel("$\pi_t$")
    plt.ylabel("$E[\pi_{t+1}/\pi_t]$")
    plt.legend()

    plt.show()
```

First, consider the case where $F_a = F_b = 1$ and $G_a = 3, G_b = 1.2$.

```
In [12]: expected_ratio()
```

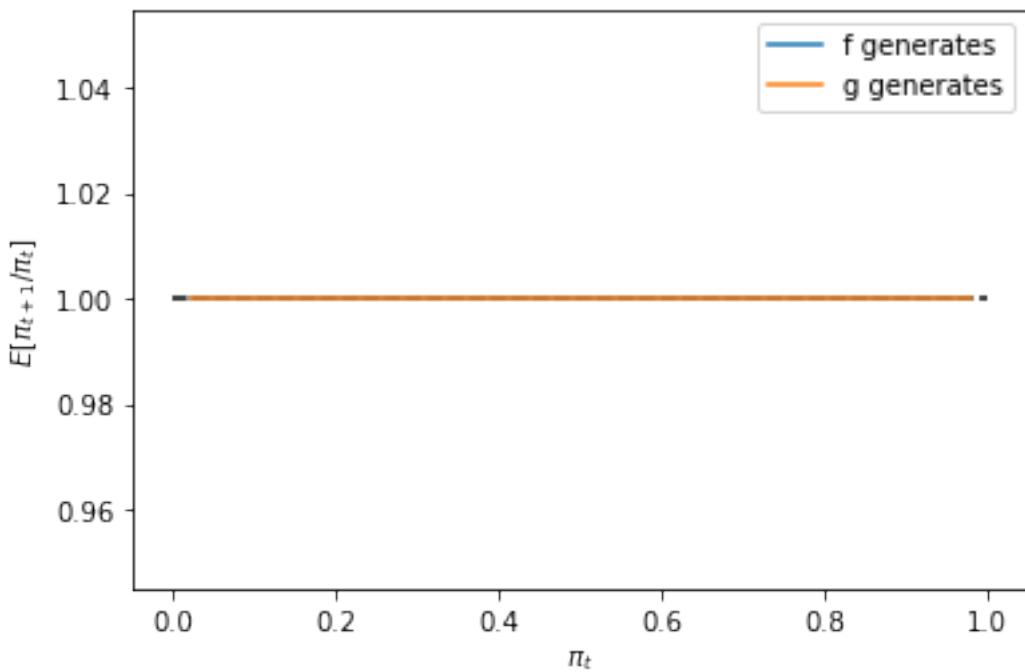


The above graphs shows that when F generates the data, π_t on average always heads north, while when G generates the data, π_t heads south.

Next, we'll look at a degenerate case in which 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.

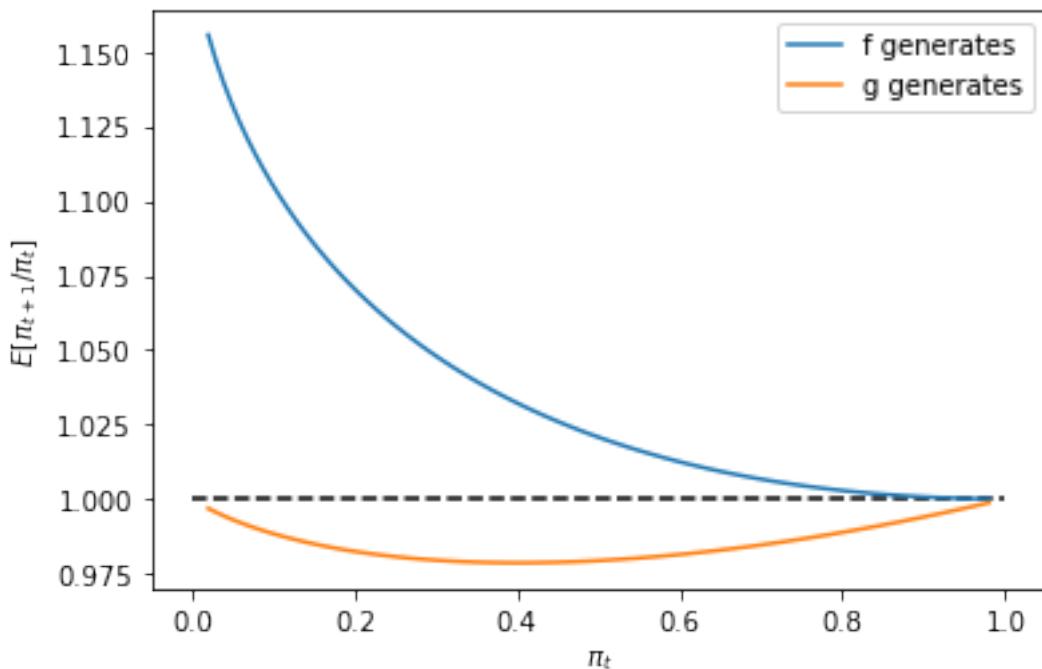
```
In [13]: expected_ratio(F_a=3, F_b=1.2)
```



The above graph says that π_t is inert and would remain 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 $G_a = 3, G_b = 1.2$.

In [14]: `expected_ratio(F_a=2, F_b=1, G_a=3, G_b=1.2)`



39.10 Sequels

We'll dig deeper into some of the ideas used here in the following lectures:

- [this lecture](#) describes **likelihood ratio processes** and their role in frequentist and Bayesian statistical theories
- [this lecture](#) returns to the subject of this lecture and studies whether the Captain's hunch that the (frequentist) decision rule that the Navy had ordered him to use can be expected to be better or worse than the rule sequential rule that Abraham Wald designed

Chapter 40

Likelihood Ratio Processes and Bayesian Learning

40.1 Contents

- Overview 40.2
- The Setting 40.3
- Likelihood Ratio Process and Bayes' Law 40.4
- Sequels 40.5

```
In [1]: import numpy as np
import matplotlib.pyplot as plt
from numba import vectorize, njit
from math import gamma
%matplotlib inline
```

40.2 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](#).

This lecture provides technical results that underly outcomes to be studied in [this lecture](#) and [this lecture](#) and [this lecture](#).

40.3 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 $w^t = \{w_1, \dots, w_t\}$ 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* 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 [?].

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 .

In [2]: # 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 = njit(lambda x: p(x, F_a, F_b))
g = njit(lambda x: p(x, G_a, G_b))
```

In [3]: @njit

```
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

In [4]: l_arr_g = simulate(G_a, G_b, N=50000)
l_seq_g = np.cumprod(l_arr_g, axis=1)

In [5]: l_arr_f = simulate(F_a, F_b, N=50000)
l_seq_f = np.cumprod(l_arr_f, axis=1)

40.4 Likelihood Ratio Process and Bayes' Law

Let π_t be a Bayesian posterior defined as

$$\pi_t = \text{Prob}(q = f | w^t)$$

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**.

Bayes' law implies that $\{\pi_t\}$ obeys the recursion

$$\pi_t = \frac{\pi_{t-1} l_t(w_t)}{\pi_{t-1} l_t(w_t) + 1 - \pi_{t-1}} \quad (1)$$

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 (1)

```
In [6]: @njit
def update(pi, l):
    "Update pi using likelihood l"

    # Update belief
    pi = pi * l / (pi * l + 1 - pi)

    return pi
```

Formula (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 $L(w^{t+1}) > 0$, 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 $L(w^{t+1})$, the likelihood ratio process at $t + 1$, and the initial prior π_0

$$\pi_{t+1} = \frac{\pi_0 L(w^{t+1})}{\pi_0 L(w^{t+1}) + 1 - \pi_0}. \quad (2)$$

Formula (2) generalizes formula (1).

Formula (2) can be regarded as a one step revision of prior probability π_0 after seeing the batch of data $\{w_i\}_{i=1}^{t+1}$.

Formula (2) shows the key role that the likelihood ratio process $L(w^{t+1})$ plays in determining the posterior probability π_{t+1} .

Formula (2) is the foundation for the insight that, because of how the likelihood ratio process behaves as $t \rightarrow +\infty$, 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 .

```
In [7]: 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 .

```
In [8]: 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])
```

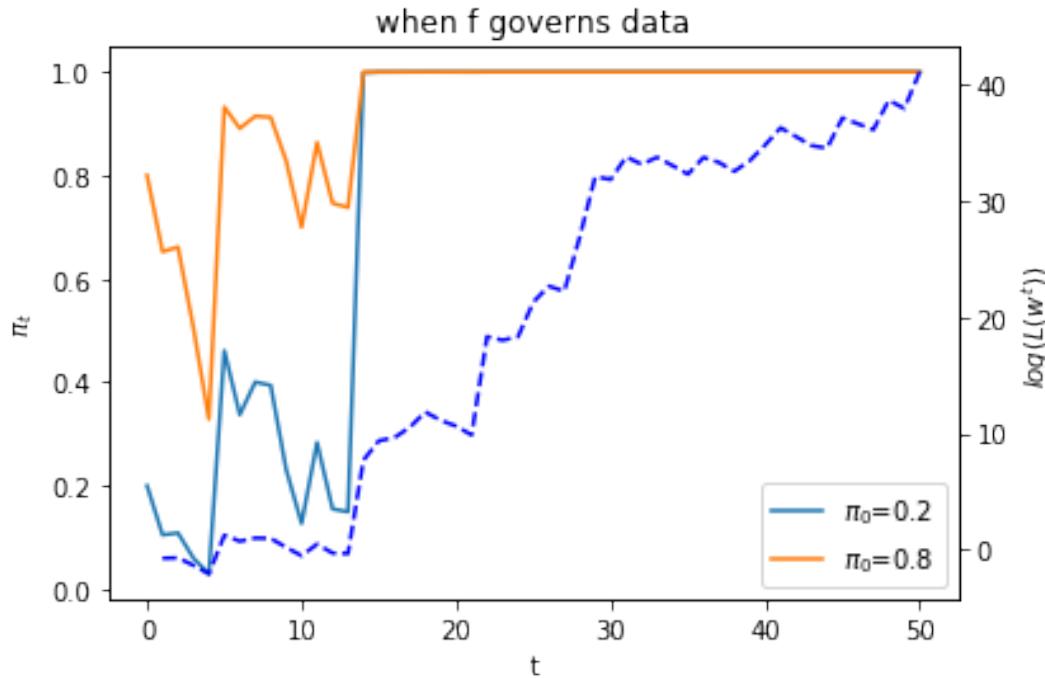
```
In [9]: fig, ax1 = plt.subplots()

for i in range(2):
    ax1.plot(range(T+1), π_seq_f[i, :], label=f"\u03c0\u2080={π_seq_f[i, 0]}")

ax1.set_ylabel("\u03c0_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

```
In [10]: T = l_arr_g.shape[1]
pi_seq_g = np.empty((2, T+1))
pi_seq_g[:, 0] = pi1, pi2

for t in range(T):
    for i in range(2):
        pi_seq_g[i, t+1] = update(pi_seq_g[i, t], l_arr_g[0, t])

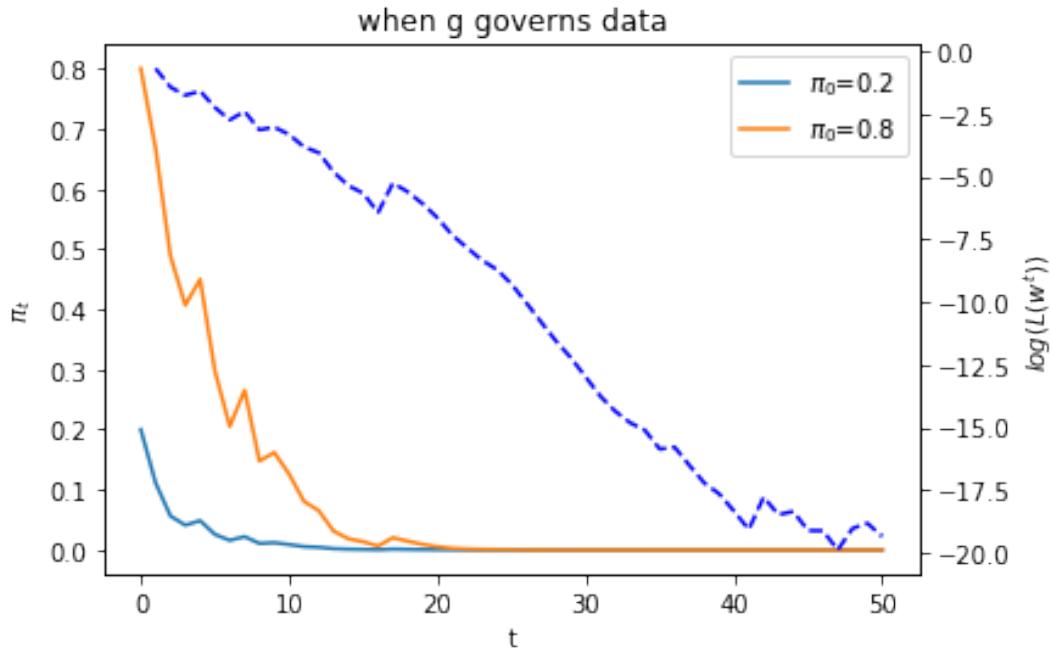
In [11]: fig, ax1 = plt.subplots()

for i in range(2):
    ax1.plot(range(T+1), pi_seq_g[i, :], label=f"$\pi_0=\{pi_seq_g[i, 0]\}$")

ax1.set_ylabel("$\pi_t$")
ax1.set_xlabel("t")
ax1.legend()
ax1.set_title("when g governs data")

ax2 = ax1.twinx()
ax2.plot(range(1, T+1), np.log(pi_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 .

```
In [12]: 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])
```

```
In [13]: np.abs(π_seq - π_seq_f).max() < 1e-10
```

```
Out[13]: True
```

We thus conclude that the likelihood ratio process is a key ingredient of the formula (2) for a Bayesian's posterior probability that nature has drawn history w^t as repeated draws from density g .

40.5 Sequels

This lecture has been devoted to building some useful infrastructure.

We'll build on results highlighted in this lectures to understand inferences that are the foundations of results described in [this lecture](#) and [this lecture](#) and [this lecture](#).

Chapter 41

Bayesian versus Frequentist Decision Rules

41.1 Contents

- Overview [41.2](#)
- Setup [41.3](#)
- Frequentist Decision Rule [41.4](#)
- Bayesian Decision Rule [41.5](#)
- Was the Navy Captain's hunch correct? [41.6](#)
- More details [41.7](#)

In addition to what's in Anaconda, this lecture will need the following libraries:

```
In [1]: !pip install quantecon
```

```
In [2]: import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from numba import njit, prange, float64, int64
from numba.experimental import jitclass
from interpolation import interp
from math import gamma
from scipy.optimize import minimize
```

41.2 Overview

This lecture follows up on ideas presented in the following lectures:

- [A Problem that Stumped Milton Friedman](#)
- [Exchangeability and Bayesian Updating](#)
- [Likelihood Ratio Processes](#)

In [A Problem that Stumped Milton Friedman](#) we described a problem that a Navy Captain presented to Milton Friedman during World War II.

The Navy had instructed the Captain to use a decision rule for quality control that the Captain suspected could be dominated by a better rule.

(The Navy had ordered the Captain to use an instance of a **frequentist decision rule**.)

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.

One of the members of the group, the great mathematician 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 **frequentist** and **Bayesian** decision rules for an instance of the Navy Captain's decision problem.

The simulations validate the Navy Captain's hunch that there is a better rule than the one the Navy had ordered him to use.

41.3 Setup

To formalize the problem of the Navy Captain whose questions posed the problem that Milton Friedman and Allan Wallis handed over to Abraham Wald, we consider a setting with the following parts.

- Each period a decision maker draws a non-negative random variable Z from a probability distribution that he does not completely understand. He knows that two probability distributions are possible, f_0 and f_1 , and that whichever distribution it is remains fixed over time. The decision maker believes that before the beginning of time, nature once and for all 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 and is worried by two types of errors and the losses that they impose on him.

- a loss \bar{L}_1 from a **type I error** that occurs when he decides that $f = f_1$ when actually $f = f_0$
- a loss \bar{L}_0 from a **type II error** that occurs when 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 Problem that Stumped Milton Friedman](#) except that we increase both \bar{L}_0 and \bar{L}_1 from 25 to 100 to encourage the frequentist Navy Captain 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.

```
In [3]: @njit
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.

```
In [4]: 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),               # size of Monte Carlo simulation
    ('mc_size', int64),                   # sequence of random values
    ('z0', float64[:]),                  # sequence of random values
    ('z1', float64[:])
]
```

```
In [5]: @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 x(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)

    return π_new
```

In [6]: wf = WaldFriedman()

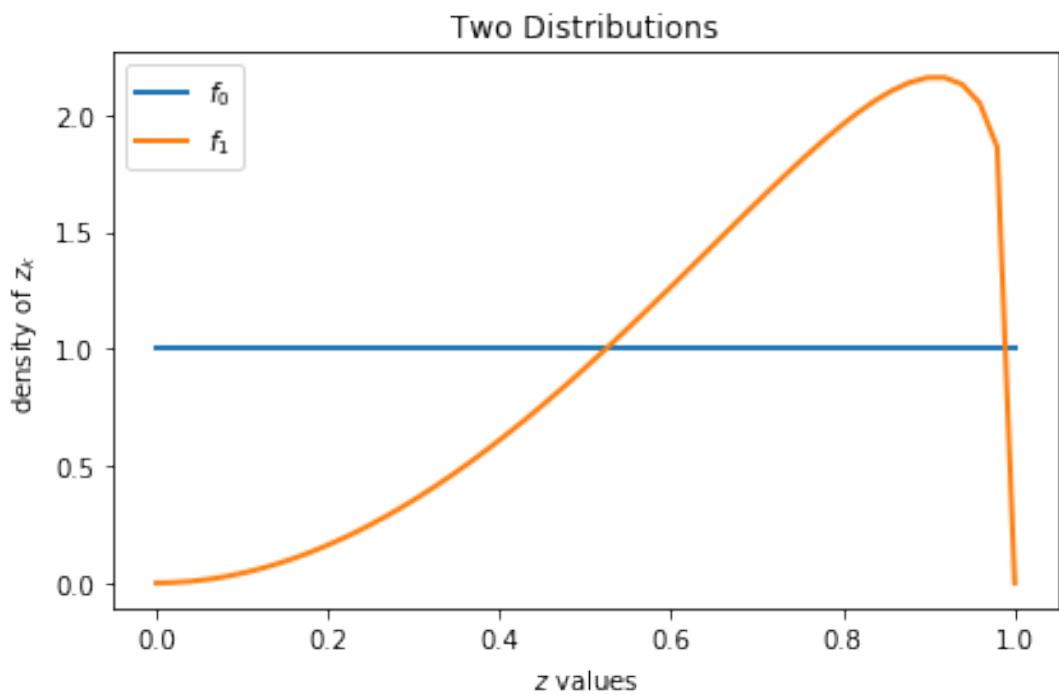
```
grid = np.linspace(0, 1, 50)

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

41.4 Frequentist Decision Rule

The Navy told the Captain to use a frequentist decision rule.

In particular, it gave him a decision rule that the Navy had designed by using frequentist statistical theory to minimize an expected loss function.

That decision rule is characterized by a sample size t and a cutoff d associated with 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

To understand how that rule was engineered, 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

$$\bar{V}_{fre}(t, d) = ct + \pi^* PFA \times \bar{L}_1 + (1 - \pi^*) (1 - PD) \times \bar{L}_0 \quad (1)$$

$$\begin{aligned} \text{where } PFA &= \Pr\{L(z^t) < d \mid q = f_0\} \\ PD &= \Pr\{L(z^t) < d \mid q = f_1\} \end{aligned}$$

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** and can be uniquely pinned down by choosing d .

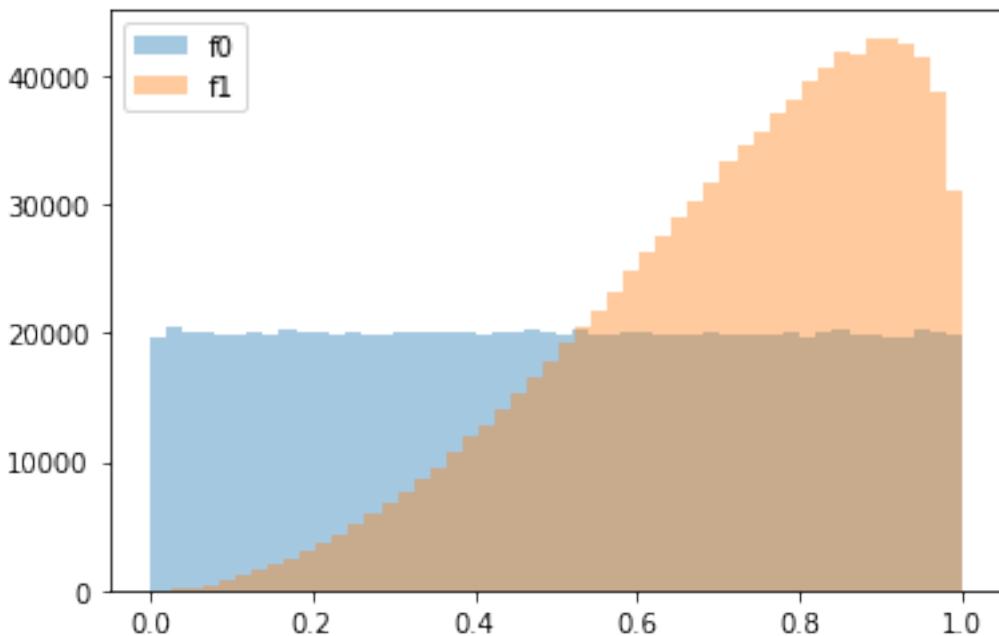
To see some receiver operating characteristic curves, please see this lecture [Likelihood Ratio Processes](#).

To solve for $\bar{V}_{fre}(t, d)$ numerically, we first simulate sequences of z when either f_0 or f_1 generates data.

```
In [7]: N = 10000
T = 100
```

```
In [8]: z0_arr = np.random.beta(wf.a0, wf.b0, (N, T))
z1_arr = np.random.beta(wf.a1, wf.b1, (N, T))
```

```
In [9]: 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.

```
In [10]: l = lambda z: wf.f0(z) / wf.f1(z)
```

```
In [11]: 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 .

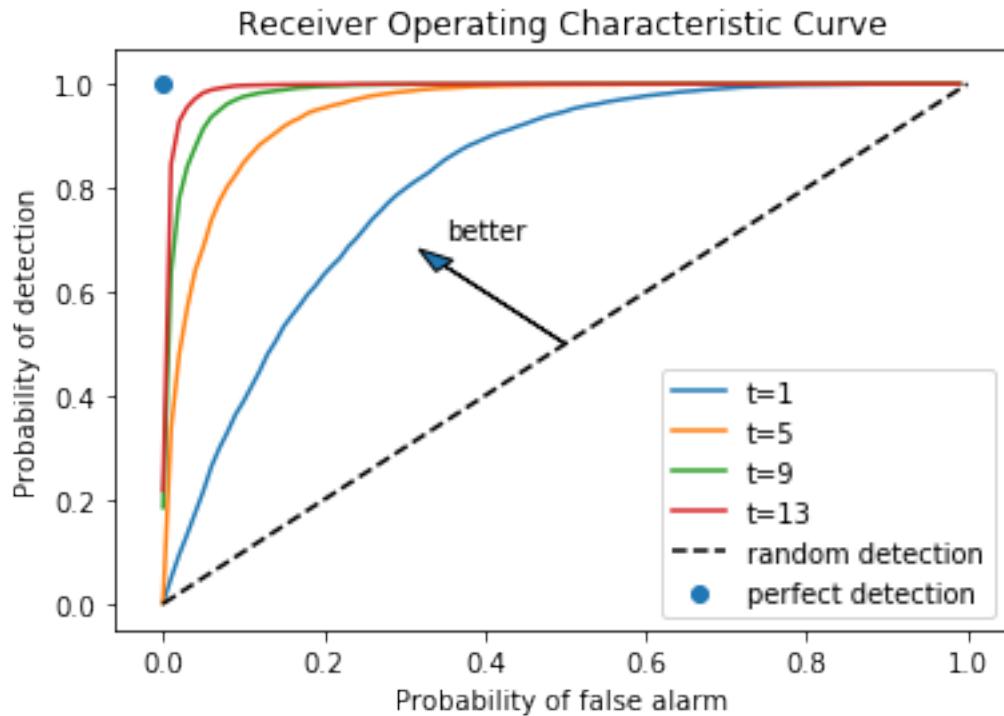
```
In [12]: PFA = np.arange(0, 100, 1)

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")
plt.show()
```



Our frequentist minimizes the expected total loss presented in equation (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.

```
In [13]: @njit
def V_fre_d_t(d, t, L0_arr, L1_arr, pi_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 = pi_star * PFA * wf.L1 + (1 - pi_star) * (1 - PD) * wf.L0
    return V

In [14]: def V_fre_t(t, L0_arr, L1_arr, pi_star, wf):
    res = minimize(V_fre_d_t, 1, args=(t, L0_arr, L1_arr, pi_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

```

In [15]: `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

```

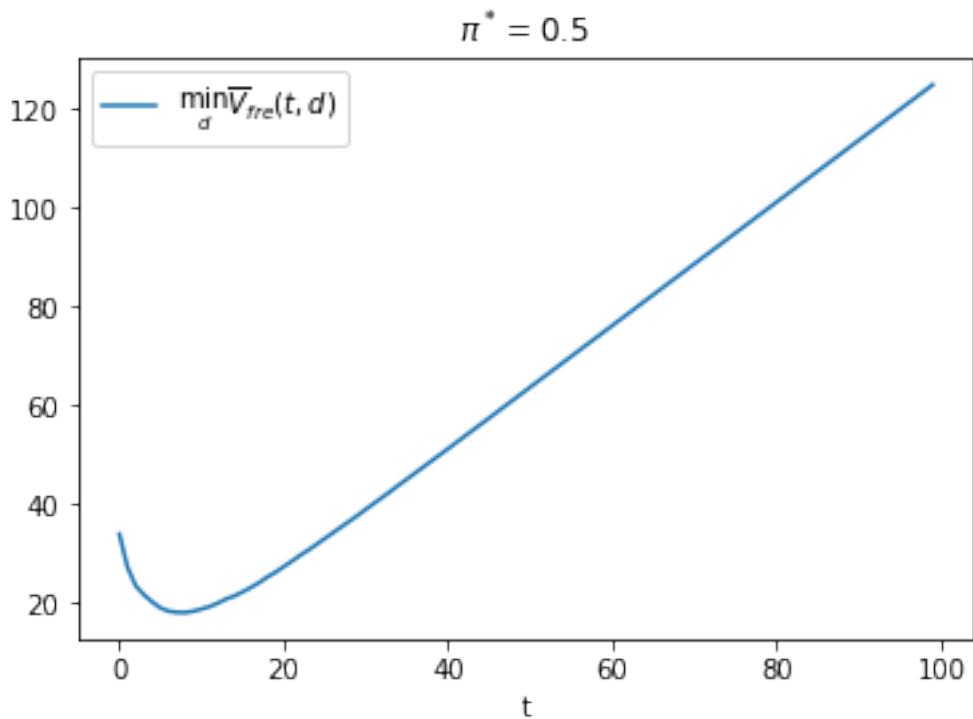
In [16]: `π_star = 0.5`

```

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='$\min_d \overline{V}_{fre}(t, d)$')
plt.xlabel('t')
plt.title('$\pi^*=0.5$')
plt.legend()
plt.show()

```



```
In [17]: t_optimal = np.argmin(V_fre_arr) + 1
```

```
In [18]: msg = f"The above graph indicates that minimizing over t tells the frequentist to draw {t_optimal} observations and then decide."
print(msg)
```

The above graph indicates that minimizing over t tells the frequentist to draw 9 observations and then decide.

Let's now change the value of π^* and watch how the decision rule changes.

```
In [19]: n_pi = 20
pi_star_arr = np.linspace(0.1, 0.9, n_pi)

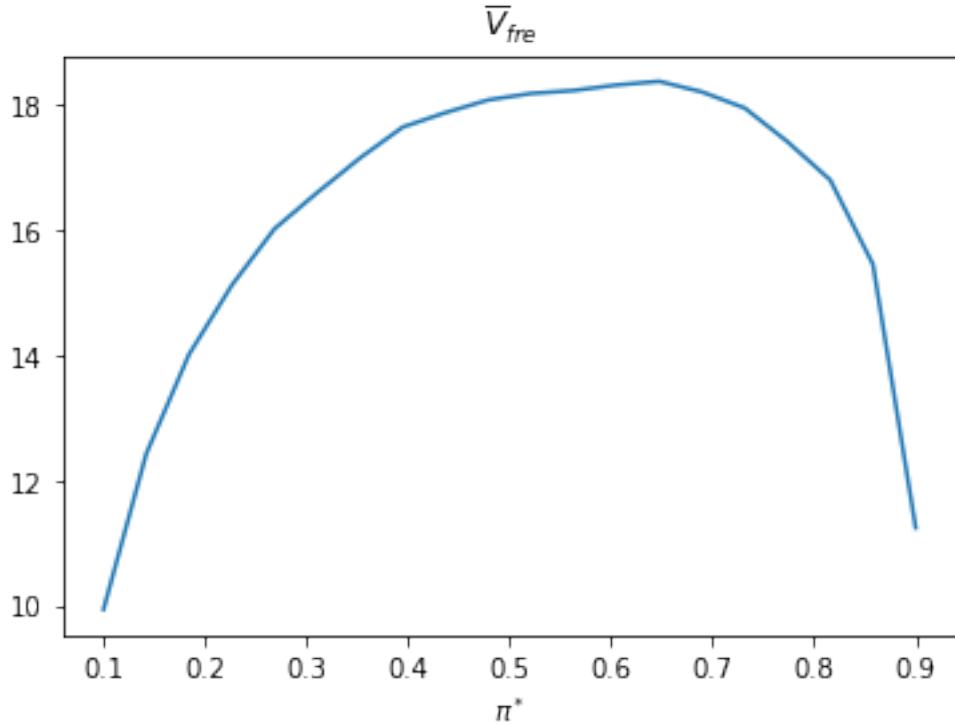
V_fre_bar_arr = np.empty(n_pi)
t_optimal_arr = np.empty(n_pi)
PFA_optimal_arr = np.empty(n_pi)
PD_optimal_arr = np.empty(n_pi)

for i, pi_star in enumerate(pi_star_arr):
    V_fre_arr, PFA_arr, PD_arr = compute_V_fre(L0_arr, L1_arr, pi_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]
```

```
In [20]: plt.plot(pi_star_arr, V_fre_bar_arr)
plt.xlabel('$\pi^*$')
plt.title('$\overline{V}_{\text{fre}}$')

plt.show()
```



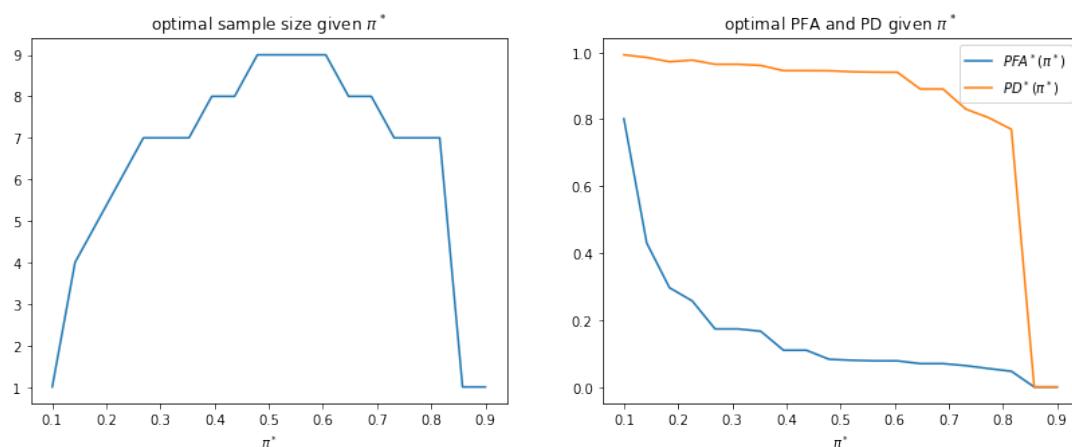
The following shows how optimal sample size t and targeted (PFA, PD) change as π^* varies.

```
In [21]: fig, axs = plt.subplots(1, 2, figsize=(14, 5))

axs[0].plot(pi_star_arr, t_optimal_arr)
axs[0].set_xlabel('$\pi^*$')
axs[0].set_title('optimal sample size given $\pi^$')

axs[1].plot(pi_star_arr, PFA_optimal_arr, label='$PFA^*(\pi^*)$')
axs[1].plot(pi_star_arr, PD_optimal_arr, label='$PD^*(\pi^*)$')
axs[1].set_xlabel('$\pi^*$')
axs[1].legend()
axs[1].set_title('optimal PFA and PD given $\pi^$')

plt.show()
```



41.5 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.

We presented a Bayesian procedure that instructed the Captain to makes decisions by comparing his current Bayesian posterior probability π with two cutoff probabilities called α and β .

To proceed, we borrow some Python code from the quantecon lecture [A Problem that Stumped Milton Friedman](#) that computes α and β .

```
In [22]: @njit(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

    x = wf.x

    h_new = np.empty_like(π_grid)
    h_func = lambda p: interp(π_grid, h, p)

    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 = x(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 = x(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
        h_new[i] = c + integral

    return h_new
```

```
In [23]: @njit
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 i == max_iter:
    print("Failed to converge!")

return h_new
```

In [24]: `h_star = solve_model(wf)`

```
<ipython-input-23-cd5766267a2f>:15: NumbaWarning: The TBB threading layer
requires TBB version 2019.5 or later i.e., TBB_INTERFACE_VERSION >= 11005. Found
TBB_INTERFACE_VERSION = 11004. The TBB threading layer is disabled.
h_new = Q(h, wf)
```

In [25]: `@njit`

```
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
    """

    pi_grid = wf.pi_grid
    L0, L1 = wf.L0, wf.L1

    # Evaluate cost at all points on grid for choosing a model
    payoff_f0 = (1 - pi_grid) * L0
    payoff_f1 = pi_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)
    beta = pi_grid[np.searchsorted(
        payoff_f1 - np.minimum(h, payoff_f0),
        1e-10)
    - 1]
    alpha = pi_grid[np.searchsorted(
        np.minimum(h, payoff_f1) - payoff_f0,
        1e-10)
    - 1]

    return (beta, alpha)
```

```
beta, alpha = find_cutoff_rule(wf, h_star)
cost_L0 = (1 - wf.pi_grid) * wf.L0
cost_L1 = wf.pi_grid * wf.L1

fig, ax = plt.subplots(figsize=(10, 6))
```

```

ax.plot(wf.pi_grid, h_star, label='continuation value')
ax.plot(wf.pi_grid, cost_L1, label='choose f1')
ax.plot(wf.pi_grid, cost_L0, label='choose f0')
ax.plot(wf.pi_grid,
        np.amin(np.column_stack([h_star, cost_L0, cost_L1]), axis=1),
        lw=15, alpha=0.1, color='b', label='minimum cost')

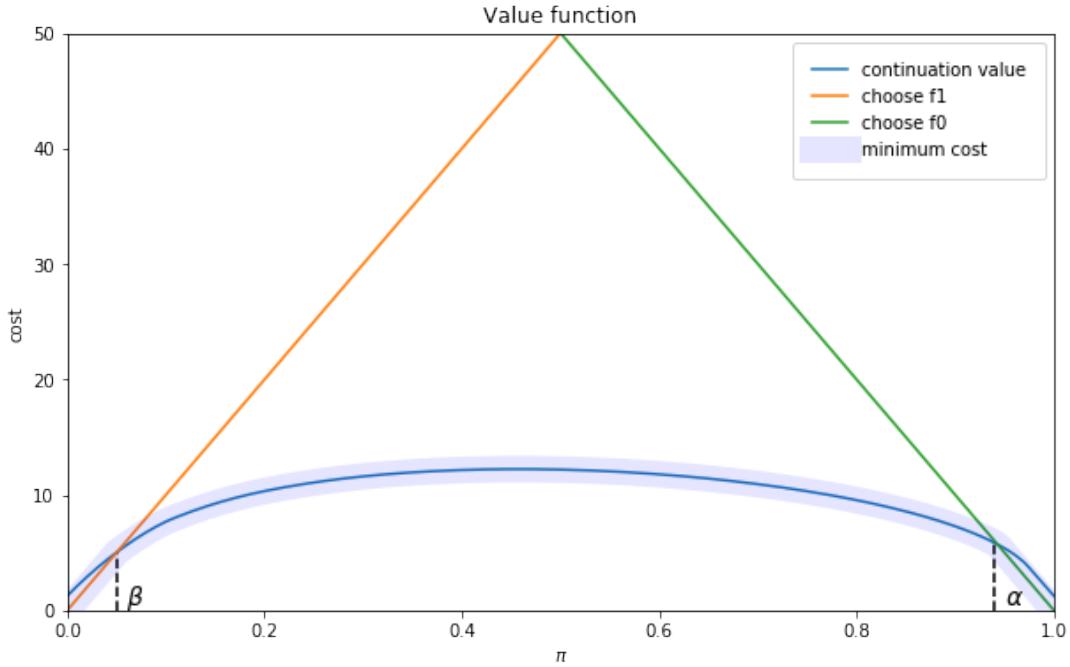
ax.annotate(r"\beta", xy=(beta + 0.01, 0.5), fontsize=14)
ax.annotate(r"\alpha", xy=(alpha + 0.01, 0.5), fontsize=14)

plt.vlines(beta, 0, beta * wf.L0, linestyle="--")
plt.vlines(alpha, 0, (1 - alpha) * wf.L1, linestyle="--")

ax.set(xlim=(0, 1), ylim=(0, 0.5 * max(wf.L0, wf.L1)), ylabel="cost",
       xlabel="\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 probabilities α and β .

The Bayesian decision rule is:

- accept H_0 if $\pi \geq \alpha$
- accept H_1 if $\pi \leq \beta$
- delay deciding and draw another z if $\beta \leq \pi \leq \alpha$

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 ,

$$V^0(\pi) = \begin{cases} 0 & \text{if } \alpha \leq \pi, \\ c + EV^0(\pi') & \text{if } \beta \leq \pi < \alpha, \\ \bar{L}_1 & \text{if } \pi < \beta. \end{cases}$$

1. under f_1

$$V^1(\pi) = \begin{cases} \bar{L}_0 & \text{if } \alpha \leq \pi, \\ c + EV^1(\pi') & \text{if } \beta \leq \pi < \alpha, \\ 0 & \text{if } \pi < \beta. \end{cases}$$

where $\pi' = \frac{\pi f_0(z')}{\pi f_0(z') + (1-\pi)f_1(z')}$.

Given a prior probability π_0 , the expected loss for the Bayesian is

$$\bar{V}_{Bayes}(\pi_0) = \pi^* V^0(\pi_0) + (1 - \pi^*) V^1(\pi_0).$$

Below we write some Python code that computes $V^0(\pi)$ and $V^1(\pi)$ numerically.

```
In [26]: @njit(parallel=True)
def V_q(wf, flag):
    V = np.zeros(wf.pi_grid_size)
    if flag == 0:
        z_arr = wf.z0
        V[wf.pi_grid < beta] = wf.L1
    else:
        z_arr = wf.z1
        V[wf.pi_grid >= alpha] = wf.L0

    V_old = np.empty_like(V)

    while True:
        V_old[:] = V[:]
        V[(beta <= wf.pi_grid) & (wf.pi_grid < alpha)] = 0

        for i in prange(len(wf.pi_grid)):
            pi = wf.pi_grid[i]

            if pi >= alpha or pi < beta:
                continue

            for j in prange(len(z_arr)):
                pi_next = wf.x(z_arr[j], pi)
                V[i] += wf.c + interp(wf.pi_grid, V_old, pi_next)

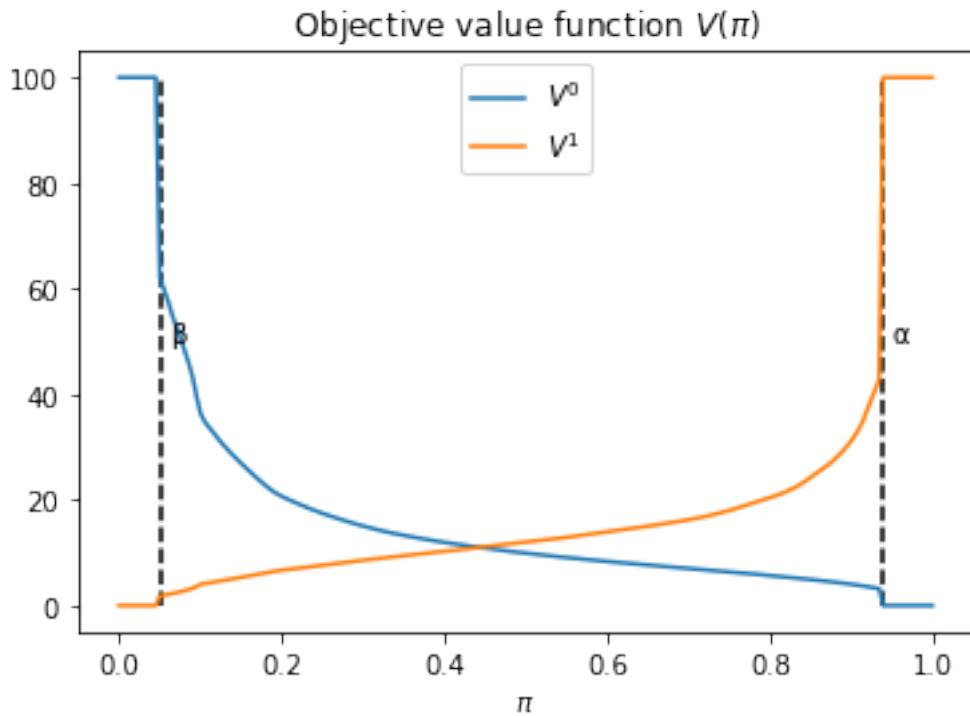
        V[i] /= wf.mc_size

        if np.abs(V - V_old).max() < 1e-5:
            break

    return V
```

```
In [27]: V0 = V_q(wf, 0)
V1 = V_q(wf, 1)

plt.plot(wf.pi_grid, V0, label='$V^0$')
plt.plot(wf.pi_grid, V1, label='$V^1$')
plt.vlines(beta, 0, wf.L0, linestyle='--')
plt.text(beta+0.01, wf.L0/2, 'β')
plt.vlines(alpha, 0, wf.L0, linestyle='--')
plt.text(alpha+0.01, wf.L0/2, 'α')
plt.xlabel('$\pi$')
plt.title('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 9 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^* .

```
In [28]: def compute_V_baye_bar(pi_star, V0, V1, wf):
```

```
V_baye = pi_star * V0 + (1 - pi_star) * V1
pi_idx = np.argmin(V_baye)
pi_optimal = wf.pi_grid[pi_idx]
V_baye_bar = V_baye[pi_idx]

return V_baye, pi_optimal, V_baye_bar
```

In [29]: `π_star_arr = [0.25, 0.3, 0.5, 0.7]`

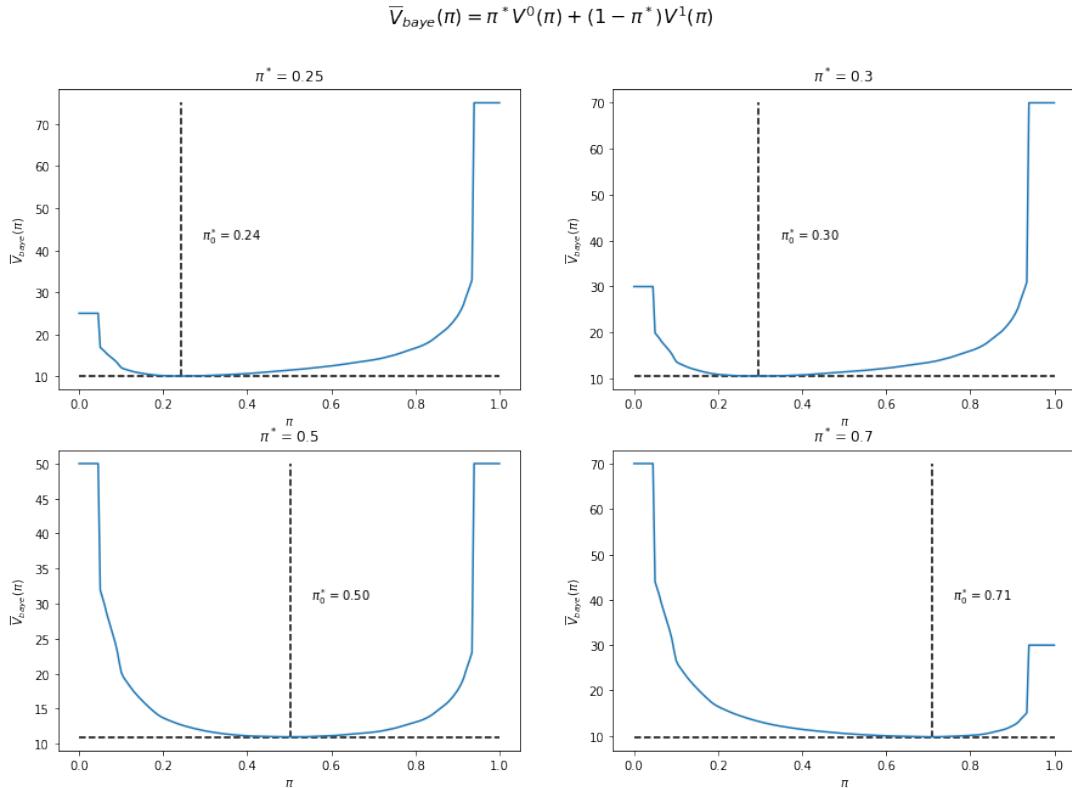
```
fig, axs = plt.subplots(2, 2, figsize=(15, 10))

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,  
                          f'${\pi_0^*}={\pi_{optimal}:0.2f}')
    axs[row_i, col_i].set_xlabel(f'${\pi}$')
    axs[row_i, col_i].set_ylabel(f'$\overline{V}_{baye}(\pi)$')
    axs[row_i, col_i].set_title(f'${\pi^*}={\pi_{star}}$')

fig.suptitle(f'$\overline{V}_{baye}(\pi)={\pi^*}V^0(\pi)+(1-{\pi^*})V^1(\pi)$',
             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^* .

```
In [30]: 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)

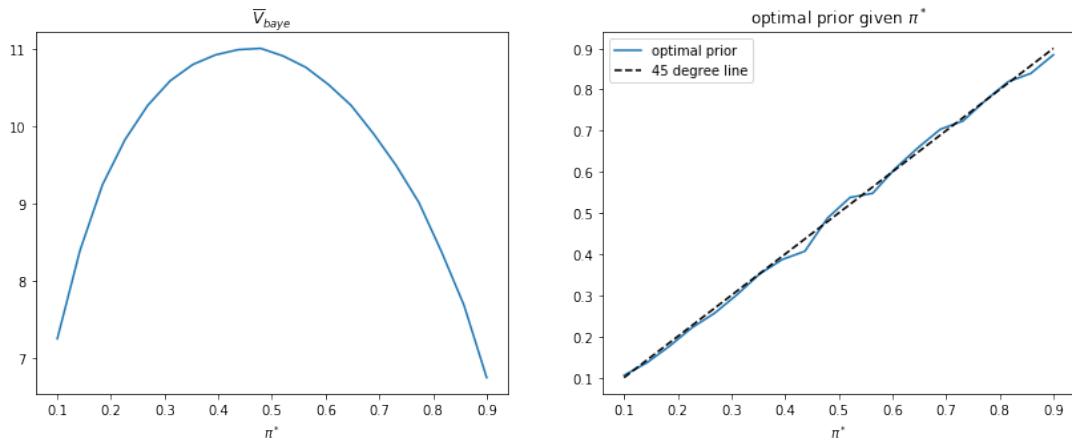
for i, π_star in enumerate(π_star_arr):
    V_baye, π_optimal, V_baye_bar = compute_V_baye_bar(π_star, V0, V1, wf)
    V_baye_bar_arr[i] = V_baye_bar
    π_optimal_arr[i] = π_optimal

fig, axs = plt.subplots(1, 2, figsize=(14, 5))

axs[0].plot(π_star_arr, V_baye_bar_arr)
axs[0].set_xlabel('$\pi^*$')
axs[0].set_title('$\overline{V}_{baye}$')

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('$\pi^*$')
axs[1].set_title('optimal prior given $\pi^*$')
axs[1].legend()

plt.show()
```



41.6 Was the Navy Captain's hunch correct?

We now compare average (i.e., frequentist) losses obtained by the frequentist and Bayesian decision rules.

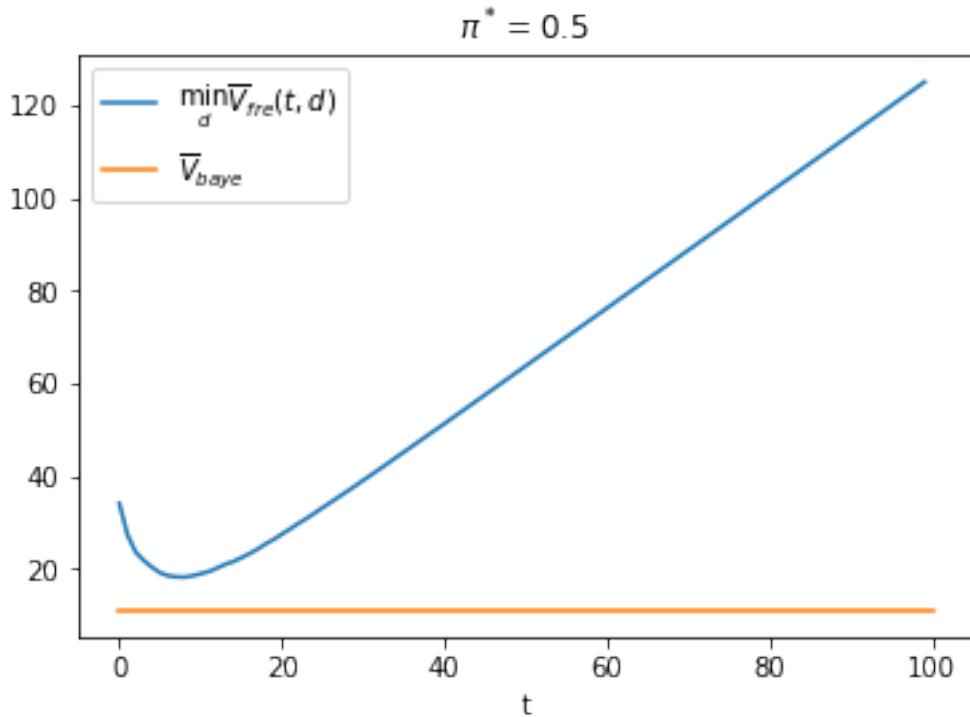
As a starting point, let's compare average loss functions when $\pi^* = 0.5$.

```
In [31]: π_star = 0.5
```

```
In [32]: # frequentist
V_fre_arr, PFA_arr, PD_arr = compute_V_fre(L0_arr, L1_arr, π_star, wf)
```

```
# bayesian
V_baye = π_star * V0 + π_star * V1
V_baye_bar = V_baye.min()
```

```
In [33]: plt.plot(range(T), V_fre_arr, label='$\min_d \overline{V}_{fre}(t, d)$')
plt.plot([0, T], [V_baye_bar, V_baye_bar], label='$\overline{V}_{baye}$')
plt.xlabel('t')
plt.title('$\pi^* = 0.5$')
plt.legend()
plt.show()
```



Evidently, there is no sample size t at which the frequentist decision rule attains a lower loss function than does the Bayesian rule.

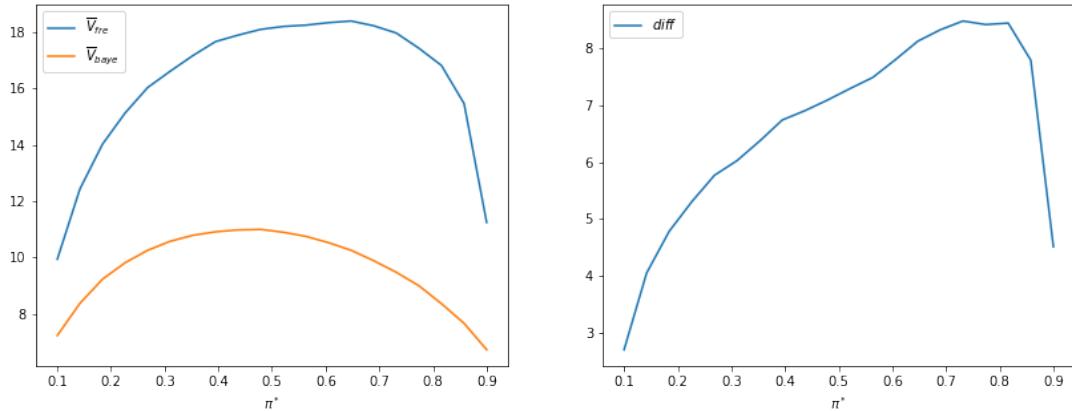
Furthermore, the following graph indicates that the Bayesian decision rule does better on average for all values of π^* .

```
In [34]: fig, axs = plt.subplots(1, 2, figsize=(14, 5))

axs[0].plot(π_star_arr, V_fre_bar_arr, label='$\overline{V}_{fre}$')
axs[0].plot(π_star_arr, V_baye_bar_arr, label='$\overline{V}_{baye}$')
axs[0].legend()
axs[0].set_xlabel('$\pi^*$')

axs[1].plot(π_star_arr, V_fre_bar_arr - V_baye_bar_arr, label='diff')
axs[1].legend()
axs[1].set_xlabel('$\pi^*$')

plt.show()
```



The right panel of the above graph plots the difference $\bar{V}_{fre} - \bar{V}_{Bayes}$.

It is always positive.

41.7 More details

We can provide more insights by focusing on the case in which $\pi^* = 0.5 = \pi_0$.

In [35]: `π_star = 0.5`

Recall that when $\pi^* = 0.5$, the frequentist decision rule sets a sample size `t_optimal ex ante`.

For our parameter settings, we can compute its value:

In [36]: `t_optimal`

Out[36]: 9

For convenience, let's define `t_idx` as the Python array index corresponding to `t_optimal` sample size.

In [37]: `t_idx = t_optimal - 1`

41.7.1 Distribution of Bayesian decision rule's times to decide

By using simulations, we compute the frequency distribution of time to deciding 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 $q = f_1$.

```
In [38]: @njit(parallel=True)
def check_results(L_arr, α, β, 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] < β) or (π_arr[i, t] > α):
                time_arr[i] = t + 1
                correctness[i] = (flag == 0 and π_arr[i, t] > α) or (flag
→== 1 and
    π_arr[i, t] < β)
                break

    return time_arr, correctness
```

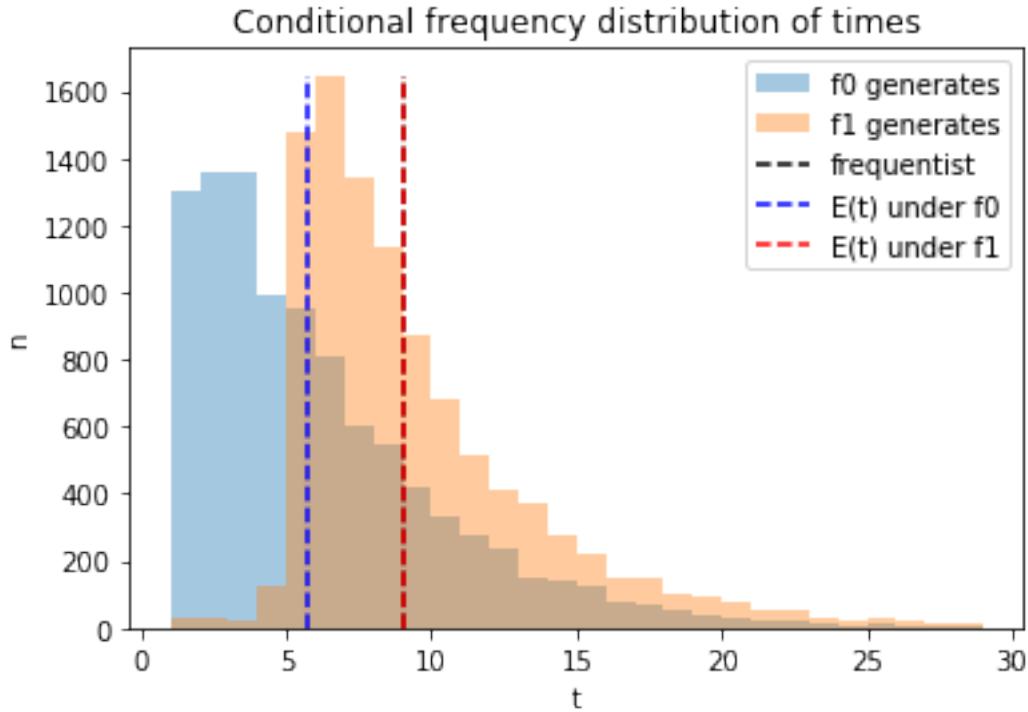
```
In [39]: time_arr0, correctness0 = check_results(L0_arr, α, β, 0, π_star)
time_arr1, correctness1 = check_results(L1_arr, α, β, 1, π_star)

# unconditional distribution
time_arr_u = np.concatenate((time_arr0, time_arr1))
correctness_u = np.concatenate((correctness0, correctness1))
```

```
In [40]: 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 two decision rules.

To begin, let's look at simulations of the Bayesian's beliefs over time.

We can easily compute the 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](#).

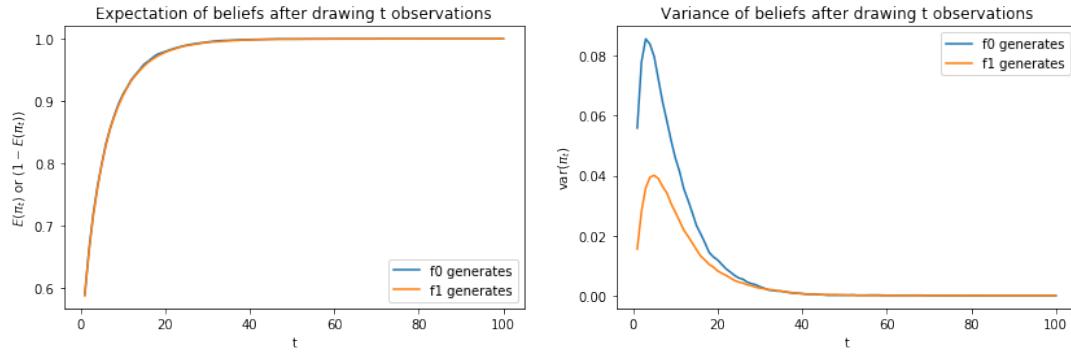
```
In [41]: π0_arr = π_star * L0_arr / (π_star * L0_arr + 1 - π_star)
π1_arr = π_star * L1_arr / (π_star * L1_arr + 1 - π_star)

In [42]: 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), color='blue',
    ↪label='f0 generates')
    axs[0].plot(np.arange(1, π1_arr.shape[1]+1), 1 - np.mean(π1_arr, 0), color='red',
    ↪label='f1 generates')
    axs[0].set_xlabel('t')
    axs[0].set_ylabel('$E(\pi_t)$ 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('var($\pi_t$)')
    axs[1].set_title('Variance of beliefs after drawing t observations')
    axs[1].legend()
```

```
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 size 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.

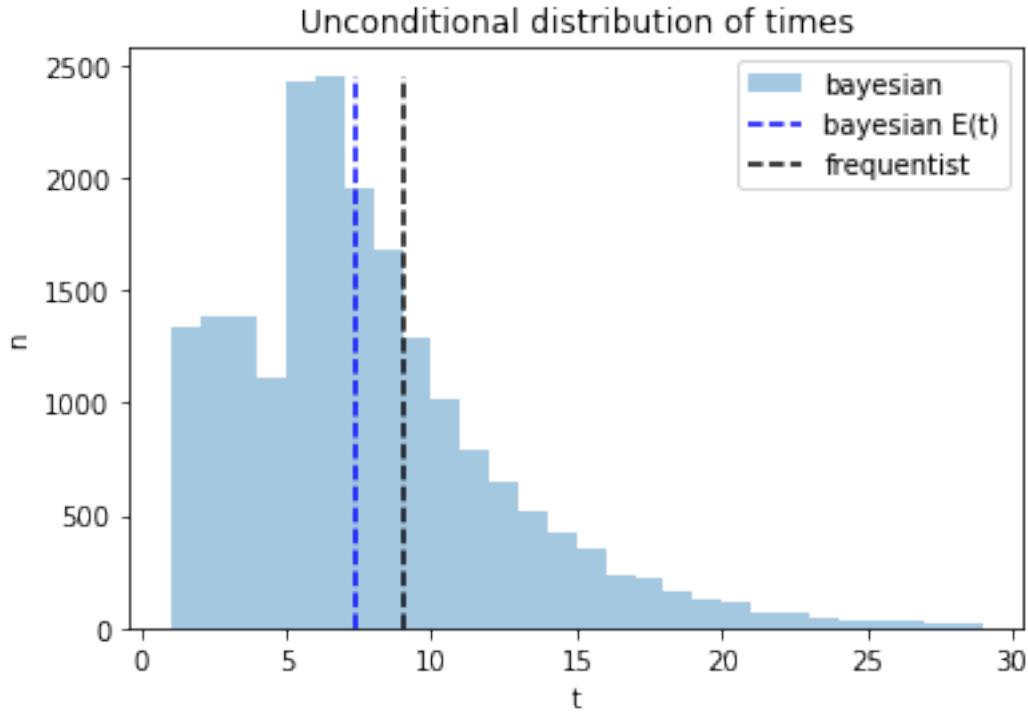
The code below plots outcomes of constructing an unconditional distribution by simply pooling the simulated data across the two possible distributions f_0 and f_1 .

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.

```
In [43]: 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()
```



41.7.2 Probability of making correct decisions

Now we use simulations to compute the fraction of samples in which the Bayesian and the frequentist decision rules decide correctly.

For the frequentist 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.

```
In [44]: # 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)

In [45]: 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],
```

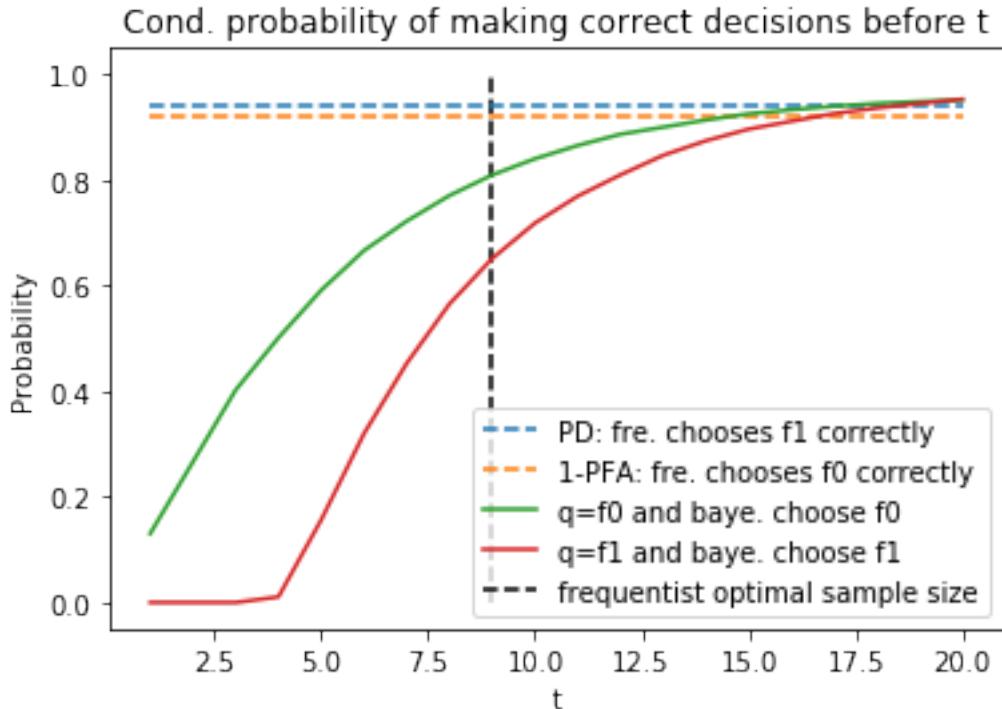
```

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.

```

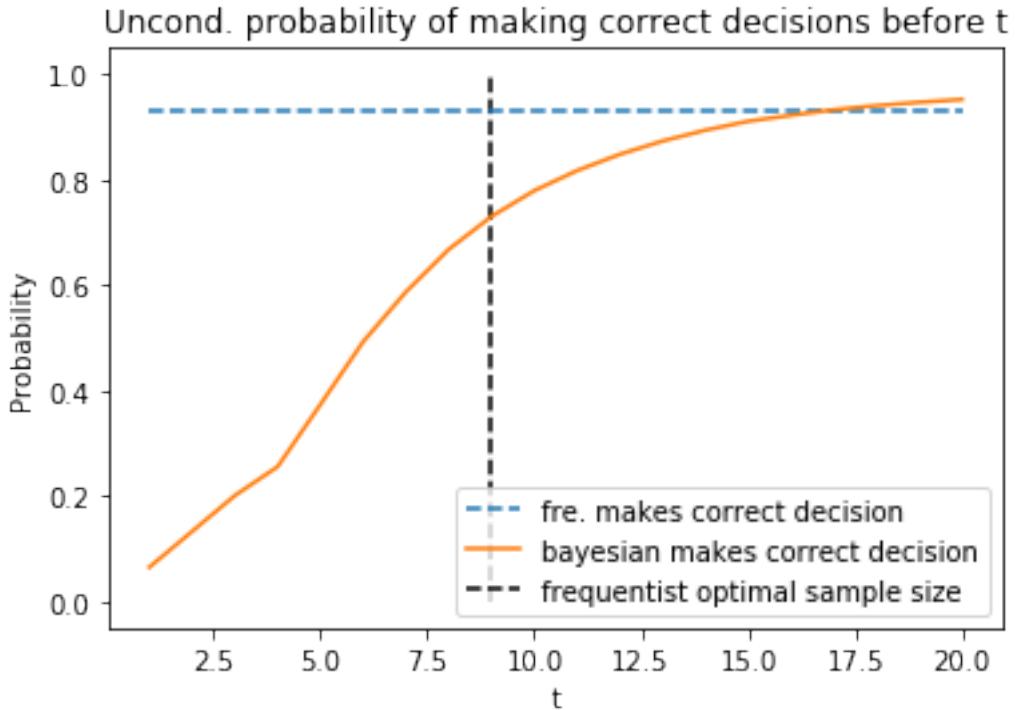
In [46]: plt.plot([1, 20], [(PD + 1 - PFA) / 2, (PD + 1 - PFA) / 2],
             linestyle='--', label='fre. makes correct decision')
plt.vlines(t_optimal, 0, 1, linestyle='--', label='frequentist optimal'
           sample size')

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()

```



41.7.3 Distribution of likelihood ratios at frequentist'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 α and β .

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.

```
In [47]: Lα = (1 - π_star) * α / (π_star - π_star * α)
Lβ = (1 - π_star) * β / (π_star - π_star * β)
```

```
In [48]: 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, color='r',
             label='f0
generates')[0]
n1 = plt.hist(np.log(L1_arr[:, t_idx]), bins=bin_range, alpha=0.4, color='b',
             label='f1
generates')[0]

plt.vlines(np.log(Lβ), 0, max(n0.max(), n1.max()), linestyle='--', color='r',
           label='log($L_β$)')
plt.vlines(np.log(Lα), 0, max(n0.max(), n1.max()), linestyle='--', color='b',
```

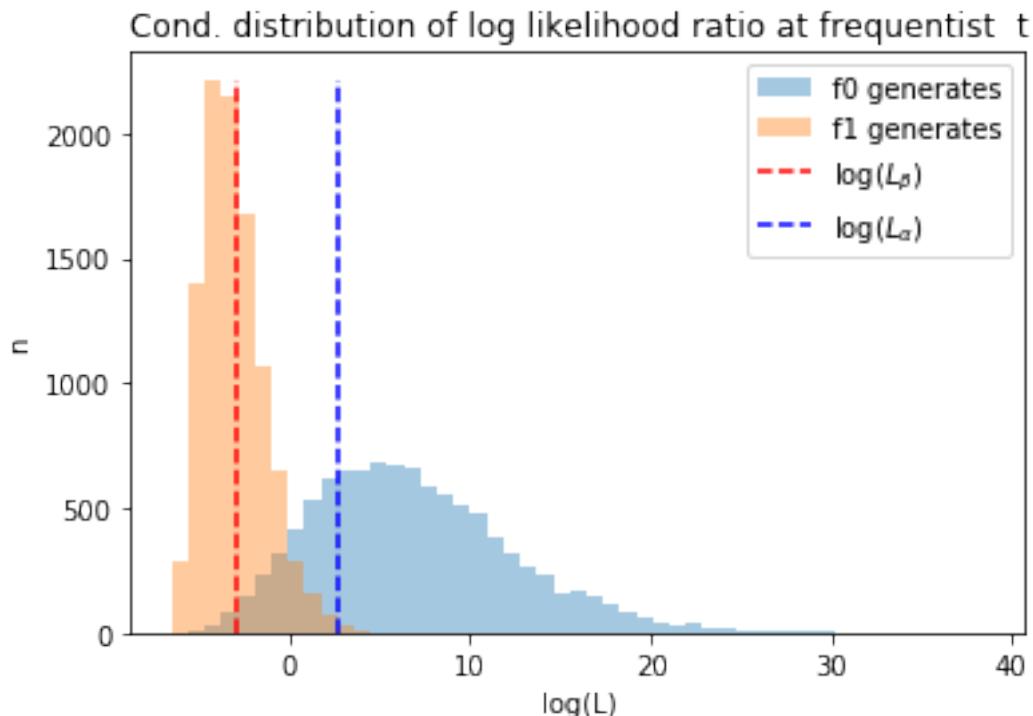
```

label='log($L_\alpha$)')
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.

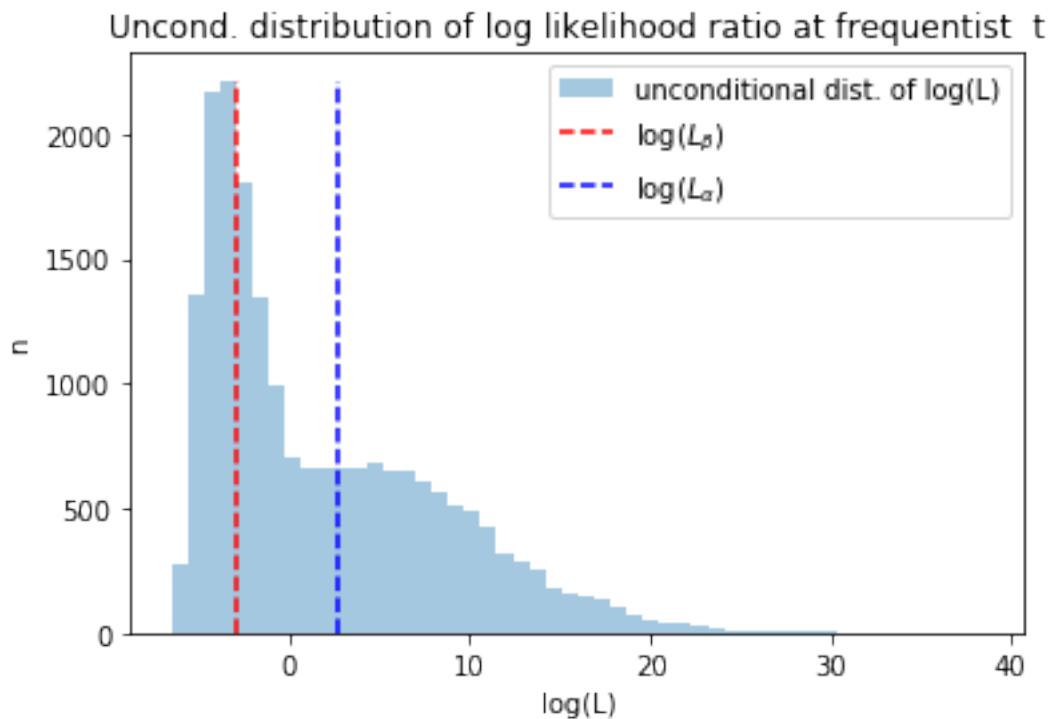
```

In [49]: 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(L_beta), 0, max(n0.max(), n1.max()), linestyle='--', color='r',
           label='log($L_\beta$)')
plt.vlines(np.log(L_alpha), 0, max(n0.max(), n1.max()), linestyle='--', color='b',
           label='log($L_\alpha$)')
plt.legend()

plt.xlabel('log(L)')
plt.ylabel('n')
plt.title('Uncond. distribution of log likelihood ratio at frequentist t')

plt.show()

```



Part VI

LQ Control

Chapter 42

LQ Control: Foundations

42.1 Contents

- Overview 42.2
- Introduction 42.3
- Optimality – Finite Horizon 42.4
- Implementation 42.5
- Extensions and Comments 42.6
- Further Applications 42.7
- Exercises 42.8
- Solutions 42.9

In addition to what's in Anaconda, this lecture will need the following libraries:

```
In [1]: !pip install quantecon
```

42.2 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 [this lecture](#))

For additional reading on LQ control, see, for example,

- [72], chapter 5
- [50], chapter 4
- [56], 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:

```
In [2]: import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from quantecon import LQ

/home/ubuntu/anaconda3/lib/python3.7/site-packages/numba/np/ufunc/parallel.py:
 355:
NumbaWarning: The TBB threading layer requires TBB version 2019.5 or later i.e.,
TBB_INTERFACE_VERSION >= 11005. Found TBB_INTERFACE_VERSION = 11004. The TBB
  threading
layer is disabled.
warnings.warn(problem)
```

42.3 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.

42.3.1 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

$$x_{t+1} = Ax_t + Bu_t + Cw_{t+1}, \quad t = 0, 1, 2, \dots \quad (1)$$

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 $\mathbb{E}w_tw_t' = I$, 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 (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

$$a_{t+1} = (1 + r)a_t - u_t - \bar{c} + \sigma w_{t+1} + \mu \tag{2}$$

How can we write this new system in the form of equation (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

$$\begin{pmatrix} a_{t+1} \\ 1 \end{pmatrix} = \begin{pmatrix} 1+r & -\bar{c} + \mu \\ 0 & 1 \end{pmatrix} \begin{pmatrix} a_t \\ 1 \end{pmatrix} + \begin{pmatrix} -1 \\ 0 \end{pmatrix} u_t + \begin{pmatrix} \sigma \\ 0 \end{pmatrix} w_{t+1} \quad (3)$$

then the first row is equivalent to (2).

Moreover, the model is now linear and can be written in the form of (1) by setting

$$x_t := \begin{pmatrix} a_t \\ 1 \end{pmatrix}, \quad A := \begin{pmatrix} 1+r & -\bar{c} + \mu \\ 0 & 1 \end{pmatrix}, \quad B := \begin{pmatrix} -1 \\ 0 \end{pmatrix}, \quad C := \begin{pmatrix} \sigma \\ 0 \end{pmatrix} \quad (4)$$

In effect, we've bought ourselves linearity by adding another state.

42.3.2 Preferences

In the LQ model, the aim is to minimize flow of losses, where time- t loss is given by the quadratic expression

$$x_t' R x_t + u_t' Q u_t \quad (5)$$

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 [50] 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 (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

$$x_t' Rx_t + u_t' Qu_t = u_t^2 = (c_t - \bar{c})^2$$

Under this specification, the household's current loss is the squared deviation of consumption from the ideal level \bar{c} .

42.4 Optimality – Finite Horizon

Let's now be precise about the optimization problem we wish to consider, and look at how to solve it.

42.4.1 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

$$\mathbb{E} \left\{ \sum_{t=0}^{T-1} \beta^t (x_t' Rx_t + u_t' Qu_t) + \beta^T x_T' R_f x_T \right\} \quad (6)$$

subject to the law of motion (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 .

42.4.2 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 .

42.4.3 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

$$\min_u \{x'_{T-1} Rx_{T-1} + u' Qu + \beta \mathbb{E} J_T(Ax_{T-1} + Bu + Cw_T)\}$$

At this stage, it is convenient to define the function

$$J_{T-1}(x) = \min_u \{x' Rx + u' Qu + \beta \mathbb{E} J_T(Ax + Bu + Cw_T)\} \quad (7)$$

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 $x_{T-1} = Ax_{T-2} + Bu + Cw_{T-1}$, 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

$$\min_u \{x'_{T-2} Rx_{T-2} + u' Qu + \beta \mathbb{E} J_{T-1}(Ax_{T-2} + Bu + Cw_{T-1})\}$$

Letting

$$J_{T-2}(x) = \min_u \{x' Rx + u' Qu + \beta \mathbb{E} J_{T-1}(Ax + Bu + Cw_{T-1})\}$$

the pattern for backward induction is now clear.

In particular, we define a sequence of value functions $\{J_0, \dots, J_T\}$ via

$$J_{t-1}(x) = \min_u \{x' Rx + u' Qu + \beta \mathbb{E} J_t(Ax + Bu + Cw_t)\} \quad \text{and} \quad J_T(x) = x' R_f x$$

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, (7) becomes

$$J_{T-1}(x) = \min_u \{x' Rx + u' Qu + \beta \mathbb{E}(Ax + Bu + Cw_T)' P_T (Ax + Bu + Cw_T)\} \quad (8)$$

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

$$u = -(Q + \beta B' P_T B)^{-1} \beta B' P_T A x \quad (9)$$

Plugging this back into (8) and rearranging yields

$$J_{T-1}(x) = x' P_{T-1} x + d_{T-1}$$

where

$$P_{T-1} = R - \beta^2 A' P_T B (Q + \beta B' P_T B)^{-1} B' P_T A + \beta A' P_T A \quad (10)$$

and

$$d_{T-1} := \beta \operatorname{trace}(C' P_T C) \quad (11)$$

(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

$$P_{t-1} = R - \beta^2 A' P_t B (Q + \beta B' P_t B)^{-1} B' P_t A + \beta A' P_t A \quad \text{with} \quad P_T = R_f \quad (12)$$

and

$$d_{t-1} = \beta(d_t + \text{trace}(C' P_t C)) \quad \text{with} \quad d_T = 0 \quad (13)$$

Recalling (9), the minimizers from these backward steps are

$$u_t = -F_t x_t \quad \text{where} \quad F_t := (Q + \beta B' P_{t+1} B)^{-1} \beta B' P_{t+1} A \quad (14)$$

These are the linear optimal control policies we discussed above.

In particular, the sequence of controls given by (14) and (1) solves our finite horizon LQ problem.

Rephrasing this more precisely, the sequence u_0, \dots, u_{T-1} given by

$$u_t = -F_t x_t \quad \text{with} \quad x_{t+1} = (A - BF_t)x_t + Cw_{t+1} \quad (15)$$

for $t = 0, \dots, T - 1$ attains the minimum of (6) subject to our constraints.

42.5 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 (12), (13) and (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

42.5.1 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, [38] or [82].)

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

$$\mathbb{E} \left\{ \sum_{t=0}^{T-1} \beta^t (c_t - \bar{c})^2 + \beta^T q a_T^2 \right\} \quad (16)$$

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 (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 (4).

To match with this state and control, the objective function (16) can be written in the form of (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 (12) and (14).

After generating shocks w_1, \dots, w_T , the dynamics for assets and consumption can be simulated via (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.

In [3]: # Model parameters

```
r = 0.05
β = 1/(1 + r)
T = 45
c_bar = 2
σ = 0.25
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 + mu],
      [0, 1]]
B = [[-1],
      [0]]
C = [[sigma],
      [0]]


# Compute solutions and simulate
lq = LQ(Q, R, A, B, C, beta=beta, 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 = sigma * wp[0, 1:] + mu # 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}

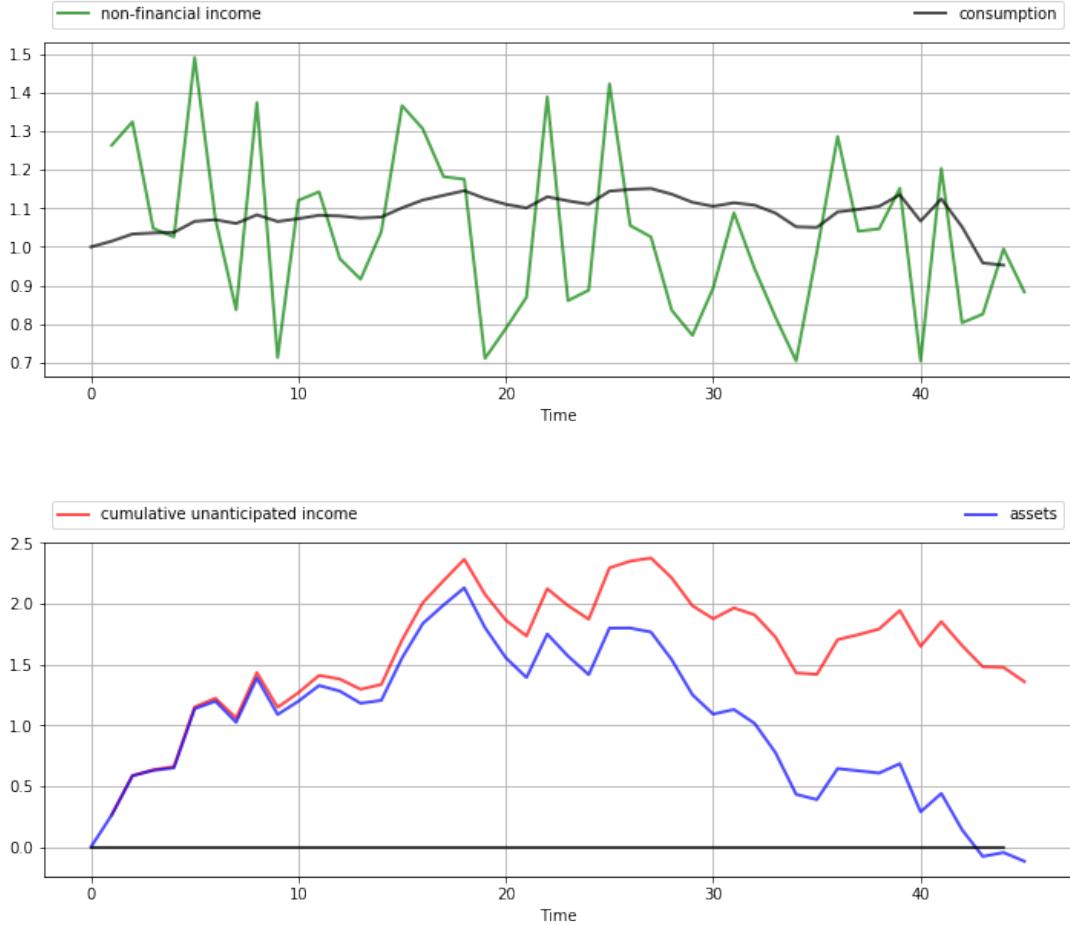
axes[0].plot(list(range(1, T+1)), income, 'g-', label="non-financial
income",
             **p_args)
axes[0].plot(list(range(T)), c, 'k-', label="consumption", **p_args)

axes[1].plot(list(range(1, T+1)), np.cumsum(income - mu), '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.

```
In [4]: # 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 = sigma * wp[0, 1:] + mu # y_t

# Plot results
n_rows = 2
fig, axes = plt.subplots(n_rows, 1, figsize=(12, 10))

plt.subplots_adjust(hspace=0.5)

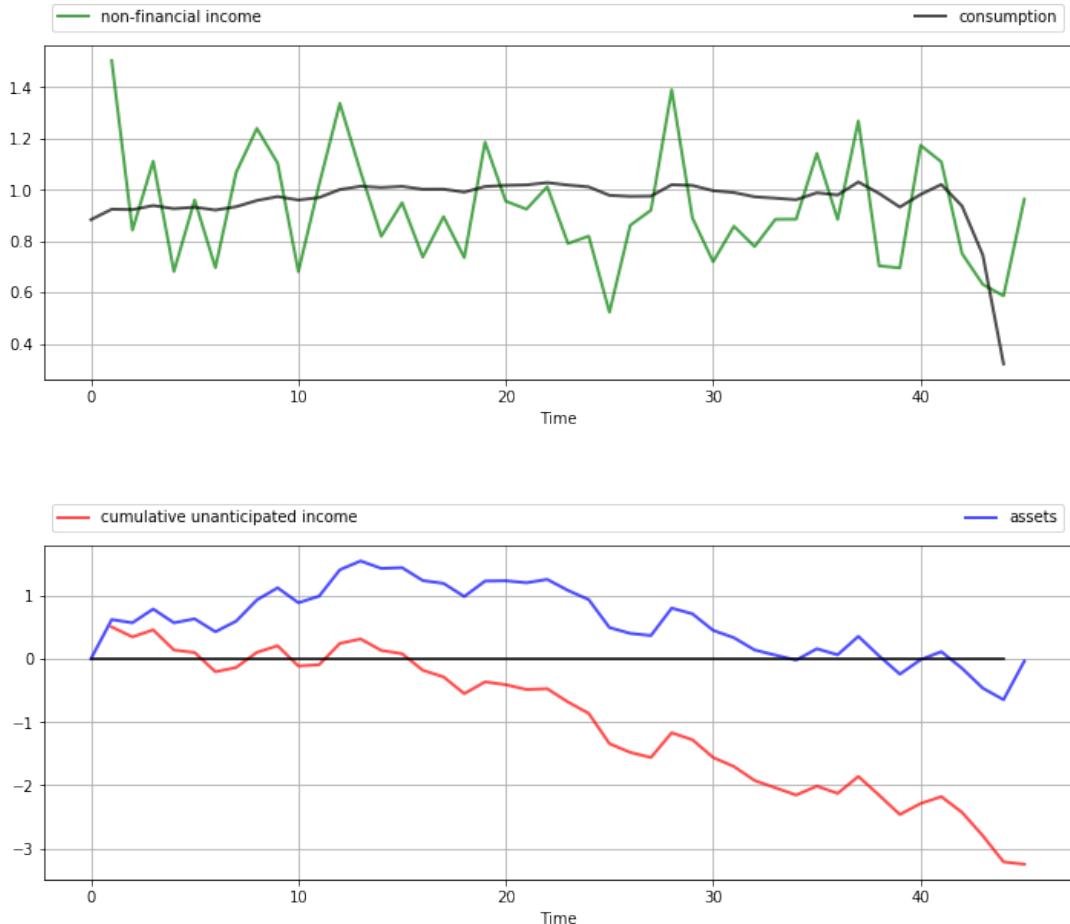
bbox = (0., 1.02, 1., .102)
legend_args = {'bbox_to_anchor': bbox, 'loc': 3, 'mode': 'expand'}
p_args = {'lw': 2, 'alpha': 0.7}

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 - mu), '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.

42.6 Extensions and Comments

Let's now consider a number of standard extensions to the LQ problem treated above.

42.6.1 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 [51], section 2.4.

42.6.2 Adding a Cross-Product Term

In some LQ problems, preferences include a cross-product term $u_t'Nx_t$, so that the objective function becomes

$$\mathbb{E} \left\{ \sum_{t=0}^{T-1} \beta^t (x_t' Rx_t + u_t' Qu_t + 2u_t' Nx_t) + \beta^T x_T' R_f x_T \right\} \quad (17)$$

Our results extend to this case in a straightforward way.

The sequence $\{P_t\}$ from (12) becomes

$$P_{t-1} = R - (\beta B' P_t A + N)' (Q + \beta B' P_t B)^{-1} (\beta B' P_t A + N) + \beta A' P_t A \quad \text{with} \quad P_T = R_f \quad (18)$$

The policies in (14) are modified to

$$u_t = -F_t x_t \quad \text{where} \quad F_t := (Q + \beta B' P_{t+1} B)^{-1} (\beta B' P_{t+1} A + N) \quad (19)$$

The sequence $\{d_t\}$ is unchanged from (13).

We leave interested readers to confirm these results (the calculations are long but not overly difficult).

42.6.3 Infinite Horizon

Finally, we consider the infinite horizon case, with **cross-product term**, unchanged dynamics and objective function given by

$$\mathbb{E} \left\{ \sum_{t=0}^{\infty} \beta^t (x_t' Rx_t + u_t' Qu_t + 2u_t' Nx_t) \right\} \quad (20)$$

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 $u_t = -Fx_t$ 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](#).

$$P = R - (\beta B' PA + N)' (Q + \beta B' PB)^{-1} (\beta B' PA + N) + \beta A' PA \quad (21)$$

Equation (21) is also called the *LQ Bellman equation*, and the map that sends a given P into the right-hand side of (21) is called the *LQ Bellman operator*.

The stationary optimal policy for this model is

$$u = -Fx \quad \text{where} \quad F = (Q + \beta B' P B)^{-1} (\beta B' P A + N) \quad (22)$$

The sequence $\{d_t\}$ from (13) is replaced by the constant value

$$d := \text{trace}(C' P C) \frac{\beta}{1 - \beta} \quad (23)$$

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](#).

42.6.4 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 (22) or (19).

It follows that we can ignore uncertainty when solving for optimal behavior, and plug it back in when examining optimal state dynamics.

42.7 Further Applications

42.7.1 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

$$\mathbb{E} \left\{ \sum_{t=0}^{T-1} \beta^t (c_t - \bar{c})^2 + \beta^T q a_T^2 \right\} \quad (24)$$

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

$$a_{t+1} = (1+r)a_t - u_t - \bar{c} + m_1 t + m_2 t^2 + \sigma w_{t+1} \quad (25)$$

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

$$x_t := \begin{pmatrix} a_t \\ 1 \\ t \\ t^2 \end{pmatrix}, \quad A := \begin{pmatrix} 1+r & -\bar{c} & m_1 & m_2 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 2 & 1 \end{pmatrix}, \quad B := \begin{pmatrix} -1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \quad C := \begin{pmatrix} \sigma \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad (26)$$

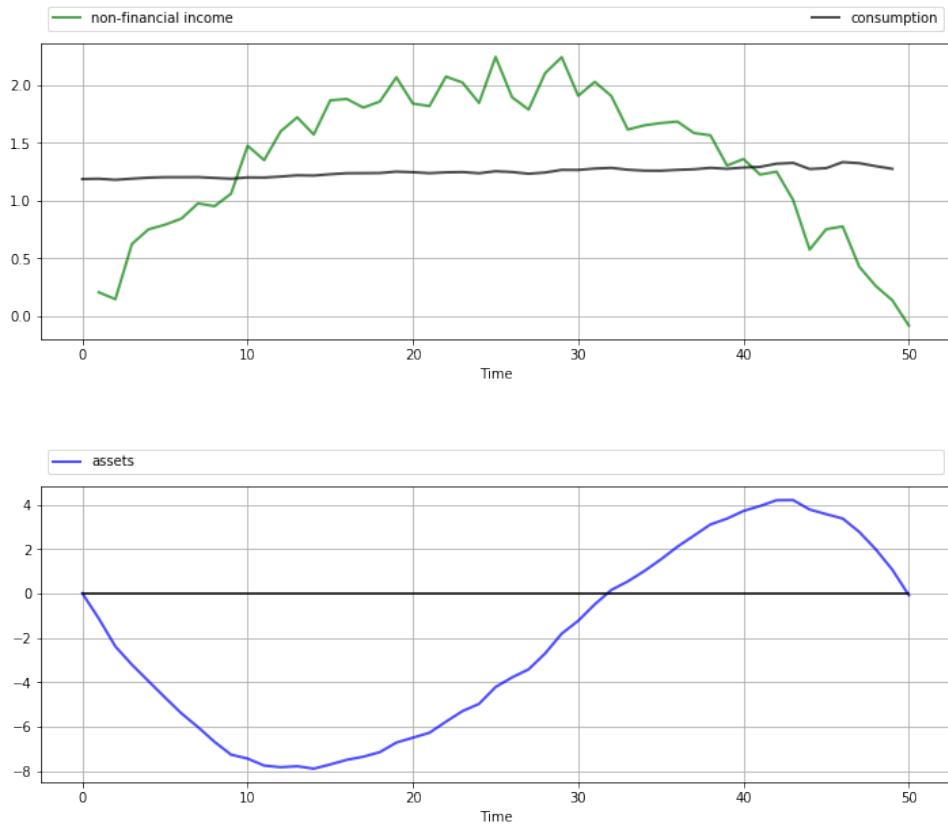
If you expand the expression $x_{t+1} = Ax_t + Bu_t + Cw_{t+1}$ using this specification, you will find that assets follow (25) as desired and that the other state variables also update appropriately.

To implement preference specification (24) we take

$$Q := 1, \quad R := \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad \text{and} \quad R_f := \begin{pmatrix} q & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad (27)$$

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 1 gives the full set of parameters used here and asks you to replicate the figure.

42.7.2 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

$$y_t = \begin{cases} p(t) + \sigma w_{t+1} & \text{if } t \leq K \\ s & \text{otherwise} \end{cases} \quad (28)$$

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 (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 (28) at K makes it very difficult to express the law of motion as a fixed-coefficient linear system.

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.)

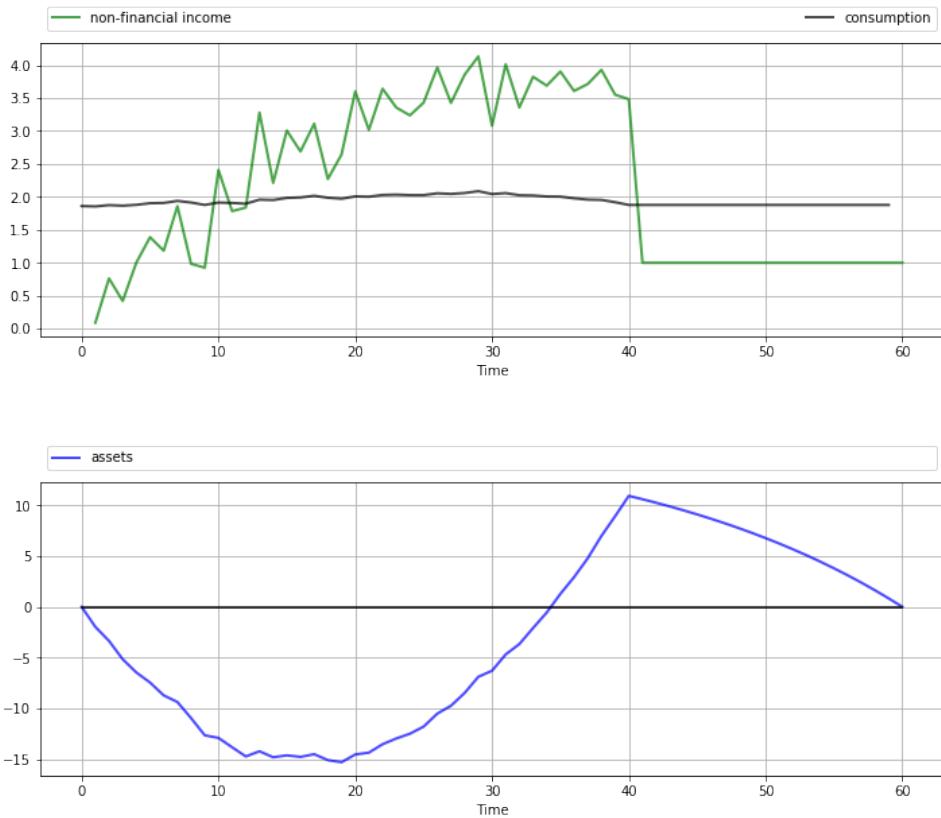
The basic idea is that although the whole problem is not a single time-invariant LQ problem, it is still a dynamic programming problem, and hence we can use appropriate Bellman equations at every stage.

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 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.

42.7.3 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

$$\mathbb{E} \left\{ \sum_{t=0}^{\infty} \beta^t \pi_t \right\} \quad \text{where} \quad \pi_t := p_t q_t - c q_t - \gamma(q_{t+1} - q_t)^2 \quad (29)$$

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

$$\bar{q}_t := \frac{a_0 - c + d_t}{2a_1}$$

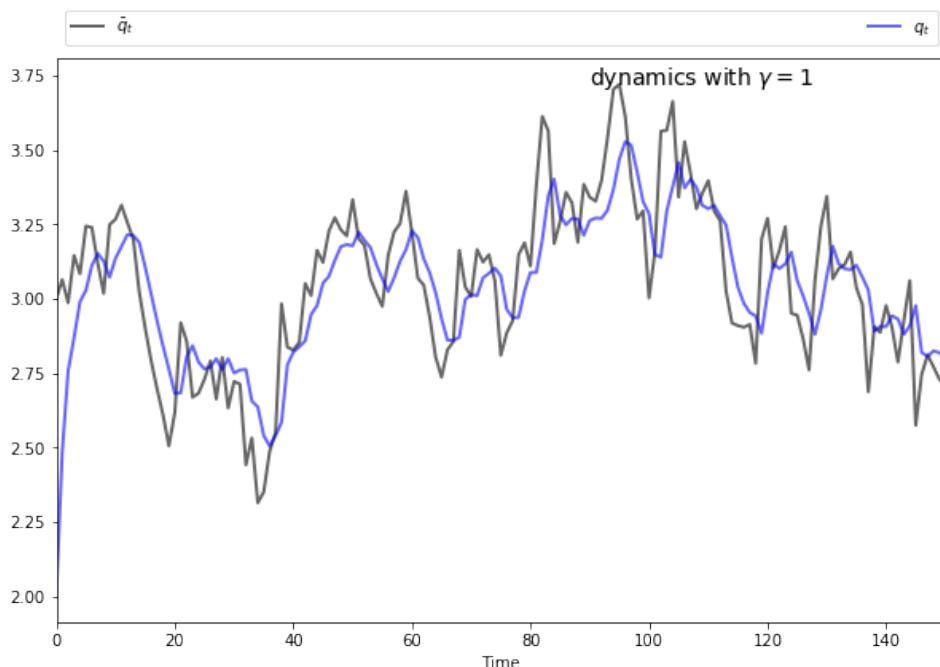
In light of this discussion, what we might expect for general γ is that

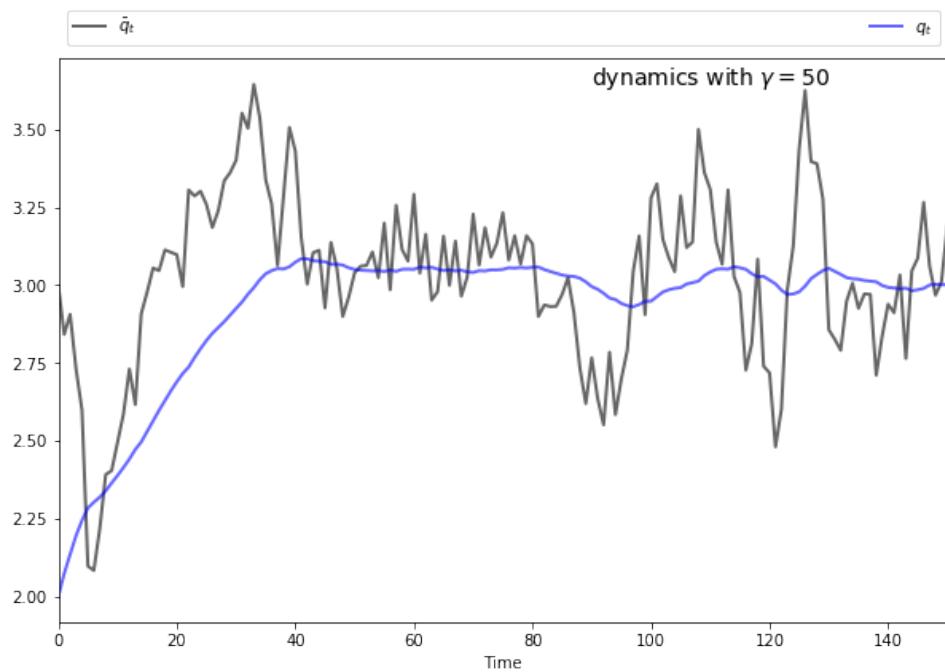
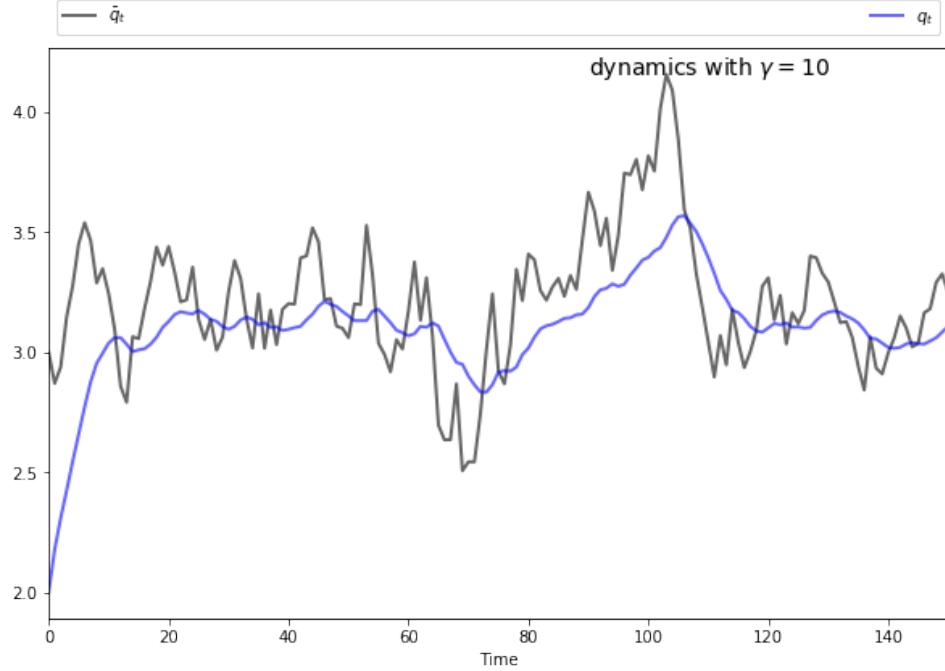
- if γ is close to zero, then q_t will track the time path of \bar{q}_t relatively closely.
- if γ is larger, then q_t will be smoother than \bar{q}_t , as the monopolist seeks to avoid adjustment costs.

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.

In (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 (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 (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

$$\min \mathbb{E} \sum_{t=0}^{\infty} \beta^t \{a_1(q_t - \bar{q}_t)^2 + \gamma u_t^2\} \quad (30)$$

It's now relatively straightforward to find R and Q such that (30) can be written as (20).

Furthermore, the matrices A , B and C from (1) can be found by writing down the dynamics of each element of the state.

Exercise 3 asks you to complete this process, and reproduce the preceding figures.

42.8 Exercises

42.8.1 Exercise 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 $q = 10^4$.

42.8.2 Exercise 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.

Some hints are as follows:

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 (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 (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.

42.8.3 Exercise 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).

42.9 Solutions

42.9.1 Exercise 1

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
- $p(T) = 0$

```
In [5]: # 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}

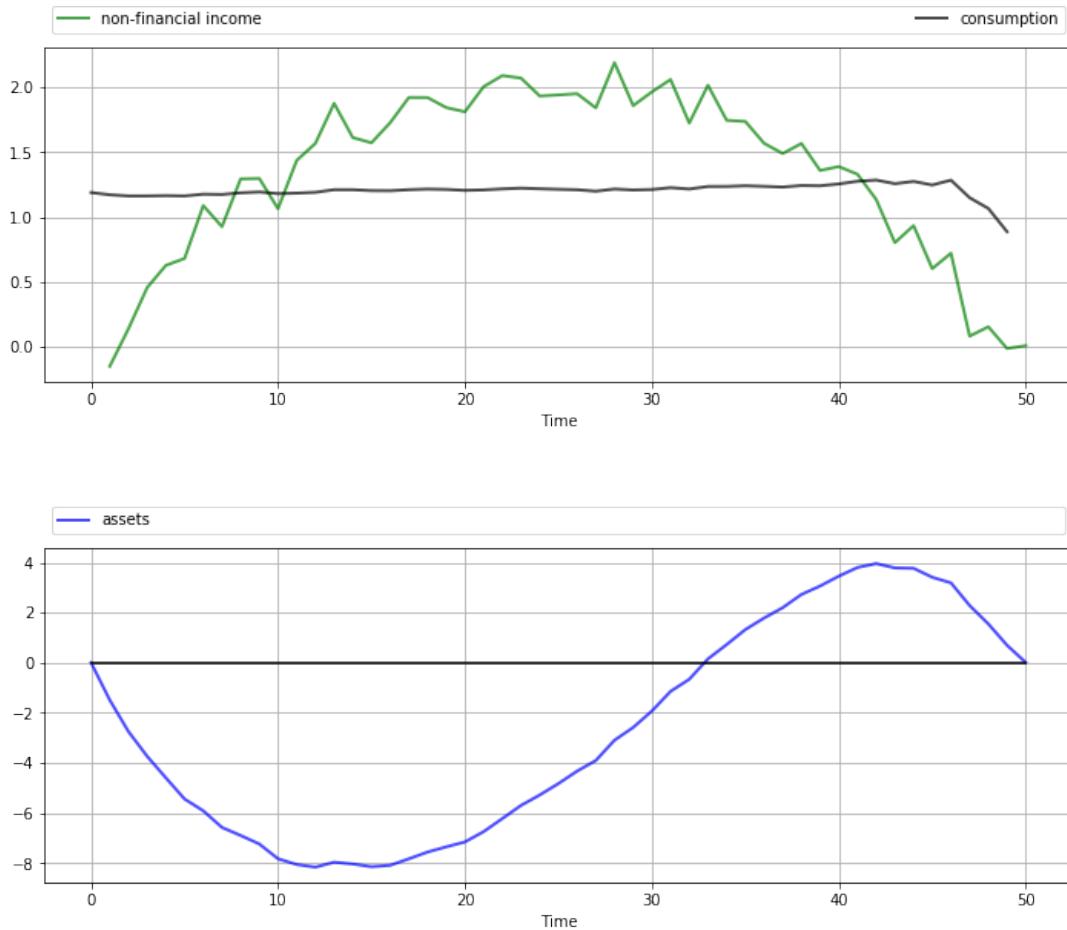
axes[0].plot(range(1, T+1), income, 'g-', label="non-financial income",
             **p_args)
axes[0].plot(range(T), c, 'k-', label="consumption", **p_args)

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()

```



42.9.2 Exercise 2

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.

In [6]: # 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.ones(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}

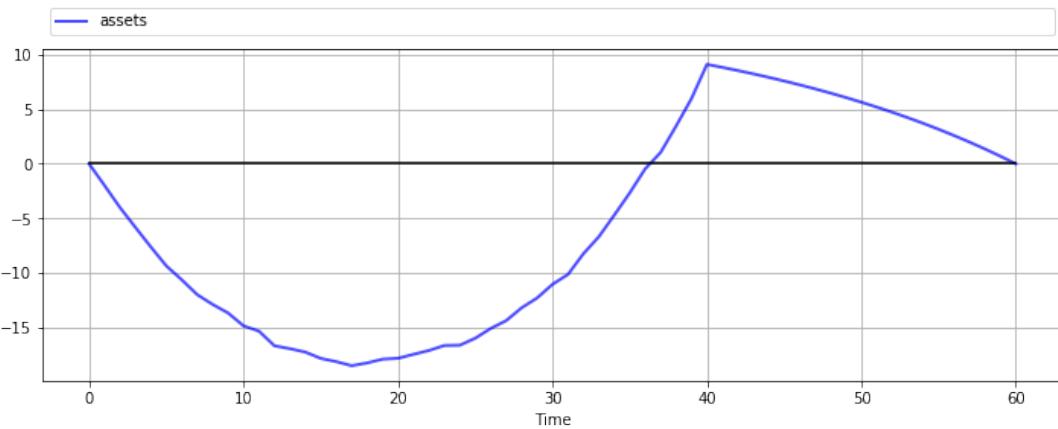
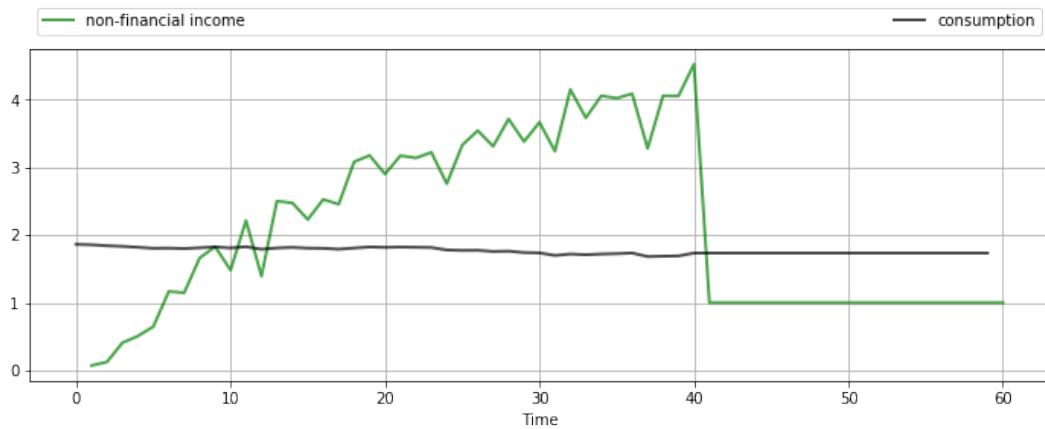
axes[0].plot(range(1, T+1), income, 'g-', label="non-financial income",
             **p_args)
axes[0].plot(range(T), c, 'k-', label="consumption", **p_args)

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()

```



42.9.3 Exercise 3

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

```
In [7]: # 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)
```

```

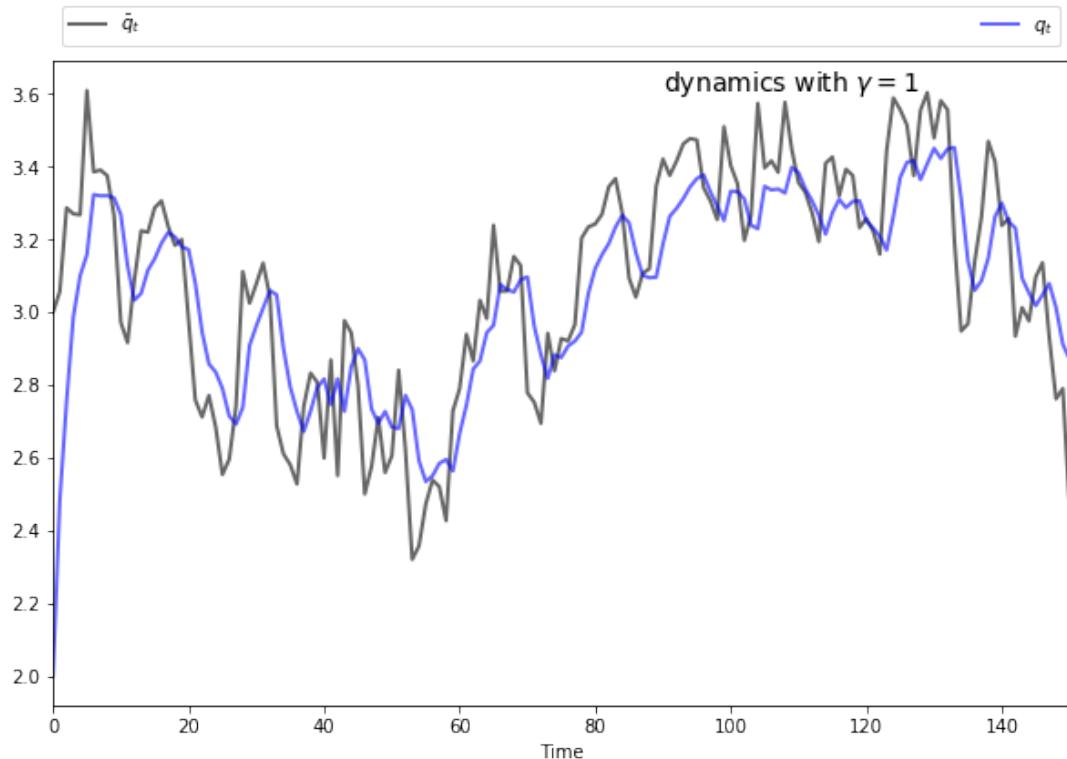
# Simulate state / control paths
x0 = (m0, 2, 1)
xp, up, wp = lq.compute_sequence(x0, ts_length=150)
q_bar = xp[0, :]
q = xp[1, :]

# 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 = f'dynamics with $\gamma = {gamma}$'
ax.text(max(time) * 0.6, 1 * q_bar.max(), s, fontsize=14)
plt.show()

```



Chapter 43

The Permanent Income Model

43.1 Contents

- Overview 43.2
- The Savings Problem 43.3
- Alternative Representations 43.4
- Two Classic Examples 43.5
- Further Reading 43.6
- Appendix: The Euler Equation 43.7

In addition to what's in Anaconda, this lecture will need the following libraries:

```
In [1]: !pip install quantecon
```

43.2 Overview

This lecture describes a rational expectations version of the famous permanent income model of Milton Friedman [38].

Robert Hall cast Friedman's model within a linear-quadratic setting [47].

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 [47] and chapter 2 of [72].

Let's start with some imports

```
In [2]: import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np
import random
from numba import njit
```

43.3 The Savings Problem

In this section, we state and solve the savings and consumption problem faced by the consumer.

43.3.1 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

$$\mathbb{E}_t[X_{t+1}] = X_t, \quad t = 0, 1, 2, \dots$$

Here $\mathbb{E}_t := \mathbb{E}[\cdot | \mathcal{F}_t]$ 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 $\mathcal{F}_t = \{X_t, X_{t-1}, \dots, X_0\}$.

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](#)).

43.3.2 The Decision Problem

A consumer has preferences over consumption streams that are ordered by the utility functional

$$\mathbb{E}_0 \left[\sum_{t=0}^{\infty} \beta^t u(c_t) \right] \tag{1}$$

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 (1) by choosing a consumption, borrowing plan $\{c_t, b_{t+1}\}_{t=0}^{\infty}$ subject to the sequence of budget constraints

$$c_t + b_t = \frac{1}{1+r} b_{t+1} + y_t \quad t \geq 0 \quad (2)$$

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.

43.3.3 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](#)

$$\begin{aligned} z_{t+1} &= Az_t + Cw_{t+1} \\ y_t &= Uz_t \end{aligned} \quad (3)$$

where

- $\{w_t\}$ is an IID vector process with $\mathbb{E}w_t = 0$ and $\mathbb{E}w_tw_t' = I$.
- 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

$$\mathbb{E}_0 \left[\sum_{t=0}^{\infty} \beta^t b_t^2 \right] < \infty \quad (4)$$

This condition rules out an always-borrow scheme that would allow the consumer to enjoy bliss consumption forever.

43.3.4 First-Order Conditions

First-order conditions for maximizing (1) subject to (2) are

$$\mathbb{E}_t[u'(c_{t+1})] = u'(c_t), \quad t = 0, 1, \dots \quad (5)$$

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, (5) has the striking implication that consumption follows a martingale:

$$\mathbb{E}_t[c_{t+1}] = c_t \quad (6)$$

(In fact, quadratic preferences are *necessary* for this conclusion Section ??.)

One way to interpret (6) is that consumption will change only when “new information” about permanent income is revealed.

These ideas will be clarified below.

43.3.5 The Optimal Decision Rule

Now let's deduce the optimal decision rule Section ??.

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 (6)
2. the period-by-period budget constraint (2), and
3. the boundary condition (4)

To accomplish this, observe first that (4) implies $\lim_{t \rightarrow \infty} \beta^{\frac{t}{2}} b_{t+1} = 0$.

Using this restriction on the debt path and solving (2) forward yields

$$b_t = \sum_{j=0}^{\infty} \beta^j (y_{t+j} - c_{t+j}) \quad (7)$$

Take conditional expectations on both sides of (7) and use the martingale property of consumption and the *law of iterated expectations* to deduce

$$b_t = \sum_{j=0}^{\infty} \beta^j \mathbb{E}_t[y_{t+j}] - \frac{c_t}{1-\beta} \quad (8)$$

Expressed in terms of c_t we get

$$c_t = (1-\beta) \left[\sum_{j=0}^{\infty} \beta^j \mathbb{E}_t[y_{t+j}] - b_t \right] = \frac{r}{1+r} \left[\sum_{j=0}^{\infty} \beta^j \mathbb{E}_t[y_{t+j}] - b_t \right] \quad (9)$$

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 $[b_t \ z_t]$.

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 (9) gives

$$c_t = \frac{r}{1+r} [U(I - \beta A)^{-1} z_t - b_t] \quad (10)$$

Using this equality to eliminate c_t in the budget constraint (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:

$$\begin{aligned}
z_{t+1} &= Az_t + Cw_{t+1} \\
b_{t+1} &= b_t + U[(I-\beta A)^{-1}(A-I)]z_t \\
y_t &= Uz_t \\
c_t &= (1-\beta)[U(I-\beta A)^{-1}z_t - b_t]
\end{aligned} \tag{11}$$

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 (11) as

$$\begin{aligned}
x_{t+1} &= \tilde{A}x_t + \tilde{C}w_{t+1} \\
\tilde{y}_t &= \tilde{U}x_t
\end{aligned} \tag{12}$$

We can use the following formulas from [linear state space models](#) to compute population mean $\mu_t = \mathbb{E}x_t$ and covariance $\Sigma_t := \mathbb{E}[(x_t - \mu_t)(x_t - \mu_t)']$

$$\mu_{t+1} = \tilde{A}\mu_t \quad \text{with } \mu_0 \text{ given} \tag{13}$$

$$\Sigma_{t+1} = \tilde{A}\Sigma_t\tilde{A}' + \tilde{C}\tilde{C}' \quad \text{with } \Sigma_0 \text{ given} \tag{14}$$

We can then compute the mean and covariance of \tilde{y}_t from

$$\begin{aligned}
\mu_{y,t} &= \tilde{U}\mu_t \\
\Sigma_{y,t} &= \tilde{U}\Sigma_t\tilde{U}'
\end{aligned} \tag{15}$$

A Simple Example with IID Income

To gain some preliminary intuition on the implications of (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

$$z_t = \begin{bmatrix} z_t^1 \\ 1 \end{bmatrix}, \quad A = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}, \quad U = [1 \ \mu], \quad C = \begin{bmatrix} \sigma \\ 0 \end{bmatrix}$$

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

$$\begin{aligned} b_t &= -\sigma \sum_{j=1}^{t-1} w_j \\ c_t &= \mu + (1 - \beta) \sigma \sum_{j=1}^t w_j \end{aligned}$$

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 $\sigma = 0.15$

```
In [3]: r = 0.05
β = 1 / (1 + r)
σ = 0.15
μ = 1
T = 60
```

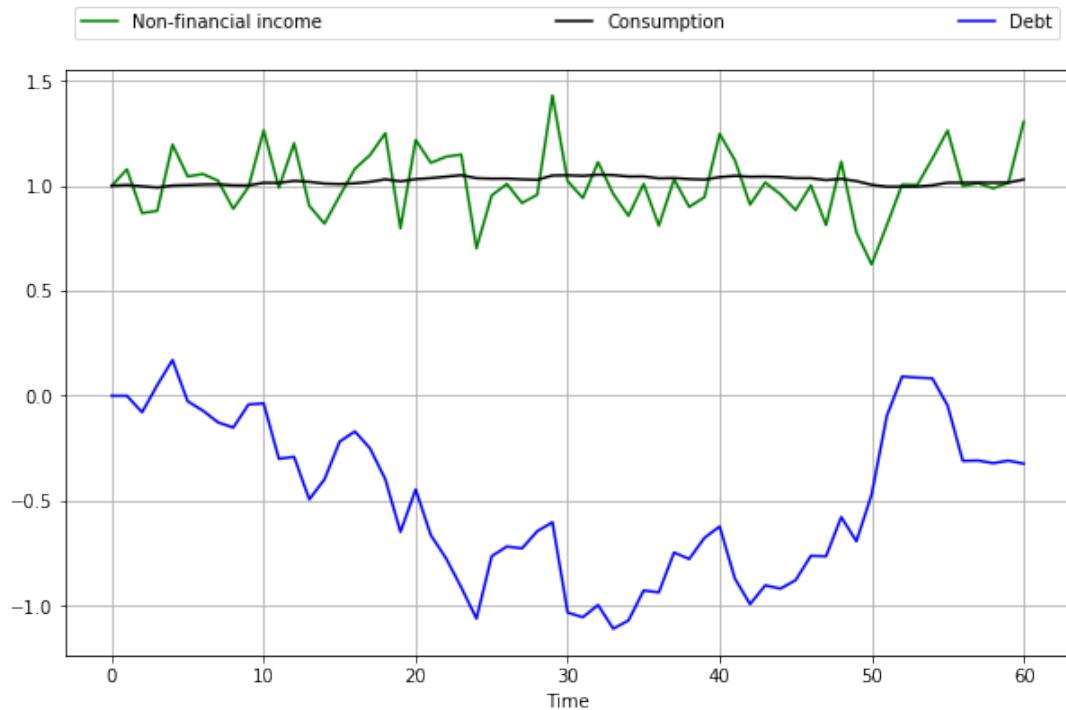
```
@njit
def time_path(T):
    w = np.random.randn(T+1) # w_0, w_1, ..., w_T
    w[0] = 0
    b = np.zeros(T+1)
    for t in range(1, T+1):
        b[t] = w[1:t].sum()
    b = -σ * b
    c = μ + (1 - β) * (σ * w - b)
    return w, b, c

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()
```



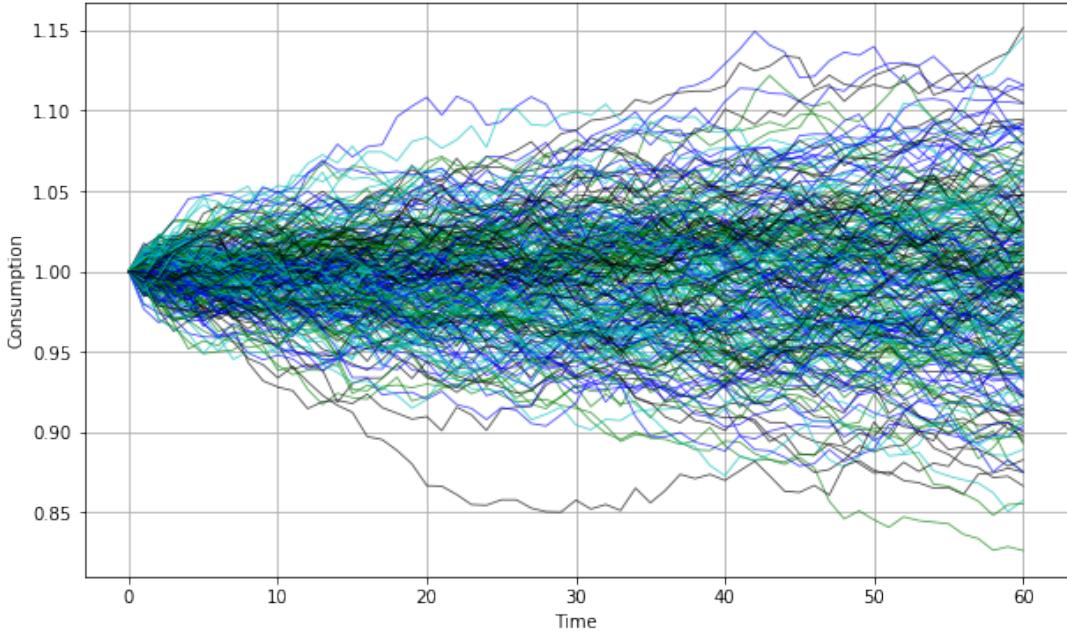
Observe that consumption is considerably smoother than income.

The figure below shows the consumption paths of 250 consumers with independent income streams

```
In [4]: 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()
```



43.4 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.

43.4.1 Hall's Representation

Hall [47] suggested an insightful way to summarize the implications of LQ permanent income theory.

First, to represent the solution for b_t , shift (9) forward one period and eliminate b_{t+1} by using (2) to obtain

$$c_{t+1} = (1 - \beta) \sum_{j=0}^{\infty} \beta^j \mathbb{E}_{t+1}[y_{t+j+1}] - (1 - \beta) [\beta^{-1}(c_t + b_t - y_t)]$$

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

$$c_{t+1} - c_t = (1 - \beta) \sum_{j=0}^{\infty} \beta^j \{ \mathbb{E}_{t+1}[y_{t+j+1}] - \mathbb{E}_t[y_{t+j+1}] \} \quad (16)$$

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 (16) and (8), which we repeat:

$$b_t = \sum_{j=0}^{\infty} \beta^j \mathbb{E}_t[y_{t+j}] - \frac{1}{1-\beta} c_t \quad (17)$$

Equation (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 (3) and substituting into (16) and (17) gives the following representation for the consumer's optimum decision rule:

$$\begin{aligned} c_{t+1} &= c_t + (1 - \beta)U(I - \beta A)^{-1} C w_{t+1} \\ b_t &= U(I - \beta A)^{-1} z_t - \frac{1}{1 - \beta} c_t \\ y_t &= U z_t \\ z_{t+1} &= A z_t + C w_{t+1} \end{aligned} \quad (18)$$

Representation (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 (6).

43.4.2 Cointegration

Representation (18) reveals that the joint process $\{c_t, b_t\}$ possesses the property that Engle and Granger [33] 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. Section ??.

Despite this, both c_t and b_t will be non-stationary because they have unit roots (see (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 (18) we have

$$(1 - \beta)b_t + c_t = (1 - \beta)U(I - \beta A)^{-1}z_t \quad (19)$$

Hence the linear combination $(1 - \beta)b_t + c_t$ is asymptotically stationary.

Accordingly, Granger and Engle would call $[(1 - \beta) \ 1]$ a **cointegrating vector** for the state.

When applied to the nonstationary vector process $[b_t \ c_t]'$, it yields a process that is asymptotically stationary.

Equation (19) can be rearranged to take the form

$$(1 - \beta)b_t + c_t = (1 - \beta)\mathbb{E}_t \sum_{j=0}^{\infty} \beta^j y_{t+j} \quad (20)$$

Equation (20) asserts that the *cointegrating residual* on the left side equals the conditional expectation of the geometric sum of future incomes on the right Section ??.

43.4.3 Cross-Sectional Implications

Consider again (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

$$c_{t+1} = c_t + (1 - \beta)U(I - \beta A)^{-1}Cw_{t+1} \quad (21)$$

or

$$c_t = c_0 + \sum_{j=1}^t \hat{w}_j \quad \text{for } \hat{w}_{t+1} := (1 - \beta)U(I - \beta A)^{-1}Cw_{t+1}$$

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

$$\text{Var}[c_t] = \text{Var}[c_0] + t \hat{\sigma}^2 \quad (22)$$

where

$$\hat{\sigma}^2 := (1 - \beta)^2 U(I - \beta A)^{-1} C C' (I - \beta A')^{-1} U'$$

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 (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., [27], [103]).

43.4.4 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 $j \geq 1$.

43.4.5 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 (3) has the moving average representation

$$y_{t+1} = d(L)w_{t+1} \quad (23)$$

where

- $d(L) = \sum_{j=0}^{\infty} d_j L^j$ for some sequence d_j , where L is the lag operator Section ??
- at time t , the consumer has an information set Section ?? $w^t = [w_t, w_{t-1}, \dots]$

Notice that

$$y_{t+j} - \mathbb{E}_t[y_{t+j}] = d_0 w_{t+j} + d_1 w_{t+j-1} + \dots + d_{j-1} w_{t+1}$$

It follows that

$$\mathbb{E}_{t+1}[y_{t+j}] - \mathbb{E}_t[y_{t+j}] = d_{j-1} w_{t+1} \quad (24)$$

Using (24) in (16) gives

$$c_{t+1} - c_t = (1 - \beta)d(\beta)w_{t+1} \quad (25)$$

The object $d(\beta)$ is the **present value of the moving average coefficients** in the representation for the endowment process y_t .

43.5 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 .

43.5.1 Example 1

Assume as before that the consumer observes the state z_t at time t .

In view of (18) we have

$$c_{t+1} - c_t = \sigma_1 w_{1t+1} + (1 - \beta) \sigma_2 w_{2t+1} \quad (26)$$

Formula (26) shows how an increment $\sigma_1 w_{1t+1}$ to the permanent component of income z_{1t+1} leads to

- 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 (11) to this example shows that

$$b_{t+1} - b_t = -z_{2t} = -\sigma_2 w_{2t} \quad (27)$$

This confirms that none of $\sigma_1 w_{1t}$ is saved, while all of $\sigma_2 w_{2t}$ is saved.

The next figure illustrates these very different reactions to transitory and permanent income shocks using impulse-response functions

```
In [5]: r = 0.05
β = 1 / (1 + r)
S = 5 # Impulse date
σ1 = σ2 = 0.15

@njit
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 = ['transitory', 'permanent']

L = 0.175

for ax, truefalse, title in zip(axes, (True, False), titles):
    b, c = time_path(T=20, permanent=truefalse)
```

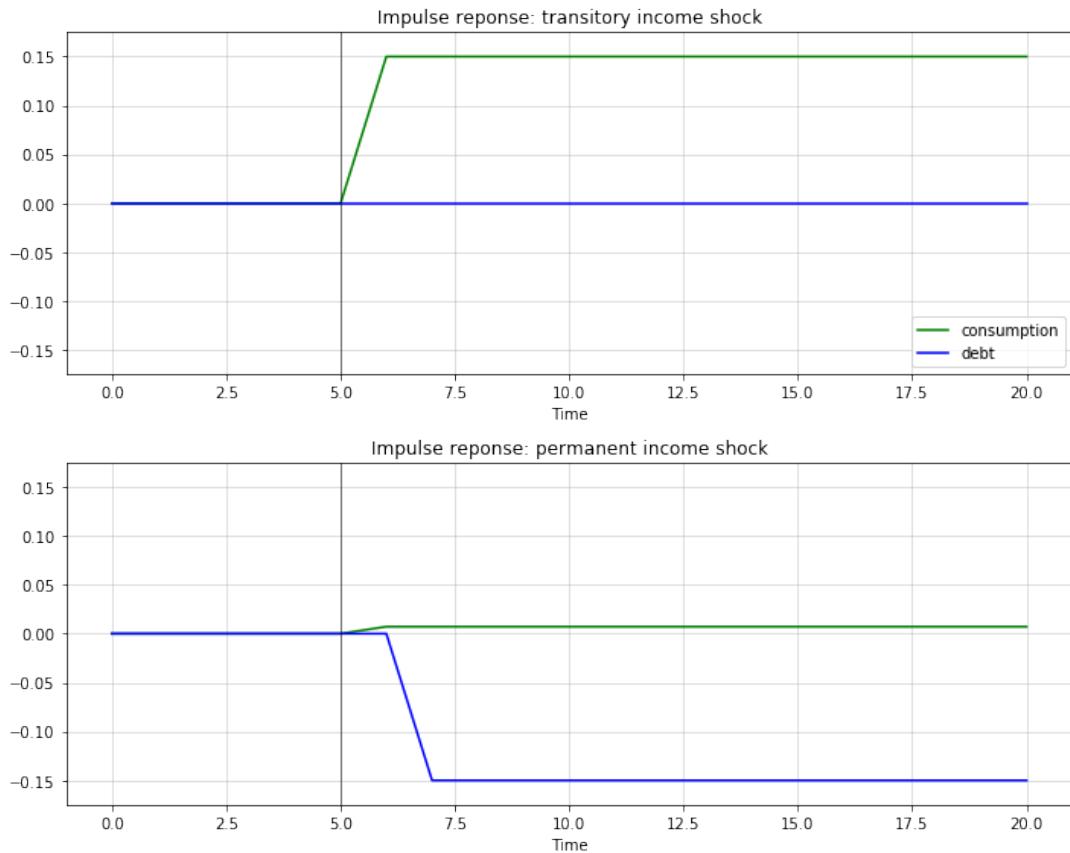
```

    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()

```



43.5.2 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 (18).

The discussion in sections 2.9.1 and 2.11.3 of [72] shows that the pertinent state space representation for y_t is

$$\begin{bmatrix} y_{t+1} \\ a_{t+1} \end{bmatrix} = \begin{bmatrix} 1 & -(1-K) \\ 0 & 0 \end{bmatrix} \begin{bmatrix} y_t \\ a_t \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix} a_{t+1}$$

$$y_t = [1 \ 0] \begin{bmatrix} y_t \\ a_t \end{bmatrix}$$

where

- $K :=$ the stationary Kalman gain
- $a_t := y_t - E[y_t | y_{t-1}, \dots, y_0]$

In the same discussion in [72] 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 (18) implies

$$c_{t+1} - c_t = [1 - \beta(1 - K)]a_{t+1} \quad (28)$$

where the endowment process can now be represented in terms of the univariate innovation to y_t as

$$y_{t+1} - y_t = a_{t+1} - (1 - K)a_t \quad (29)$$

Equation (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 (29), the first difference of income is a first-order moving average.

Equation (28) asserts that the first difference of consumption is IID.

Application of formula to this example shows that

$$b_{t+1} - b_t = (K - 1)a_t \quad (30)$$

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.

43.6 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., [48], [86], [26], [20].

43.7 Appendix: The Euler Equation

Where does the first-order condition (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 (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 (5) in the two-period case.

The proof for the general case is similar.

Footnotes

[1] A linear marginal utility is essential for deriving (6) from (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 (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 (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.

[3] Representation (3) implies that $d(L) = U(I - AL)^{-1}C$.

[4] This would be the case if, for example, the **spectral radius** of A is strictly less than one.

[5] 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.

[6] See [61], [69], [70] for interesting applications of related ideas.

Chapter 44

Permanent Income II: LQ Techniques

44.1 Contents

- Overview [44.2](#)
- Setup [44.3](#)
- The LQ Approach [44.4](#)
- Implementation [44.5](#)
- Two Example Economies [44.6](#)

In addition to what's in Anaconda, this lecture will need the following libraries:

```
In [1]: !pip install quantecon
```

44.2 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 [\[47\]](#) 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 [\[8\]](#) 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 [14].

(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:

```
In [2]: import quantecon as qe
import numpy as np
import scipy.linalg as la
import matplotlib.pyplot as plt
%matplotlib inline

/home/ubuntu/anaconda3/lib/python3.7/site-packages/numba/np/ufunc/parallel.py:
 355:
NumbaWarning: The TBB threading layer requires TBB version 2019.5 or later i.e.,
TBB_INTERFACE_VERSION >= 11005. Found TBB_INTERFACE_VERSION = 11004. The TBB
  threading
layer is disabled.
  warnings.warn(problem)
```

44.3 Setup

Let's recall the basic features of the model discussed in the [permanent income model](#).

Consumer preferences are ordered by

$$E_0 \sum_{t=0}^{\infty} \beta^t u(c_t) \quad (1)$$

where $u(c) = -(c - \gamma)^2$.

The consumer maximizes (1) by choosing a consumption, borrowing plan $\{c_t, b_{t+1}\}_{t=0}^{\infty}$ subject to the sequence of budget constraints

$$c_t + b_t = \frac{1}{1+r} b_{t+1} + y_t, \quad t \geq 0 \quad (2)$$

and the no-Ponzi condition

$$E_0 \sum_{t=0}^{\infty} \beta^t b_t^2 < \infty \quad (3)$$

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

$$\begin{aligned} z_{t+1} &= Az_t + Cw_{t+1} \\ y_t &= Uz_t \end{aligned} \quad (4)$$

The restrictions on the shock process and parameters are the same as in our [previous lecture](#).

44.3.1 Digression on a Useful Isomorphism

The LQ permanent income model of consumption is mathematically isomorphic with a version of Barro's [8] 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.

44.3.2 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 (4), as discussed in our lecture on [linear models](#).

To do so we take

$$z_t = \begin{bmatrix} 1 \\ y_t \\ y_{t-1} \end{bmatrix}, \quad A = \begin{bmatrix} 1 & 0 & 0 \\ \alpha & \rho_1 & \rho_2 \\ 0 & 1 & 0 \end{bmatrix}, \quad C = \begin{bmatrix} 0 \\ \sigma \\ 0 \end{bmatrix}, \quad \text{and} \quad U = [0 \ 1 \ 0]$$

44.4 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

44.4.1 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

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t \{x_t' R x_t + u_t' Q u_t\},$$

subject to x_0 given and the law of motion

$$x_{t+1} = \tilde{A}x_t + \tilde{B}u_t + \tilde{C}w_{t+1}, \quad t \geq 0, \tag{5}$$

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 (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 $x_{t+1} = (\tilde{A} - \tilde{B}F)x_t + \tilde{C}w_{t+1}$.

44.4.2 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 $U_\gamma := [\gamma \ 0 \ 0]$, we can write the state dynamics as in (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 (5) match the dynamics of z_t and b_t described above.

To map utility into the quadratic form $x_t'Rx_t + u_t'Qu_t$ 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 (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.

44.5 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.

44.5.1 The Exogenous Nonfinancial Income Process

First, we create the objects for the optimal linear regulator

```
In [3]: # 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 = Lz.stationary_distributions()
```

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

```
In [4]: 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

```
In [5]: 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

```
In [6]: 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

```
In [7]: P, F, d = lqpi.stationary_values() # Compute value function and decision rule
ABF = ALQ - BLQ @ F # Form closed loop system
```

44.5.2 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

```
In [8]: # 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

```
In [9]: ABF - A_LSS
```

```
Out[9]: 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],
   [-9.51248178e-06,  9.51247915e-08,  3.36117263e-17,
  -1.99999923e-08]])
```

Now compare pertinent elements of `c_pol` and `F`

```
In [10]: print(c_pol, "\n", -F)
```

```
[[6.55172414e+01 3.44827586e-01 1.68058632e-18]
 [[ 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.

44.6 Two Example Economies

In the spirit of Bewley models [14], 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.

44.6.1 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

```
In [11]: lss = qe.LinearStateSpace(A_LSS, C_LSS, G_LSS, mu_0=mu_0, Sigma_0=Sigma_0)
```

44.6.2 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.

```
In [12]: def income_consumption_debt_series(A, C, G, mu_0, Sigma_0, T=150, npaths=25):
    """
    This function takes initial conditions ( $\mu_0$ ,  $\Sigma_0$ ) 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=mu_0, Sigma_0=Sigma_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], Σ_y[1, 1]
    debt_mean[t], debt_var[t] = μ_x[3], 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")
    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

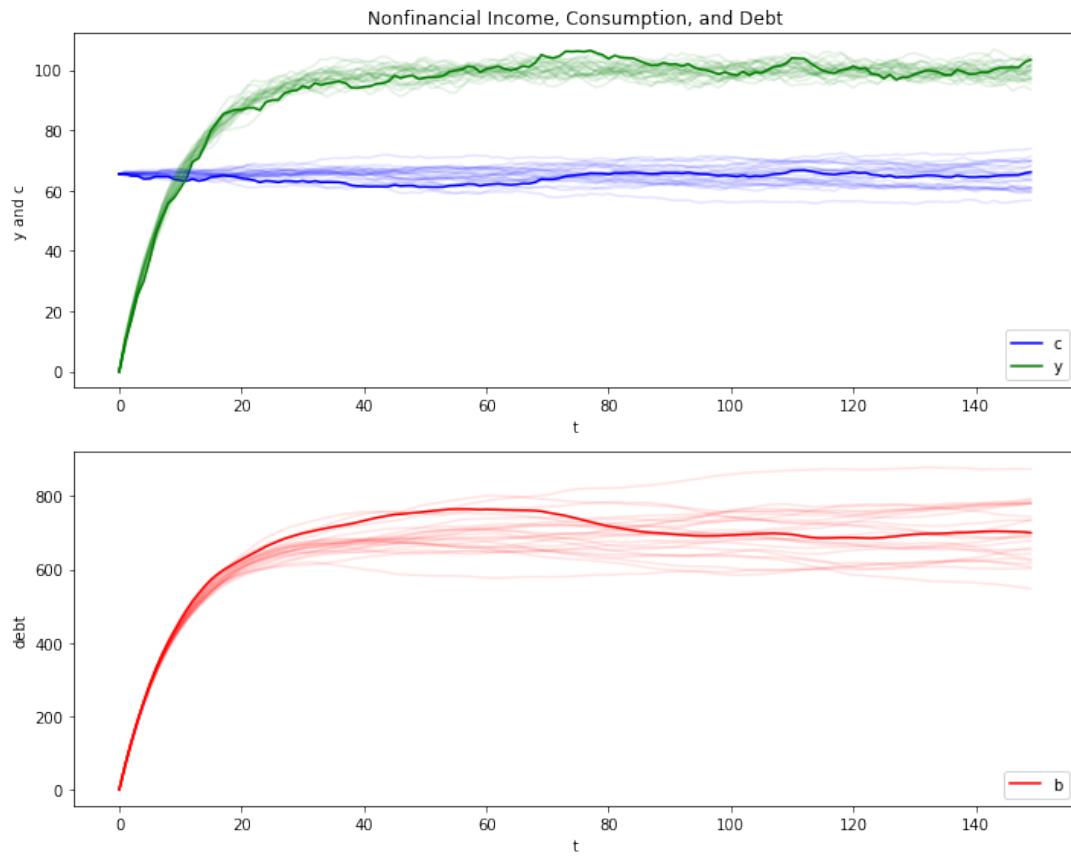
```

Now let's create figures with initial conditions of zero for y_0 and b_0

```
In [13]: out = income_consumption_debt_series(A_LSS, C_LSS, G_LSS, mu_0, Sigma_0)
bsim0, csim0, ysim0 = out[:3]
cons_mean0, cons_var0, debt_mean0, debt_var0 = out[3:]

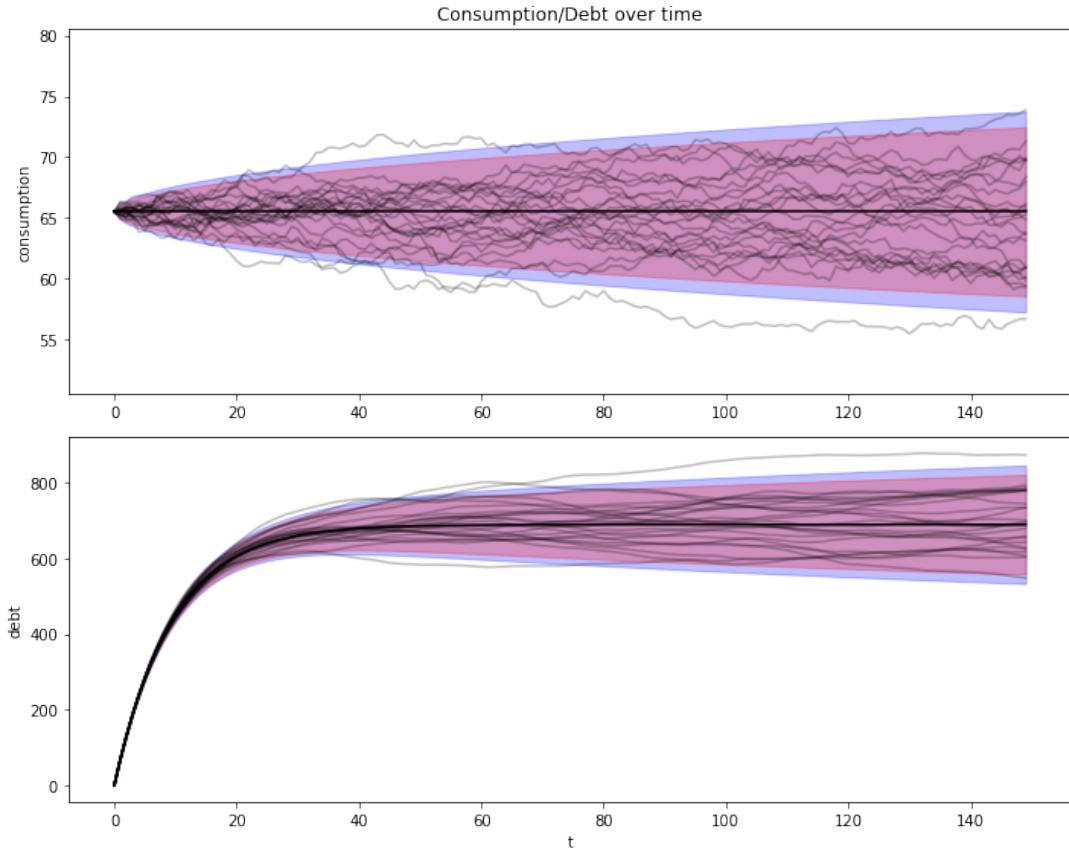
consumption_income_debt_figure(bsim0, csim0, ysim0)

plt.show()
```



```
In [14]: 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**

$$(1 - \beta)b_t + c_t = (1 - \beta)E_t \sum_{j=0}^{\infty} \beta^j y_{t+j} \quad (6)$$

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 (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.

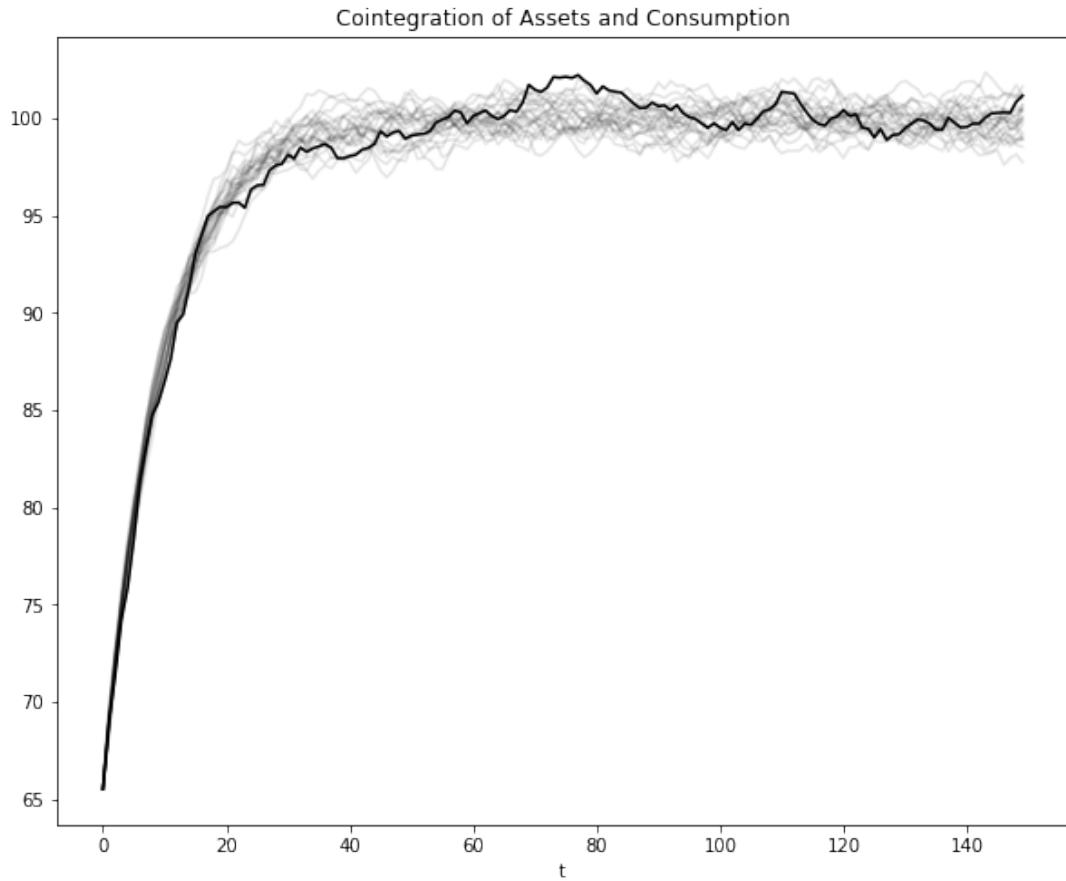
By altering initial conditions, we shall remove this transient in our second example to be presented below

```
In [15]: 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
```

```
In [16]: cointegration_figure(bsim0, csim0)
plt.show()
```



44.6.3 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 [14] 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.

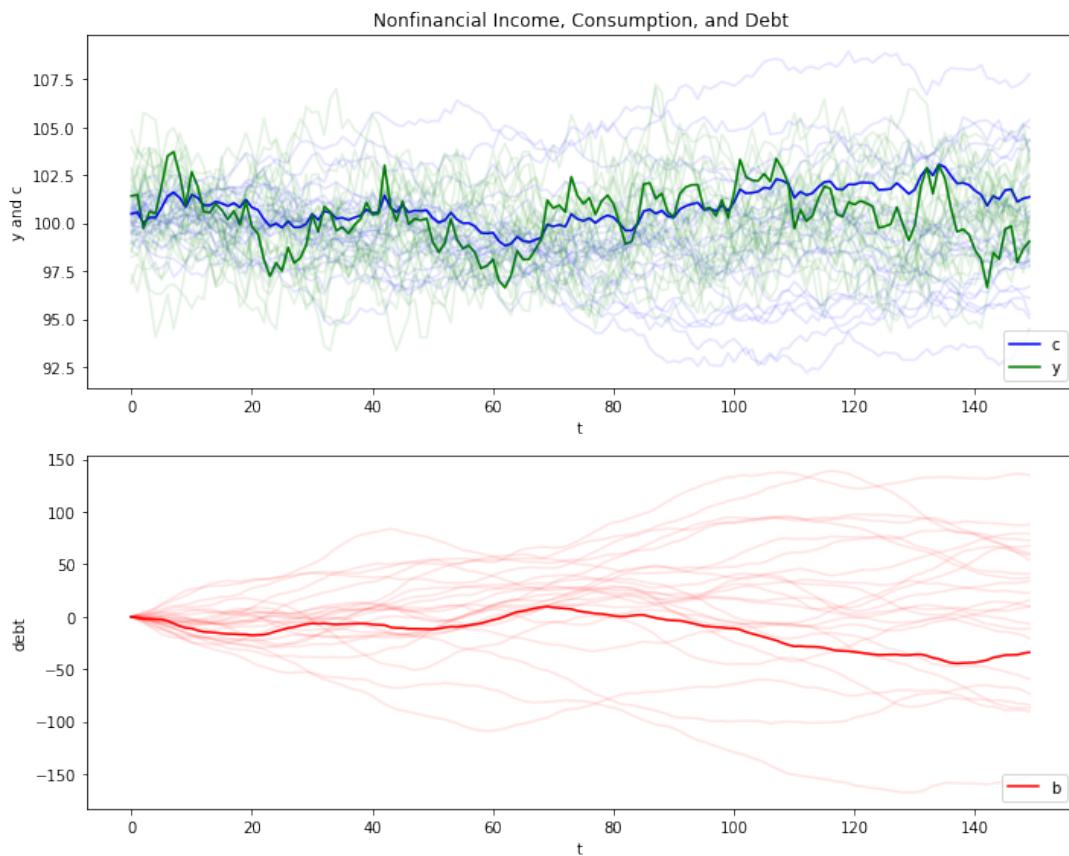
There is no need for foreigners to lend to our group.

Let's have a look at the corresponding figures

```
In [17]: 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:]

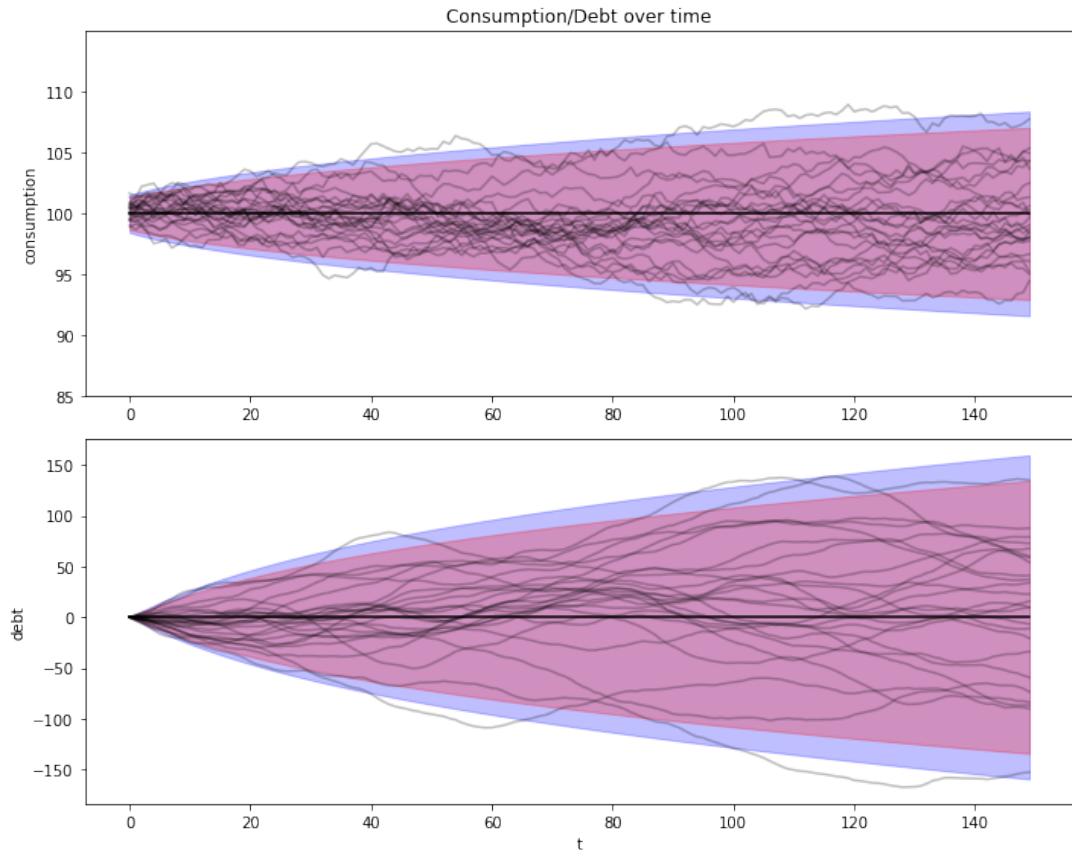
consumption_income_debt_figure(bsimb, csimb, ysimb)

plt.show()
```



```
In [18]: consumption_debt_fanchart(csimb, cons_meanb, cons_varb,
                                    bsimb, debt_meanb, debt_varb)

plt.show()
```



The graphs confirm the following outcomes:

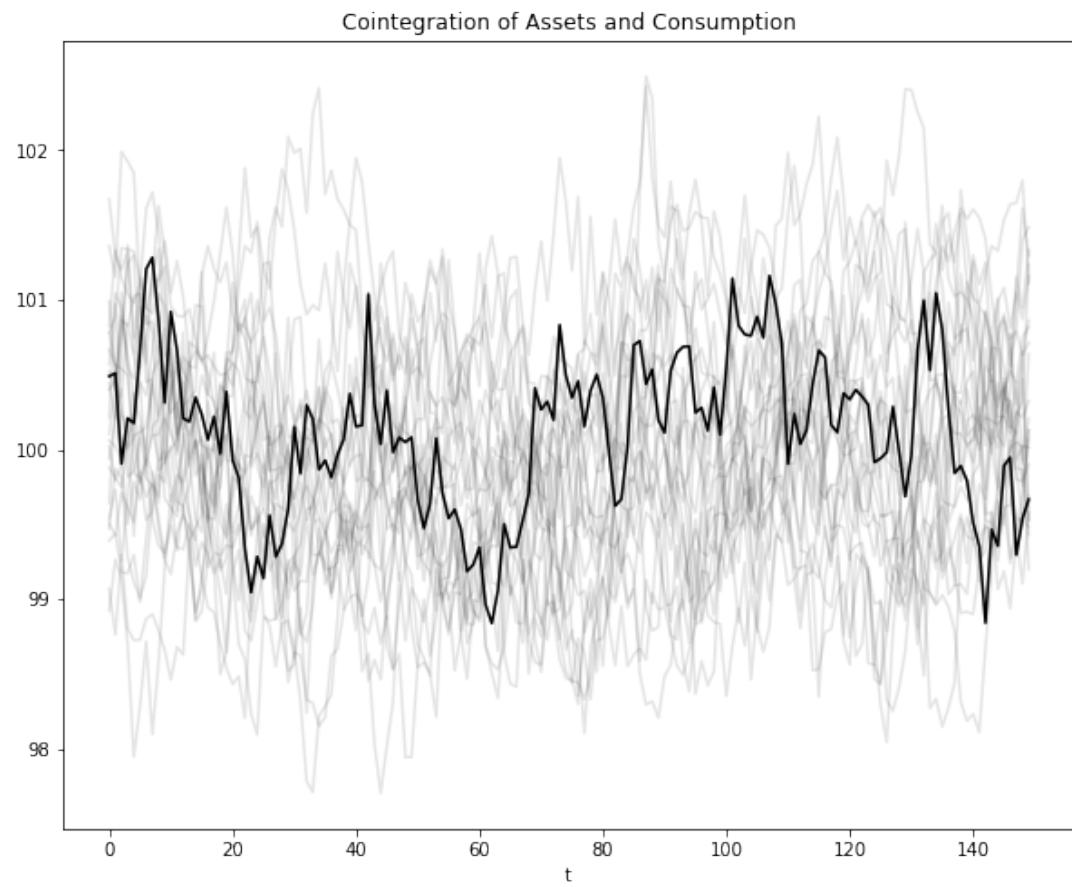
- As before, the consumption distribution spreads out over time.

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

```
In [19]: cointegration_figure(bsimb, csimb)
plt.show()
```



Chapter 45

Production Smoothing via Inventories

45.1 Contents

- Overview [45.2](#)
- Example 1 [45.3](#)
- Inventories Not Useful [45.4](#)
- Inventories Useful but are Hardwired to be Zero Always [45.5](#)
- Example 2 [45.6](#)
- Example 3 [45.7](#)
- Example 4 [45.8](#)
- Example 5 [45.9](#)
- Example 6 [45.10](#)
- Exercises [45.11](#)

In addition to what's in Anaconda, this lecture employs the following library:

```
In [1]: !pip install quantecon
```

45.2 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 [?], 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

$$\begin{bmatrix} I_{t+1} \\ z_t \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & A_{22} \end{bmatrix} \begin{bmatrix} I_t \\ z_t \end{bmatrix} + \begin{bmatrix} 1 & -1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} Q_t \\ S_t \end{bmatrix} + \begin{bmatrix} 0 \\ C_2 \end{bmatrix} \epsilon_{t+1}$$

or

$$x_{t+1} = Ax_t + Bu_t + C\epsilon_{t+1}$$

(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 $u_t' Qu_t$ 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

$$\pi_t = -(x_t' Rx_t + u_t' Qu_t + 2u_t' Nx_t)$$

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

$$\begin{aligned}\pi_t &= p_t S_t - c(Q_t) - d(I_t, S_t) \\ &= (a_0 - a_1 S_t + v_t) S_t - c_1 Q_t - c_2 Q_t^2 - d_1 I_t - d_2 (S_t - I_t)^2 \\ &= a_0 S_t - a_1 S_t^2 + G z_t S_t - c_1 Q_t - c_2 Q_t^2 - d_1 I_t - d_2 S_t^2 - d_2 I_t^2 + 2d_2 S_t I_t \\ &= - \left(\underbrace{d_1 I_t + d_2 I_t^2 + a_1 S_t^2 + d_2 S_t^2 + c_2 Q_t^2}_{x_t' Rx_t} - \underbrace{a_0 S_t - G z_t S_t + c_1 Q_t - 2d_2 S_t I_t}_{u_t' Qu_t} \right) \\ &= - \left([I_t \ z_t'] \underbrace{\begin{bmatrix} d_2 & \frac{d_1}{2} S_c \\ \frac{d_1}{2} S_c' & 0 \end{bmatrix}}_{\equiv R} [I_t \ z_t] + [Q_t \ S_t] \underbrace{\begin{bmatrix} c_2 & 0 \\ 0 & a_1 + d_2 \end{bmatrix}}_{\equiv Q} [Q_t \ S_t] + 2 [Q_t \ S_t] \underbrace{\begin{bmatrix} 0 & \frac{c_1}{2} S_c \\ -d_2 & -\frac{a_0}{2} S_c \end{bmatrix}}_{\equiv N} \right)\end{aligned}$$

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

$$u_t = -Fx_t$$

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.

```
In [2]: import numpy as np
import quantecon as qe
import matplotlib.pyplot as plt
%matplotlib inline
```

```
/home/ubuntu/anaconda3/lib/python3.7/site-packages/numba/np/ufunc/parallel.py:
 355:
NumbaWarning: The TBB threading layer requires TBB version 2019.5 or later i.e.,
TBB_INTERFACE_VERSION >= 11005. Found TBB_INTERFACE_VERSION = 11004. The TBB
  threading
layer is disabled.
```

```
warnings.warn(problem)
```

```
In [3]: 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,                         # Cost-of-holding inventories
                 d1=1,                         # Inverse demand function
                 d2=1,
                 a0=10,                        # z process
                 a1=1,
                 A22=[[1, 0],
                       [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
    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]
    l_path = (G @ z_path)[0, :]

    Q_path = u_path[0, :]
    S_path = u_path[1, :]

    revenue = (a0 - a1 * S_path + l_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 + l_path - c1) / (2 * (a1 + c2))
    Q_hardwired = (a0 + l_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), l_path, color='r')
    ax1_.set_ylabel("demand shock", color='r')
    span = max(abs(l_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: $a_0 = 10, a_1 = 1$
- cost of production $c_1 = 1, c_2 = 1$
- costs of holding inventories $d_1 = 1, d_2 = 1$

In the examples below, we alter some or all of these parameter values.

45.3 Example 1

In this example, the demand shock follows AR(1) process:

$$\nu_t = \alpha + \rho \nu_{t-1} + \epsilon_t,$$

which implies

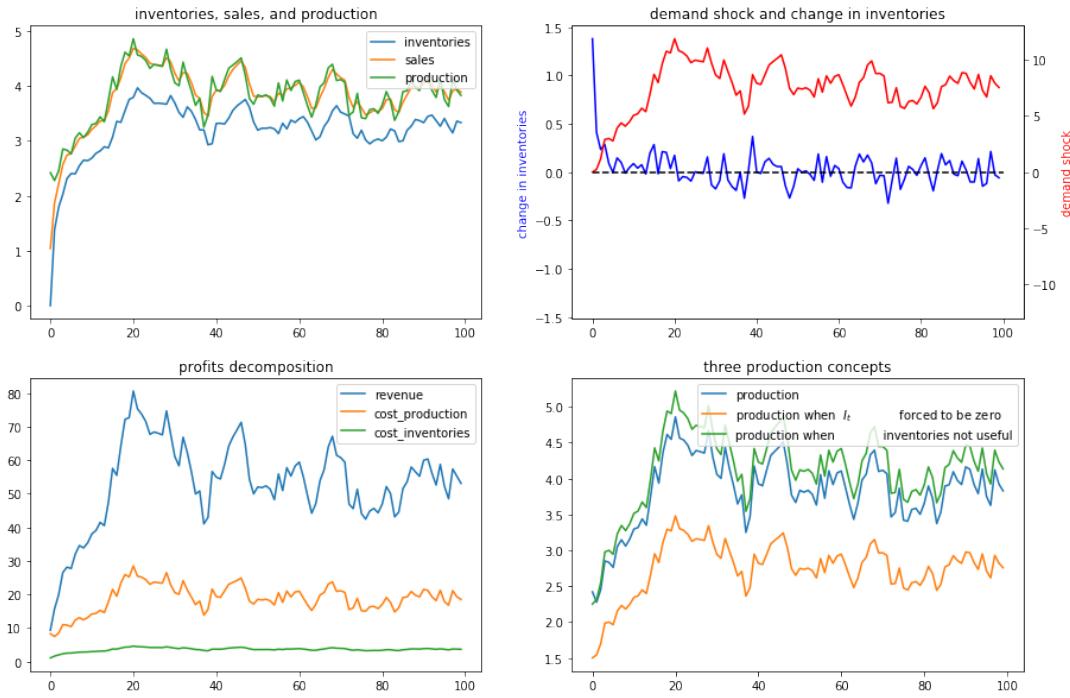
$$z_{t+1} = \begin{bmatrix} 1 \\ v_{t+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ \alpha & \rho \end{bmatrix} \underbrace{\begin{bmatrix} 1 \\ v_t \end{bmatrix}}_{z_t} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} \epsilon_{t+1}.$$

We set $\alpha = 1$ and $\rho = 0.9$, their default values.

We'll calculate and display outcomes, then discuss them below the pertinent figures.

In [4]: `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.

45.4 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

$$\sum_{t=0}^{\infty} \beta^t \{p_t Q_t - C(Q_t)\}.$$

The optimum decision rule is

$$Q_t^{ni} = \frac{a_0 + \nu_t - c_1}{c_2 + a_1}.$$

45.5 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

$$\max_{\{I_t, Q_t, S_t\}} \sum_{t=0}^{\infty} \beta^t \{p_t S_t - C(Q_t) - d(I_t, S_t)\}$$

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

$$\max_{Q_t} \sum_{t=0}^{\infty} \beta^t \{p_t Q_t - C(Q_t) - d(0, Q_t)\}$$

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.

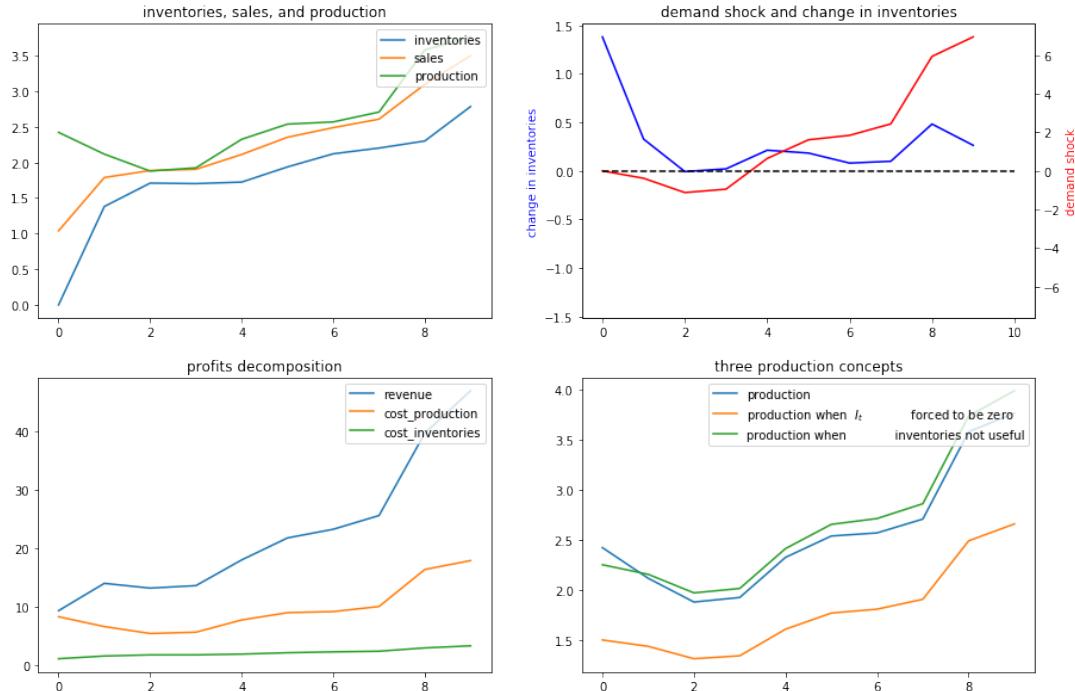
High optimal production in the original model early on occurs because the firm wants to accumulate inventories quickly in order to acquire high inventories for use in later periods.

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:

In [5]: `ex1.simulate(x0, T=10)`



45.6 Example 2

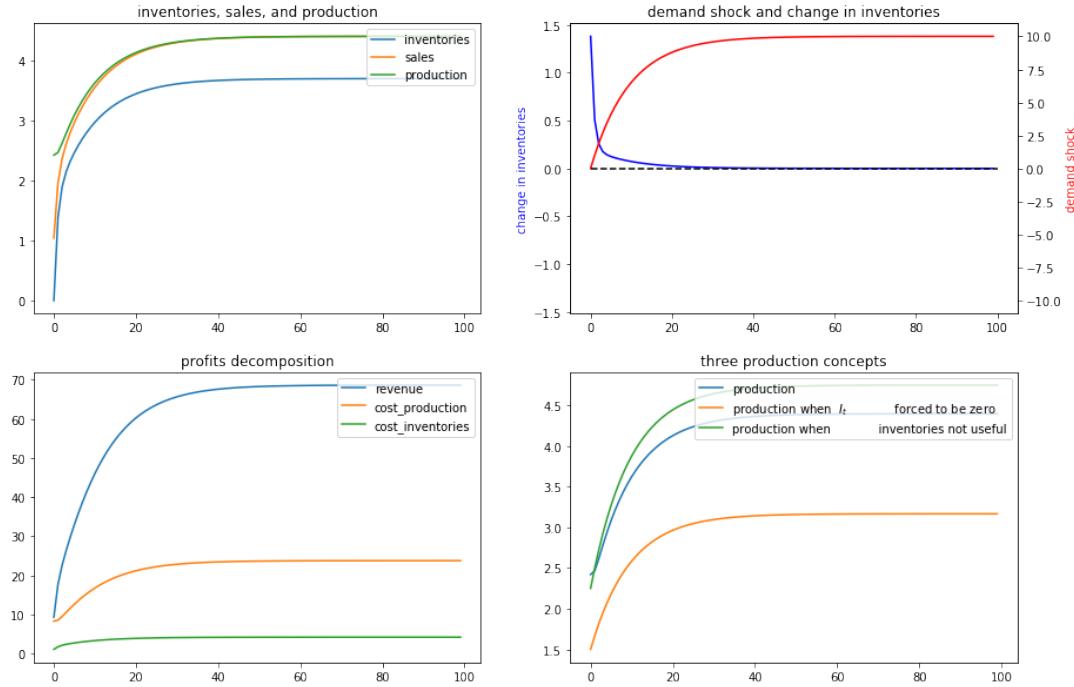
Next, we shut down randomness in demand and assume that the demand shock ν_t follows a deterministic path:

$$\nu_t = \alpha + \rho \nu_{t-1}$$

Again, we’ll compute and display outcomes in some figures

```
In [6]: ex2 = SmoothingExample(C2=[[0], [0]])
```

```
x0 = [0, 1, 0]
ex2.simulate(x0)
```



45.7 Example 3

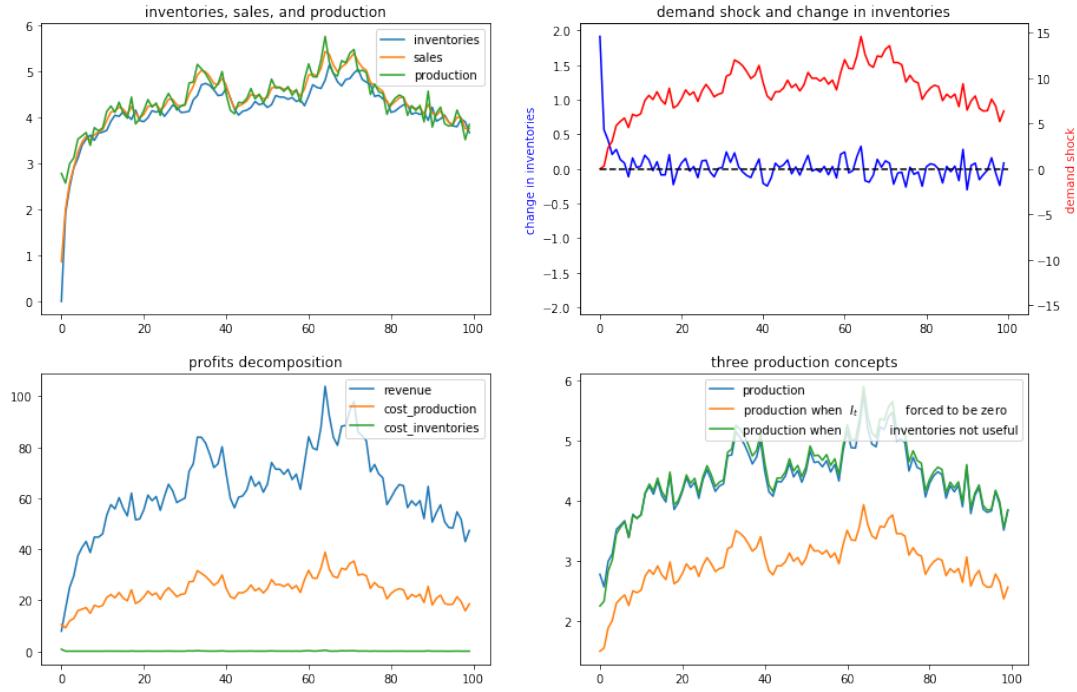
Now we'll put randomness back into the demand shock process and also assume that there are zero costs of holding inventories.

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

```
In [7]: ex3 = SmoothingExample(d1=0)
```

```
x0 = [0, 1, 0]
ex3.simulate(x0)
```



45.8 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.

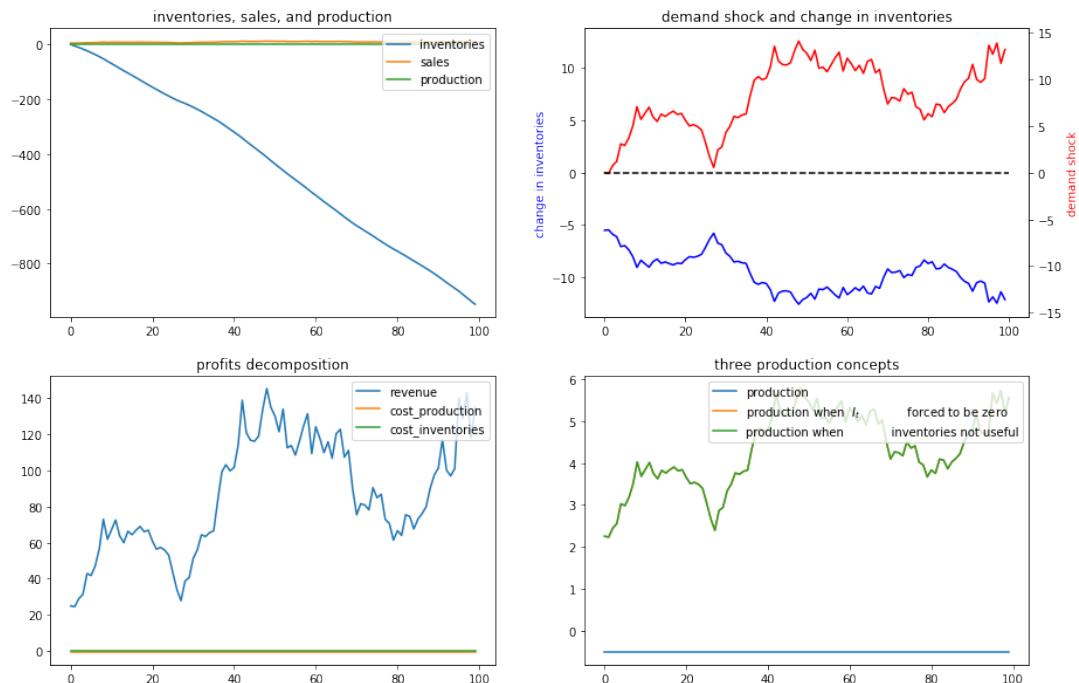
The dynamic program instructs the firm to set production costs to zero and to **run a Ponzi scheme** by running inventories down forever.

(We can interpret this as the firm somehow **going short in** or **borrowing** inventories)

The following figures confirm that inventories head south without limit

```
In [8]: ex4 = SmoothingExample(d1=0, d2=0)
```

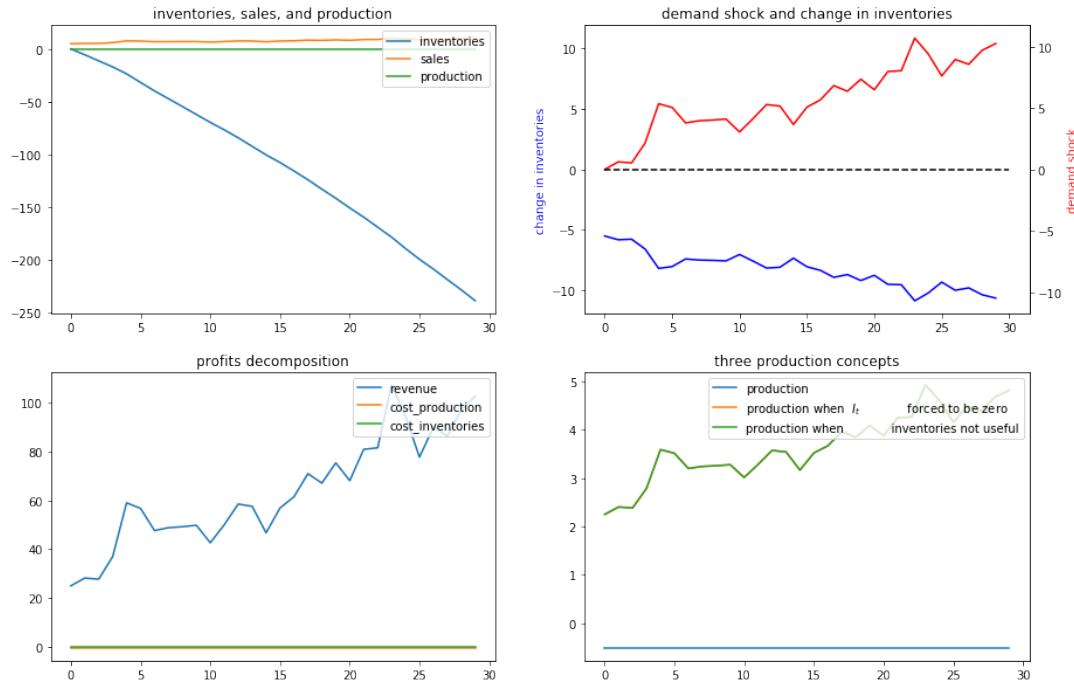
```
x0 = [0, 1, 0]
ex4.simulate(x0)
```



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

```
In [9]: # shorter period
ex4.simulate(x0, T=30)
```



45.9 Example 5

Now we'll assume that the demand shock that follows a linear time trend

$$v_t = b + at, a > 0, b > 0$$

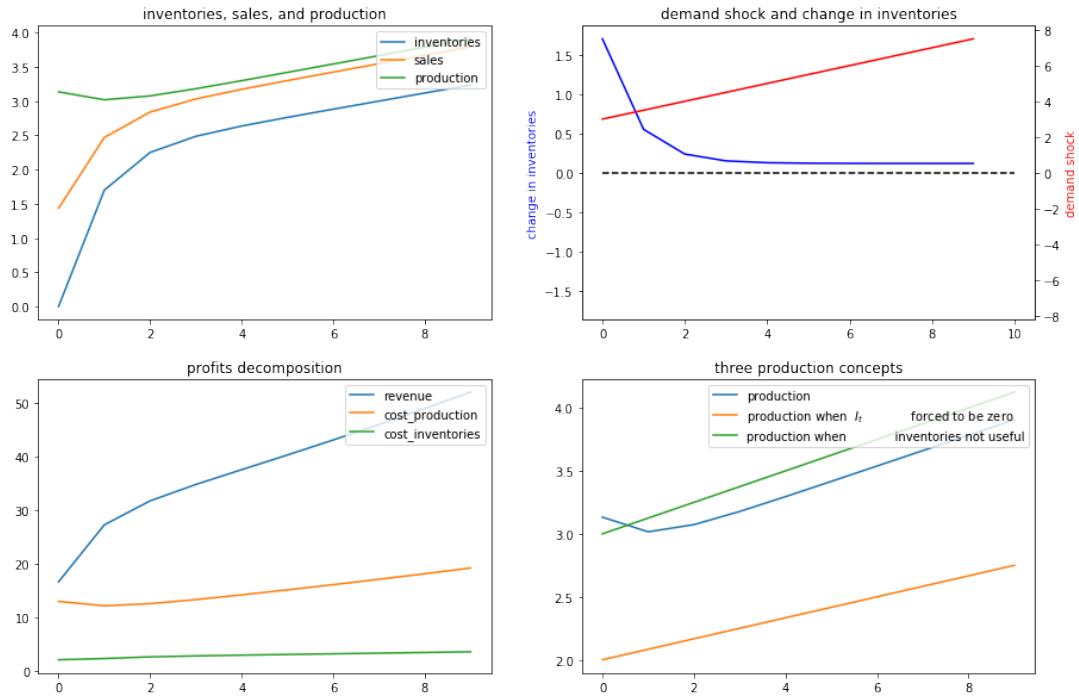
To represent this, we set $C_2 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$ and

$$A_{22} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}, x_0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, G = [b \ a]$$

```
In [10]: # Set parameters
a = 0.5
b = 3.
```

```
In [11]: 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)
```



45.10 Example 6

Now we'll assume a deterministically seasonal demand shock.

To represent this we'll set

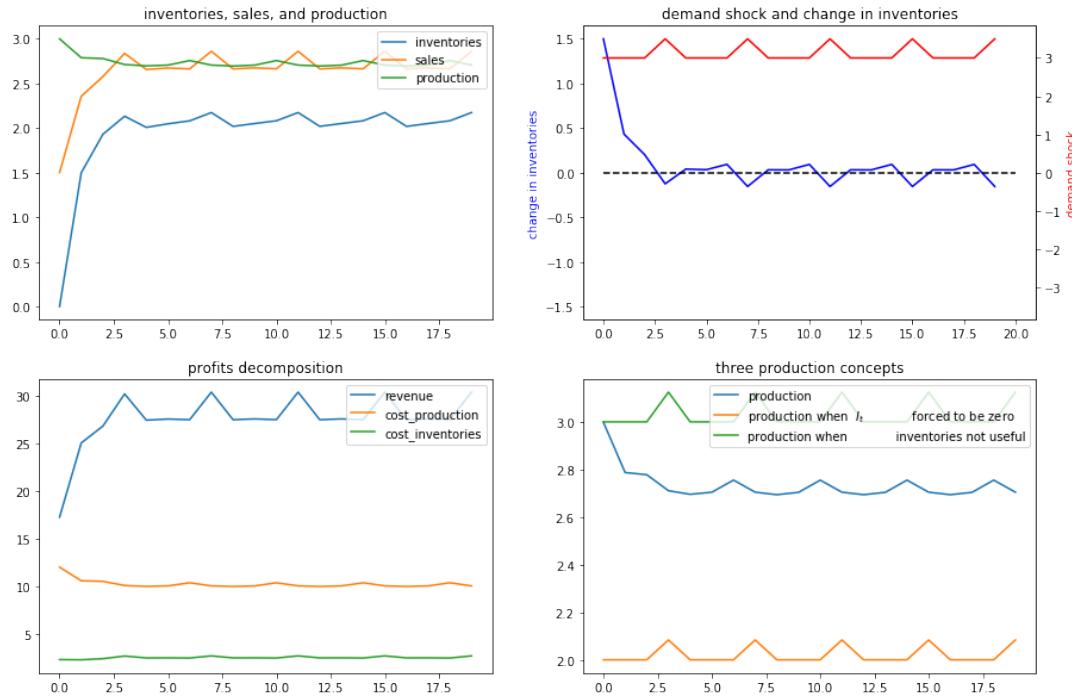
$$A_{22} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}, C_2 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, G' = \begin{bmatrix} b \\ a \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

where $a > 0, b > 0$ and

$$x_0 = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

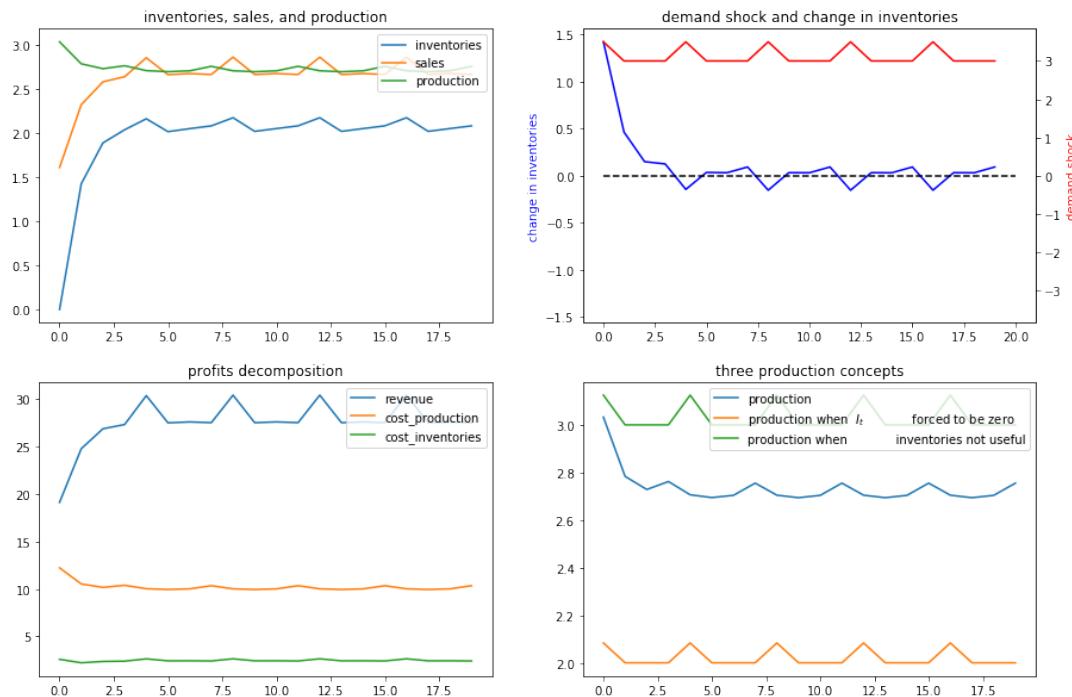
```
In [12]: ex6 = SmoothingExample(A22=[[1, 0, 0, 0, 0],
[0, 0, 0, 0, 1],
[0, 1, 0, 0, 0],
[0, 0, 1, 0, 0],
[0, 0, 0, 1, 0]],
C2=[[0], [0], [0], [0], [0]],
G=[b, a, 0, 0, 0])
```

```
x00 = [0, 1, 0, 1, 0, 0] # Set the initial inventory as 0
ex6.simulate(x00, T=20)
```

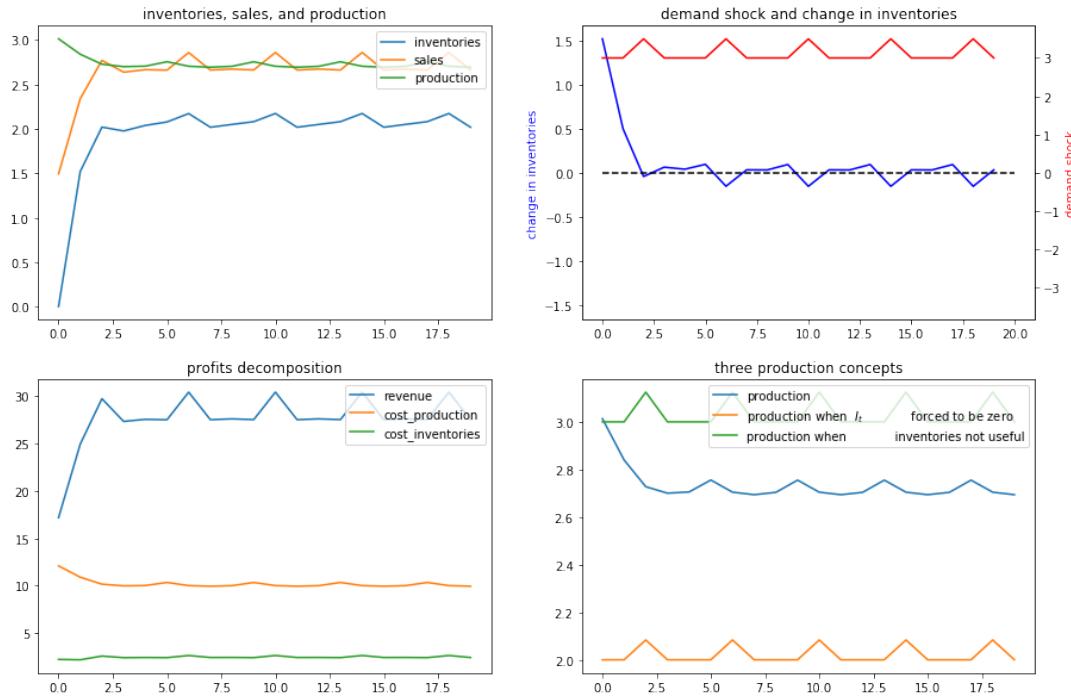


Now we'll generate some more examples that differ simply from the initial **season** of the year in which we begin the demand shock

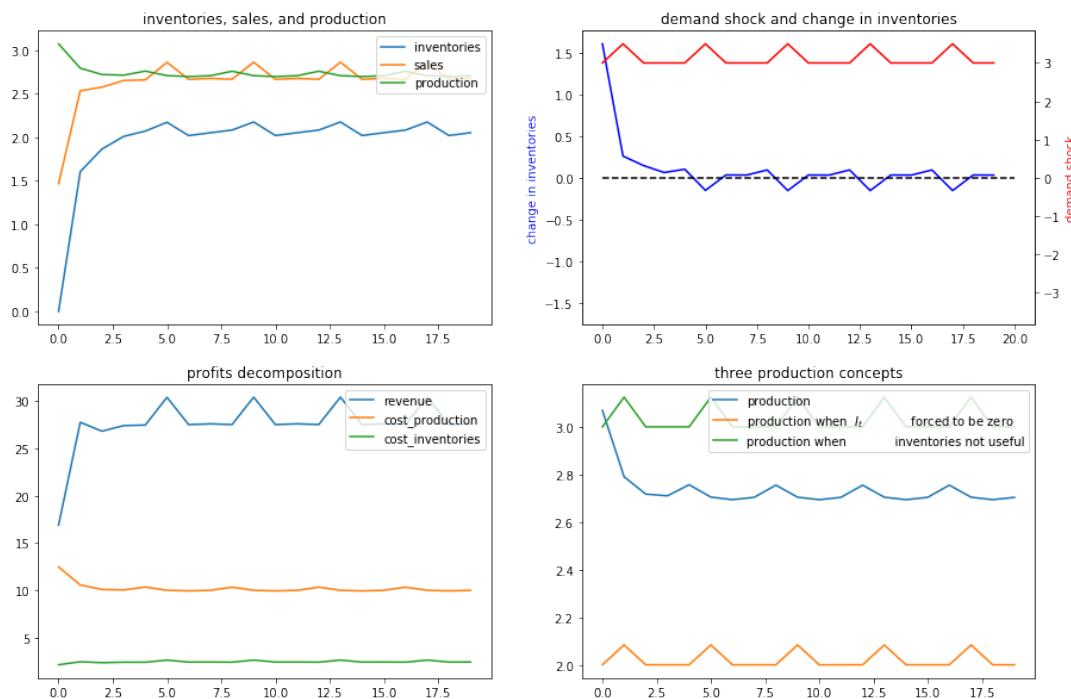
```
In [13]: x01 = [0, 1, 1, 0, 0, 0]
ex6.simulate(x01, T=20)
```



In [14]: `x02 = [0, 1, 0, 0, 1, 0]`
`ex6.simulate(x02, T=20)`



In [15]: `x03 = [0, 1, 0, 0, 0, 1]`
`ex6.simulate(x03, T=20)`



45.11 Exercises

Please try to analyze some inventory sales smoothing problems using the `SmoothingExample` class.

45.11.1 Exercise 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.

45.11.2 Exercise 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 $d_2 = 5$.

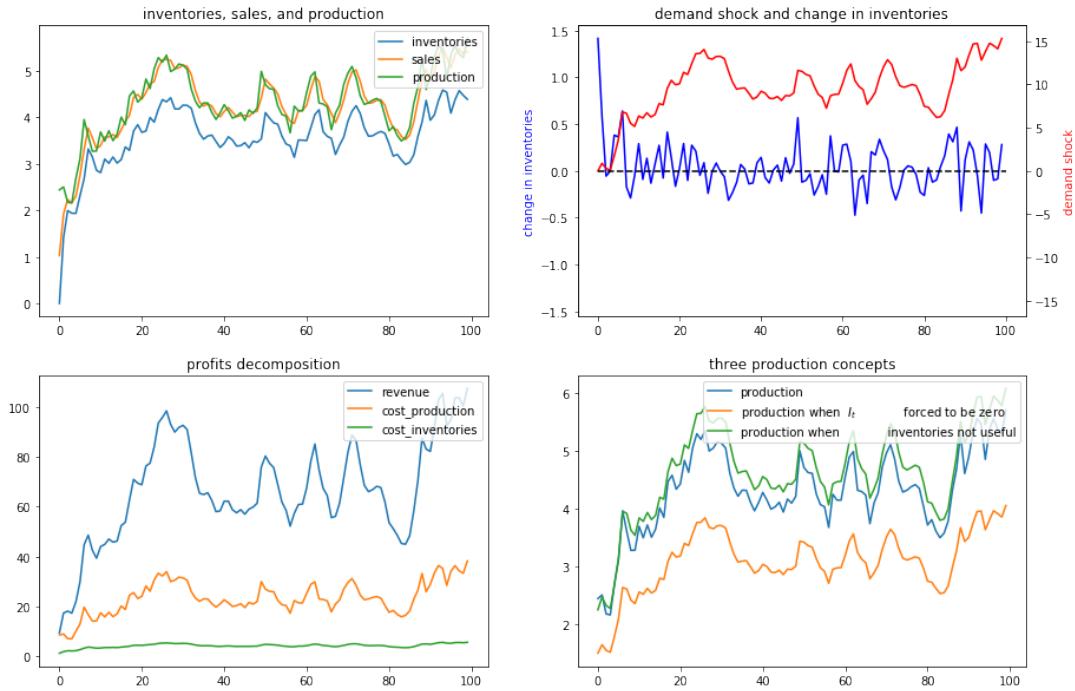
45.11.3 Solution 1

```
In [16]: # set parameters
α = 1
ρ1 = 1.2
ρ2 = -.3

In [17]: # construct matrices
A22 = [[1, 0, 0],
       [1, ρ1, ρ2],
       [0, 1, 0]]
C2 = [[0], [1], [0]]
G = [0, 1, 0]
```

```
In [18]: ex1 = SmoothingExample(A22=A22, C2=C2, G=G)

x0 = [0, 1, 0, 0] # initial condition
ex1.simulate(x0)
```



```
In [19]: # now silence the noise
ex1_no_noise = SmoothingExample(A22=A22, C2=[[0], [0], [0]], G=G)

# initial condition
x0 = [0, 1, 0, 0]

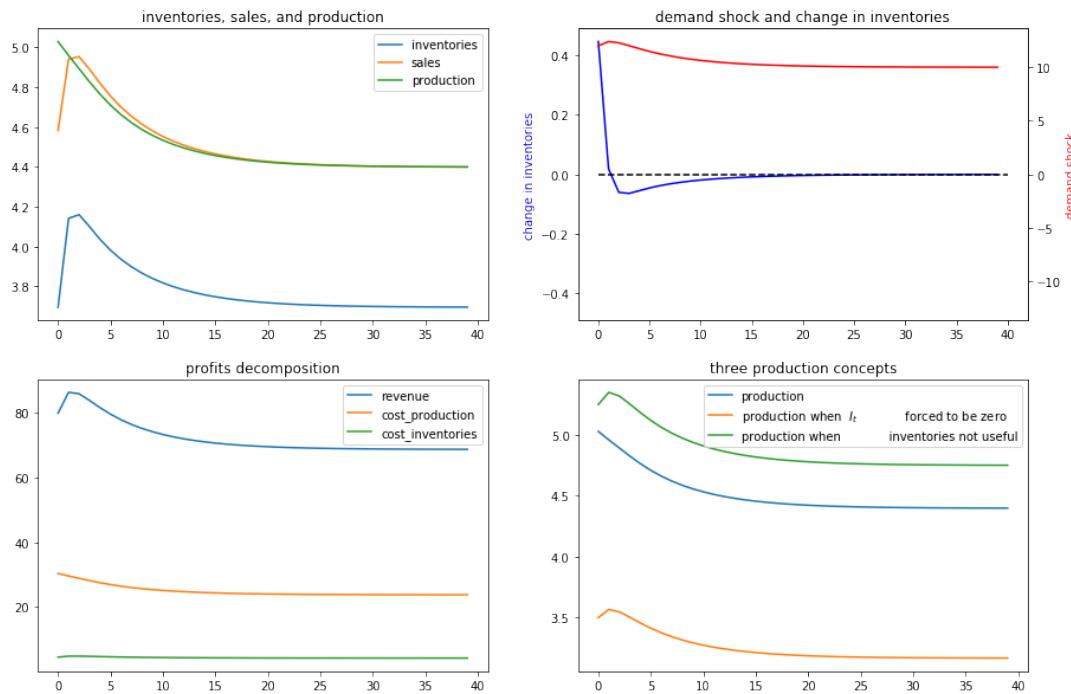
# compute stationary states
x_bar = ex1_no_noise.LQ.compute_sequence(x0, ts_length=250)[0][:, -1]
x_bar
```

```
Out[19]: 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.

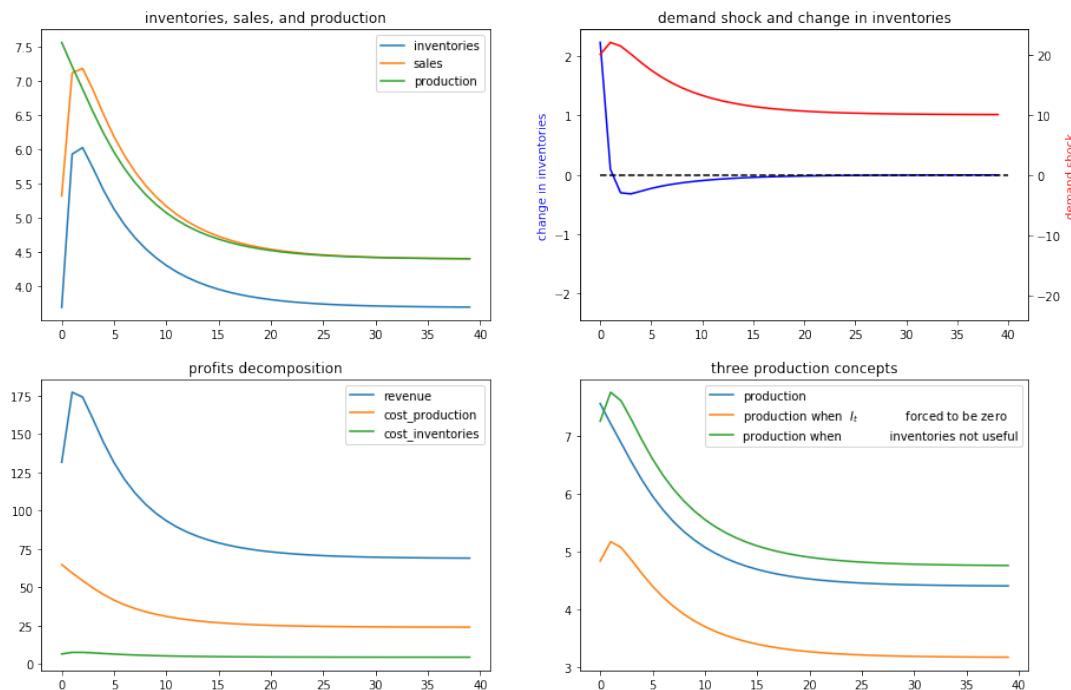
```
In [20]: T = 40
```

```
In [21]: # small shock
x_bar1 = x_bar.copy()
x_bar1[2] += 2
ex1_no_noise.simulate(x_bar1, T=T)
```



```
In [22]: # large shock
```

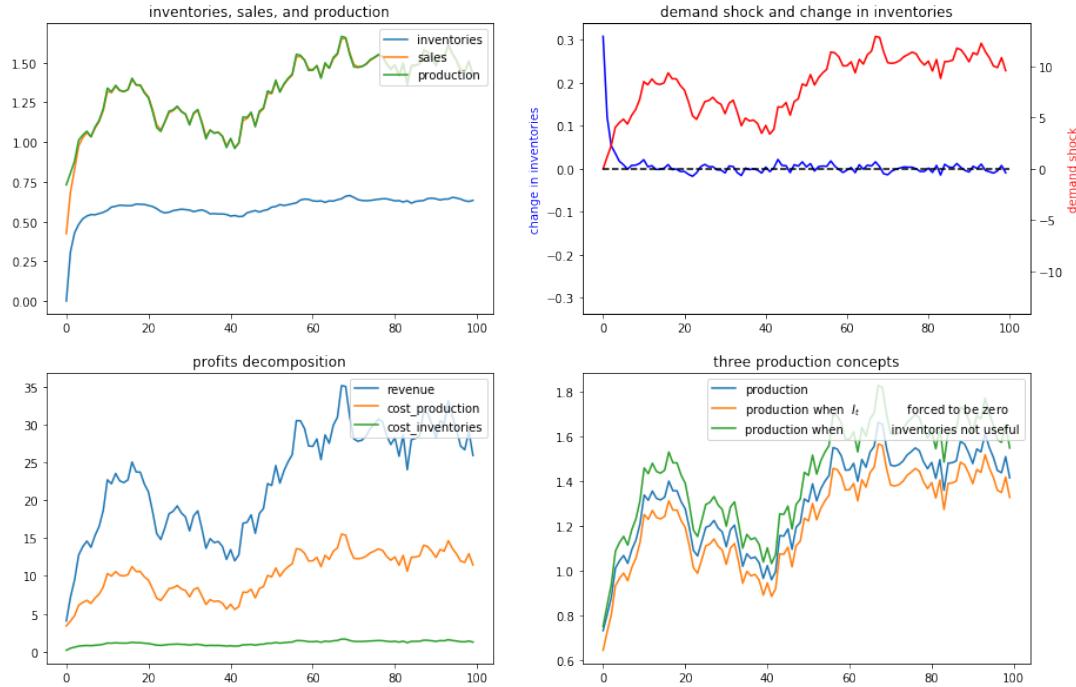
```
x_bar1 = x_bar.copy()
x_bar1[2] += 10
ex1_no_noise.simulate(x_bar1, T=T)
```



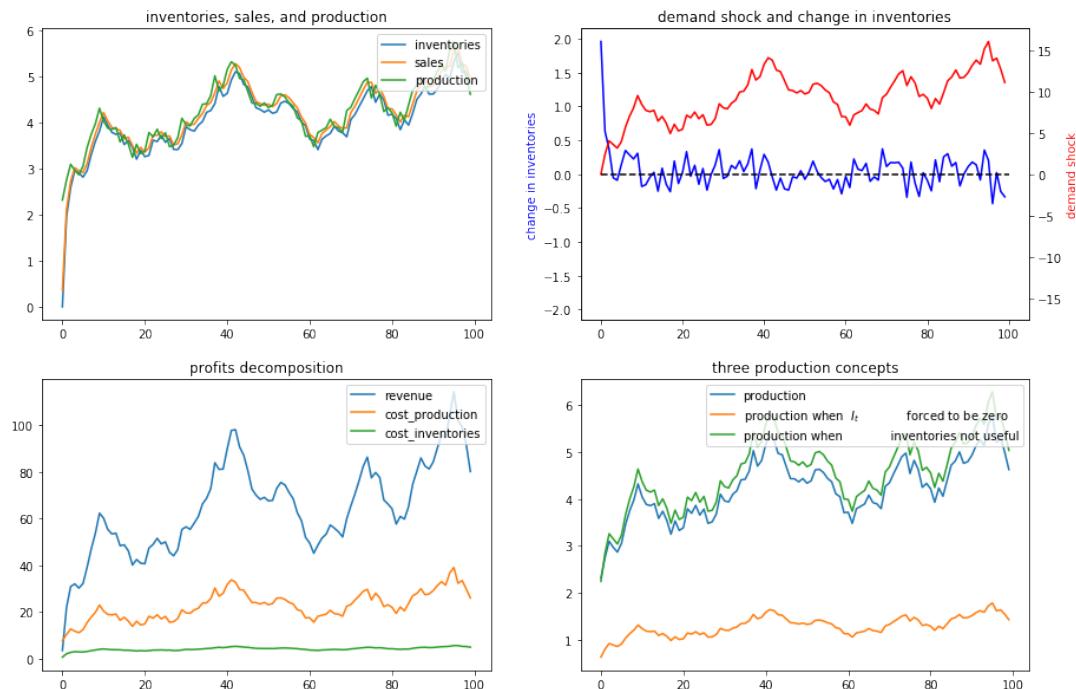
45.11.4 Solution 2

In [23]: `x0 = [0, 1, 0]`

In [24]: `SmoothingExample(c2=5).simulate(x0)`



In [25]: `SmoothingExample(d2=5).simulate(x0)`



Part VII

Multiple Agent Models

Chapter 46

Schelling's Segregation Model

46.1 Contents

- Outline 46.2
- The Model 46.3
- Results 46.4
- Exercises 46.5
- Solutions 46.6

46.2 Outline

In 1969, Thomas C. Schelling developed a simple but striking model of racial segregation [98].

His model studies the dynamics of racially mixed neighborhoods.

Like much of Schelling's work, the model shows how local interactions can lead to surprising aggregate structure.

In particular, it shows that relatively mild preference for neighbors of similar race can lead in aggregate to the collapse of mixed neighborhoods, and high levels of segregation.

In recognition of this and other research, Schelling was awarded the 2005 Nobel Prize in Economic Sciences (joint with Robert Aumann).

In this lecture, we (in fact you) will build and run a version of Schelling's model.

Let's start with some imports:

```
In [1]: from random import uniform, seed
        from math import sqrt
        import matplotlib.pyplot as plt
        %matplotlib inline
```

46.3 The Model

We will cover a variation of Schelling's model that is easy to program and captures the main idea.

46.3.1 Set-Up

Suppose we have two types of people: orange people and green people.

For the purpose of this lecture, we will assume there are 250 of each type.

These agents all live on a single unit square.

The location of an agent is just a point (x, y) , where $0 < x, y < 1$.

46.3.2 Preferences

We will say that an agent is *happy* if half or more of her 10 nearest neighbors are of the same type.

Here ‘nearest’ is in terms of [Euclidean distance](#).

An agent who is not happy is called *unhappy*.

An important point here is that agents are not averse to living in mixed areas.

They are perfectly happy if half their neighbors are of the other color.

46.3.3 Behavior

Initially, agents are mixed together (integrated).

In particular, the initial location of each agent is an independent draw from a bivariate uniform distribution on $S = (0, 1)^2$.

Now, cycling through the set of all agents, each agent is now given the chance to stay or move.

We assume that each agent will stay put if they are happy and move if unhappy.

The algorithm for moving is as follows

1. Draw a random location in S
2. If happy at new location, move there
3. Else, go to step 1

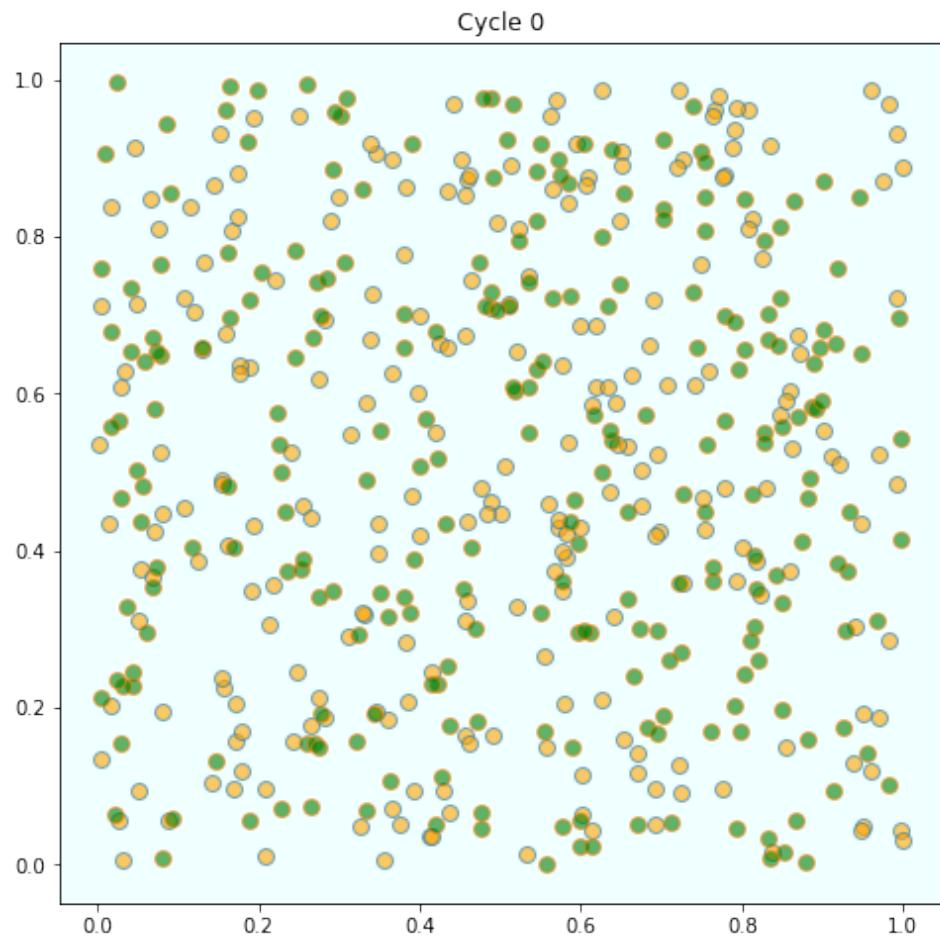
In this way, we cycle continuously through the agents, moving as required.

We continue to cycle until no one wishes to move.

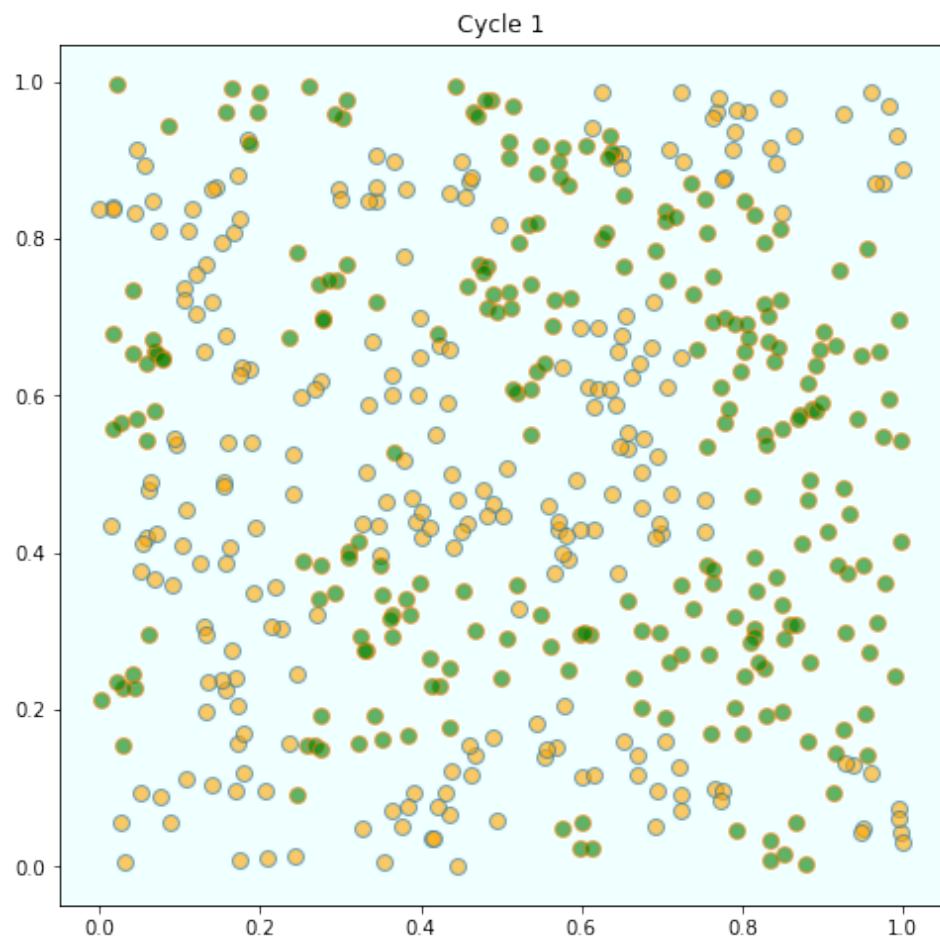
46.4 Results

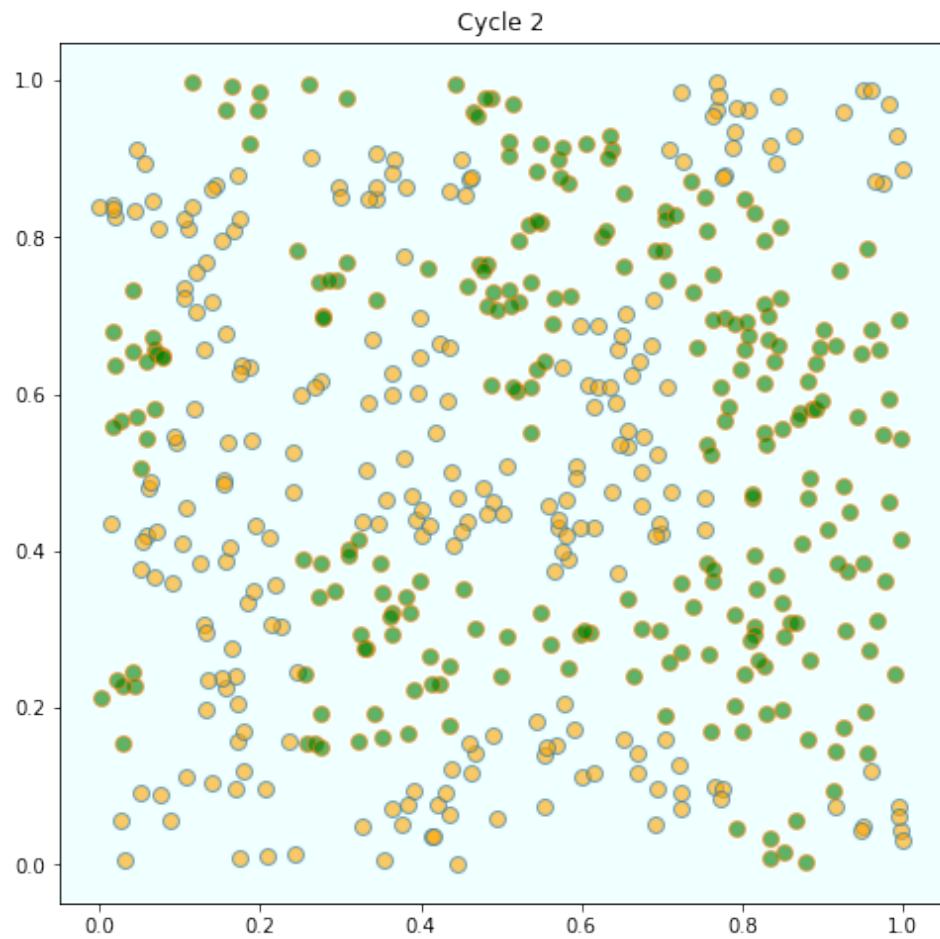
Let’s have a look at the results we got when we coded and ran this model.

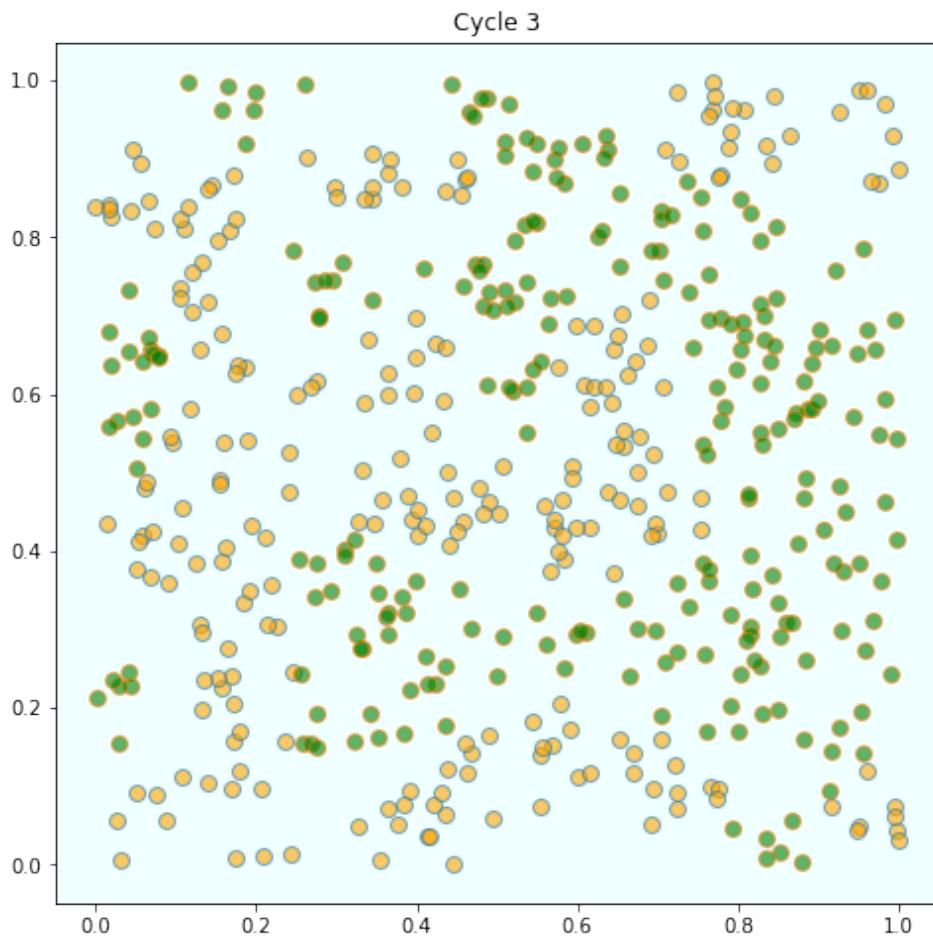
As discussed above, agents are initially mixed randomly together.



But after several cycles, they become segregated into distinct regions.







In this instance, the program terminated after 4 cycles through the set of agents, indicating that all agents had reached a state of happiness.

What is striking about the pictures is how rapidly racial integration breaks down.

This is despite the fact that people in the model don't actually mind living mixed with the other type.

Even with these preferences, the outcome is a high degree of segregation.

46.5 Exercises

46.5.1 Exercise 1

Implement and run this simulation for yourself.

Consider the following structure for your program.

Agents can be modeled as [objects](#).

Here's an indication of how they might look

* **Data:**

- * type (green or orange)
- * location

- * Methods:
 - * determine whether happy or not given locations of other agents
 - * If not happy, move
 - * find a new location where happy

And here's some pseudocode for the main loop

```
while agents are still moving
  for agent in agents
    give agent the opportunity to move
```

Use 250 agents of each type.

46.6 Solutions

46.6.1 Exercise 1

Here's one solution that does the job we want.

If you feel like a further exercise, you can probably speed up some of the computations and then increase the number of agents.

```
In [2]: seed(10) # For reproducible random numbers

class Agent:

    def __init__(self, type):
        self.type = type
        self.draw_location()

    def draw_location(self):
        self.location = uniform(0, 1), uniform(0, 1)

    def get_distance(self, other):
        "Computes the euclidean distance between self and other agent."
        a = (self.location[0] - other.location[0])**2
        b = (self.location[1] - other.location[1])**2
        return sqrt(a + b)

    def happy(self, agents):
        "True if sufficient number of nearest neighbors are of the same
        type."
        distances = []
        # distances is a list of pairs (d, agent), where d is distance from
        # agent to self
        for agent in agents:
```

```

        if self != agent:
            distance = self.get_distance(agent)
            distances.append((distance, agent))
    # == Sort from smallest to largest, according to distance == #
    distances.sort()
    # == Extract the neighboring agents == #
    neighbors = [agent for d, agent in distances[:num_neighbors]]
    # == Count how many neighbors have the same type as self == #
    num_same_type = sum(self.type == agent.type for agent in neighbors)
    return num_same_type >= require_same_type

def update(self, agents):
    "If not happy, then randomly choose new locations until happy."
    while not self.happy(agents):
        self.draw_location()

def plot_distribution(agents, cycle_num):
    "Plot the distribution of agents after cycle_num rounds of the loop."
    x_values_0, y_values_0 = [], []
    x_values_1, y_values_1 = [], []
    # == Obtain locations of each type == #
    for agent in agents:
        x, y = agent.location
        if agent.type == 0:
            x_values_0.append(x)
            y_values_0.append(y)
        else:
            x_values_1.append(x)
            y_values_1.append(y)
    fig, ax = plt.subplots(figsize=(8, 8))
    plot_args = {'markersize': 8, 'alpha': 0.6}
    ax.set_facecolor('azure')
    ax.plot(x_values_0, y_values_0, 'o', markerfacecolor='orange', **plot_args)
    ax.plot(x_values_1, y_values_1, 'o', markerfacecolor='green', **plot_args)
    ax.set_title(f'Cycle {cycle_num-1}')
    plt.show()

# == Main == #

num_of_type_0 = 250
num_of_type_1 = 250
num_neighbors = 10      # Number of agents regarded as neighbors
require_same_type = 5   # Want at least this many neighbors to be same type

# == Create a list of agents == #
agents = [Agent(0) for i in range(num_of_type_0)]
agents.extend(Agent(1) for i in range(num_of_type_1))

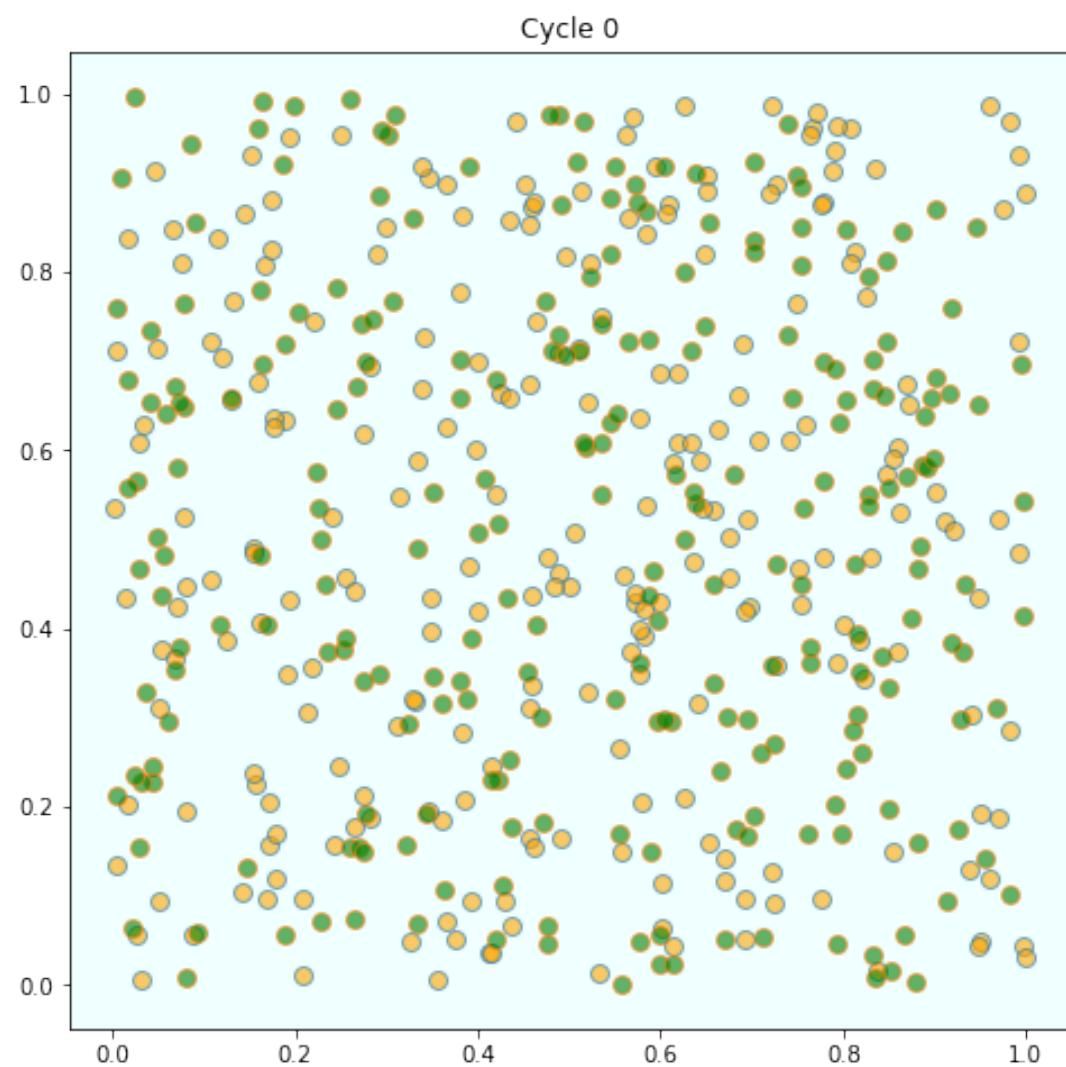
count = 1
# == Loop until none wishes to move == #
while True:
    print('Entering loop ', count)
    plot_distribution(agents, count)
    count += 1

```

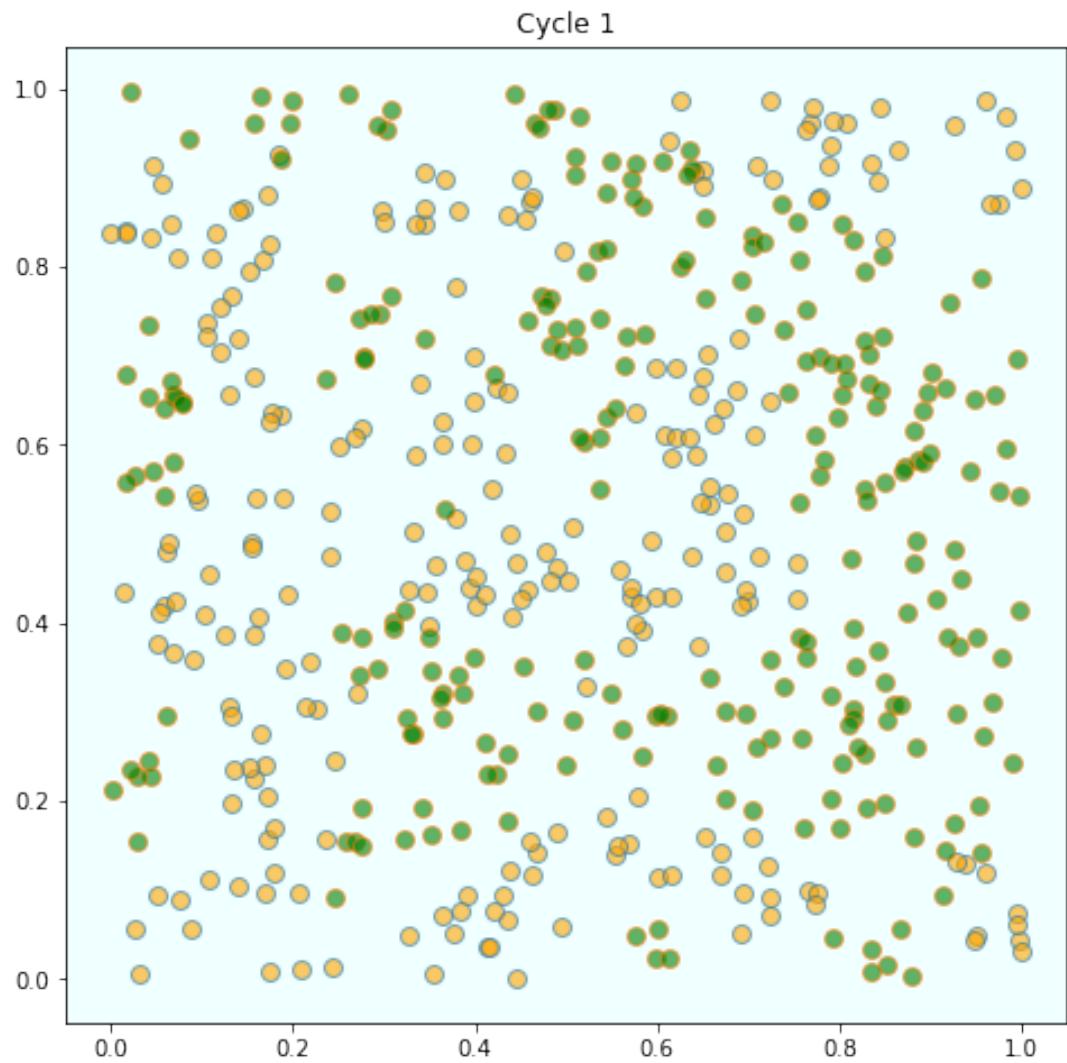
```
no_one_moved = True
for agent in agents:
    old_location = agent.location
    agent.update(agents)
    if agent.location != old_location:
        no_one_moved = False
if no_one_moved:
    break

print('Converged, terminating.')
```

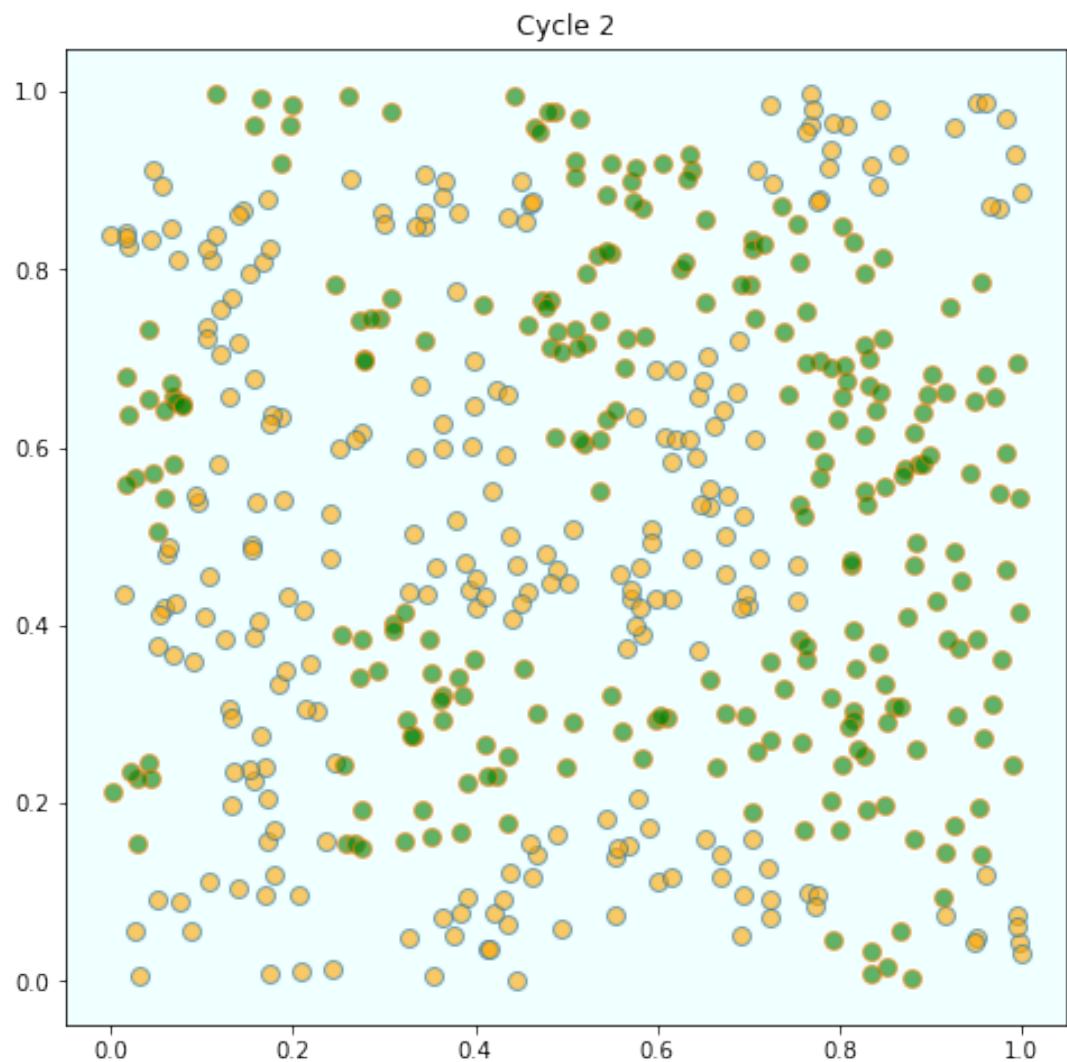
Entering loop 1



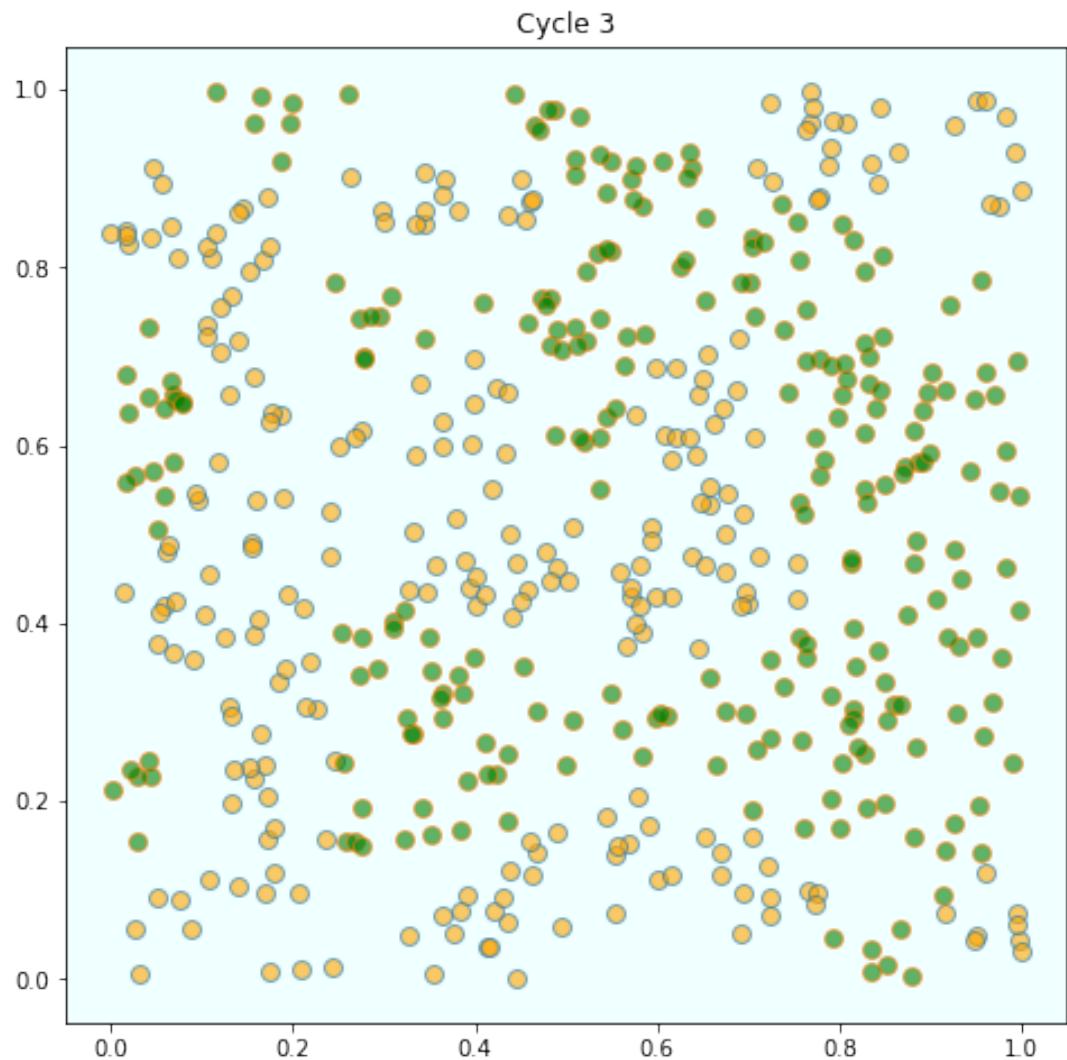
Entering loop 2



Entering loop 3



Entering loop 4



Converged, terminating.

Chapter 47

A Lake Model of Employment and Unemployment

47.1 Contents

- Overview 47.2
- The Model 47.3
- Implementation 47.4
- Dynamics of an Individual Worker 47.5
- Endogenous Job Finding Rate 47.6
- Exercises 47.7
- Solutions 47.8

In addition to what's in Anaconda, this lecture will need the following libraries:

```
In [1]: !pip install quantecon
```

47.2 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 jobs 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:

```
In [2]: import numpy as np
        from quantecon import MarkovChain
        import matplotlib.pyplot as plt
        %matplotlib inline
        from scipy.stats import norm
        from scipy.optimize import brentq
        from quantecon.distributions import BetaBinomial
        from numba import jit

/home/ubuntu/anaconda3/lib/python3.7/site-packages/numba/np/ufunc/parallel.py:
  ↪355:
NumbaWarning: The TBB threading layer requires TBB version 2019.5 or later i.e.,
TBB_INTERFACE_VERSION >= 11005. Found TBB_INTERFACE_VERSION = 11004. The TBB
  ↪threading
layer is disabled.
warnings.warn(problem)
```

47.2.1 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.

47.3 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 $g = b - d$.

47.3.1 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

We also want to know 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)

47.3.2 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

$$X_{t+1} = AX_t \quad \text{where} \quad A := \begin{pmatrix} (1 - d)(1 - \lambda) + b & (1 - d)\alpha + b \\ (1 - d)\lambda & (1 - d)(1 - \alpha) \end{pmatrix}$$

This law tells us how total employment and unemployment evolve over time.

47.3.3 Laws of Motion for Rates

Now let's derive the law of motion for rates.

To get these we can divide both sides of $X_{t+1} = AX_t$ by N_{t+1} to get

$$\begin{pmatrix} U_{t+1}/N_{t+1} \\ E_{t+1}/N_{t+1} \end{pmatrix} = \frac{1}{1+g} A \begin{pmatrix} U_t/N_t \\ E_t/N_t \end{pmatrix}$$

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

$$x_{t+1} = \hat{A}x_t \quad \text{where} \quad \hat{A} := \frac{1}{1+g} A$$

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 \hat{A} sum to 1.

47.4 Implementation

Let's code up these equations.

To do this we're going to use a class that we'll call `LakeModel`.

This class will

1. store the primitives α, λ, b, d
2. compute and store the implied objects g, A, \hat{A}
3. provide methods to simulate dynamics of the stocks and rates
4. provide a method to compute the steady state of the rate

Please be careful because the implied objects g, A, \hat{A} will not change if you only change the primitives.

For example, if you would like to update a primitive like $\alpha = 0.03$, you need to create an instance and update it by `lm = LakeModel(alpha=0.03)`.

In the exercises, we show how to avoid this issue by using getter and setter methods.

```
In [3]: class LakeModel:
    """
    Solves the lake model and computes dynamics of unemployment stocks and
    rates.

    Parameters:
    -----
    λ : scalar
        The job finding rate for currently unemployed workers
    α : scalar
        The dismissal rate for currently employed workers
    """

    def __init__(self, λ, α):
        self.λ = λ
        self.α = α
        self._compute_implied()

    def _compute_implied(self):
        g = self.λ / (1 + self.λ)
        A = np.array([[1 - self.α, self.α], [g, 1 - g]])
        self._A = A
        self._g = g
        self._hat_A = A / (1 + g)

    def simulate(self, x0, t_end, dt=1):
        x = x0
        for t in range(t_end):
            x = self._hat_A @ x
        return x

    def steady_state(self):
        return self._g / (1 + self._g)
```

```

b : scalar
    Entry rate into the labor force
d : scalar
    Exit rate from the labor force

"""
def __init__(self, λ=0.283, α=0.013, b=0.0124, d=0.00822):
    self.λ, self.α, self.b, self.d = λ, α, b, d

    λ, α, b, d = self.λ, self.α, self.b, self.d
    self.g = b - d
    self.A = np.array([[((1-d) ** (1-λ)) + b,          (1 - d) ** α + b],
                      [(1-d) ** λ,          (1 - d) ** (1 - α)]])

    self.A_hat = self.A / (1 + self.g)

"""

def rate_steady_state(self, tol=1e-6):
    """
    Finds the steady state of the system :math:`\mathbf{x}_{t+1} = \hat{A} \mathbf{x}_t`

    Returns
    -----
    xbar : steady state vector of employment and unemployment rates
    """

    x = 0.5 * np.ones(2)
    error = tol + 1
    while error > tol:
        new_x = self.A_hat @ x
        error = np.max(np.abs(new_x - x))
        x = new_x
    return x

def simulate_stock_path(self, x0, T):
    """
    Simulates the sequence of Employment and Unemployment stocks

    Parameters
    -----
    X0 : array
        Contains initial values (E0, U0)
    T : int
        Number of periods to simulate

    Returns
    -----
    X : iterator
        Contains sequence of employment and unemployment stocks
    """

    X = np.atleast_1d(x0) # Recast as array just in case
    for t in range(T):
        yield X
        X = self.A @ X

def simulate_rate_path(self, x0, T):
    """
    Simulates the sequence of employment and unemployment rates

```

```

Parameters
-----
x0 : array
    Contains initial values (e0, u0)
T : int
    Number of periods to simulate

Returns
-----
x : iterator
    Contains sequence of employment and unemployment rates

"""
x = np.atleast_1d(x0) # Recast as array just in case
for t in range(T):
    yield x
    x = self.A_hat @ x

```

As explained, if we create an instance and update it by `lm = LakeModel($\alpha=0.03$)`, derived objects like A will also change.

```
In [4]: lm = LakeModel()
lm. $\alpha$ 
```

```
Out[4]: 0.013
```

```
In [5]: lm.A
```

```
Out[5]: array([[0.72350626, 0.02529314],
               [0.28067374, 0.97888686]])
```

```
In [6]: lm = LakeModel( $\alpha = 0.03$ )
lm.A
```

```
Out[6]: array([[0.72350626, 0.0421534 ],
               [0.28067374, 0.9620266 ]])
```

47.4.1 Aggregate Dynamics

Let's run a simulation under the default parameters (see above) starting from $X_0 = (12, 138)$

```

In [7]: lm = LakeModel()
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

fig, axes = plt.subplots(3, 1, figsize=(10, 8))
X_0 = (U_0, E_0)

```

```

X_path = np.vstack(tuple(lm.simulate_stock_path(X_0, T)))

axes[0].plot(X_path[:, 0], lw=2)
axes[0].set_title('Unemployment')

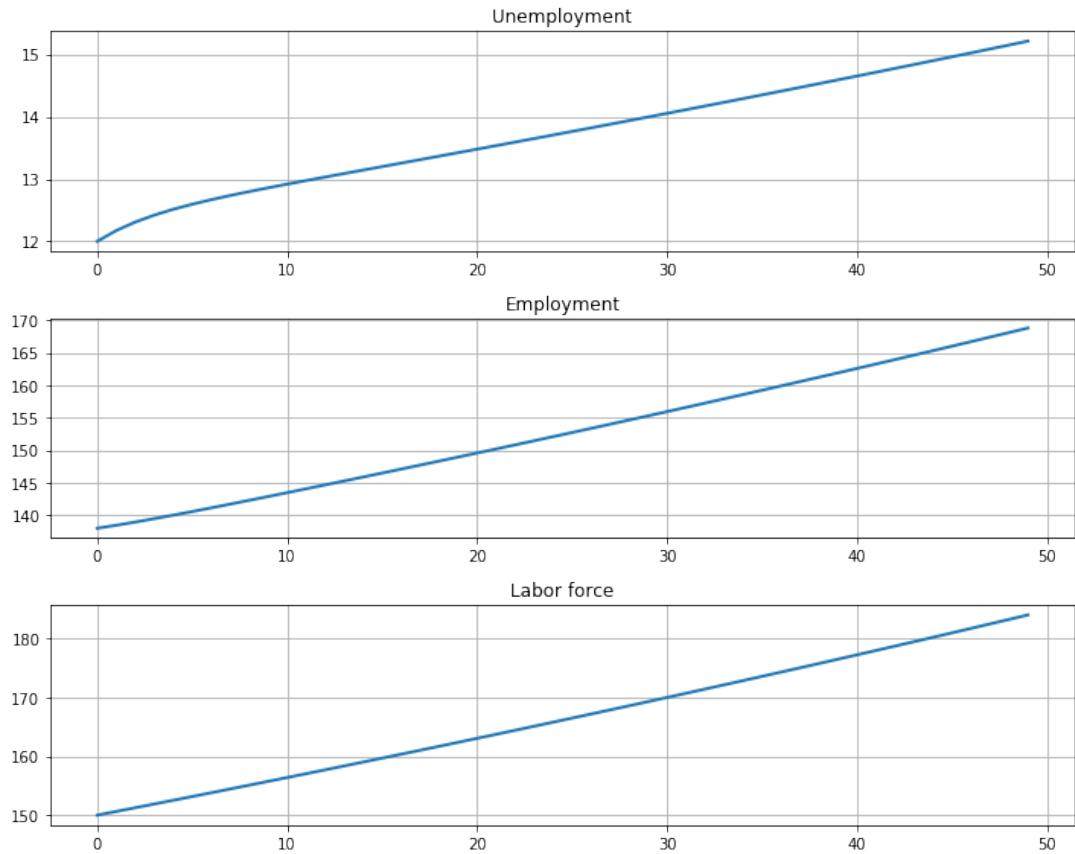
axes[1].plot(X_path[:, 1], lw=2)
axes[1].set_title('Employment')

axes[2].plot(X_path.sum(1), lw=2)
axes[2].set_title('Labor force')

for ax in axes:
    ax.grid()

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 .

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

- $\bar{x} = \hat{A}\bar{x}$
- the components satisfy $\bar{e} + \bar{u} = 1$

This equation tells us that a steady state level \bar{x} is an eigenvector of \hat{A} associated with a unit eigenvalue.

We also have $x_t \rightarrow \bar{x}$ as $t \rightarrow \infty$ provided that the remaining eigenvalue of \hat{A} has modulus less

than 1.

This is the case for our default parameters:

```
In [8]: lm = LakeModel()
e, f = np.linalg.eigvals(lm.A_hat)
abs(e), abs(f)
```

```
Out[8]: (0.6953067378358462, 1.0)
```

Let's look at the convergence of the unemployment and employment rate to steady state levels (dashed red line)

```
In [9]: lm = LakeModel()
e_0 = 0.92      # Initial employment rate
u_0 = 1 - e_0  # Initial unemployment rate
T = 50          # Simulation length

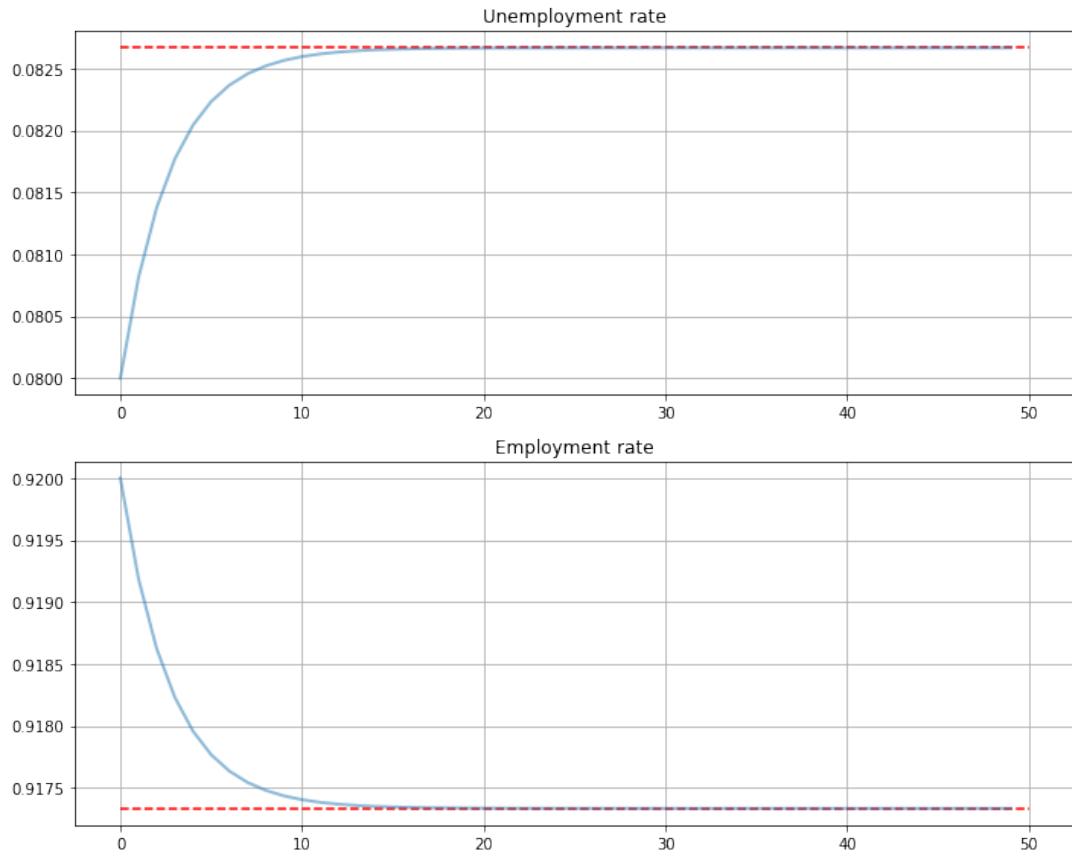
xbar = lm.rate_steady_state()

fig, axes = plt.subplots(2, 1, figsize=(10, 8))
x_0 = (u_0, e_0)
x_path = np.vstack(tuple(lm.simulate_rate_path(x_0, T)))

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, 'r', '--')
    axes[i].set_title(title)
    axes[i].grid()

plt.tight_layout()
plt.show()
```



47.5 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.

47.5.1 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

$$\lim_{T \rightarrow \infty} \bar{s}_{u,T} = \psi^*[0] \quad \text{and} \quad \lim_{T \rightarrow \infty} \bar{s}_{e,T} = \psi^*[1]$$

with probability one.

Inspection tells us that P is exactly the transpose of \hat{A} 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.

47.5.2 Convergence Rate

How long does it take for time series sample averages to converge to cross-sectional averages?

We can use [QuantEcon.py](#)'s `MarkovChain` class to investigate this.

Let's plot the path of the sample averages over 5,000 periods

```
In [10]: lm = LakeModel(d=0, b=0)
T = 5000 # Simulation length

α, λ = lm.α, lm.λ

P = [[1 - λ, λ],
      [α, 1 - α]]

mc = MarkovChain(P)

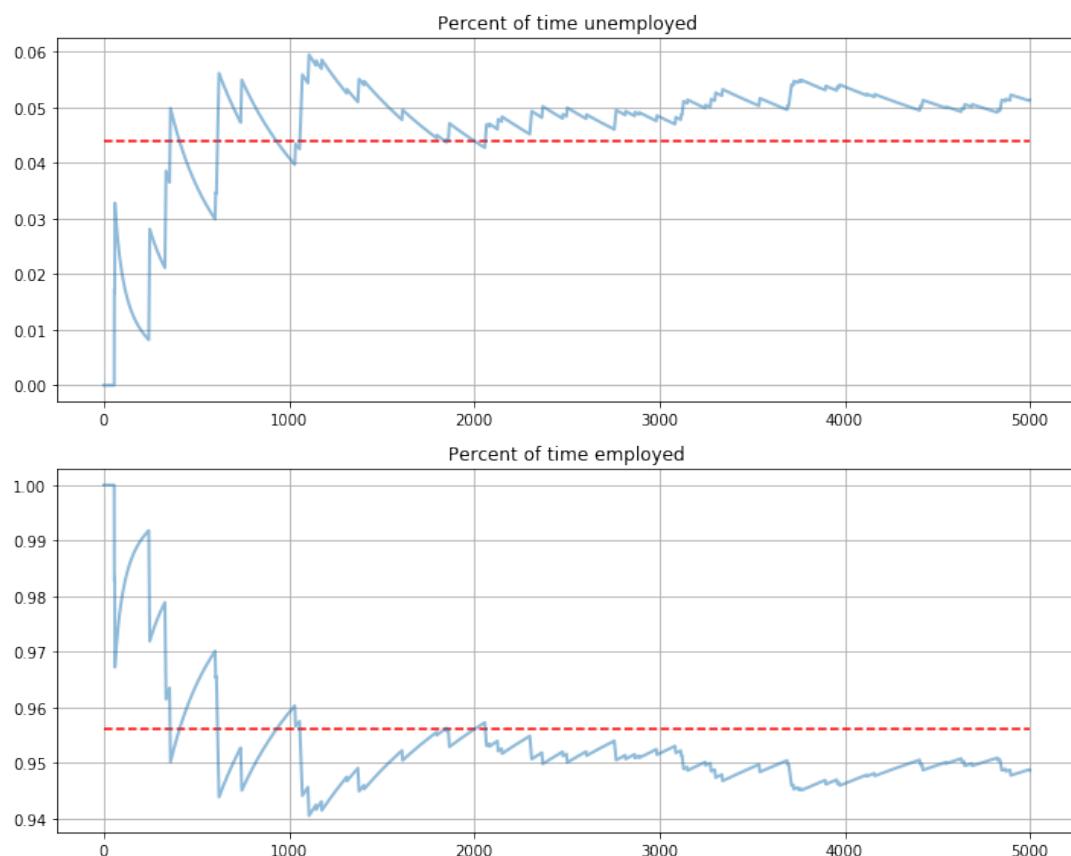
xbar = lm.rate_steady_state()

fig, axes = plt.subplots(2, 1, figsize=(10, 8))
s_path = mc.simulate(T, init=1)
s_bar_e = s_path.cumsum() / range(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, 'r', '--')
    axes[i].set_title(titles[i])
    axes[i].grid()

plt.tight_layout()
plt.show()
```



The stationary probabilities are given by the dashed red 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.

47.6 Endogenous Job Finding Rate

We now make the hiring rate endogenous.

The transition rate from unemployment to employment will be determined by the McCall search model [80].

All details relevant to the following discussion can be found in [our treatment](#) of that model.

47.6.1 Reservation Wage

The most important thing to remember about the model is that optimal decisions are 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.

As we saw in [our discussion of the model](#), 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

47.6.2 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

$$\lambda = \gamma \mathbb{P}\{w_t \geq \bar{w}\} = \gamma \sum_{w' \geq \bar{w}} p(w') \quad (1)$$

47.6.3 Fiscal Policy

We can use the McCall search version of the Lake Model 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.

Thus, 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 (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

$$\tau = u(c, \tau)c$$

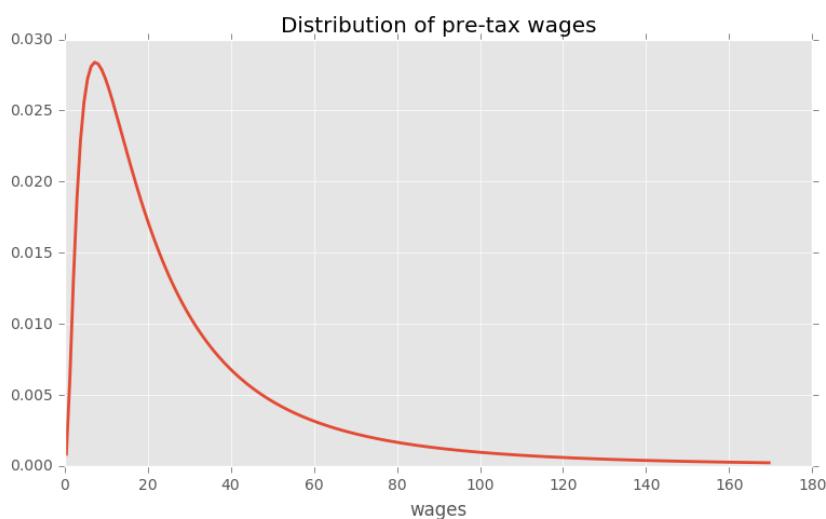
To evaluate alternative government tax-unemployment compensation pairs, we require a welfare criterion.

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 in the [McCall search model lecture](#).

The wage offer distribution will be a discretized version of the lognormal distribution $LN(\log(20), 1)$, as shown in the next figure



We take a period to be a month.

We set b and d to match monthly [birth](#) and [death rates](#), respectively, in the U.S. population

- $b = 0.0124$
- $d = 0.00822$

Following [24], we set α , the hazard rate of leaving employment, to

- $\alpha = 0.013$

47.6.4 Fiscal Policy Code

We will make use of techniques from the [McCall model lecture](#)

The first piece of code implements value function iteration

In [11]: # A default utility function

```
@jit
def u(c, σ):
    if c > 0:
        return (c**(1 - σ) - 1) / (1 - σ)
    else:
        return -10e6

class McCallModel:
    """
    Stores the parameters and functions associated with a given model.
    """

    def __init__(self,
                 α=0.2,          # Job separation rate
                 β=0.98,         # Discount rate
                 γ=0.7,          # Job offer rate
                 c=6.0,          # Unemployment compensation
                 σ=2.0,          # Utility parameter
                 w_vec=None,     # Possible wage values
                 p_vec=None):   # Probabilities over w_vec

        self.α, self.β, self.γ, self.c = α, β, γ, c
        self.σ = σ

        # Add a default wage vector and probabilities over the vector using
        # the beta-binomial distribution
        if w_vec is None:
            n = 60 # Number of possible outcomes for wage
            # Wages between 10 and 20
            self.w_vec = np.linspace(10, 20, n)
            a, b = 600, 400 # Shape parameters
            dist = BetaBinomial(n-1, a, b)
            self.p_vec = dist.pdf()
        else:
            self.w_vec = w_vec
            self.p_vec = p_vec

    @jit
    def _update_bellman(α, β, γ, c, σ, w_vec, p_vec, V, V_new, U):
        """
        A jitted function to update the Bellman equations. Note that V_new is
        
```

modified in place (i.e., modified by this function). The new value of U is returned.

```

"""
for w_idx, w in enumerate(w_vec):
    # w_idx indexes the vector of possible wages
    V_new[w_idx] = u(w, sigma) + beta * ((1 - alpha) * V[w_idx] + alpha * U)

U_new = u(c, sigma) + beta * (1 - gamma) * U + \
        beta * gamma * np.sum(np.maximum(U, V) * p_vec)

return U_new
"""

def solve_mccall_model(mcm, tol=1e-5, max_iter=2000):
    """
    Iterates to convergence on the Bellman equations

    Parameters
    -----
    mcm : an instance of McCallModel
    tol : float
        error tolerance
    max_iter : int
        the maximum number of iterations
    """

    V = np.ones(len(mcm.w_vec)) # Initial guess of V
    V_new = np.empty_like(V)    # To store updates to V
    U = 1                      # Initial guess of U
    i = 0
    error = tol + 1

    while error > tol and i < max_iter:
        U_new = _update_bellman(mcm.alpha, mcm.beta, mcm.gamma,
                                 mcm.c, mcm.sigma, mcm.w_vec, mcm.p_vec, V, V_new, U)
        error_1 = np.max(np.abs(V_new - V))
        error_2 = np.abs(U_new - U)
        error = max(error_1, error_2)
        V[:] = V_new
        U = U_new
        i += 1

    return V, U

```

The second piece of code is used to complete the reservation wage:

```
In [12]: def compute_reservation_wage(mcm, return_values=False):
    """
    Computes the reservation wage of an instance of the McCall model
    by finding the smallest  $w$  such that  $V(w) > U$ .

    If  $V(w) > U$  for all  $w$ , then the reservation wage  $w_{\bar{w}}$  is set to
    the lowest wage in mcm.w_vec.

    If  $V(w) < U$  for all  $w$ , then  $w_{\bar{w}}$  is set to np.inf.
    """

    if return_values:
        V = np.zeros(len(mcm.w_vec))
        for w in mcm.w_vec:
            V[w] = u(w, sigma)
        V[V <= U] = np.inf
        V[V > U] = 0
        return V[V != np.inf].min()
    else:
        V = np.zeros(len(mcm.w_vec))
        for w in mcm.w_vec:
            V[w] = u(w, sigma)
        V[V <= U] = np.inf
        V[V > U] = 0
        return V[V != np.inf].min()

```

Parameters

mcm : an instance of McCallModel
return_values : bool (optional, default=False)
Return the value functions as well

Returns

w_bar : scalar
The reservation wage

"""

```
V, U = solve_mccall_model(mcm)
w_idx = np.searchsorted(V - U, 0)

if w_idx == len(V):
    w_bar = np.inf
else:
    w_bar = mcm.w_vec[w_idx]

if return_values == False:
    return w_bar
else:
    return w_bar, V, U
```

Now let's compute and plot welfare, employment, unemployment, and tax revenue as a function of the unemployment compensation rate

In [13]: # Some global variables that will stay constant

```
α = 0.013
α_q = (1-(1-α)**3)    # Quarterly (α is monthly)
b = 0.0124
d = 0.00822
β = 0.98
γ = 1.0
σ = 2.0

# The default wage distribution --- a discretized lognormal
log_wage_mean, wage_grid_size, max_wage = 20, 200, 170
logw_dist = norm(np.log(log_wage_mean), 1)
w_vec = np.linspace(1e-8, max_wage, wage_grid_size + 1)
cdf = logw_dist.cdf(np.log(w_vec))
pdf = cdf[1:] - cdf[:-1]
p_vec = pdf / pdf.sum()
w_vec = (w_vec[1:] + w_vec[:-1]) / 2

def compute_optimal_quantities(c, τ):
    """
    Compute the reservation wage, job finding rate and value functions
    of the workers given c and τ.
    """
    mcm = McCallModel(α=α_q,
                      β=β,
```

```

    γ=γ,
    c=c-τ,           # Post tax compensation
    σ=σ,
    w_vec=w_vec-τ,  # Post tax wages
    p_vec=p_vec)

w_bar, V, U = compute_reservation_wage(mcm, return_values=True)
λ = γ * np.sum(p_vec[w_vec - τ > w_bar])
return w_bar, λ, V, U

def compute_steady_state_quantities(c, τ):
    """
    Compute the steady state unemployment rate given c and τ using optimal
    quantities from the McCall model and computing corresponding steady
    state quantities
    """

    w_bar, λ, V, U = compute_optimal_quantities(c, τ)

    # Compute steady state employment and unemployment rates
    lm = LakeModel(α=α_q, λ=λ, b=b, d=d)
    x = lm.rate_steady_state()
    u, e = x

    # Compute steady state welfare
    w = np.sum(V * p_vec * (w_vec - τ > w_bar)) / np.sum(p_vec * (w_vec -
        τ > w_bar))
    welfare = e * w + u * U

    return e, u, welfare

def find_balanced_budget_tax(c):
    """
    Find the tax level that will induce a balanced budget.
    """

    def steady_state_budget(t):
        e, u, w = compute_steady_state_quantities(c, t)
        return t - u * c

    τ = brentq(steady_state_budget, 0.0, 0.9 * c)
    return τ

# Levels of unemployment insurance we wish to study
c_vec = np.linspace(5, 140, 60)

tax_vec = []
unempl_vec = []
empl_vec = []
welfare_vec = []

for c in c_vec:
    t = find_balanced_budget_tax(c)
    e_rate, u_rate, welfare = compute_steady_state_quantities(c, t)
    tax_vec.append(t)
    unempl_vec.append(u_rate)
    empl_vec.append(e_rate)
    welfare_vec.append(welfare)

```

```

empl_vec.append(e_rate)
welfare_vec.append(welfare)

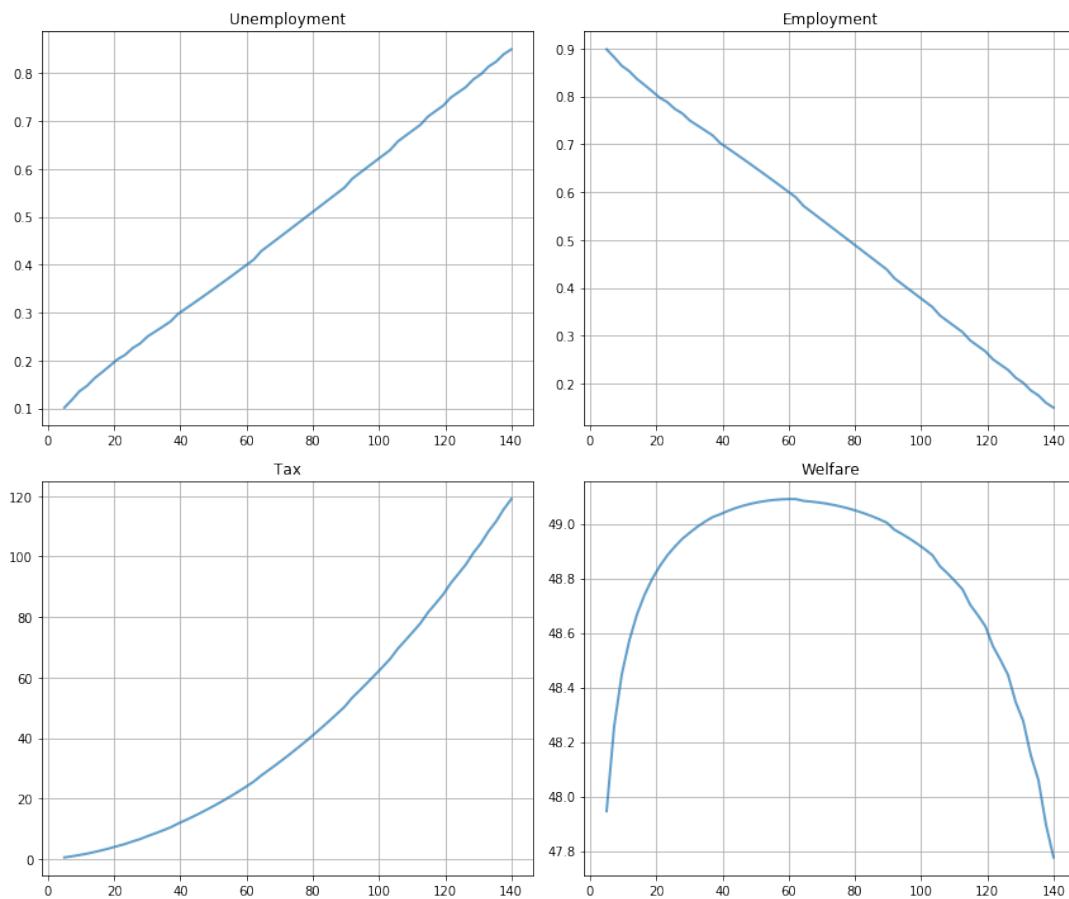
fig, axes = plt.subplots(2, 2, figsize=(12, 10))

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)
    ax.grid()

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.

47.7 Exercises

47.7.1 Exercise 1

In the Lake Model, there is derived data such as A which depends on primitives like α and λ .

So, when a user alters these primitives, we need the derived data to update automatically.

(For example, if a user changes the value of b for a given instance of the class, we would like $g = b - d$ to update automatically)

In the code above, we took care of this issue by creating new instances every time we wanted to change parameters.

That way the derived data is always matched to current parameter values.

However, we can use descriptors instead, so that derived data is updated whenever parameters are changed.

This is safer and means we don't need to create a fresh instance for every new parameterization.

(On the other hand, the code becomes denser, which is why we don't always use the descriptor approach in our lectures.)

In this exercise, your task is to arrange the `LakeModel` class by using descriptors and decorators such as `@property`.

(If you need to refresh your understanding of how these work, consult [this lecture](#).)

47.7.2 Exercise 2

Consider an economy with an initial stock of workers $N_0 = 100$ at the steady state level of employment in the baseline parameterization

- $\alpha = 0.013$
- $\lambda = 0.283$
- $b = 0.0124$
- $d = 0.00822$

(The values for α and λ follow [24])

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?

Note: it may be easier to use the class created in exercise 1 to help with changing variables.

47.7.3 Exercise 3

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?

47.8 Solutions

47.8.1 Exercise 1

```
In [14]: class LakeModelModified:
    """
    Solves the lake model and computes dynamics of unemployment stocks and
    rates.

    Parameters:
    -----
    λ : scalar
        The job finding rate for currently unemployed workers
    α : scalar
        The dismissal rate for currently employed workers
    b : scalar
        Entry rate into the labor force
    d : scalar
        Exit rate from the labor force
    """

    def __init__(self, λ=0.283, α=0.013, b=0.0124, d=0.00822):
        self._λ, self._α, self._b, self._d = λ, α, b, d
        self.compute_derived_values()

    def compute_derived_values(self):
        # Unpack names to simplify expression
        λ, α, b, d = self._λ, self._α, self._b, self._d

        self._g = b - d
        self._A = np.array([[((1-d) ** (1-λ)) + b, ((1 - d) ** α + b)],
                           [(1-d) ** λ, ((1 - d) ** (1 - α))]])

        self._A_hat = self._A / (1 + self._g)

    @property
    def g(self):
        return self._g

    @property
    def A(self):
        return self._A

    @property
    def A_hat(self):
        return self._A_hat

    @property
    def λ(self):
        return self._λ

    @λ.setter
    def λ(self, new_value):
        self._λ = new_value
        self.compute_derived_values()

    @property
```

```

def α(self):
    return self._α

@α.setter
def α(self, new_value):
    self._α = new_value
    self.compute_derived_values()

@property
def b(self):
    return self._b

@b.setter
def b(self, new_value):
    self._b = new_value
    self.compute_derived_values()

@property
def d(self):
    return self._d

@d.setter
def d(self, new_value):
    self._d = new_value
    self.compute_derived_values()

def rate_steady_state(self, tol=1e-6):
    """
    Finds the steady state of the system :math:`\hat{x}_{t+1} = \hat{A} x_t`
    """

    Returns
    -----
    xbar : steady state vector of employment and unemployment rates
    """
    x = 0.5 * np.ones(2)
    error = tol + 1
    while error > tol:
        new_x = self.A_hat @ x
        error = np.max(np.abs(new_x - x))
        x = new_x
    return x

def simulate_stock_path(self, X0, T):
    """
    Simulates the sequence of Employment and Unemployment stocks
    """

    Parameters
    -----
    X0 : array
        Contains initial values ( $E_0, U_0$ )
    T : int
        Number of periods to simulate

    Returns
    -----
    X : iterator
        Contains sequence of employment and unemployment stocks

```

```

"""
X = np.atleast_1d(x0) # Recast as array just in case
for t in range(T):
    yield X
    X = self.A @ X

def simulate_rate_path(self, x0, T):
"""
Simulates the sequence of employment and unemployment rates

Parameters
-----
x0 : array
    Contains initial values ( $e_0, u_0$ )
T : int
    Number of periods to simulate

Returns
-----
x : iterator
    Contains sequence of employment and unemployment rates

"""
x = np.atleast_1d(x0) # Recast as array just in case
for t in range(T):
    yield X
    X = self.A_hat @ X

```

47.8.2 Exercise 2

We begin by constructing the class containing the default parameters and assigning the steady state values to x_0

```
In [15]: lm = LakeModelModified()
x0 = lm.rate_steady_state()
print(f"Initial Steady State: {x0}")
```

```
Initial Steady State: [0.08266806 0.91733194]
```

Initialize the simulation values

```
In [16]: N0 = 100
T = 50
```

New legislation changes λ to 0.2

```
In [17]: lm.lam = 0.2
```

```
xbar = lm.rate_steady_state() # new steady state
X_path = np.vstack(tuple(lm.simulate_stock_path(x0 * N0, T)))
x_path = np.vstack(tuple(lm.simulate_rate_path(x0, T)))
print(f"New Steady State: {xbar}")
```

```
New Steady State: [0.11309573 0.88690427]
```

Now plot stocks

```
In [18]: fig, axes = plt.subplots(3, 1, figsize=[10, 9])

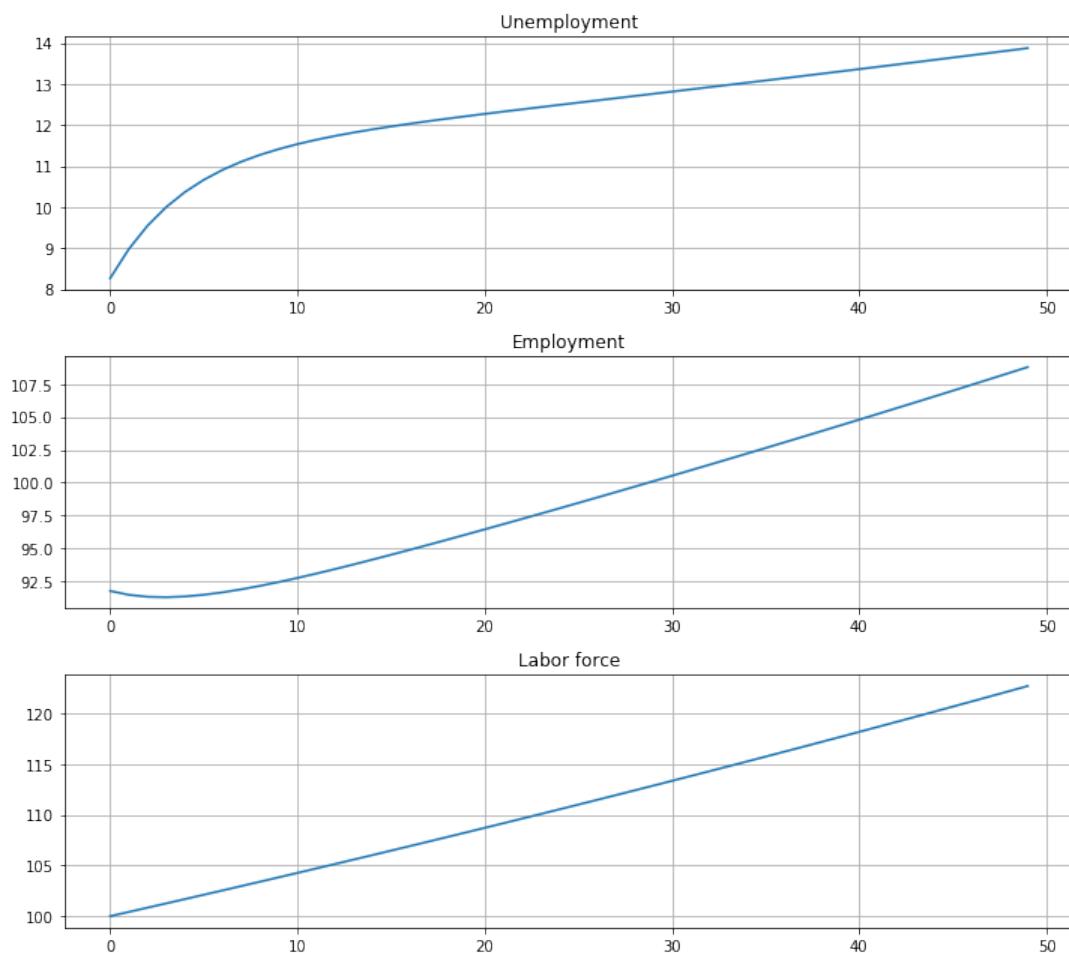
axes[0].plot(X_path[:, 0])
axes[0].set_title('Unemployment')

axes[1].plot(X_path[:, 1])
axes[1].set_title('Employment')

axes[2].plot(X_path.sum(1))
axes[2].set_title('Labor force')

for ax in axes:
    ax.grid()

plt.tight_layout()
plt.show()
```



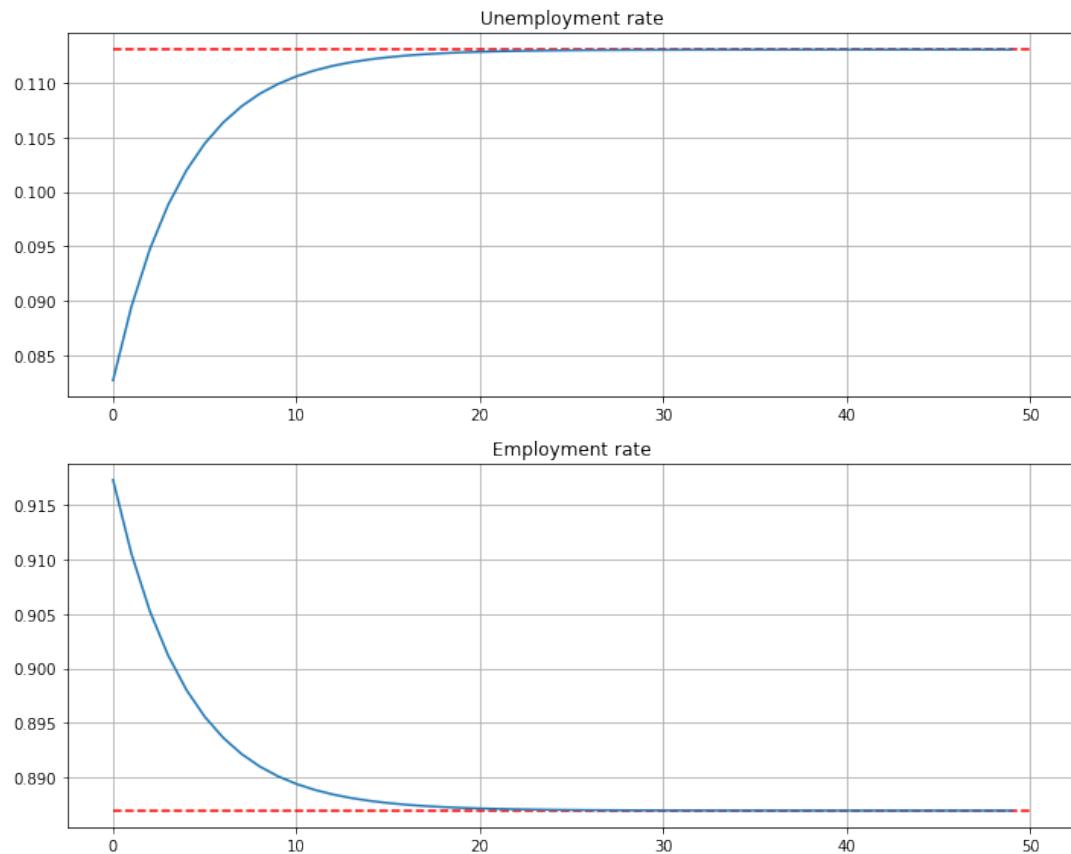
And how the rates evolve

```
In [19]: 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, 'r', '--')
    axes[i].set_title(title)
    axes[i].grid()

plt.tight_layout()
plt.show()
```



We see that it takes 20 periods for the economy to converge to its new steady state levels.

47.8.3 Exercise 3

This next 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

```
In [20]: lm = LakeModelModified()
x0 = lm.rate_steady_state()
```

Here are the other parameters:

```
In [21]: b_hat = 0.025
T_hat = 20
```

Let's increase b to the new value and simulate for 20 periods

```
In [22]: lm.b = b_hat
# Simulate stocks
X_path1 = np.vstack(tuple(lm.simulate_stock_path(x0 * N0, T_hat)))
# Simulate rates
x_path1 = np.vstack(tuple(lm.simulate_rate_path(x0, T_hat)))
```

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

```
In [23]: lm.b = 0.0124
# Simulate stocks
X_path2 = np.vstack(tuple(lm.simulate_stock_path(X_path1[-1, :2], T-
T_hat+1)))
# Simulate rates
x_path2 = np.vstack(tuple(lm.simulate_rate_path(x_path1[-1, :2], T-
T_hat+1)))
```

Finally, we combine these two paths and plot

```
In [24]: # note [1:] to avoid doubling period 20
x_path = np.vstack([x_path1, x_path2[1:]])
X_path = np.vstack([X_path1, X_path2[1:]])

fig, axes = plt.subplots(3, 1, figsize=[10, 9])

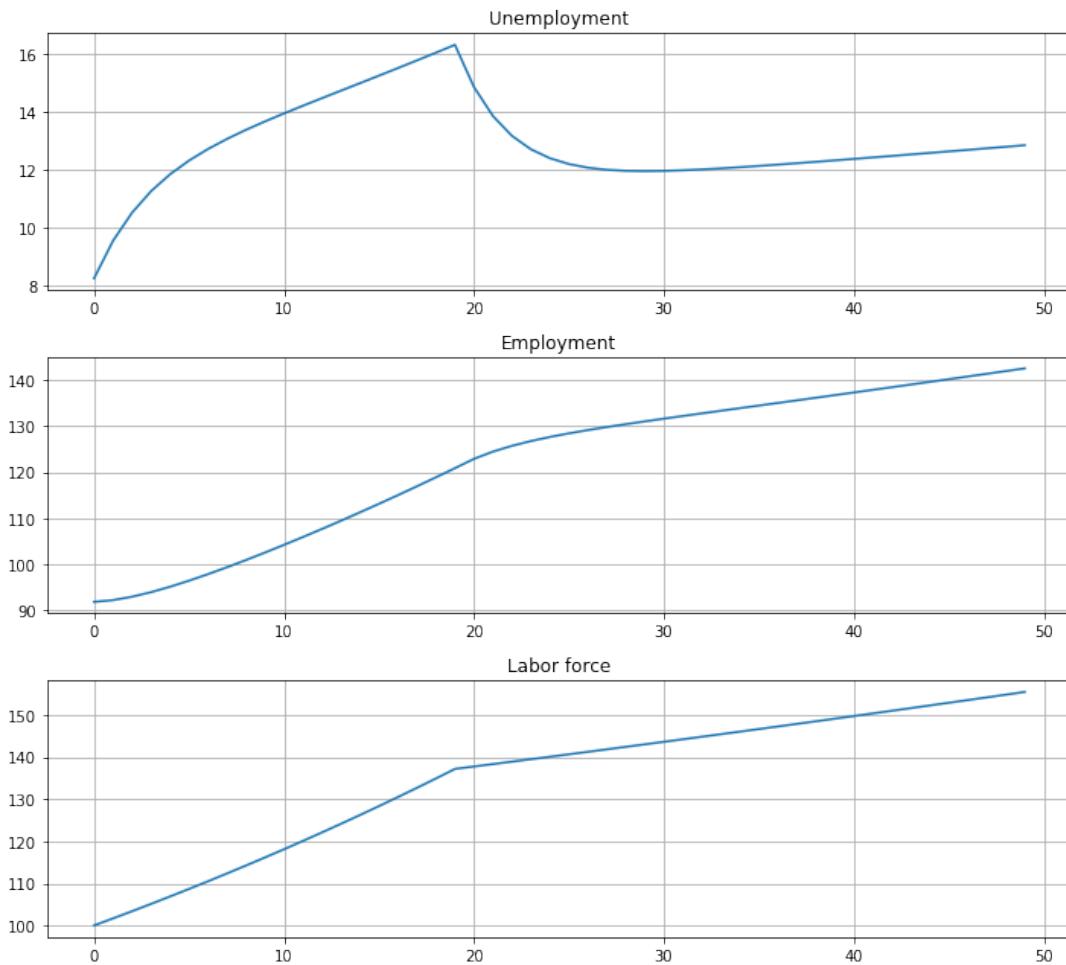
axes[0].plot(X_path[:, 0])
axes[0].set_title('Unemployment')

axes[1].plot(X_path[:, 1])
axes[1].set_title('Employment')

axes[2].plot(X_path.sum(1))
axes[2].set_title('Labor force')

for ax in axes:
    ax.grid()

plt.tight_layout()
plt.show()
```



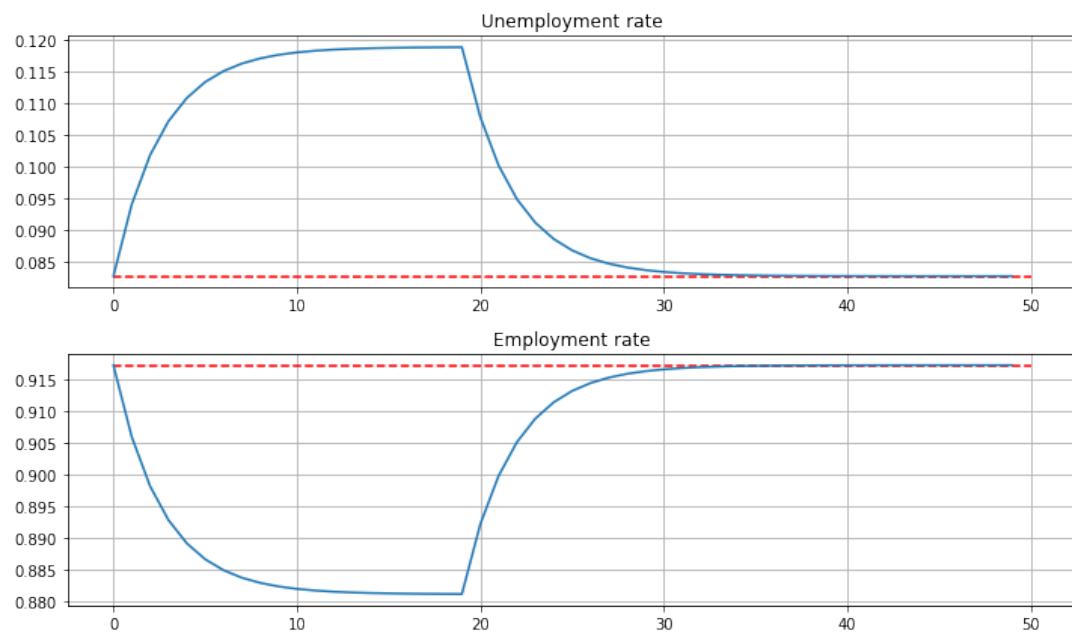
And the rates

```
In [25]: 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, 'r', '--')
    axes[i].set_title(title)
    axes[i].grid()

plt.tight_layout()
plt.show()
```



Chapter 48

Rational Expectations Equilibrium

48.1 Contents

- Overview 48.2
- Defining Rational Expectations Equilibrium 48.3
- Computation of an Equilibrium 48.4
- Exercises 48.5
- Solutions 48.6

“If you’re so smart, why aren’t you rich?”

In addition to what’s in Anaconda, this lecture will need the following libraries:

In [1]: `!pip install quantecon`

48.2 Overview

This lecture introduces the concept of *rational expectations equilibrium*.

To illustrate it, we describe a linear quadratic version of a famous and important model due to Lucas and Prescott [74].

This 1971 paper is one of a small number of research articles that kicked off the *rational expectations revolution*.

We follow Lucas and Prescott by employing a setting that is readily “Bellmanized” (i.e., capable of being formulated in terms of dynamic programming problems).

Because we use linear quadratic setups for demand and costs, we can adapt 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 rational expectations quantities.

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:

```
In [2]: import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

We’ll also use the `LQ` class from `QuantEcon.py`.

```
In [3]: from quantecon import LQ
```

```
/home/ubuntu/anaconda3/lib/python3.7/site-packages/numba/np/ufunc/parallel.py:
 355:
NumbaWarning: The TBB threading layer requires TBB version 2019.5 or later i.e.,
TBB_INTERFACE_VERSION >= 11005. Found TBB_INTERFACE_VERSION = 11004. The TBB
threading
layer is disabled.
warnings.warn(problem)
```

48.2.1 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.

We want to impose that

- The representative firm or individual takes *aggregate Y* as given when it chooses individual y , but
- At the end of the day, $Y = y$, 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 collection of n firms produce a homogeneous good that is sold in a competitive market.

Each of these n firms sells output y .

The price p of the good lies on an inverse demand curve

$$p = a_0 - a_1 Y \quad (1)$$

where

- $a_i > 0$ for $i = 0, 1$
- $Y = ny$ is the market-wide level of output

Each firm has a total cost function

$$c(y) = c_1 y + 0.5 c_2 y^2, \quad c_i > 0 \text{ for } i = 1, 2$$

The profits of a representative firm are $py - c(y)$.

Using (1), we can express the problem of the representative firm as

$$\max_y \left[(a_0 - a_1 Y) y - c_1 y - 0.5 c_2 y^2 \right] \quad (2)$$

In posing problem (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 (2).

This assures that the firm is a price taker.

The first-order condition for problem (2) is

$$a_0 - a_1 Y - c_1 - c_2 y = 0 \quad (3)$$

At this point, *but not before*, we substitute $Y = ny$ into (3) to obtain the following linear equation

$$a_0 - c_1 - (a_1 + n^{-1} c_2) Y = 0 \quad (4)$$

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 (1).

48.2.2 Further Reading

References for this lecture include

- [74]
- [95], chapter XIV
- [72], chapter 7

48.3 Defining Rational Expectations Equilibrium

Our first illustration of a rational expectations equilibrium involves a market with n firms, each of which seeks to maximize the discounted present value of profits in the face of adjust-

ment 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 supplies.

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.

48.3.1 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 of these n firms sell output y_t .

The price p_t of the good lies on the inverse demand curve

$$p_t = a_0 - a_1 Y_t \quad (5)$$

where

- $a_i > 0$ for $i = 0, 1$
- $Y_t = ny_t$ 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

$$\sum_{t=0}^{\infty} \beta^t r_t \quad (6)$$

where

$$r_t := p_t y_t - \frac{\gamma(y_{t+1} - y_t)^2}{2}, \quad y_0 \text{ given} \quad (7)$$

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 (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.

We assume that n is such a large number that the output of any single firm has a negligible effect on aggregate output.

That justifies firms in regarding their forecasts of aggregate output as being unaffected by their own output decisions.

The Firm's Beliefs

We suppose the firm believes that market-wide output Y_t follows the law of motion

$$Y_{t+1} = H(Y_t) \quad (8)$$

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 (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

$$v(y, Y) = \max_{y'} \left\{ a_0 y - a_1 y Y - \frac{\gamma(y' - y)^2}{2} + \beta v(y', H(Y)) \right\} \quad (9)$$

Let's denote the firm's optimal policy function by h , so that

$$y_{t+1} = h(y_t, Y_t) \quad (10)$$

where

$$h(y, Y) := \operatorname{argmax}_{y'} \left\{ a_0 y - a_1 y Y - \frac{\gamma(y' - y)^2}{2} + \beta v(y', H(Y)) \right\} \quad (11)$$

Evidently v and h both depend on H .

A First-Order Characterization

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

$$-\gamma(y' - y) + \beta v_y(y', H(Y)) = 0 \quad (12)$$

An important useful envelope result of Benveniste-Scheinkman [11] implies that to differentiate v with respect to y we can naively differentiate the right side of (9), giving

$$v_y(y, Y) = a_0 - a_1 Y + \gamma(y' - y)$$

Substituting this equation into (12) gives the *Euler equation*

$$-\gamma(y_{t+1} - y_t) + \beta[a_0 - a_1 Y_{t+1} + \gamma(y_{t+2} - y_{t+1})] = 0 \quad (13)$$

The firm optimally sets an output path that satisfies (13), taking (8) as given, and subject to

- the initial conditions for (y_0, Y_0) .
- the terminal condition $\lim_{t \rightarrow \infty} \beta^t y_t v_y(y_t, Y_t) = 0$.

This last condition is called the *transversality condition*, and acts as a first-order necessary condition “at infinity”.

The firm’s decision rule solves the difference equation (13) subject to the given initial condition y_0 and the transversality condition.

Note that solving the Bellman equation (9) for v and then h in (11) yields a decision rule that automatically imposes both the Euler equation (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 $Y_t = ny_t$, the *actual law of motion* for market-wide output is then

$$Y_{t+1} = nh(Y_t/n, Y_t) \quad (14)$$

Thus, when firms believe that the law of motion for market-wide output is (8), their optimizing behavior makes the actual law of motion be (14).

48.3.2 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 $H(Y) = nh(Y/n, Y)$ for all Y .

Thus, a rational expectations equilibrium equates the perceived and actual laws of motion (8) and (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 operations, taking a perceived law of motion into a decision rule via (9)–(11), and a decision rule into an actual law via (14).

The H component of a rational expectations equilibrium is a fixed point of Φ .

48.4 Computation of an Equilibrium

Now let's consider the problem of computing the rational expectations equilibrium.

48.4.1 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.

In particular, there is no guarantee that direct iterations on Φ converge Section ??.

Furthermore, there are examples in which these iterations diverge.

Fortunately, there is another method that works here.

The method exploits a connection between equilibrium and Pareto optimality expressed in the fundamental theorems of welfare economics (see, e.g, [79]).

Lucas and Prescott [74] used this method to construct a rational expectations equilibrium.

The details follow.

48.4.2 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](#)).

The 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.

For convenience, in this section, we set $n = 1$.

We first compute a sum of consumer and producer surplus at time t

$$s(Y_t, Y_{t+1}) := \int_0^{Y_t} (a_0 - a_1 x) dx - \frac{\gamma(Y_{t+1} - Y_t)^2}{2} \quad (15)$$

The first term is the area under the demand curve, while the second measures the social costs of changing output.

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 .

48.4.3 Solution of the Planning Problem

Evaluating the integral in (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

$$V(Y) = \max_{Y'} \left\{ a_0 Y - \frac{a_1}{2} Y^2 - \frac{\gamma(Y' - Y)^2}{2} + \beta V(Y') \right\} \quad (16)$$

The associated first-order condition is

$$-\gamma(Y' - Y) + \beta V'(Y') = 0 \quad (17)$$

Applying the same Benveniste-Scheinkman formula gives

$$V'(Y) = a_0 - a_1 Y + \gamma(Y' - Y)$$

Substituting this into equation (17) and rearranging leads to the Euler equation

$$\beta a_0 + \gamma Y_t - [\beta a_1 + \gamma(1 + \beta)] Y_{t+1} + \gamma \beta Y_{t+2} = 0 \quad (18)$$

48.4.4 The Key Insight

Return to equation (13) and set $y_t = Y_t$ for all t .

(Recall that for this section we've set $n = 1$ to simplify the calculations)

A small amount of algebra will convince you that when $y_t = Y_t$, equations (18) and (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 $Y_t = ny_t$ 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 (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 problem implies an optimal policy — and hence aggregate law of motion — taking the form

$$Y_{t+1} = \kappa_0 + \kappa_1 Y_t \quad (19)$$

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 (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

$$y_{t+1} = h_0 + h_1 y_t + h_2 Y_t \quad (20)$$

Hence a rational expectations equilibrium will be defined by the parameters $(\kappa_0, \kappa_1, h_0, h_1, h_2)$ in (19)–(20).

48.5 Exercises

48.5.1 Exercise 1

Consider the firm problem described above.

Let the firm's belief function H be as given in (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 (20) and give the values for each h_j .

If there were n identical competitive firms all behaving according to (20), what would (20) imply for the *actual* law of motion (8) for market supply.

48.5.2 Exercise 2

Consider the following κ_0, κ_1 pairs as candidates for the aggregate law of motion component of a rational expectations equilibrium (see (19)).

Extending the program that you wrote for exercise 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 1 to compute a rational expectations equilibrium.

(You are not being asked actually to use the algorithm you are suggesting)

48.5.3 Exercise 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
 - $a_0 = 100, a_1 = 0.05, \beta = 0.95, \gamma = 10$
1. Represent the solution in the form $Y_{t+1} = \kappa_0 + \kappa_1 Y_t$.
2. Compare your answer with the results from exercise 2.

48.5.4 Exercise 4

A monopolist faces the industry demand curve (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 the previous exercise.

In particular, solve for the parameters in

$$Y_{t+1} = m_0 + m_1 Y_t$$

Compare your results with the previous exercise – comment.

48.6 Solutions

48.6.1 Exercise 1

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

$$x_t = \begin{bmatrix} y_t \\ Y_t \\ 1 \end{bmatrix}, \quad u_t = y_{t+1} - y_t$$

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

- $x'_t R x_t + u'_t Q u_t = -r_t$
- $x_{t+1} = Ax_t + Bu_t$

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

In [4]: # Model parameters

```
a0 = 100
a1 = 0.05
β = 0.95
γ = 10.0

# Beliefs

x0 = 95.5
x1 = 0.95

# Formulate the LQ problem

A = np.array([[1, 0, 0], [0, x1, x0], [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 $Y_t = ny_t$, which, combined with the previous equation, yields

$$Y_{t+1} = n(96.949 + y_t - 0.046 Y_t) = n96.949 + (1 - n0.046)Y_t$$

48.6.2 Exercise 2

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

```

In [5]: candidates = ((94.0886298678, 0.923409232937),
                     (93.2119845412, 0.984323478873),
                     (95.0818452486, 0.952459076301))

for x0, x1 in candidates:

    # Form the associated law of motion
    A = np.array([[1, 0, 0], [0, x1, x0], [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((x0, x1), (h0, h1 + h2)):
    print(f'Equilibrium pair = {x0}, {x1}')
    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 $(h_0, h_1, h_2) = (95.0819, 1.0000, -0.475)$.

(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)

48.6.3 Exercise 3

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

- $x'_t R x_t + u'_t Q u_t = -s(Y_t, Y_{t+1})$
- $x_{t+1} = Ax_t + Bu_t$

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:

In [6]: # 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()
x0, x1 = -F[1], 1 - F[0]
print(x0, x1)

95.08187459215002 0.9524590627039248
```

The output yields the same (κ_0, κ_1) pair obtained as an equilibrium from the previous exercise.

48.6.4 Exercise 4

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

```
In [7]: 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)

73.47294403502818 0.9265270559649701
```

We see that the law of motion for the monopolist is approximately $Y_{t+1} = 73.4729 + 0.9265Y_t$.

In the rational expectations case, the law of motion was approximately $Y_{t+1} = 95.0818 + 0.9525Y_t$.

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

Footnotes

- [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 [77] and [36] 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.

Chapter 49

Stability in Linear Rational Expectations Models

49.1 Contents

- Overview 49.2
- Linear difference equations 49.3
- Illustration: Cagan's Model 49.4
- Some Python code 49.5
- Alternative code 49.6
- Another perspective 49.7
- Log money supply feeds back on log price level 49.8
- Big P , little p interpretation 49.9
- Fun with SymPy code 49.10

In addition to what's in Anaconda, this lecture deploys the following libraries:

```
In [1]: !pip install quantecon
```

```
In [2]: import numpy as np
import quantecon as qe
import matplotlib.pyplot as plt
%matplotlib inline
from sympy import *
init_printing()

/home/ubuntu/anaconda3/lib/python3.7/site-packages/numba/np/ufunc/parallel.py:
  ↵355:
NumbaWarning: The TBB threading layer requires TBB version 2019.5 or later i.e.,
TBB_INTERFACE_VERSION >= 11005. Found TBB_INTERFACE_VERSION = 11004. The TBB
  ↵threading
layer is disabled.
warnings.warn(problem)
```

49.2 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 [18] linking the price level to the money supply.

Cagan did not use a rational expectations version of his model, but Sargent [94] 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) [?] and Whiteman (1983) [110] 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

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.

49.3 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.

49.3.1 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

$$(1 - \lambda L)y_t = u_t, \forall t \quad (1)$$

has solutions

$$y_t = (1 - \lambda L)^{-1}u_t + k\lambda^t \quad (2)$$

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 (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 (1) as

$$y_{t-1} = \lambda^{-1}y_t - \lambda^{-1}u_t, \forall t \quad (3)$$

or

$$(1 - \lambda^{-1}L^{-1})y_t = -\lambda^{-1}u_{t+1}. \quad (4)$$

A solution is

$$y_t = -\lambda^{-1} \left(\frac{1}{1 - \lambda^{-1}L^{-1}} \right) u_{t+1} + k\lambda^t \quad (5)$$

for any k .

To verify that this is a solution, check the consequences of operating on both sides of equation (5) by $(1 - \lambda L)$ and compare to equation (1).

For any bounded $\{u_t\}$ sequence, solution (2) exists for $|\lambda| < 1$ because the **distributed lag** in u converges.

Solution (5) exists when $|\lambda| > 1$ because the **distributed lead** in u converges.

When $|\lambda| > 1$, the distributed lag in u in (2) may diverge, in which case a solution of this form does not exist.

The distributed lead in u in (5) need not converge when $|\lambda| < 1$.

49.3.2 Second order

Now consider the second order difference equation

$$(1 - \lambda_1 L)(1 - \lambda_2 L)y_{t+1} = u_t \quad (6)$$

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 (6). Using insights from our analysis of the first-order equation, operate on both sides of (6) by the forward inverse of $(1 - \lambda_2 L)$ to rewrite equation (6) as

$$(1 - \lambda_1 L)y_{t+1} = -\frac{\lambda_2^{-1}}{1 - \lambda_2^{-1}L^{-1}}u_{t+1}$$

or

$$y_{t+1} = \lambda_1 y_t - \lambda_2^{-1} \sum_{j=0}^{\infty} \lambda_2^{-j} u_{t+j+1}. \quad (7)$$

Thus, we obtained equation (7) by solving a stable root (in this case λ_1) **backward**, and an unstable root (in this case λ_2) **forward**.

Equation (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**

49.4 Illustration: Cagan's Model

Now let's use linear difference equations to represent and solve Sargent's [94] rational expectations version of Cagan's model [18] 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 [94]

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

$$p_t = (1 - \lambda)m_t + \lambda p_{t+1} \quad (8)$$

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 (8) forward gives

$$p_t = (1 - \lambda) \sum_{j=0}^{\infty} \lambda^j m_{t+j}, \quad (9)$$

which is the unique **stable** solution of difference equation (8) among a class of more general solutions

$$p_t = (1 - \lambda) \sum_{j=0}^{\infty} \lambda^j m_{t+j} + c\lambda^{-t} \quad (10)$$

that is indexed by the real number $c \in \mathbf{R}$.

Because we want to focus on stable solutions, we set $c = 0$.

Equation (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

$$\begin{aligned} m_t &= Gx_t \\ x_{t+1} &= Ax_t \end{aligned} \quad (11)$$

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 into state space system (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 (8) that obeys the system comprised of (8)-(11).

By stable or non-explosive, we mean that neither m_t nor p_t diverges as $t \rightarrow +\infty$.

This requires that we shut down the term $c\lambda^{-t}$ in equation (10) above by setting $c = 0$

The solution we are after is

$$p_t = Fx_t \quad (12)$$

where

$$F = (1 - \lambda)G(I - \lambda A)^{-1} \quad (13)$$

Note: As mentioned above, an *explosive solution* of difference equation (8) can be constructed by adding to the right hand of (12) a sequence $c\lambda^{-t}$ where c is an arbitrary positive constant.

49.5 Some Python code

We'll construct examples that illustrate (11).

Our first example takes as the law of motion for the log money supply the second order difference equation

$$m_{t+1} = \alpha + \rho_1 m_t + \rho_2 m_{t-1} \quad (14)$$

that is parameterized by ρ_1, ρ_2, α

To capture this parameterization with system (9) we set

$$x_t = \begin{bmatrix} 1 \\ m_t \\ m_{t-1} \end{bmatrix}, \quad A = \begin{bmatrix} 1 & 0 & 0 \\ \alpha & \rho_1 & \rho_2 \\ 0 & 1 & 0 \end{bmatrix}, \quad G = [0 \ 1 \ 0]$$

Here is Python code

In [3]: `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.

In [4]: `eigvals = np.linalg.eigvals(A)`
`print(eigvals)`

```
[ -0.05249378  0.95249378  1. ]
```

In [5]: `(abs(eigvals) <= 1).all()`

Out[5]: True

Now let's compute F in formulas (12) and (13).

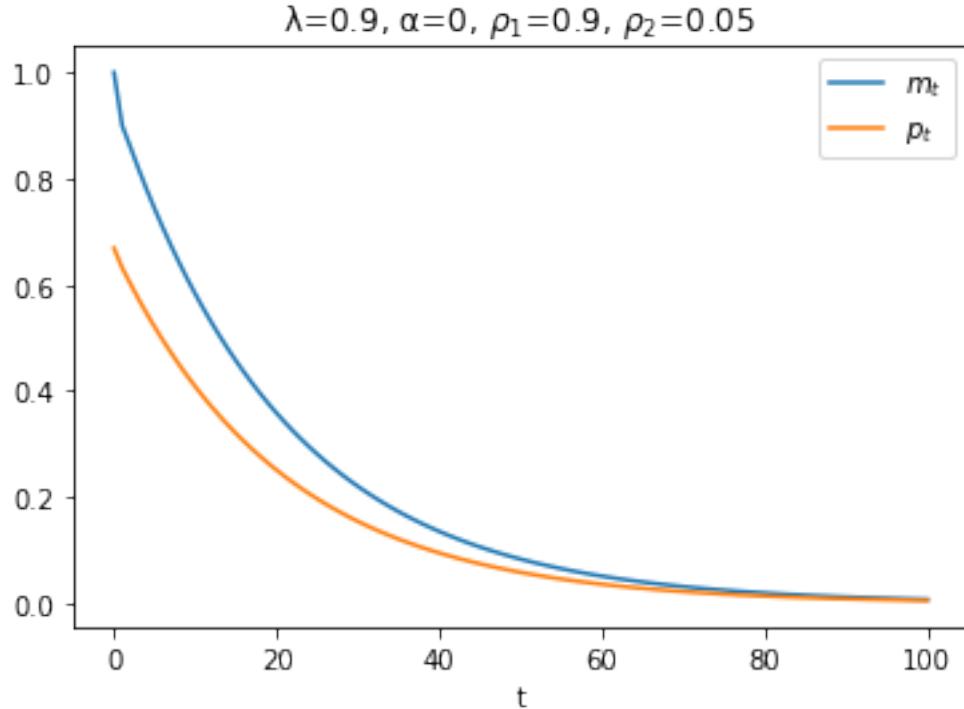
In [6]: `# compute the solution, i.e. formula (3)`
`F = (1 - λ) * G @ np.linalg.inv(np.eye(A.shape[0])) - λ * A`
`print("F=", F)`

```
F= [[0.          0.66889632  0.03010033]]
```

Now let's simulate paths of m_t and p_t starting from an initial value x_0 .

In [7]: `# 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`

In [8]: `plt.figure()`
`plt.plot(range(T+1), m_seq, label='m_t')`
`plt.plot(range(T+1), p_seq, label='p_t')`
`plt.xlabel('t')`
`plt.title(f'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 (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

49.6 Alternative code

We could also have run the simulation using the quantecon **LinearStateSpace** code.

The following code block performs the calculation with that code.

In [9]: # 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)
```

In [10]: T = 100

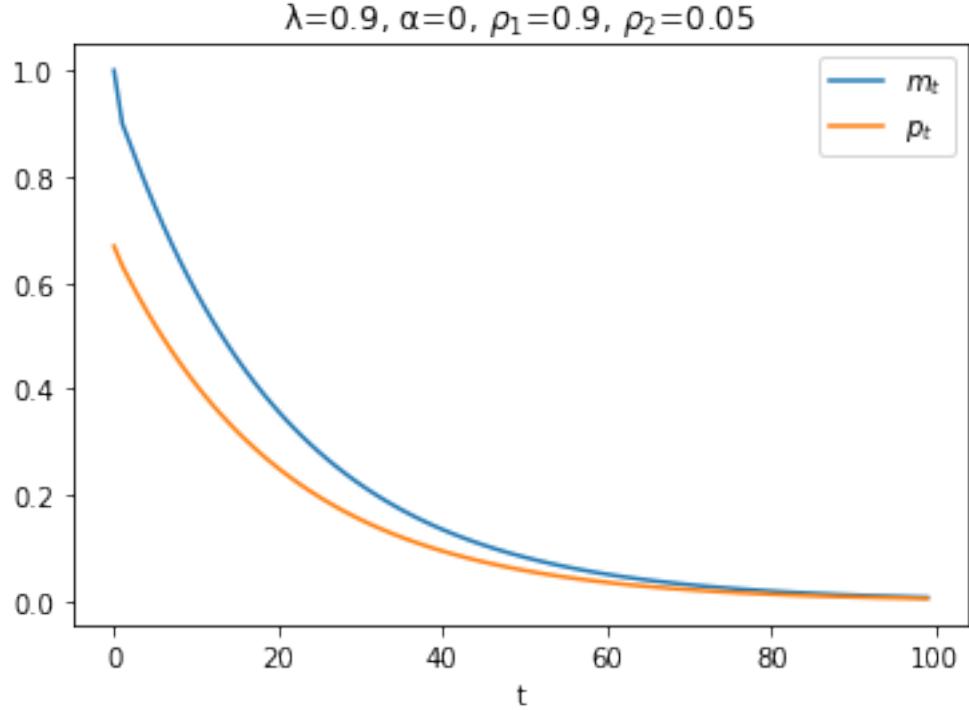
```
# simulate using LinearStateSpace
x, y = ss.simulate(ts_length=T)

# 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'$\lambda=\lambda$, $\alpha=\alpha$, $\rho_1=\rho_1$, $\rho_2=\rho_2$')
plt.legend()
plt.show()

```



49.6.1 Special case

To simplify our presentation in ways that will let focus on an important idea, in the above second-order difference equation (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

$$m_{t+1} = \rho m_t \quad (15)$$

and the state x_t becomes

$$x_t = m_t.$$

Consequently, we can set $G = 1$, $A = \rho$ making our formula (13) for F become

$$F = (1 - \lambda)(1 - \lambda\rho)^{-1}.$$

so that the log price level satisfies

$$p_t = Fm_t.$$

Please keep these formulas in mind as we investigate an alternative route to and interpretation of our formula for F .

49.7 Another perspective

Above, we imposed stability or non-explosiveness on the solution of the key difference equation (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 (8) and (15) to form the system

$$\begin{bmatrix} m_{t+1} \\ p_{t+1} \end{bmatrix} = \begin{bmatrix} \rho & 0 \\ -(1-\lambda)/\lambda & \lambda^{-1} \end{bmatrix} \begin{bmatrix} m_t \\ p_t \end{bmatrix} \quad (16)$$

or

$$y_{t+1} = Hy_t, \quad t \geq 0 \quad (17)$$

where

$$H = \begin{bmatrix} \rho & 0 \\ -(1-\lambda)/\lambda & \lambda^{-1} \end{bmatrix}. \quad (18)$$

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 (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 $t \rightarrow +\infty$.

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

$$y_0^* = \begin{bmatrix} y_{1,0}^* \\ 0 \end{bmatrix} \quad (19)$$

the path of y_t^* and therefore the paths of both components of $y_t = Qy_t^*$ will diverge in absolute value as $t \rightarrow +\infty$. (We say that the paths *explode*)

Equation (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 (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

$$Q^{21}m_0 + Q^{22}p_0 = 0 \quad (20)$$

where Q^{ij} denotes the (i, j) component of Q^{-1} .

Solving this equation for p_0 , we find

$$p_0 = -(Q^{22})^{-1}Q^{21}m_0. \quad (21)$$

This is the unique **stabilizing value** of p_0 expressed as a function of m_0 .

49.7.1 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

$$p_0 = Q_{21}Q_{11}^{-1}m_0. \quad (22)$$

It can be verified that this formula replicates itself over time in the sense that

$$p_t = Q_{21}Q_{11}^{-1}m_t. \quad (23)$$

To implement formula (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

$$Q_1 = \begin{bmatrix} Q_{11} \\ Q_{21} \end{bmatrix}.$$

49.7.2 Some remarks about feedback

We have expressed (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} , $j = 0, 1, 2, \dots$.

A tell-tale sign that we should look beyond its superficial “feedback” form is that $\lambda^{-1} > 1$ so that the matrix H in (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.

49.8 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

$$m_{t+1} = \rho m_t + \delta p_t \quad (24)$$

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 (8) continues to describe equality between the demand and supply of money.

We assume that equations (8) and (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

$$y_{t+1} = Hy_t$$

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 .

```
In [11]: def construct_H(p, λ, δ):
    "construct matrix H given parameters."
    H = np.empty((2, 2))
    H[0, :] = p, δ
    H[1, :] = - (1 - λ) / λ, 1 / λ

    return H

def H_eigvals(p=.9, λ=.5, δ=0):
    "compute the eigenvalues of matrix H given parameters."
    # construct H matrix
    H = construct_H(p, λ, δ)

    # compute eigenvalues
    eigvals = np.linalg.eigvals(H)

    return eigvals
```

```
In [12]: H_eigvals()
```

```
Out[12]: 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.

```
In [13]: # small negative δ
H_eigvals(δ=-0.05)
```

```
Out[13]: array([0.8562829, 2.0437171])
```

```
In [14]: # large negative δ
H_eigvals(δ=-1.5)
```

```
Out[14]: array([0.10742784, 2.79257216])
```

A sufficiently small positive δ also causes no problem.

```
In [15]: # sufficiently small positive δ
H_eigvals(δ=0.05)
```

```
Out[15]: array([0.94750622, 1.95249378])
```

But a large enough positive δ makes both eigenvalues of H strictly greater than unity in modulus.

For example,

In [16]: `H_eigvals(δ=0.2)`

Out[16]: `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} .

In [17]: `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

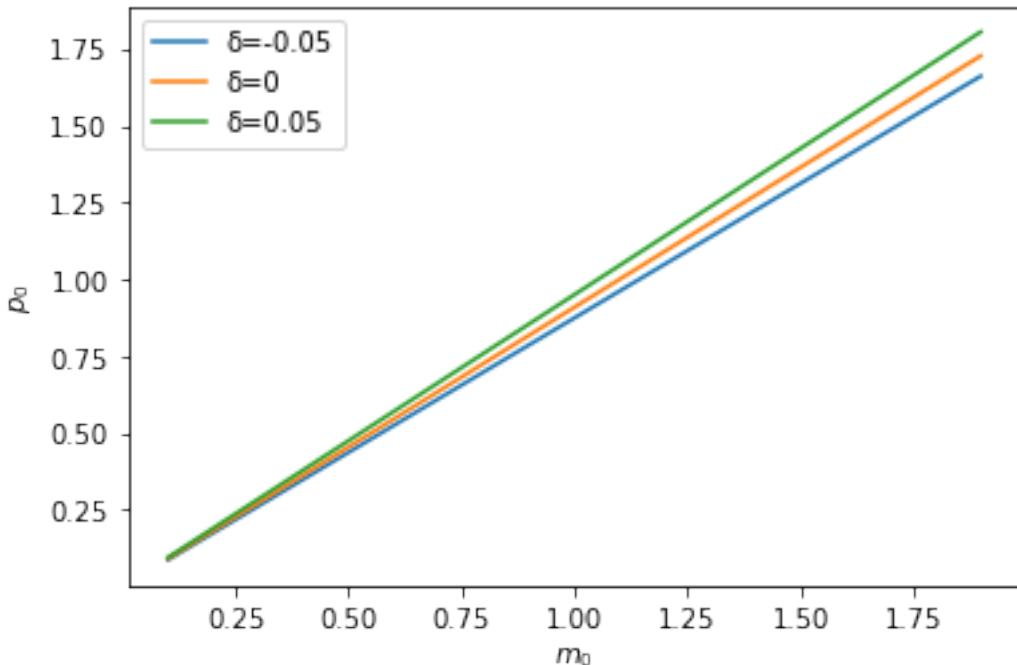
return p0
```

Let's plot how the solution p_0 changes as m_0 changes for different settings of δ .

In [18]: `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("$m_0$")
plt.ylabel("$p_0$")
plt.show()
```



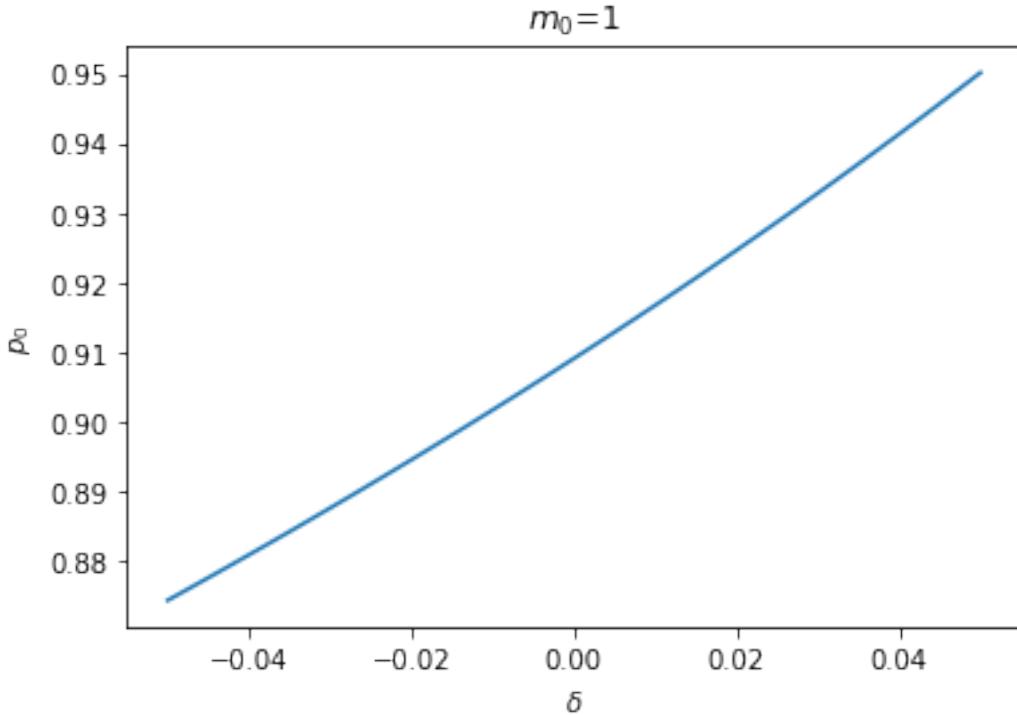
To look at things from a different angle, we can fix the initial value m_0 and see how p_0 changes as δ changes.

In [19]: `m0 = 1`

```

delta_range = np.linspace(-0.05, 0.05, 100)
plt.plot(delta_range, [magic_p0(m0, delta) for delta in delta_range])
plt.xlabel('$\delta$')
plt.ylabel('$p_0$')
plt.title(f'$m_0=${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.

In [20]: `magic_p0(1, δ=0.2)`

```
both eigenvalues exceed unity in modulus
```

49.9 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 (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

$$x_{t+1} = Ax_t$$

where $x_t = \begin{bmatrix} m_t \\ P_t \end{bmatrix}$.

Apply formula (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

$$p_t = [F_1 \quad F_2] \begin{bmatrix} m_t \\ F^*m_t \end{bmatrix} = F_1 m_t + F_2 F^* m_t$$

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 (11), (12), and (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 $G = [1 \quad 0]$ and we compute $[F_1 \quad F_2] \equiv F$ from equation (13) above.
3. We compute $F_1 + F_2 F^*$ and compare it with F^* and check for the anticipated equality.

In [21]: # set parameters

```
ρ = .9
λ = .5
δ = .05
```

In [22]: # 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
```

Out[22]: 0.9501243788791095

```
In [23]: # 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
```

Out[23]: array([0.92755597, 0.02375311])

Compare F^* with $F_1 + F_2 F^*$

```
In [24]: F_check[0] + F_check[1] * F_star, F_star
```

Out[24]: (0.9501243788791096, 0.9501243788791095)

49.10 Fun with SymPy code

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 .

```
In [25]: λ, δ, ρ = symbols('λ, δ, ρ')
```

```
In [26]: H1 = Matrix([[ρ, δ], [- (1 - λ) / λ, λ ** -1]])
```

```
In [27]: H1
```

```
Out[27]: ⎡ ρ   δ ⎤
          ⎣ λ-1   1 ⎦
```

```
In [28]: H1.eigenvals()
```

```
Out[28]: ⎧ λρ+1 - √4δλ²-4δλ+λ²ρ²-2λρ+1  : 1, λρ+1 + √4δλ²-4δλ+λ²ρ²-2λρ+1  : 1 ⎫
```

```
In [29]: H1.eigenvecs()
```

```
Out[29]: ⎢ ⎛ λρ+1 - √4δλ²-4δλ+λ²ρ²-2λρ+1 ⎞ ⎤ ⎢ ⎛ δ ⎞ ⎤ ⎢ ⎛ λρ+1 + √4δλ²-4δλ+λ²ρ²-2λρ+1 ⎞ ⎤ ⎣ ⎝ 1 ⎠ ⎦ ⎣ ⎝ 1 ⎠ ⎦ ⎣ ⎝ 1 ⎠ ⎦
```

Now let's compute H when δ is zero.

```
In [30]: H2 = Matrix([[ρ, 0], [- (1 - λ) / λ, λ ** -1]])
```

```
In [31]: H2
```

```
Out[31]: ⎡ ρ 0 ⎤
          ⎣ λ-1 1 ⎦
```

```
In [32]: H2.eigenvals()
```

```
Out[32]: {1/λ : 1, ρ : 1}
```

```
In [33]: H2.eigenvects()
```

```
Out[33]: ⎡ ⎛ 1 0 ⎞ ⎛ 0 1 ⎞ ⎤
          ⎢ ⎜ λ 1 ⎟ , ⎜ ρ -λ(-ρ+λ) ⎟ ⎥
          ⎣ ⎝ 1 0 ⎠ ⎝ λ-1 1 ⎠ ⎦
```

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)

```
In [34]: # 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])
```

```
In [35]: Q
```

```
Out[35]: ⎡ -λ(-ρ+λ) 0 ⎤
          ⎣ 1 λ-1 1 ⎦
```

Q^{-1}

```
In [36]: Q_inv = Q ** (-1)
Q_inv
```

Out[36]: $\begin{bmatrix} -\frac{\lambda-1}{\lambda(-\rho+\frac{1}{\lambda})} & 0 \\ \frac{\lambda-1}{\lambda(-\rho+\frac{1}{\lambda})} & 1 \end{bmatrix}$

$Q_{21}Q_{11}^{-1}$

In [37]: $Q[1, 0] / Q[0, 0]$

Out[37]: $-\frac{\lambda-1}{\lambda(-\rho+\frac{1}{\lambda})}$

$-(Q^{22})^{-1}Q^{21}$

In [38]: $-Q_inv[1, 0] / Q_inv[1, 1]$

Out[38]: $-\frac{\lambda-1}{\lambda(-\rho+\frac{1}{\lambda})}$

Chapter 50

Markov Perfect Equilibrium

50.1 Contents

- Overview 50.2
- Background 50.3
- Linear Markov Perfect Equilibria 50.4
- Application 50.5
- Exercises 50.6
- Solutions 50.7

In addition to what's in Anaconda, this lecture will need the following libraries:

```
In [1]: !pip install quantecon
```

50.2 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 [72].

Let's start with some standard imports:

```
In [2]: import numpy as np
import quantecon as qe
import matplotlib.pyplot as plt
%matplotlib inline
```

```
/home/ubuntu/anaconda3/lib/python3.7/site-packages/numba/np/ufunc/parallel.py:
355:
```

```
NumbaWarning: The TBB threading layer requires TBB version 2019.5 or later i.e.,
TBB_INTERFACE_VERSION >= 11005. Found TBB_INTERFACE_VERSION = 11004. The TBB
threading
layer is disabled.
warnings.warn(problem)
```

50.3 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., [34], [92], [29]).
- Rate of extraction from a shared natural resource, such as a fishery (e.g., [71], [107]).

Let's examine a model of the first type.

50.3.1 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

$$p = a_0 - a_1(q_1 + q_2) \quad (1)$$

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 (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:

$$\pi_i = pq_i - \gamma(\hat{q}_i - q_i)^2, \quad \gamma > 0, \quad (2)$$

Substituting the inverse demand curve (1) into (2) lets us express the one-period payoff as

$$\pi_i(q_i, q_{-i}, \hat{q}_i) = a_0 q_i - a_1 q_i^2 - a_1 q_i q_{-i} - \gamma(\hat{q}_i - q_i)^2, \quad (3)$$

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

$$v_i(q_i, q_{-i}) = \max_{\hat{q}_i} \{ \pi_i(q_i, q_{-i}, \hat{q}_i) + \beta v_i(\hat{q}_i, f_{-i}(q_{-i}, q_i)) \} \quad (4)$$

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 (4).
- The maximizer on the right side of (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.

50.3.2 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

$$v_i^{j+1}(q_i, q_{-i}) = \max_{\hat{q}_i} \{ \pi_i(q_i, q_{-i}, \hat{q}_i) + \beta v_i^j(\hat{q}_i, f_{-i}(q_{-i}, q_i)) \} \quad (5)$$

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.

50.4 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.

50.4.1 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

$$\sum_{t=t_0}^{t_1-1} \beta^{t-t_0} \{x_t' R_i x_t + u_{it}' Q_i u_{it} + u_{-it}' S_i u_{-it} + 2x_t' W_i u_{it} + 2u_{-it}' M_i u_{it}\} \quad (6)$$

while the state evolves according to

$$x_{t+1} = Ax_t + B_1 u_{1t} + B_2 u_{2t} \quad (7)$$

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 $n \times k_i$

50.4.2 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 $t = t_0, \dots, t_1 - 1$ 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 (6) and (7), then player 1’s problem becomes minimization of

$$\sum_{t=t_0}^{t_1-1} \beta^{t-t_0} \{x_t' \Pi_{1t} x_t + u_{1t}' Q_1 u_{1t} + 2u_{1t}' \Gamma_{1t} x_t\} \quad (8)$$

subject to

$$x_{t+1} = \Lambda_{1t}x_t + B_1u_{1t}, \quad (9)$$

where

- $\Lambda_{it} := A - B_{-i}F_{-it}$
- $\Pi_{it} := R_i + F'_{-it}S_iF_{-it}$
- $\Gamma_{it} := W'_i - M'_iF_{-it}$

This is an LQ dynamic programming problem that can be solved by working backwards.

Decision rules that solve this problem are

$$F_{1t} = (Q_1 + \beta B'_1 P_{1t+1} B_1)^{-1} (\beta B'_1 P_{1t+1} \Lambda_{1t} + \Gamma_{1t}) \quad (10)$$

where P_{1t} solves the matrix Riccati difference equation

$$P_{1t} = \Pi_{1t} - (\beta B'_1 P_{1t+1} \Lambda_{1t} + \Gamma_{1t})' (Q_1 + \beta B'_1 P_{1t+1} B_1)^{-1} (\beta B'_1 P_{1t+1} \Lambda_{1t} + \Gamma_{1t}) + \beta \Lambda'_{1t} P_{1t+1} \Lambda_{1t} \quad (11)$$

Similarly, decision rules that solve player 2's problem are

$$F_{2t} = (Q_2 + \beta B'_2 P_{2t+1} B_2)^{-1} (\beta B'_2 P_{2t+1} \Lambda_{2t} + \Gamma_{2t}) \quad (12)$$

where P_{2t} solves

$$P_{2t} = \Pi_{2t} - (\beta B'_2 P_{2t+1} \Lambda_{2t} + \Gamma_{2t})' (Q_2 + \beta B'_2 P_{2t+1} B_2)^{-1} (\beta B'_2 P_{2t+1} \Lambda_{2t} + \Gamma_{2t}) + \beta \Lambda'_{2t} P_{2t+1} \Lambda_{2t} \quad (13)$$

Here, in all cases $t = t_0, \dots, t_1 - 1$ and the terminal conditions are $P_{it_1} = 0$.

The solution procedure is to use equations (10), (11), (12), and (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 (10) contain F_{2t}
- some terms on the right-hand side of (12) contain F_{1t}

we need to solve these $k_1 + k_2$ equations simultaneously.

Key Insight

A key insight is that equations (10) and (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 (11) and (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 $t_1 \rightarrow +\infty$.

In practice, we usually fix t_1 and compute the equilibrium of an infinite horizon game by driving $t_0 \rightarrow -\infty$.

This is the approach we adopt in the next section.

50.4.3 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.

50.5 Application

Let's use these procedures to treat some applications, starting with the duopoly model.

50.5.1 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 (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:

$$x_{t+1} = (A - B_1 F_1 - B_2 F_2) x_t \tag{14}$$

50.5.2 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

```
In [3]: 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

# 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 F_2 as computed above, plug it into (8) and (9) to get firm 1's problem and solve it using LQ.

We hope that the resulting policy will agree with F_1 as computed above

```
In [4]: Λ1 = A - B2 @ F2
lq1 = qe.LQ(Q1, R1, Λ1, B1, beta=β)
P1_ih, F1_ih, d = lq1.stationary_values()
F1_ih
```

```
Out[4]: 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

```
In [5]: np.allclose(F1, F1_ih)
```

```
Out[5]: True
```

50.5.3 Dynamics

Let's now investigate the dynamics of price and output in this simple duopoly model under the MPE policies.

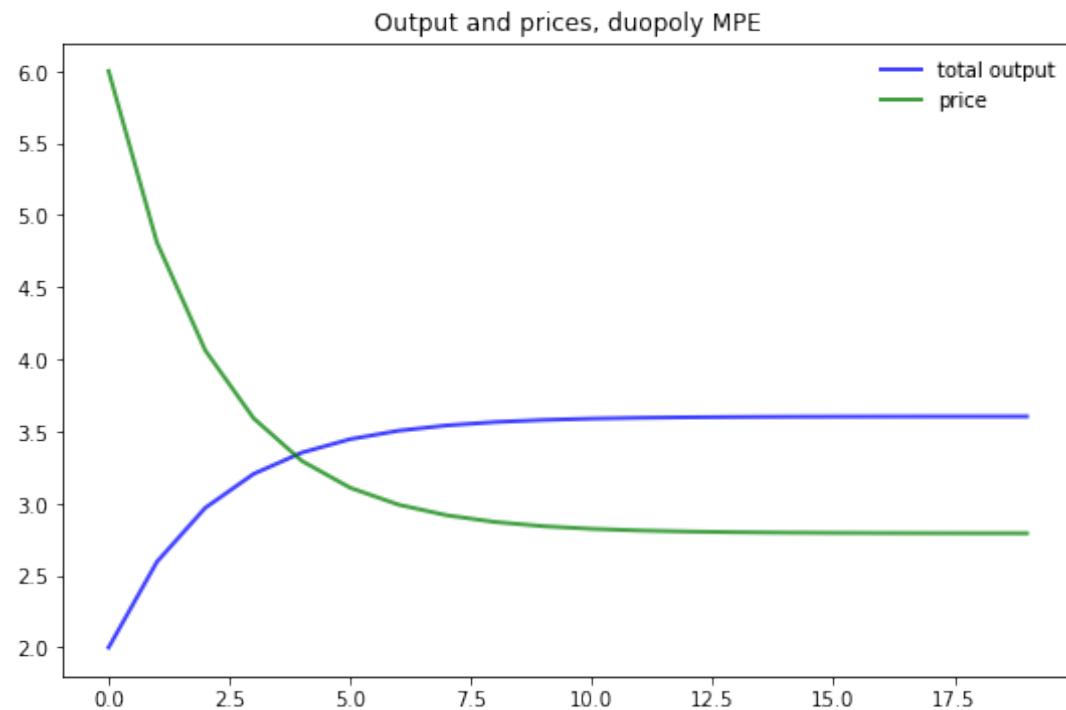
Given our optimal policies F_1 and F_2 , the state evolves according to (14).

The following program

- imports F_1 and F_2 from the previous program along with all parameters.
- computes the evolution of x_t using (14).
- extracts and plots industry output $q_t = q_{1t} + q_{2t}$ and price $p_t = a_0 - a_1 q_t$.

```
In [6]: 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

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()
```

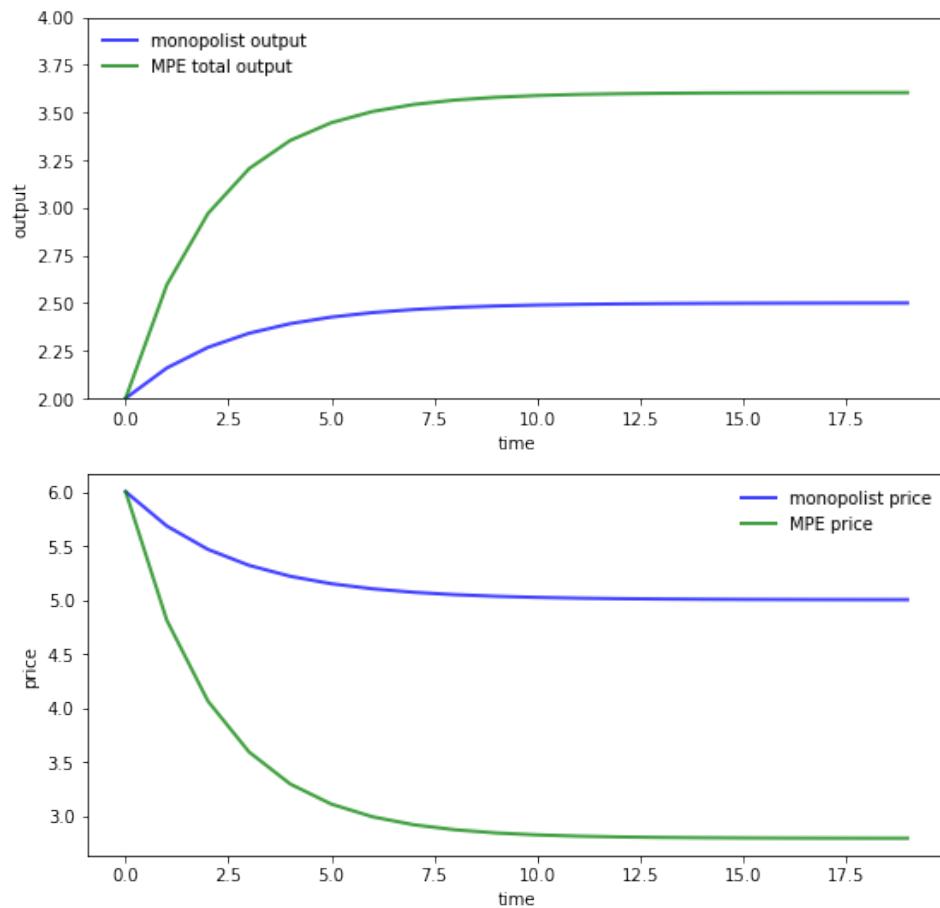


Note that the initial condition has been set to $q_{10} = q_{20} = 1.0$.

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.

50.6 Exercises

50.6.1 Exercise 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.

50.6.2 Exercise 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 [63].

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

- $C_{it} = c_{i1} + c_{i2}I_{it} + 0.5c_{i3}I_{it}^2$
- $E_{it} = e_{i1} + e_{i2}q_{it} + 0.5e_{i3}q_{it}^2$ 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

$$S_t = Dp_{it} + b$$

where

- $S_t = [S_{1t} \quad S_{2t}]'$
- D is a 2×2 negative definite matrix and
- b is a vector of constants

Firm i maximizes the undiscounted sum

$$\lim_{T \rightarrow \infty} \frac{1}{T} \sum_{t=0}^T (p_{it}S_{it} - E_{it} - C_{it})$$

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 $u_{it} = -F_i x_t$.

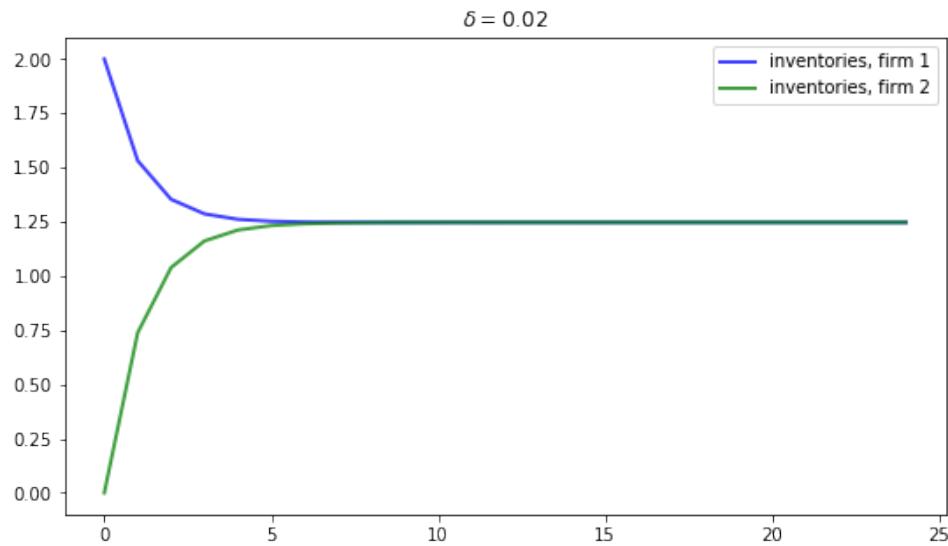
The Markov perfect equilibrium of Judd's model can be computed by filling in the matrices appropriately.

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

In [7]: $\delta = 0.02$

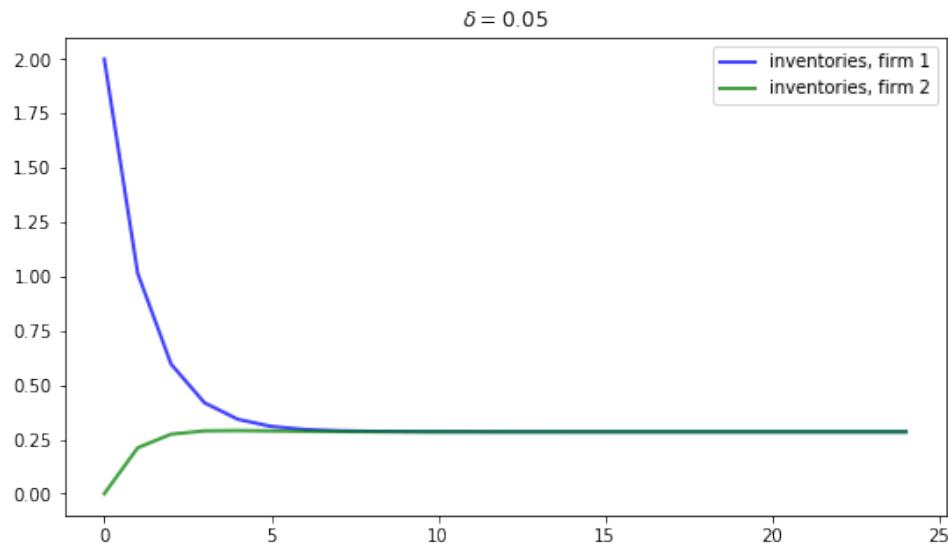
```
D = np.array([[-1, 0.5], [0.5, -1]])
b = np.array([25, 25])
c1 = c2 = np.array([1, -2, 1])
e1 = e2 = np.array([10, 10, 3])
```



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



50.7 Solutions

50.7.1 Exercise 1

First, let's compute the duopoly MPE under the stated parameters

```
In [8]: # == Parameters ==
a0 = 10.0
a1 = 2.0
```

```

 $\beta = 0.96$ 
 $\gamma = 12.0$ 

# == 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 = gamma
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=beta)

```

Now we evaluate the time path of industry output and prices given initial condition $q_{10} = q_{20} = 1$.

```
In [9]: 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 $x_{t+1} = Ax_t + Bu_t$.

We solve for the optimal policy $u_t = -Fx_t$ and track the resulting dynamics of $\{q_t\}$, starting at $q_0 = 2.0$.

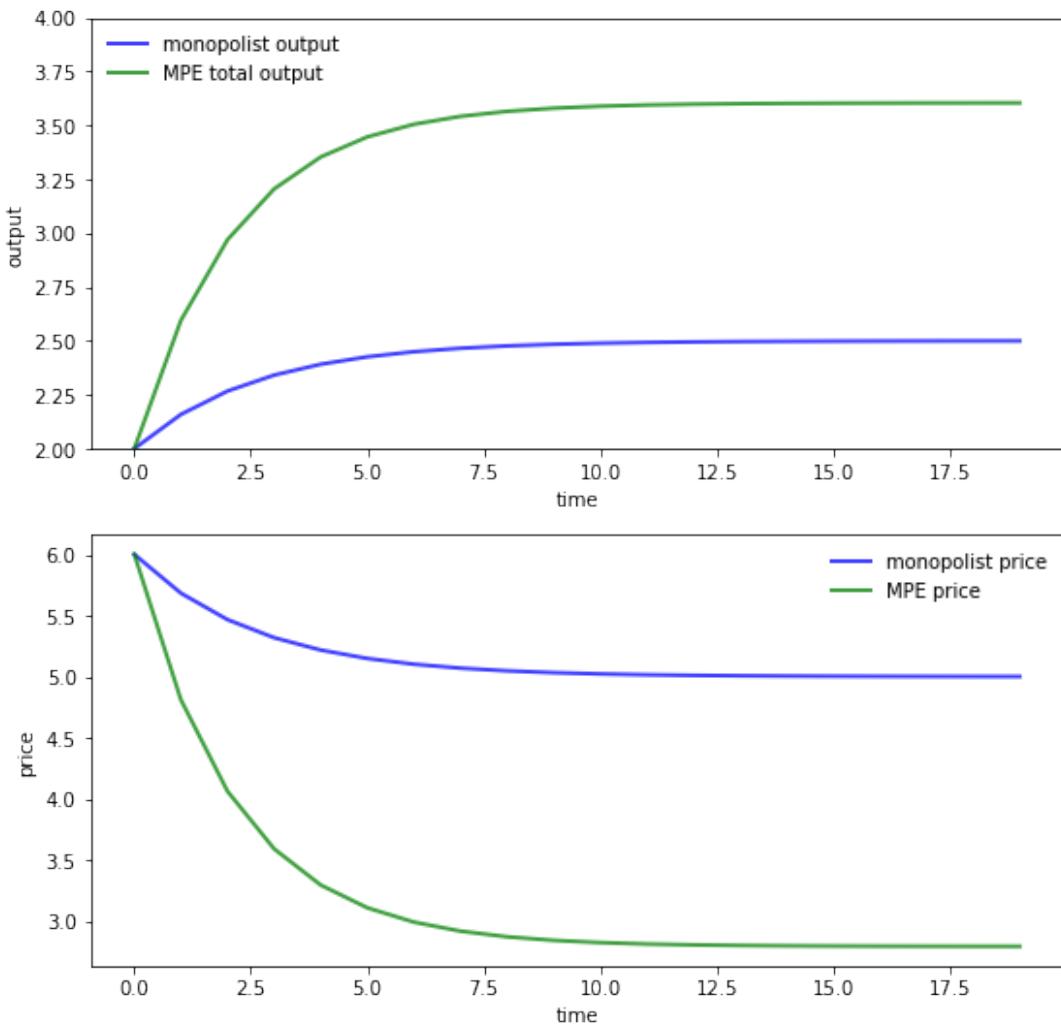
```
In [10]: R = a1
Q = γ
A = B = 1
lq_alt = qe.LQ(Q, R, A, B, beta=β)
P, F, d = lq_alt.stationary_values()
q_bar = a0 / (2.0 * a1)
qm = np.empty(n)
qm[0] = 2
x0 = qm[0] - q_bar
x = x0
for i in range(1, n):
    x = A * x - B * F * x
    qm[i] = float(x) + q_bar
pm = a0 - a1 * qm
```

Let's have a look at the different time paths

```
In [11]: 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()
```



50.7.2 Exercise 2

We treat the case $\delta = 0.02$

```
In [12]: 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:

In [13]: # == 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, 0],
               [-0.5 * e1[1], b[0] / 2.]])
W2 = np.array([[0, 0, 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`

In [14]: `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]]
```

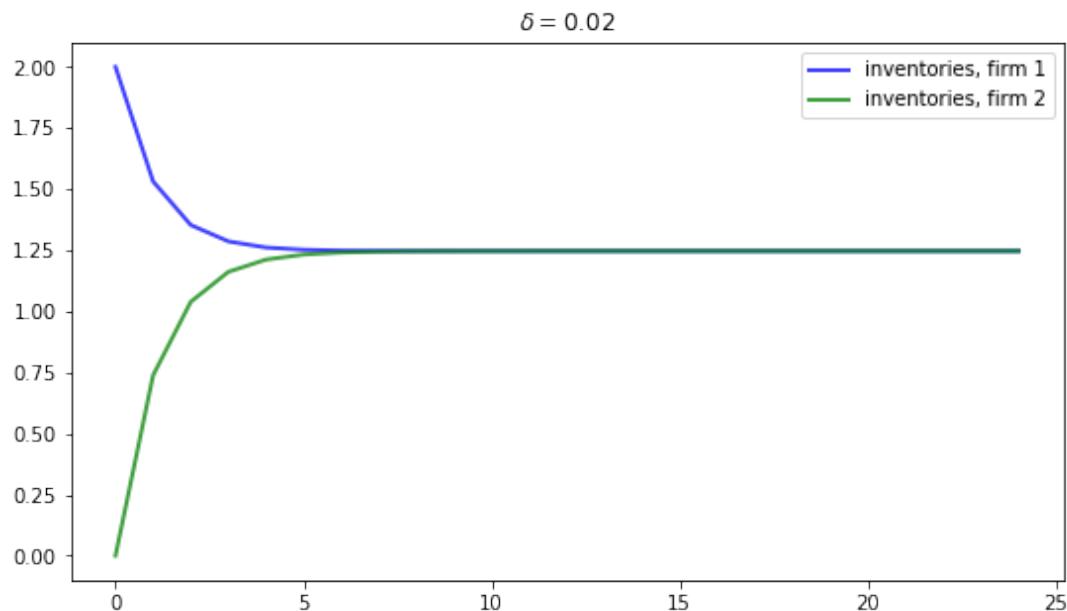
Firm 2's feedback rule:

```
[[ 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 $\delta = 0.02$

```
In [15]: 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'$\delta = \{delta\}$')
ax.legend()
plt.show()
```



Chapter 51

Uncertainty Traps

51.1 Contents

- Overview 51.2
- The Model 51.3
- Implementation 51.4
- Results 51.5
- Exercises 51.6
- Solutions 51.7

51.2 Overview

In this lecture, we study a simplified version of an uncertainty traps model of Fajgelbaum, Schaal and Taschereau-Dumouchel [37].

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:

```
In [1]: import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np
import itertools
```

51.3 The Model

The original model described in [37] has many interesting moving parts.

Here we examine a simplified version that nonetheless captures many of the key ideas.

51.3.1 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.

51.3.2 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

$$x_m = \theta + \epsilon_m \quad \text{where} \quad \epsilon_m \sim N(0, \gamma_x^{-1}) \quad (1)$$

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.

51.3.3 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

$$\mu' = \rho \frac{\gamma \mu + M \gamma_x X}{\gamma + M \gamma_x} \quad (2)$$

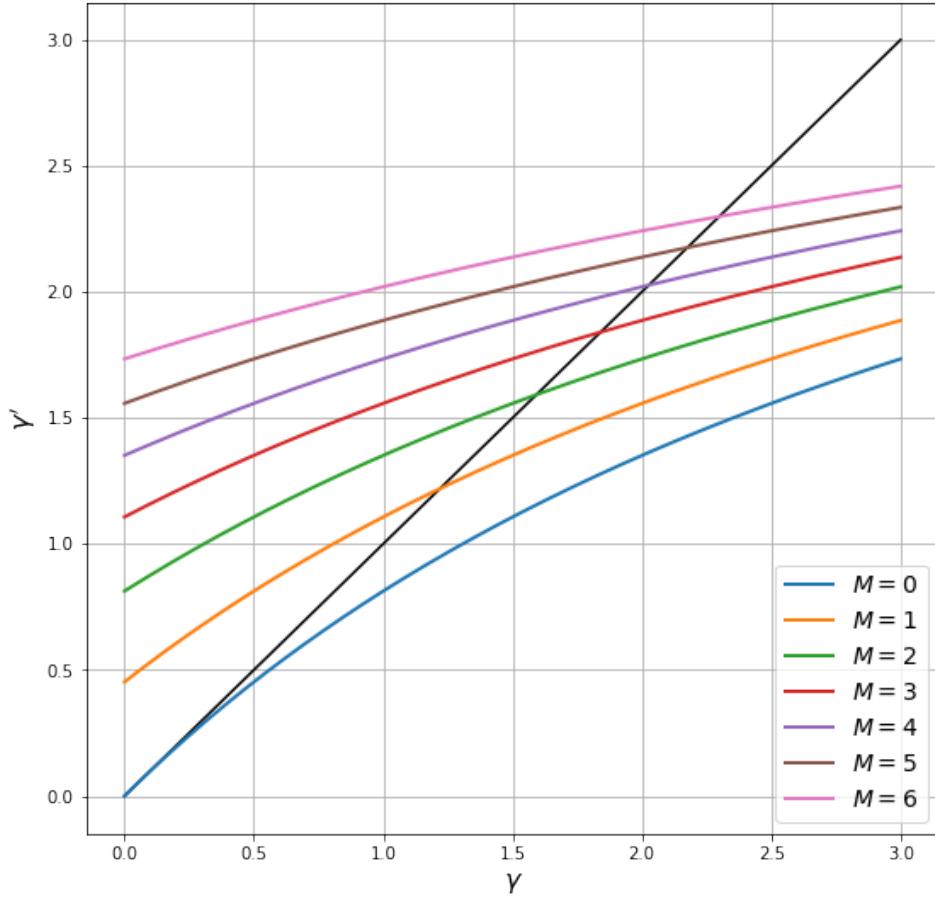
$$\gamma' = \left(\frac{\rho^2}{\gamma + M \gamma_x} + \sigma_\theta^2 \right)^{-1} \quad (3)$$

These are standard Kalman filtering results applied to the current setting.

Exercise 1 provides more details on how (2) and (3) are derived and then asks you to fill in remaining steps.

The next figure plots the law of motion for the precision in (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.

51.3.4 Participation

Omitting time subscripts once more, entrepreneurs enter the market in the current period if

$$\mathbb{E}[u(x_m - F_m)] > c \quad (4)$$

Here

- the mathematical expectation of x_m is based on (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 (4).

The utility function has the constant absolute risk aversion form

$$u(x) = \frac{1}{a} (1 - \exp(-ax)) \quad (5)$$

where a is a positive parameter.

Combining (4) and (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

$$\psi(\mu, \gamma, F_m) := \frac{1}{a} \left(1 - \exp \left(-a\mu + aF_m + \frac{a^2 \left(\frac{1}{\gamma} + \frac{1}{\gamma_x} \right)}{2} \right) \right) - c > 0 \quad (6)$$

51.4 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 $F_1, \dots, F_{\bar{M}}$ and tests condition (6) for each firm.

The `init` method encodes as default values the parameters we'll use in the simulations below

In [2]: `class UncertaintyTrapEcon:`

```
def __init__(self,
            a=1.5,                      # Risk aversion
            gamma_x=0.5,                 # Production shock precision
            rho=0.99,                    # Correlation coefficient for theta
            sigma_theta=0.5,              # Standard dev of theta shock
            num_firms=100,                # Number of firms
            sigma_F=1.5,                 # Standard dev of fixed costs
            c=-420,                      # External opportunity cost
            mu_init=0,                   # Initial value for mu
            gamma_init=4,                # Initial value for gamma
            theta_init=0):               # Initial value for theta

        # == Record values == #
        self.a, self.gamma_x, self.rho, self.sigma_theta = a, gamma_x, rho, sigma_theta
```

```

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.
    """
    self.θ = self.ρ * self.θ + self.σ_θ * w

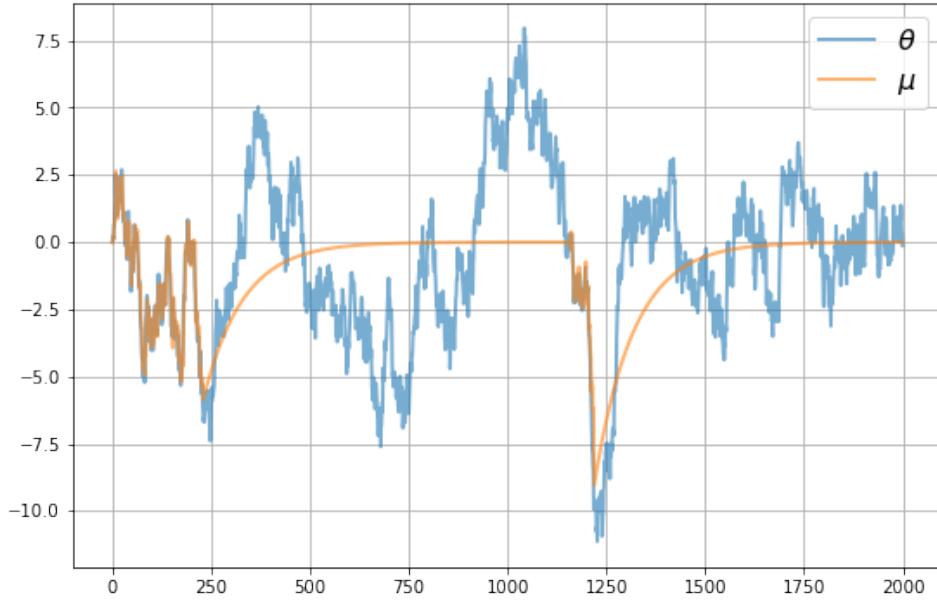
def gen_aggregates(self):
    """
    Generate aggregates based on current beliefs (μ, γ). This
    is a simulation step that depends on the draws for F.
    """
    F_vals = self.σ_F * np.random.randn(self.num_firms)
    M = np.sum(self.ψ(F_vals) > 0) # Counts number of active firms
    if M > 0:
        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.

51.5 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.

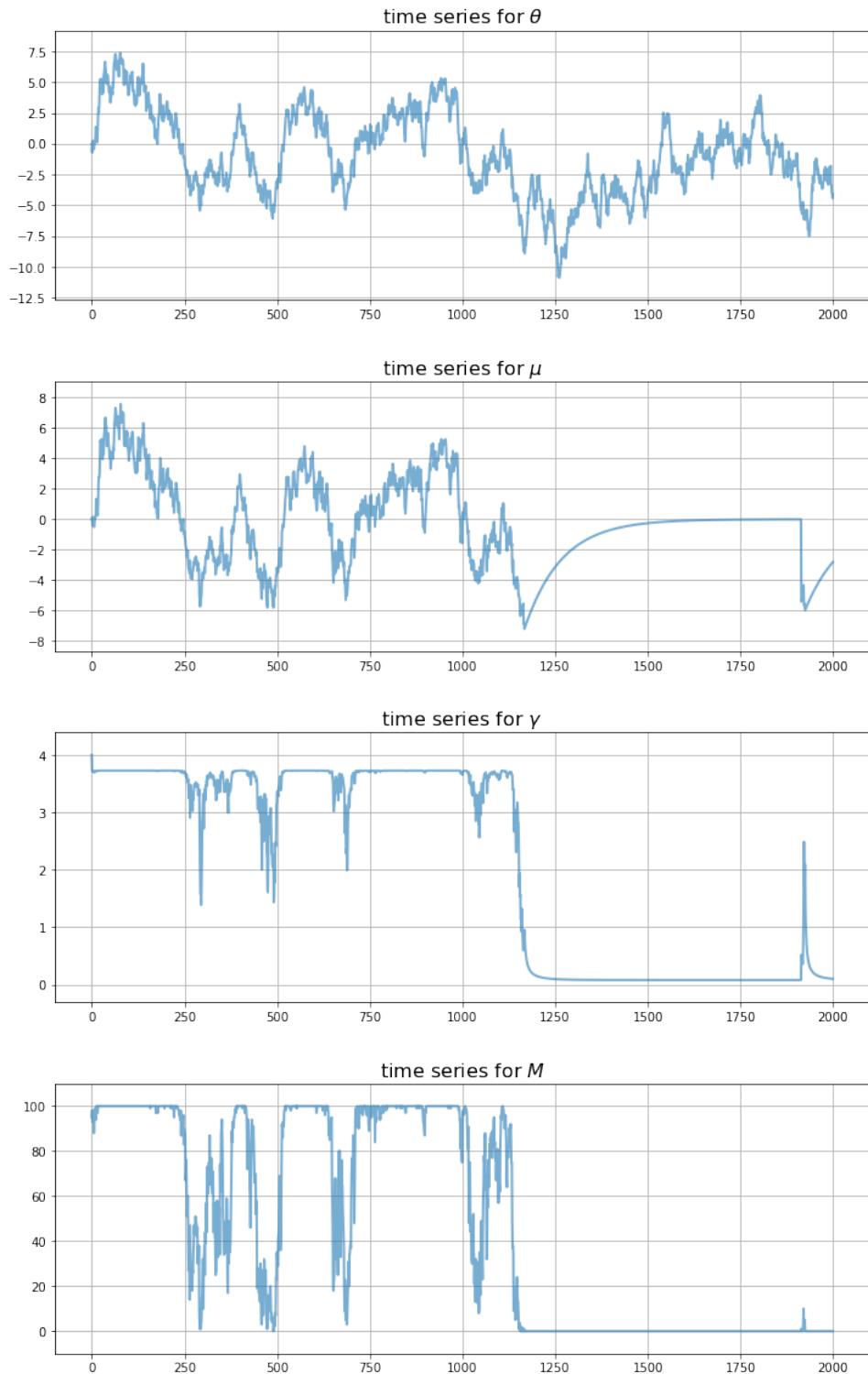
However, there are times when μ tracks θ poorly due to insufficient information.

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.

51.6 Exercises

51.6.1 Exercise 1

Fill in the details behind (2) and (3) based on the following standard result (see, e.g., p. 24 of [112]).

Fact Let $\mathbf{x} = (x_1, \dots, x_M)$ 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

$$\mu_0 = \frac{\mu\gamma + M\bar{x}\gamma_x}{\gamma + M\gamma_x} \quad \text{and} \quad \gamma_0 = \gamma + M\gamma_x$$

51.6.2 Exercise 2

Modulo randomness, replicate the simulation figures shown above.

- Use the parameter values listed as defaults in the `init` method of the `UncertaintyTrapEcon` class.

51.7 Solutions

51.7.1 Exercise 1

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.

51.7.2 Exercise 2

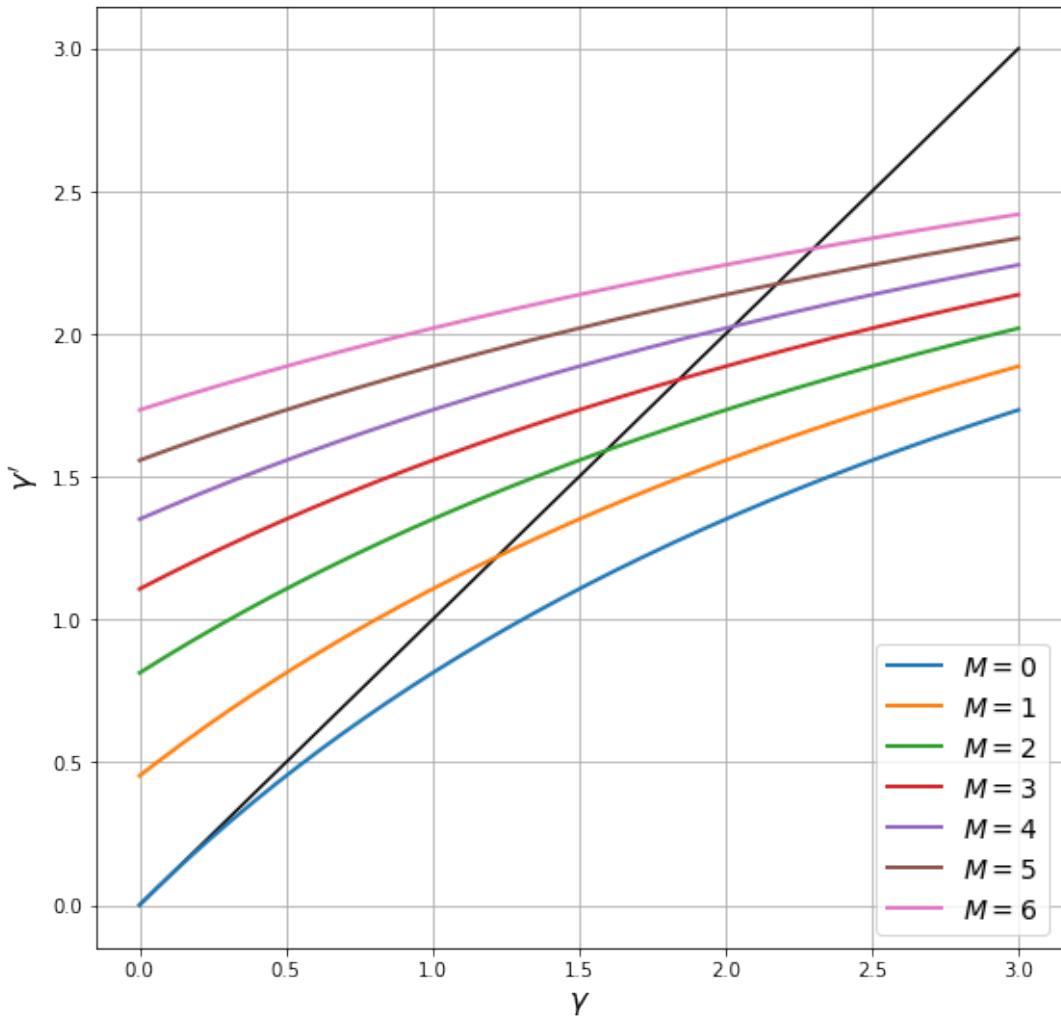
First, let's replicate the plot that illustrates the law of motion for precision, which is

$$\gamma_{t+1} = \left(\frac{\rho^2}{\gamma_t + M\gamma_x} + \sigma_\theta^2 \right)^{-1}$$

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

```
In [3]: 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

In [4]: `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] = θ

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])

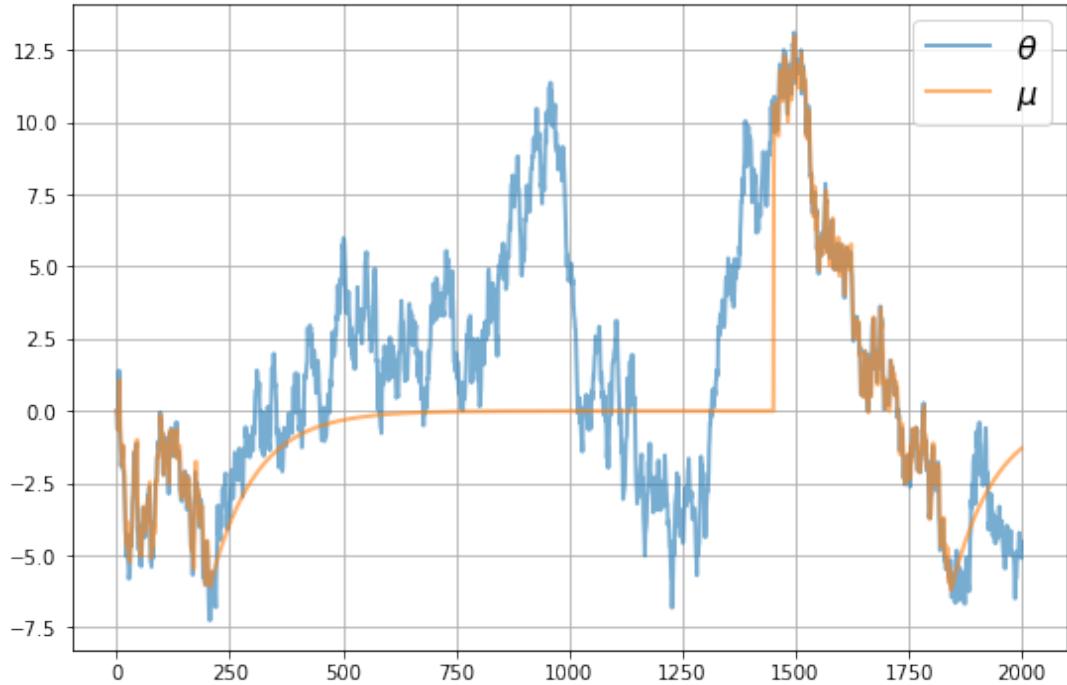
    μ_vec[t+1] = econ.μ
    γ_vec[t+1] = econ.γ
    θ_vec[t+1] = econ.θ

# 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

In [5]: `fig, ax = plt.subplots(figsize=(9, 6))
ax.plot(range(sim_length), θ_vec, alpha=0.6, lw=2, label=r"θ")
ax.plot(range(sim_length), μ_vec, alpha=0.6, lw=2, label=r"μ")
ax.legend(fontsize=16)
ax.grid()
plt.show()`



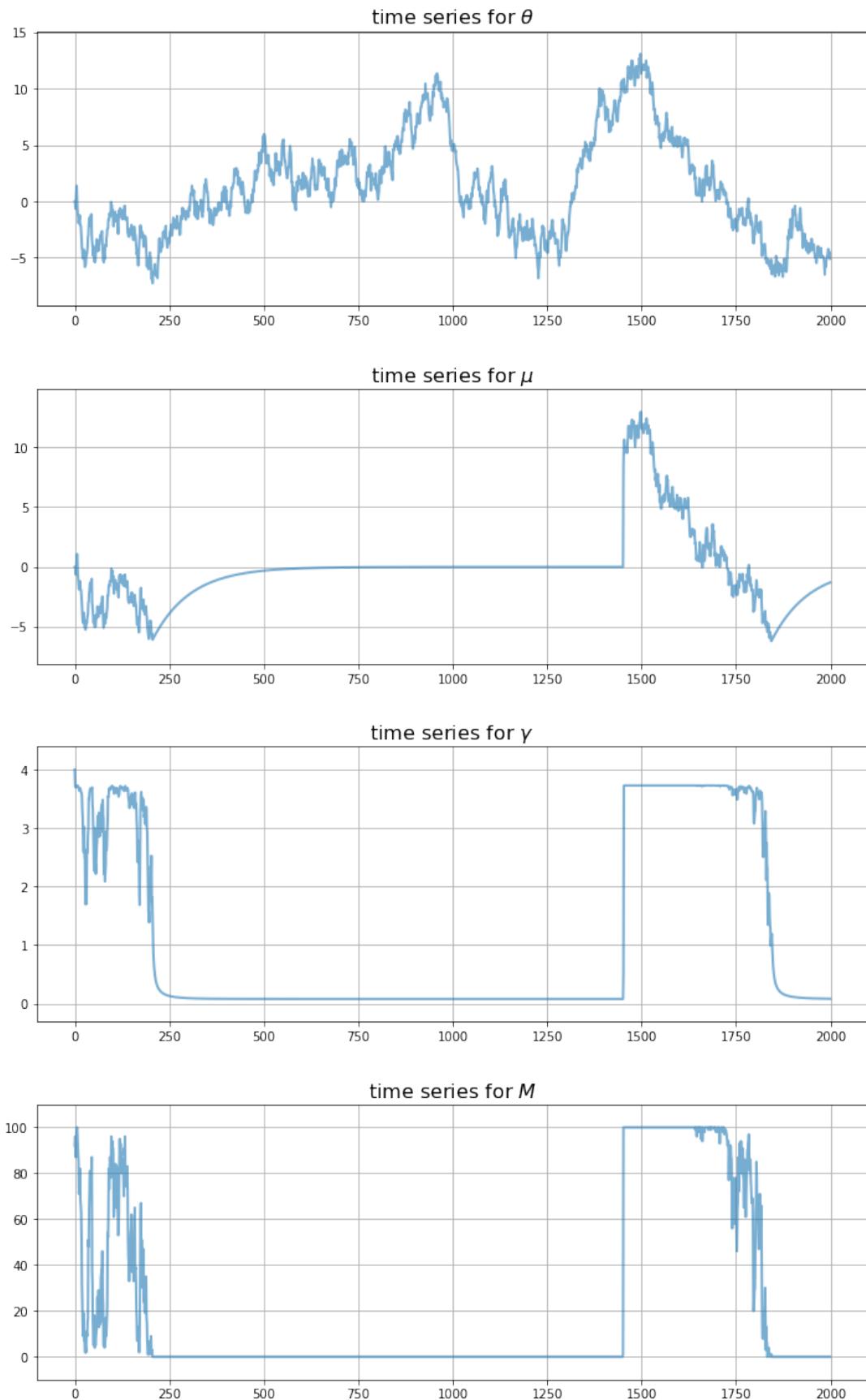
Now let's plot the whole thing together

```
In [6]: fig, axes = plt.subplots(4, 1, figsize=(12, 20))
# Add some spacing
fig.subplots_adjust(hspace=0.3)

series = (theta_vec, mu_vec, gamma_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()
```



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 52

The Aiyagari Model

52.1 Contents

- Overview 52.2
- The Economy 52.3
- Firms 52.4
- Code 52.5

In addition to what's in Anaconda, this lecture will need the following libraries:

```
In [1]: !pip install quantecon
```

52.2 Overview

In this lecture, we describe the structure of a class of models that build on work by Truman Bewley [13].

We begin by discussing an example of a Bewley model due to Rao Aiyagari.

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 [4]
- risk sharing and asset pricing [55]
- the shape of the wealth distribution [10]
- etc., etc., etc.

Let's start with some imports:

```
In [2]: import numpy as np
import quantecon as qe
import matplotlib.pyplot as plt
%matplotlib inline
from quantecon.markov import DiscreteDP
from numba import jit
```

```
/home/ubuntu/anaconda3/lib/python3.7/site-packages/numba/np/ufunc/parallel.py:
  ↵355:
NumbaWarning: The TBB threading layer requires TBB version 2019.5 or later i.e.,
TBB_INTERFACE_VERSION >= 11005. Found TBB_INTERFACE_VERSION = 11004. The TBB
  ↵threading
layer is disabled.
    warnings.warn(problem)
```

52.2.1 References

The primary reference for this lecture is [4].

A textbook treatment is available in chapter 18 of [72].

A continuous time version of the model by SeHyoun Ahn and Benjamin Moll can be found [here](#).

52.3 The Economy

52.3.1 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

$$a_{t+1} + c_t \leq wz_t + (1+r)a_t \quad c_t \geq 0, \quad \text{and} \quad a_t \geq -B$$

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.

52.4 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

$$Y_t = AK_t^\alpha N^{1-\alpha}$$

where

- A and α are parameters with $A > 0$ and $\alpha \in (0, 1)$
- K_t is aggregate capital
- N is total labor supply (which is constant in this simple version of the model)

The firm's problem is

$$\max_{K,N} \{AK_t^\alpha N^{1-\alpha} - (r + \delta)K - wN\}$$

The parameter δ is the depreciation rate.

From the first-order condition with respect to capital, the firm's inverse demand for capital is

$$r = A\alpha \left(\frac{N}{K}\right)^{1-\alpha} - \delta \quad (1)$$

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

$$w(r) = A(1 - \alpha)(A\alpha/(r + \delta))^{\alpha/(1-\alpha)} \quad (2)$$

52.4.1 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

In practice, once parameter values are set, we can check for an SREE by the following steps

1. pick a proposed quantity K for aggregate capital

2. determine corresponding prices, with interest rate r determined by (1) and a wage rate $w(r)$ as given in (2)
3. determine the common optimal savings policy of the households given these prices
4. compute aggregate capital as the mean of steady state capital given this savings policy

If this final quantity agrees with K then we have a SREE.

52.5 Code

Let's look at how we might compute such an equilibrium in practice.

To solve the household's dynamic programming problem we'll use the `DiscreteDP` class from [QuantEcon.py](#).

Our first task is the least exciting one: write code that maps parameters for a household problem into the `R` and `Q` matrices needed to generate an instance of `DiscreteDP`.

Below is a piece of boilerplate code that does just this.

In reading the code, the following information will be helpful

- `R` needs to be a matrix where `R[s, a]` is the reward at state `s` under action `a`.
- `Q` needs to be a three-dimensional array where `Q[s, a, s']` is the probability of transitioning to state `s'` when the current state is `s` and the current action is `a`.

(A more detailed discussion of `DiscreteDP` is available in the [Discrete State Dynamic Programming](#) lecture in the [Advanced Quantitative Economics with Python](#) lecture series.)

Here we take the state to be $s_t := (a_t, z_t)$, where a_t is assets and z_t is the shock.

The action is the choice of next period asset level a_{t+1} .

We use Numba to speed up the loops so we can update the matrices efficiently when the parameters change.

The class also includes a default set of parameters that we'll adopt unless otherwise specified.

In [3]: `class Household:`

`"""`

`This class takes the parameters that define a household asset`

`accumulation`

`problem and computes the corresponding reward and transition matrices R`
 `and Q required to generate an instance of DiscreteDP, and thereby solve`
 `for the optimal policy.`

`Comments on indexing: We need to enumerate the state space S as a`

`sequence`

`S = {0, ..., n}. To this end, (a_i, z_i) index pairs are mapped to s_i`
`indices according to the rule`

`s_i = a_i * z_size + z_i`

`To invert this map, use`

`a_i = s_i // z_size (integer division)`
`z_i = s_i % z_size`

```

"""
def __init__(self,
            r=0.01,                                # Interest rate
            w=1.0,                                   # Wages
            β=0.96,                                  # Discount factor
            a_min=1e-10,                             # Markov chain
            Π=[[0.9, 0.1], [0.1, 0.9]],             # Exogenous states
            z_vals=[0.1, 1.0],                      # Exogenous states
            a_max=18,
            a_size=200):

    # Store values, set up grids over a and z
    self.r, self.w, self.β = r, w, β
    self.a_min, self.a_max, self.a_size = a_min, a_max, a_size

    self.Π = np.asarray(Π)
    self.z_vals = np.asarray(z_vals)
    self.z_size = len(z_vals)

    self.a_vals = np.linspace(a_min, a_max, a_size)
    self.n = a_size * self.z_size

    # Build the array Q
    self.Q = np.zeros((self.n, a_size, self.n))
    self.build_Q()

    # Build the array R
    self.R = np.empty((self.n, a_size))
    self.build_R()

def set_prices(self, r, w):
    """
    Use this method to reset prices. Calling the method will trigger a
    re-build of R.
    """
    self.r, self.w = r, w
    self.build_R()

def build_Q(self):
    populate_Q(self.Q, self.a_size, self.z_size, self.Π)

def build_R(self):
    self.R.fill(-np.inf)
    populate_R(self.R,
               self.a_size,
               self.z_size,
               self.a_vals,
               self.z_vals,
               self.r,
               self.w)

# Do the hard work using JIT-ed functions

@jit(nopython=True)

```

```

def populate_R(R, a_size, z_size, a_vals, z_vals, r, w):
    n = a_size * z_size
    for s_i in range(n):
        a_i = s_i // z_size
        z_i = s_i % z_size
        a = a_vals[a_i]
        z = z_vals[z_i]
        for new_a_i in range(a_size):
            a_new = a_vals[new_a_i]
            c = w * z + (1 + r) * a - a_new
            if c > 0:
                R[s_i, new_a_i] = np.log(c) # Utility

@jit(nopython=True)
def populate_Q(Q, a_size, z_size, Π):
    n = a_size * z_size
    for s_i in range(n):
        z_i = s_i % z_size
        for a_i in range(a_size):
            for next_z_i in range(z_size):
                Q[s_i, a_i, a_i*z_size + next_z_i] = Π[z_i, next_z_i]

@jit(nopython=True)
def asset_marginal(s_probs, a_size, z_size):
    a_probs = np.zeros(a_size)
    for a_i in range(a_size):
        for z_i in range(z_size):
            a_probs[a_i] += s_probs[a_i*z_size + z_i]
    return a_probs

```

As a first example of what we can do, let's compute and plot an optimal accumulation policy at fixed prices.

```

In [4]: # Example prices
r = 0.03
w = 0.956

# Create an instance of Household
am = Household(a_max=20, r=r, w=w)

# Use the instance to build a discrete dynamic program
am_ddp = DiscreteDP(am.R, am.Q, am.β)

# Solve using policy function iteration
results = am_ddp.solve(method='policy_iteration')

# Simplify names
z_size, a_size = am.z_size, am.a_size
z_vals, a_vals = am.z_vals, am.a_vals
n = a_size * z_size

# Get all optimal actions across the set of a indices with z fixed in each
row
a_star = np.empty((z_size, a_size))
for s_i in range(n):
    a_i = s_i // z_size

```

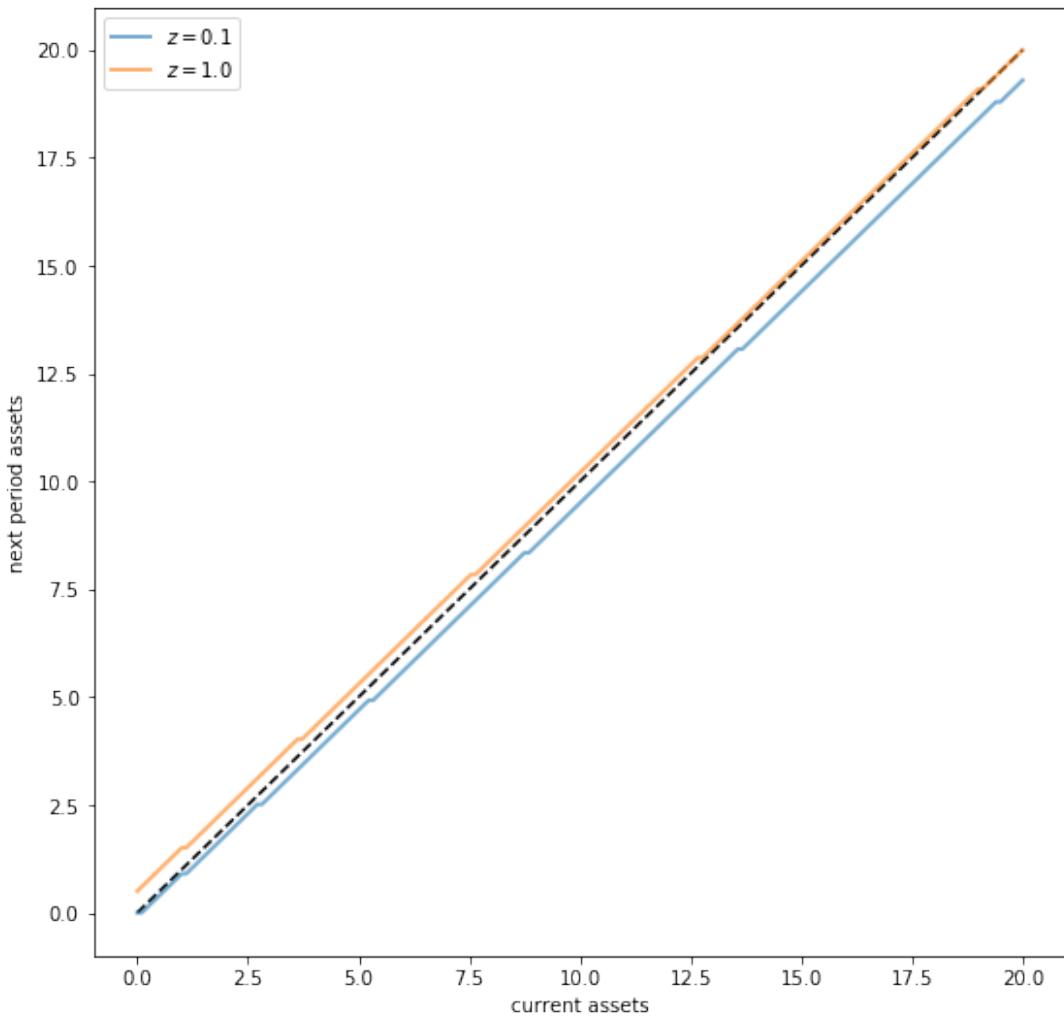
```

z_i = s_i % z_size
a_star[z_i, a_i] = a_vals[results.sigma[s_i]]

fig, ax = plt.subplots(figsize=(9, 9))
ax.plot(a_vals, a_vals, 'k--') # 45 degrees
for i in range(z_size):
    lb = f'$z = {z_vals[i]:.2}$'
    ax.plot(a_vals, a_star[i, :], 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.

Now we want to calculate the equilibrium.

Let's do this visually as a first pass.

The following code draws aggregate supply and demand curves.

The intersection gives equilibrium interest rates and capital.

```
In [5]: A = 1.0
N = 1.0
α = 0.33
β = 0.96
δ = 0.05

def r_to_w(r):
    """
    Equilibrium wages associated with a given interest rate r.
    """
    return A * (1 - α) * (A * α / (r + δ))**(α / (1 - α))

def rd(K):
    """
    Inverse demand curve for capital. The interest rate associated with a
    given demand for capital K.
    """
    return A * α * (N / K)**(1 - α) - δ

def prices_to_capital_stock(am, r):
    """
    Map prices to the induced level of capital stock.

    Parameters:
    -----
    am : Household
        An instance of an aiyagari_household.Household
    r : float
        The interest rate
    """
    w = r_to_w(r)
    am.set_prices(r, w)
    aiyagari_ddp = DiscreteDP(am.R, am.Q, β)
    # Compute the optimal policy
    results = aiyagari_ddp.solve(method='policy_iteration')
    # Compute the stationary distribution
    stationary_probs = results.mc.stationary_distributions[0]
    # Extract the marginal distribution for assets
    asset_probs = asset_marginal(stationary_probs, am.a_size, am.z_size)
    # Return K
    return np.sum(asset_probs * am.a_vals)

# Create an instance of Household
am = Household(a_max=20)

# Use the instance to build a discrete dynamic program
am_ddp = DiscreteDP(am.R, am.Q, am.β)

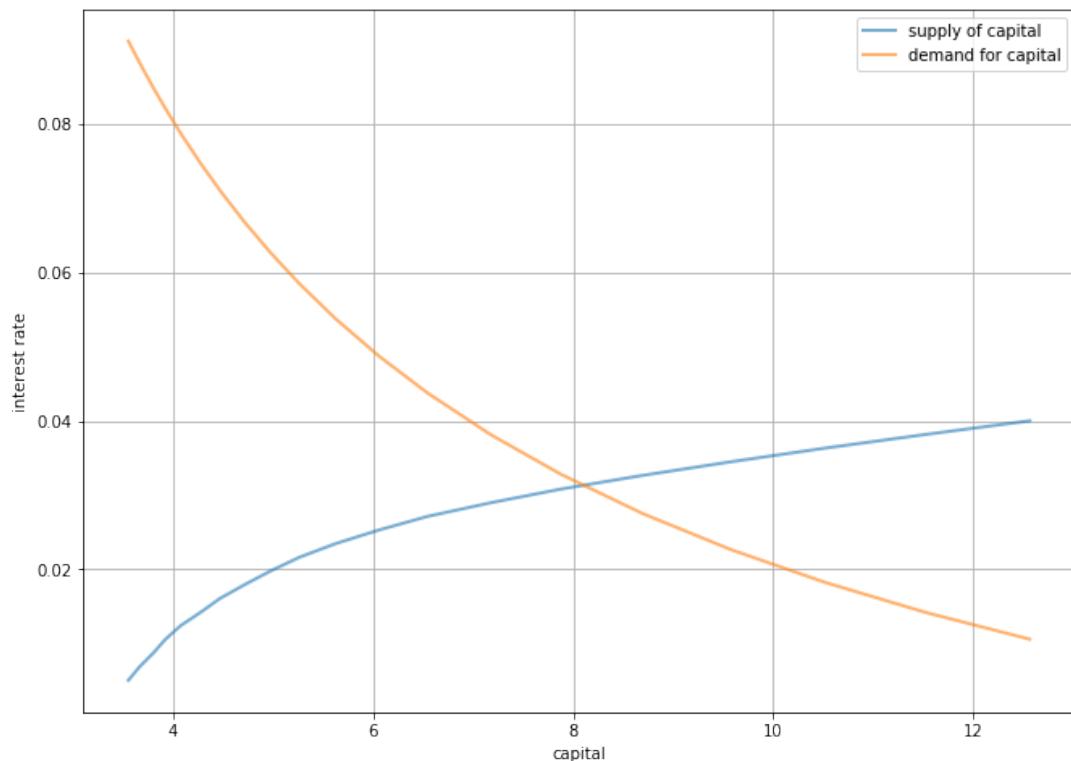
# Create a grid of r values at which to compute demand and supply of capital
num_points = 20
r_vals = np.linspace(0.005, 0.04, num_points)

# Compute supply of capital
k_vals = np.empty(num_points)
```

```
for i, r in enumerate(r_vals):
    k_vals[i] = prices_to_capital_stock(am, r)

# 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, rd(k_vals), lw=2, alpha=0.6, label='demand for capital')
ax.grid()
ax.set_xlabel('capital')
ax.set_ylabel('interest rate')
ax.legend(loc='upper right')

plt.show()
```



Part VIII

Asset Pricing and Finance

Chapter 53

Asset Pricing: Finite State Models

53.1 Contents

- Overview 53.2
- Pricing Models 53.3
- Prices in the Risk-Neutral Case 53.4
- Risk Aversion and Asset Prices 53.5
- Exercises 53.6
- Solutions 53.7

“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:

```
In [1]: !pip install quantecon
```

53.2 Overview

An asset is a claim on one or more future payoffs.

The spot price of an asset depends primarily on

- the anticipated dynamics for the stream of income accruing to the owners
- attitudes to 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 by repackaging income streams.

Key tools for the lecture are

- 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:

```
In [2]: import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import quantecon as qe
from numpy.linalg import eigvals, solve

/home/ubuntu/anaconda3/lib/python3.7/site-packages/numba/np/ufunc/parallel.py:
 355:
NumbaWarning: The TBB threading layer requires TBB version 2019.5 or later i.e.,
TBB_INTERFACE_VERSION >= 11005. Found TBB_INTERFACE_VERSION = 11004. The TBB
threading
layer is disabled.
warnings.warn(problem)
```

53.3 Pricing Models

In what follows 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}, \dots
- A time- t **ex-dividend** asset is a claim to the stream d_{t+1}, d_{t+2}, \dots

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).

53.3.1 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

$$p_t = \beta \mathbb{E}_t[d_{t+1} + p_{t+1}] \quad (1)$$

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 .

53.3.2 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 [54] and Lars Peter Hansen and Scott Richard [52] showed that in quite general settings the price of an ex-dividend asset obeys

$$p_t = \mathbb{E}_t[m_{t+1}(d_{t+1} + p_{t+1})] \quad (2)$$

for some **stochastic discount factor** m_{t+1} .

The fixed discount factor β in (1) has been replaced by the random variable m_{t+1} .

The way anticipated future payoffs are evaluated can now depend on various random outcomes.

One example of this idea is that assets that tend to have good payoffs in bad states of the world might be regarded as more valuable.

This is because they pay well when funds are more urgently wanted.

We give examples of how the stochastic discount factor has been modeled below.

53.3.3 Asset Pricing and Covariances

Recall that, from the definition of a conditional covariance $\text{cov}_t(x_{t+1}, y_{t+1})$, we have

$$\mathbb{E}_t(x_{t+1}y_{t+1}) = \text{cov}_t(x_{t+1}, y_{t+1}) + \mathbb{E}_t x_{t+1} \mathbb{E}_t y_{t+1} \quad (3)$$

If we apply this definition to the asset pricing equation (2) we obtain

$$p_t = \mathbb{E}_t m_{t+1} \mathbb{E}_t(d_{t+1} + p_{t+1}) + \text{cov}_t(m_{t+1}, d_{t+1} + p_{t+1}) \quad (4)$$

It is useful to regard equation (4) as a generalization of equation (1)

- In equation (1), the stochastic discount factor $m_{t+1} = \beta$, a constant.
- In equation (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 (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 (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](#).

53.3.4 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 (2) by d_t to get

$$v_t = \mathbb{E}_t \left[m_{t+1} \frac{d_{t+1}}{d_t} (1 + v_{t+1}) \right] \quad (5)$$

Below we'll discuss the implication of this equation.

53.4 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 the asset prices depends on the dividend process.

53.4.1 Example 1: Constant Dividends

The simplest case is risk-neutral pricing in the face of a constant, non-random dividend stream $d_t = d > 0$.

Removing the expectation from (1) and iterating forward gives

$$\begin{aligned} p_t &= \beta(d + p_{t+1}) \\ &= \beta(d + \beta(d + p_{t+2})) \\ &\quad \vdots \\ &= \beta(d + \beta d + \beta^2 d + \cdots + \beta^{k-2} d + \beta^{k-1} p_{t+k}) \end{aligned}$$

Unless prices explode in the future, this sequence converges to

$$\bar{p} := \frac{\beta d}{1 - \beta} \tag{6}$$

This price is the equilibrium price in the constant dividend case.

Indeed, simple algebra shows that setting $p_t = \bar{p}$ for all t satisfies the equilibrium condition $p_t = \beta(d + p_{t+1})$.

53.4.2 Example 2: Dividends with Deterministic Growth Paths

Consider a growing, non-random dividend process $d_{t+1} = gd_t$ where $0 < g\beta < 1$.

While prices are not usually constant when dividends grow over time, the price dividend-ratio might be.

If we guess this, substituting $v_t = v$ into (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*.

53.4.3 Example 3: Markov Growth, Risk-Neutral Pricing

Next, we consider a dividend process

$$d_{t+1} = g_{t+1} d_t \quad (7)$$

The stochastic growth factor $\{g_t\}$ is given by

$$g_t = g(X_t), \quad t = 1, 2, \dots$$

where

1. $\{X_t\}$ is a finite Markov chain with state space S and transition probabilities

$$P(x, y) := \mathbb{P}\{X_{t+1} = y \mid X_t = x\} \quad (x, y \in S)$$

1. g is a given function on S taking positive 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.

```
In [3]: mc = qe.tauchen(0.96, 0.25, n=25)
sim_length = 80

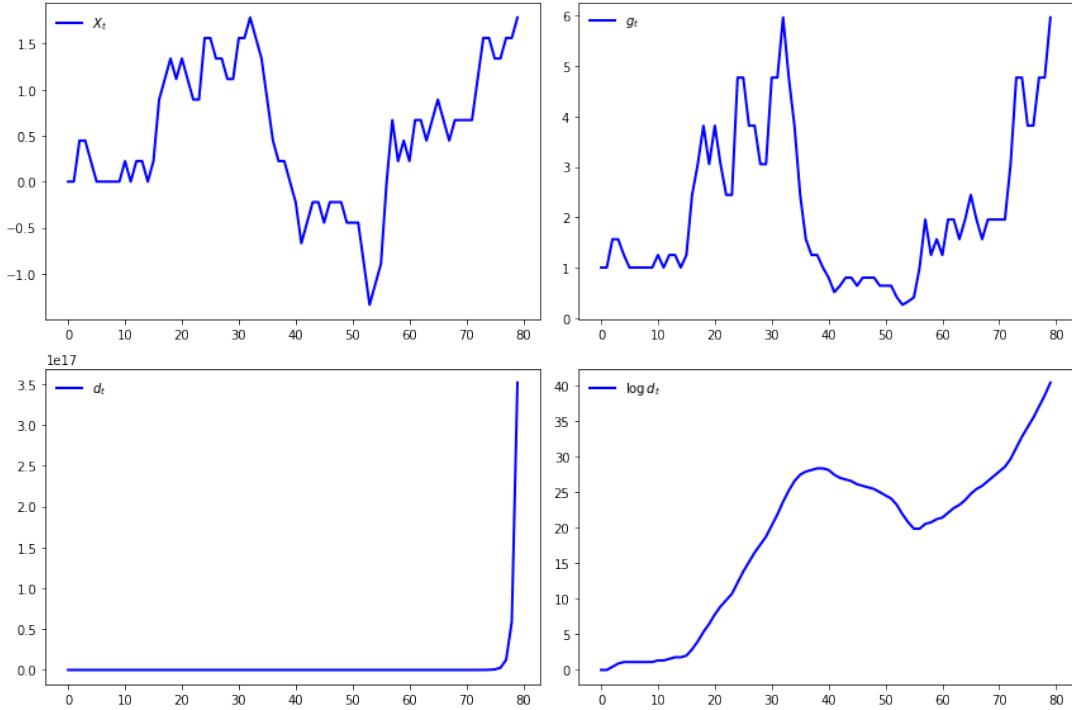
x_series = mc.simulate(sim_length, init=np.median(mc.state_values))
g_series = np.exp(x_series)
d_series = np.cumprod(g_series) # Assumes d_0 = 1

series = [x_series, g_series, d_series, np.log(d_series)]
labels = ['$X_t$', '$g_t$', '$d_t$', r'$\log \backslash, d_t$']
```

```

fig, axes = plt.subplots(2, 2, figsize=(12, 8))
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

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 constant given the state X_t .

In other words, we are looking for a fixed function v such that the price-dividend ratio satisfies $v_t = v(X_t)$.

We can substitute this guess into (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

$$v(x) = \beta \sum_{y \in S} K(x, y)(1 + v(y)) \quad \text{where} \quad K(x, y) := g(y)P(x, y) \quad (8)$$

Suppose that there are n possible states x_1, \dots, x_n .

We can then think of (8) as n stacked equations, one for each state, and write it in matrix form as

$$v = \beta K(\mathbb{1} + v) \quad (9)$$

Here

- v is understood to be the column vector $(v(x_1), \dots, v(x_n))'$.
- 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 (9) have a unique solution?

From the [Neumann series lemma](#) and Gelfand's formula, this will be the case if βK has spectral radius strictly less than one.

In other words, we require that the eigenvalues of K be strictly less than β^{-1} in modulus.

The solution is then

$$v = (I - \beta K)^{-1} \beta K \mathbb{1} \quad (10)$$

53.4.4 Code

Let's calculate and plot the price-dividend ratio at a set of 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

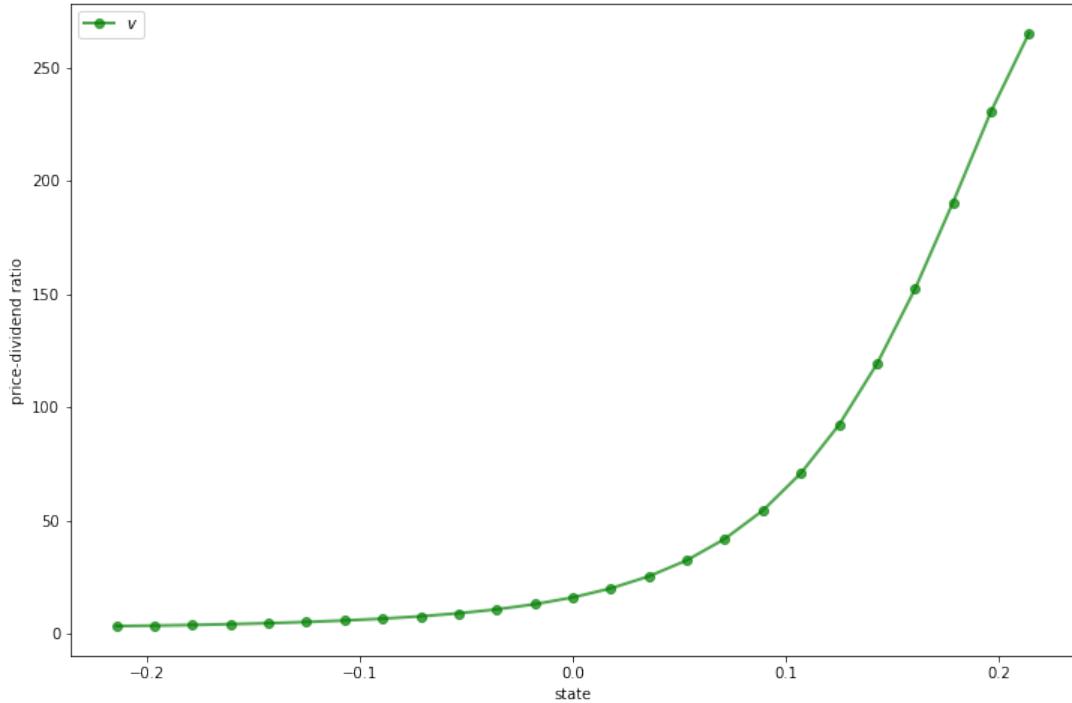
```
In [4]: n = 25 # Size of state space
β = 0.9
mc = qe.tauchen(0.96, 0.02, n=n)

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(figsize=(12, 8))
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.

53.5 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

53.5.1 Pricing a Lucas Tree

Let's start with a version of the celebrated asset pricing model of Robert E. Lucas, Jr. [73].

As in [73], suppose that the stochastic discount factor takes the form

$$m_{t+1} = \beta \frac{u'(c_{t+1})}{u'(c_t)} \quad (11)$$

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 (7).

The asset being priced is a claim on the endowment process.

Following [73], suppose further that in equilibrium, consumption is equal to the endowment, so that $d_t = c_t$ for all t .

For utility, we'll assume the **constant relative risk aversion** (CRRA) specification

$$u(c) = \frac{c^{1-\gamma}}{1-\gamma} \text{ with } \gamma > 0 \quad (12)$$

When $\gamma = 1$ we let $u(c) = \ln c$.

Inserting the CRRA specification into (11) and using $c_t = d_t$ gives

$$m_{t+1} = \beta \left(\frac{c_{t+1}}{c_t} \right)^{-\gamma} = \beta g_{t+1}^{-\gamma} \quad (13)$$

Substituting this into (5) gives the price-dividend ratio formula

$$v(X_t) = \beta \mathbb{E}_t [g(X_{t+1})^{1-\gamma} (1 + v(X_{t+1}))]$$

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 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

$$v = (I - \beta J)^{-1} \beta J \mathbb{1} \quad (14)$$

We will define a function tree_price to solve for v given parameters stored in the class AssetPriceModel

```
In [5]: class AssetPriceModel:
    """
```

A class that stores the primitives of the asset pricing model.

Parameters

```

 $\beta$  : scalar, float
    Discount factor
mc : MarkovChain
    Contains the transition matrix and set of state values for the state
    process
 $\gamma$  : scalar(float)
    Coefficient of risk aversion
g : callable
    The function mapping states to growth rates

"""
def __init__(self,  $\beta$ =0.96, mc=None,  $\gamma$ =2.0, g=np.exp):
    self. $\beta$ , self. $\gamma$  =  $\beta$ ,  $\gamma$ 
    self.g = g

    # A default process for the Markov chain
    if mc is None:
        self. $\rho$  = 0.9
        self. $\sigma$  = 0.02
        self.mc = qe.tauchen(self. $\rho$ , self. $\sigma$ , n=25)
    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. $\beta$ :
        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
    -----
    v : array_like(float)
        Lucas tree price-dividend ratio

    """
    # Simplify names, set up matrices
     $\beta$ ,  $\gamma$ , P, y = ap. $\beta$ , ap. $\gamma$ , ap.mc.P, ap.mc.state_values
    J = P * ap.g(y)**(1 -  $\gamma$ )

    # 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 - beta * J, beta * J @ Ones)

return v
```

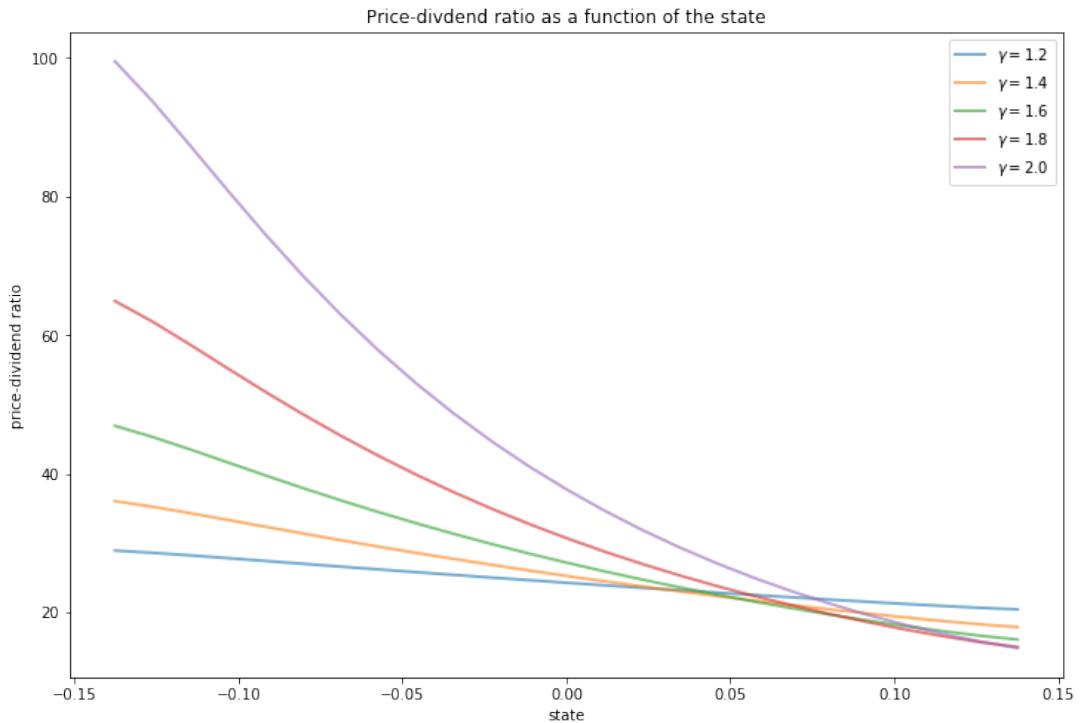
Here's a plot of v as a function of the state for several values of γ , with a positively correlated Markov process and $g(x) = \exp(x)$

```
In [6]: gams = [1.2, 1.4, 1.6, 1.8, 2.0]
ap = AssetPriceModel()
states = ap.mc.state_values

fig, ax = plt.subplots(figsize=(12, 8))

for gamma in gams:
    ap.gamma = gamma
    v = tree_price(ap)
    ax.plot(states, v, lw=2, alpha=0.6, label=rf"$\gamma = {gamma}$")

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 suggest higher future consumption growth.

In the stochastic discount factor (13), higher growth decreases the discount factor, lowering the weight placed on future returns.

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 (10).

This is as expected, since $\gamma = 0$ implies $u(c) = c$ (and hence agents are risk-neutral).

53.5.2 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 the 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 (2) with $d_t = \zeta$, or

$$p_t = \mathbb{E}_t [m_{t+1}(\zeta + p_{t+1})]$$

We maintain the stochastic discount factor (13), so this becomes

$$p_t = \mathbb{E}_t [\beta g_{t+1}^{-\gamma} (\zeta + p_{t+1})] \tag{15}$$

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

$$p = (I - \beta M)^{-1} \beta M \zeta \mathbb{1} \tag{16}$$

The above is implemented in the function `consol_price`.

```
In [7]: def consol_price(ap, zeta):
    """
```

Computes price of a consol bond with payoff ζ

Parameters

*ap: AssetPriceModel
An instance of AssetPriceModel containing primitives*

*ζ : scalar(float)
Coupon of the consol*

Returns

*p : array_like(float)
Console bond prices*

"""

*# Simplify names, set up matrices
 $\beta, \gamma, P, y = ap.\beta, ap.\gamma, ap.mc.P, ap.mc.state_values$
 $M = P * ap.g(y)^{**(-\gamma)}$*

*# Make sure that a unique solution exists
ap.test_stability(M)*

*# Compute price
I = np.identity(ap.n)
Ones = np.ones(ap.n)
p = solve(I - beta * M, beta * zeta * M @ Ones)*

return p

53.5.3 Pricing an Option to Purchase the Consol

Let's now price options of varying 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

$$w(x, p_S) = \max \left\{ \beta \sum_{y \in S} P(x, y) g(y)^{-\gamma} w(y, p_S), p(x) - p_S \right\} \quad (17)$$

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 (17) as the nonlinear vector equation

$$w = \max\{\beta M w, p - p_S \mathbf{1}\} \quad (18)$$

To solve (18), form the operator T mapping vector w into vector Tw via

$$Tw = \max\{\beta M w, p - p_S \mathbf{1}\}$$

Start at some initial w and iterate with T to convergence .

We can find the solution with the following function call_option

```
In [8]: 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 consol

    p_s : scalar(float)
        Strike price

    ε : scalar(float), optional(default=1e-8)
        Tolerance for infinite horizon problem

    Returns
    -----
    w : array_like(float)
        Option price
```

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 $P_S = 40$

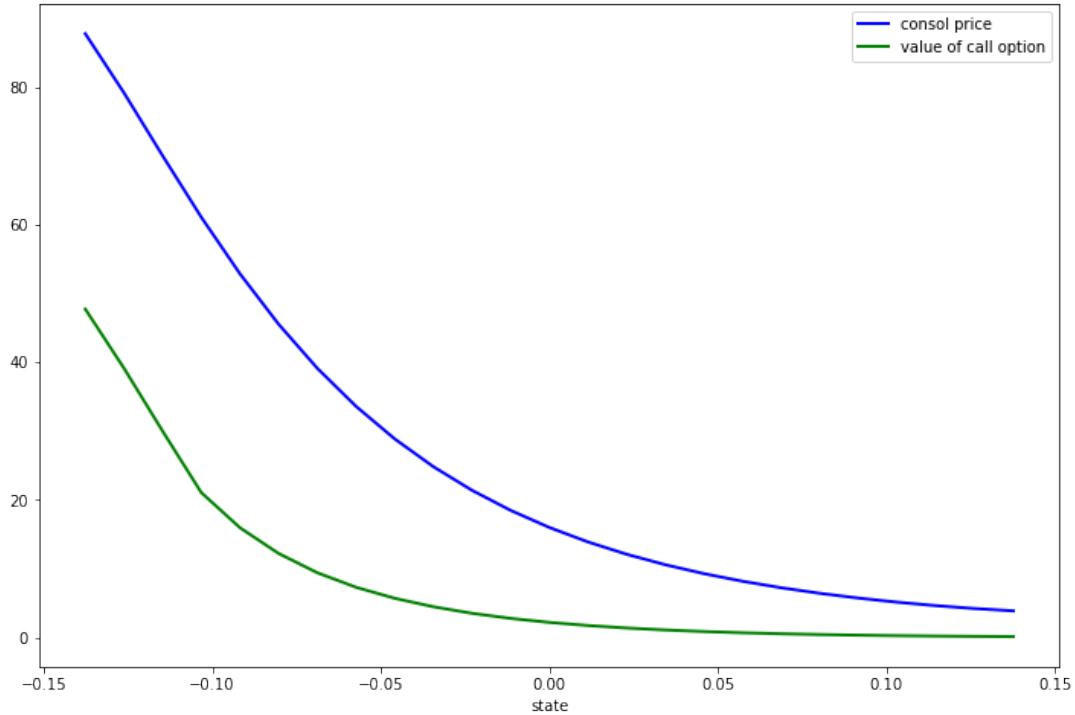
```

In [9]: 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(figsize=(12, 8))
ax.plot(x, p, 'b-', lw=2, label='consol price')
ax.plot(x, w, 'g-', lw=2, label='value of call option')
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.

53.5.4 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

$$m_1 = \beta M \mathbf{1}$$

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 $m_{j+1} = Mm_j$ for $j \geq 1$.

53.6 Exercises

53.6.1 Exercise 1

In the lecture, we considered **ex-dividend assets**.

A **cum-dividend** asset is a claim to the stream d_t, d_{t+1}, \dots

Following (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 $d_t = gd_t$ where $0 < g\beta < 1$, what is the equilibrium price of a cum-dividend asset?

53.6.2 Exercise 2

Consider the following primitives

```
In [10]: n = 5
P = 0.0125 * np.ones((n, n))
P += np.diag(0.95 - 0.0125 * np.ones(5))
# 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 $p_S = 150.0$

53.6.3 Exercise 3

Let's consider finite horizon call options, which are more common than the infinite horizon variety.

Finite horizon options obey functional equations closely related to (17).

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 $0, 1, \dots, k - 1$.

The option expires at time k .

Thus, for $k = 1, 2, \dots$, 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 the preceding as the sequence of nonlinear vector equations

$$w_k = \max\{\beta M w_{k-1}, p - p_S \mathbb{1}\} \quad k = 1, 2, \dots \quad \text{with } w_0 = 0$$

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 1.

Is one higher than the other? Can you give intuition?

53.7 Solutions

53.7.1 Exercise 1

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

$$p_t = \frac{1}{1 - \beta g} d_t$$

53.7.2 Exercise 2

First, let's enter the parameters:

```
In [11]: n = 5
P = 0.0125 * np.ones((n, n))
P += np.diag(0.95 - 0.0125 * np.ones(5))
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

```
In [12]: apm = AssetPriceModel(β=β, mc=mc, γ=γ, g=lambda x: x)
```

Now we just need to call the relevant functions on the data:

```
In [13]: tree_price(apm)
```

```
Out[13]: array([29.47401578, 21.93570661, 17.57142236, 14.72515002, 12.72221763])
```

```
In [14]: consol_price(apm, ζ)
```

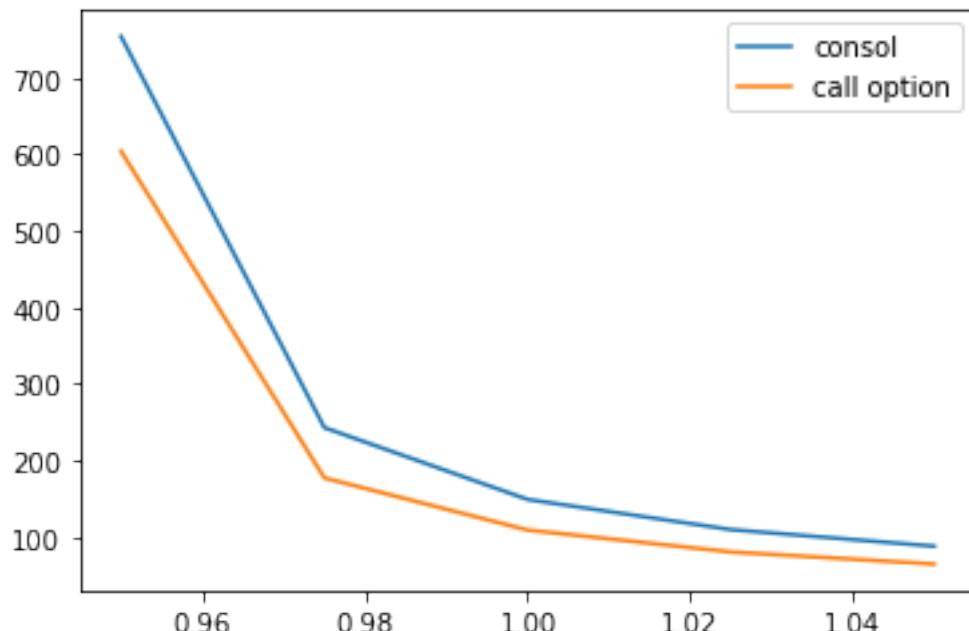
```
Out[14]: array([753.87100476, 242.55144082, 148.67554548, 109.25108965,
 87.56860139])
```

```
In [15]: call_option(apm, ζ, p_s)
```

```
Out[15]: array([603.87100476, 176.8393343 , 108.67734499, 80.05179254,
 64.30843748])
```

Let's show the last two functions as a plot

```
In [16]: 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()
```



53.7.3 Exercise 3

Here's a suitable function:

```
In [17]: 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)

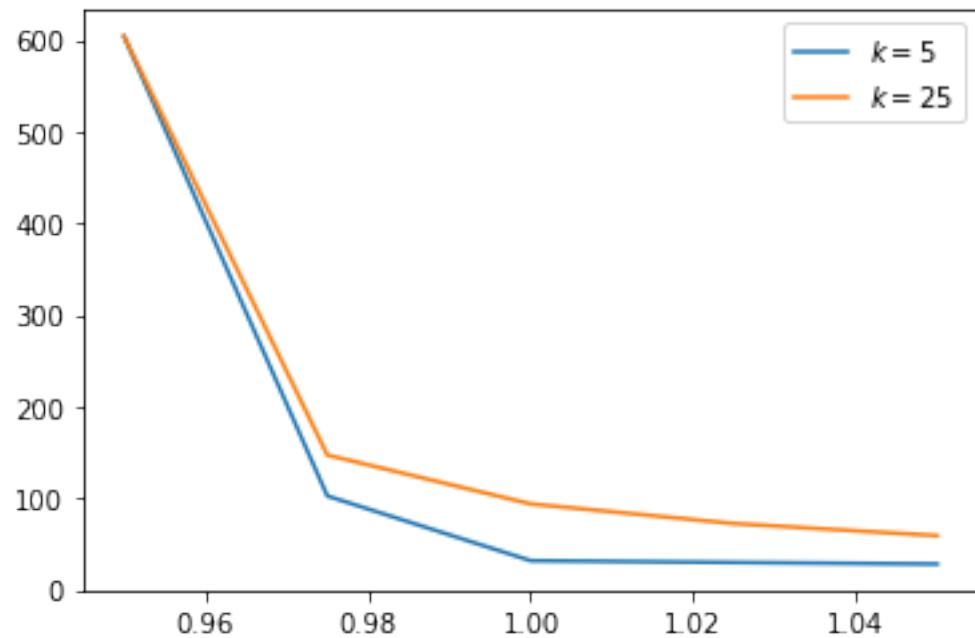
    # Make sure that a unique solution exists
    ap.test_stability(M)

    # 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 $k=25$

```
In [18]: fig, ax = plt.subplots()
for k in [5, 25]:
    w = finite_horizon_call_option(apm, ζ, p_s, k)
    ax.plot(s, w, label=r'$k = \{k\}$')
ax.legend()
plt.show()
```



Not surprisingly, the option has greater value with larger k .

This is because the owner has a longer time horizon over which he or she may exercise the option.

Chapter 54

Heterogeneous Beliefs and Bubbles

54.1 Contents

- Overview 54.2
- Structure of the Model 54.3
- Solving the Model 54.4
- Exercises 54.5
- Solutions 54.6

In addition to what's in Anaconda, this lecture uses following libraries:

```
In [1]: !pip install quantecon
```

54.2 Overview

This lecture describes a version of a model of Harrison and Kreps [53].

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:

```
In [2]: import numpy as np
import quantecon as qe
import scipy.linalg as la
```

/home/ubuntu/anaconda3/lib/python3.7/site-packages/numba/np/ufunc/parallel.py:
355:

```
NumbaWarning: The TBB threading layer requires TBB version 2019.5 or later i.e.,
TBB_INTERFACE_VERSION >= 11005. Found TBB_INTERFACE_VERSION = 11004. The TBB
threading
layer is disabled.
warnings.warn(problem)
```

54.2.1 References

Prior to reading the following, you might like to review our lectures on

- [Markov chains](#)
- [Asset pricing with finite state space](#)

54.2.2 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.

54.3 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 $S \in \{0, 1\}$.

Thus, the stock is traded **ex dividend**.

The dividend obeys

$$d_t = \begin{cases} 0 & \text{if } s_t = 0 \\ 1 & \text{if } s_t = 1 \end{cases}$$

The owner of a share at the beginning of time t is entitled to the dividend paid at time t .

The owner of the share at the beginning of time t is also entitled to sell the share to another investor during time t .

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:

```
In [3]: 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
```

```
Out[3]: array([[0.57142857, 0.42857143]])
```

```
In [4]: mcb.stationary_distributions
```

```
Out[4]: array([[0.42857143, 0.57142857]])
```

The stationary distribution of P_a is approximately $\pi_a = [.57 \quad .43]$.

The stationary distribution of P_b is approximately $\pi_b = [.43 \quad .57]$.

Thus, a type a investor is more pessimistic on average.

54.3.1 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 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 Section ??.
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.

54.3.2 Short Sales Prohibited

No short sales are allowed.

This matters because it limits how pessimists can express their opinion.

- They **can** express themselves by selling their shares.
- They **cannot** express themselves more loudly by artificially “manufacturing shares” – that is, they cannot borrow shares from more optimistic investors and then immediately sell them.

54.3.3 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.

However, the stationary distributions $\pi_a = [.57 \quad .43]$ and $\pi_b = [.43 \quad .57]$ tell us that a type B person is more optimistic about the dividend process in the long run than is a type A person.

54.3.4 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 .

54.4 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.

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.

54.4.1 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:

$$p_h(s) = \beta (P_h(s, 0)(0 + p_h(0)) + P_h(s, 1)(1 + p_h(1))), \quad s = 0, 1$$

These equations imply that the equilibrium price vector is

$$\begin{bmatrix} p_h(0) \\ p_h(1) \end{bmatrix} = \beta[I - \beta P_h]^{-1} P_h \begin{bmatrix} 0 \\ 1 \end{bmatrix} \quad (1)$$

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

```
In [5]: 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 investor h 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.

54.4.2 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:

$$\bar{p}(s) = \beta \max \{P_a(s, 0)\bar{p}(0) + P_a(s, 1)(1 + \bar{p}(1)), P_b(s, 0)\bar{p}(0) + P_b(s, 1)(1 + \bar{p}(1))\} \quad (2)$$

for $s = 0, 1$.

In the above equation, the *max* on the right side is evidently over 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 (2) 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 (2), namely

$$\bar{p}^{j+1}(s) = \beta \max \{ P_a(s, 0)\bar{p}^j(0) + P_a(s, 1)(1 + \bar{p}^j(1)), P_b(s, 0)\bar{p}^j(0) + P_b(s, 1)(1 + \bar{p}^j(1)) \} \quad (3)$$

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.

The reason that an investor is willing to pay more than what he believes is warranted by fundamental value of the prospective dividend stream is he expects to have the option to sell the asset later to another investor who will value the asset more highly than he will.

- 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 the 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

```
In [6]: 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 [θ, θ]
    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],
                            1)
        # 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],
                           1)
    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
```

54.4.3 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 (2), the equilibrium price satisfies

$$\check{p}(s) = \beta \min \{P_a(s, 1)\check{p}(0) + P_a(s, 1)(1 + \check{p}(1)), P_b(s, 1)\check{p}(0) + P_b(s, 1)(1 + \check{p}(1))\} \quad (4)$$

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 (4), 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 (2), 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

```
In [7]: 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 converge
    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],
                            1)
    # 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
```

54.4.4 Further Interpretation

[97] 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:

- Compared to the homogeneous beliefs setting leading to the pricing formula, high volume occurs when the Harrison-Kreps pricing formula 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 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.

54.5 Exercises

54.5.1 Exercise 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.

54.6 Solutions

54.6.1 Exercise 1

First, we will obtain equilibrium price vectors with homogeneous beliefs, including when all investors are optimistic or pessimistic.

```
In [8]: 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
qpeess = 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.

```

In [9]: 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.

Footnotes

- [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.

Part IX

Data and Empirics

Chapter 55

Pandas for Panel Data

55.1 Contents

- Overview 55.2
- Slicing and Reshaping Data 55.3
- Merging Dataframes and Filling NaNs 55.4
- Grouping and Summarizing Data 55.5
- Final Remarks 55.6
- Exercises 55.7
- Solutions 55.8

55.2 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.

55.3 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:

```
In [1]: url1 = 'https://raw.githubusercontent.com/QuantEcon/lecture-
python/master/source/_static/lecture_specific/pandas_panel/realwage.csv'
```

```
In [2]: 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

```
In [3]: realwage.head() # Show first 5 rows
```

```
Out[3]:   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

```
In [4]: realwage = realwage.pivot_table(values='value',
                                      index='Time',
                                      columns=['Country', 'Series', 'Pay period'])
realwage.head()
```

```
Out[4]: Country                               Australia      \
Series      In 2015 constant prices at 2015 USD PPPS
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`

```
In [5]: realwage.index = pd.to_datetime(realwage.index)
type(realwage.index)
```

```
Out[5]: 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

```
In [6]: type(realwage.columns)
```

```
Out[6]: pandas.core.indexes.multi.MultiIndex
```

```
In [7]: realwage.columns.names
```

```
Out[7]: FrozenList(['Country', 'Series', 'Pay period'])
```

Like before, we can select the country (the top level of our `MultiIndex`)

```
In [8]: realwage['United States'].head()
```

```
Out[8]: 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)

```
In [9]: realwage.stack().head()
```

```
Out[9]: 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	...	
	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		United Kingdom	...	\
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	...	

We can also pass in an argument to select the level we would like to stack

```
In [10]: realwage.stack(level='Country').head()
```

```
Out[10]: 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

```
In [11]: realwage['2015'].stack(level=(1, 2)).transpose().head()
```

```
Out[11]: 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)

```
In [12]: realwage_f = realwage.xs(['Hourly', 'In 2015 constant prices at 2015 USD\\
                                   exchange
                                   rates'],
                                 level=('Pay period', 'Series'), axis=1)
realwage_f.head()
```

```
Out[12]: 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]
```

55.4 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:

```
In [13]: url2 = 'https://raw.githubusercontent.com/QuantEcon/lecture-
python/master/source/_static/lecture_specific/pandas_panel/countries.csv'
```

```
In [14]: worlddata = pd.read_csv(url2, sep=';')
worlddata.head()
```

```
Out[14]:   Country (en) Country (de)          Country (local) ... Deathrate \
0    Afghanistan  Afghanistan  Afganistan/Afghanestan ...  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/Åland/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'

```
In [15]: worlddata = worlddata[['Country (en)', 'Continent']]
worlddata = worlddata.rename(columns={'Country (en)': 'Country'})
worlddata.head()
```

```
Out[15]:   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

```
In [16]: realwage_f.transpose().head()
```

```
Out[16]: 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
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]
```

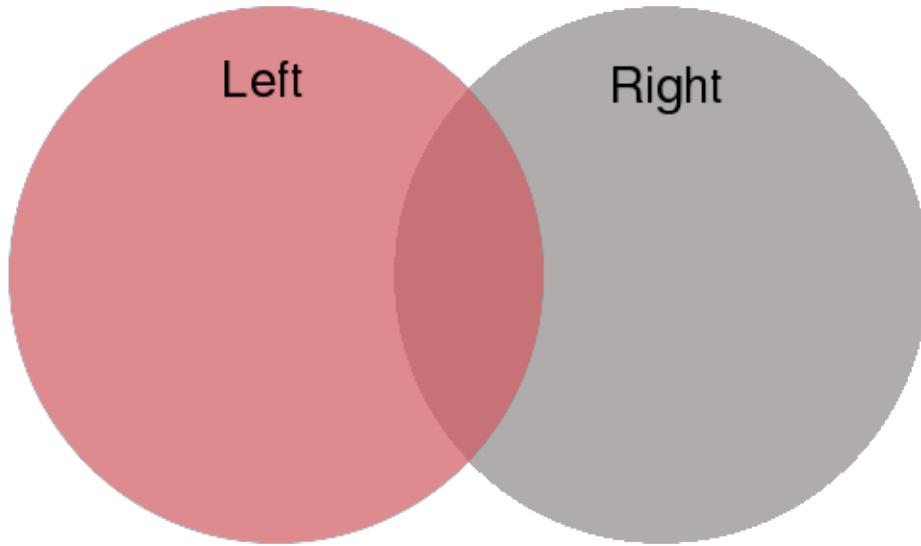
We can use either left, right, inner, or outer join to merge our datasets:

- 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'`

```
In [17]: merged = pd.merge(realwage_f.transpose(), worlddata,
                           how='left', left_index=True, right_on='Country')
merged.head()
```

```
Out[17]:      2006-01-01 00:00:00  2007-01-01 00:00:00  2008-01-01 00:00:00  ... \
17                  12.06                12.46              12.24  ...
23                  9.70                9.82              9.87  ...
32                  0.87                0.92              0.96  ...
100                 6.89                6.96              7.24  ...
38                  1.42                1.45              1.44  ...

      2016-01-01 00:00:00      Country      Continent
17                  12.98    Australia     Australia
23                  9.76    Belgium       Europe
32                  1.24    Brazil    South America
100                 8.48    Canada   North America
38                  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

```
In [18]: merged[merged['Continent'].isnull()]
```

```
Out[18]:    2006-01-01 00:00:00  2007-01-01 00:00:00  2008-01-01 00:00:00  ... \
247          3.42            3.74            3.87  ...
247          0.23            0.45            0.39  ...
247          1.50            1.64            1.71  ...

              2016-01-01 00:00:00           Country  Continent
247          5.28             Korea        NaN
247          0.55  Russian Federation      NaN
247          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`

```
In [19]: missing_continents = {'Korea': 'Asia',
                             'Russian Federation': 'Europe',
                             'Slovak Republic': 'Europe'}
```

`merged['Country'].map(missing_continents)`

```
Out[19]: 17      NaN
23      NaN
32      NaN
100     NaN
38      NaN
108     NaN
41      NaN
225     NaN
53      NaN
58      NaN
45      NaN
68      NaN
233     NaN
86      NaN
88      NaN
91      NaN
247     Asia
117     NaN
122     NaN
123     NaN
138     NaN
153     NaN
151     NaN
174     NaN
175     NaN
247     Europe
247     Europe
198     NaN
200     NaN
227     NaN
```

```
241      NaN
240      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

```
In [20]: merged['Continent'] =
    merged['Continent'].fillna(merged['Country'].map(missing_continents))

    # Check for whether continents were correctly mapped

    merged[merged['Country'] == 'Korea']

Out[20]:      2006-01-01 00:00:00  2007-01-01 00:00:00  2008-01-01 00:00:00 ... \
247              3.42                  3.74                  3.87      ...
247  2016-01-01 00:00:00  Country  Continent
                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

```
In [21]: replace = ['Central America', 'North America', 'South America']

for country in replace:
    merged['Continent'].replace(to_replace=country,
                                value='America',
                                inplace=True)
```

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

```
In [22]: merged = merged.set_index(['Continent', 'Country']).sort_index()
merged.head()
```

```
Out[22]:      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 date-time format

In [23]: `merged.columns`

```
Out[23]: 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()`

```
In [24]: merged.columns = pd.to_datetime(merged.columns)
merged.columns = merged.columns.rename('Time')
merged.columns
```

```
Out[24]: 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`

```
In [25]: merged = merged.transpose()
merged.head()
```

```
Out[25]: 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]

55.5 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)

In [26]: `merged.mean().head(10)`

```
Out[26]: 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
```

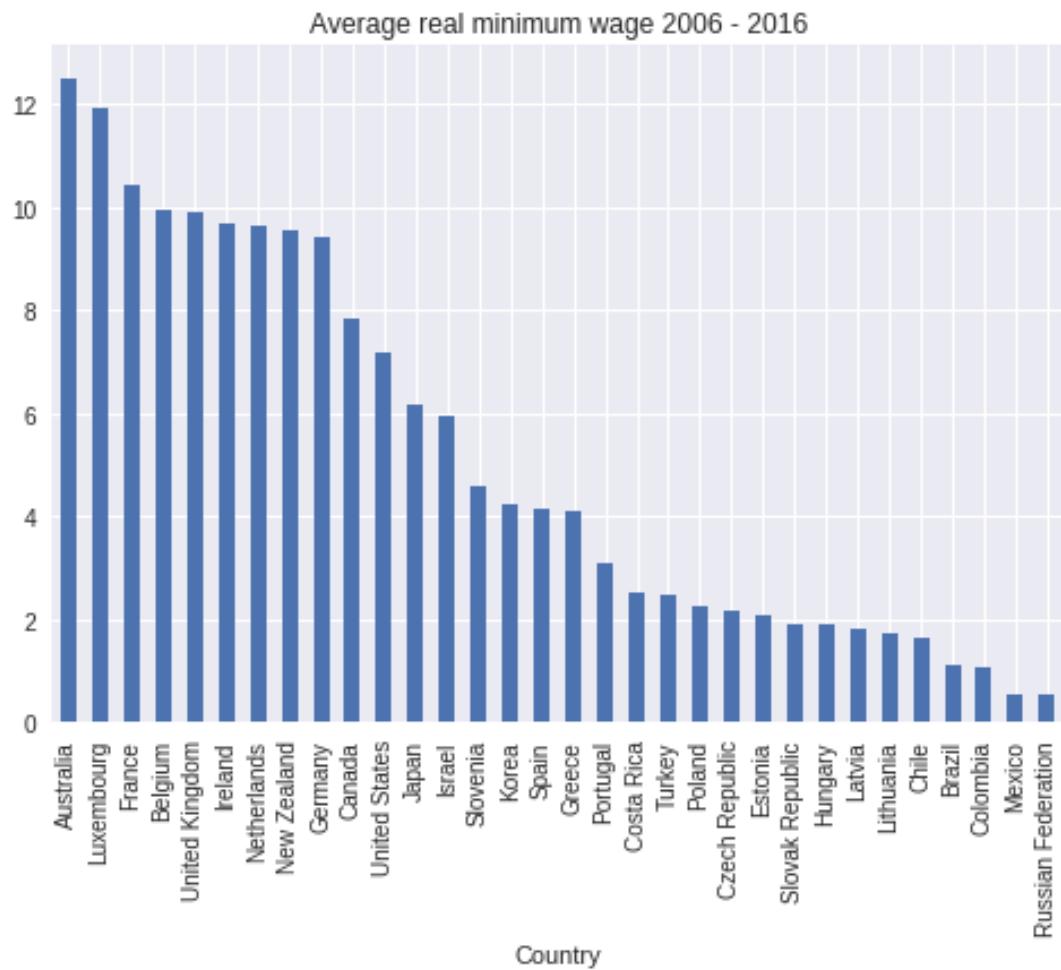
Using this series, we can plot the average real minimum wage over the past decade for each country in our data set

```
In [27]: import matplotlib.pyplot as plt
%matplotlib inline
import matplotlib
matplotlib.style.use('seaborn')

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)

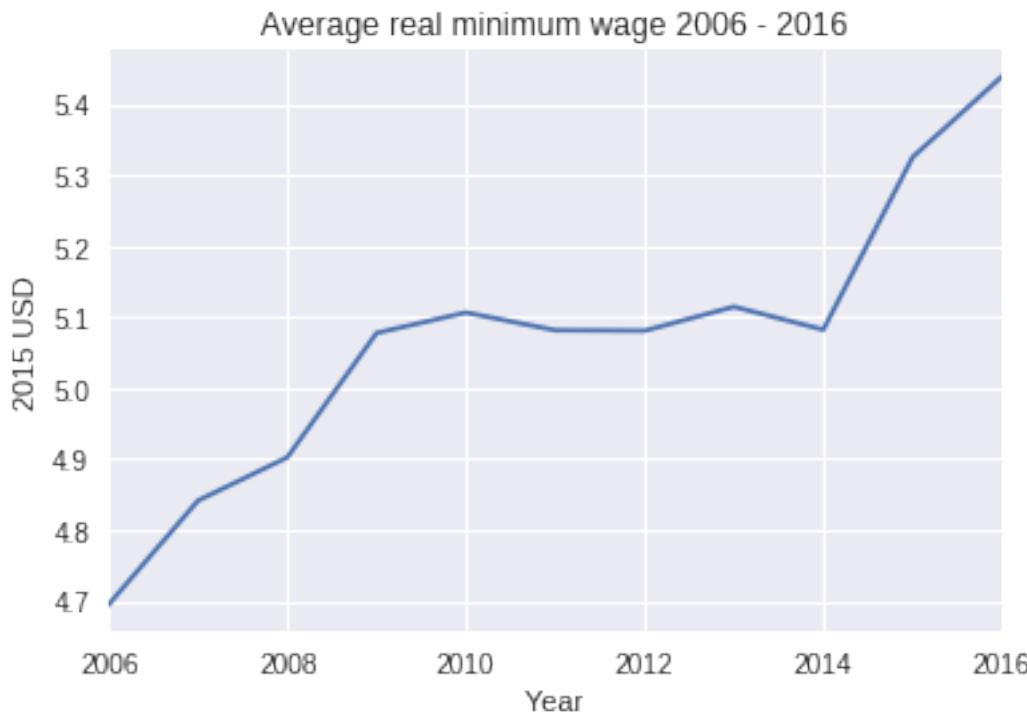
In [28]: `merged.mean(axis=1).head()`

Out[28]: 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

In [29]: `merged.mean(axis=1).plot()`
`plt.title('Average real minimum wage 2006 - 2016')`
`plt.ylabel('2015 USD')`
`plt.xlabel('Year')`
`plt.show()`



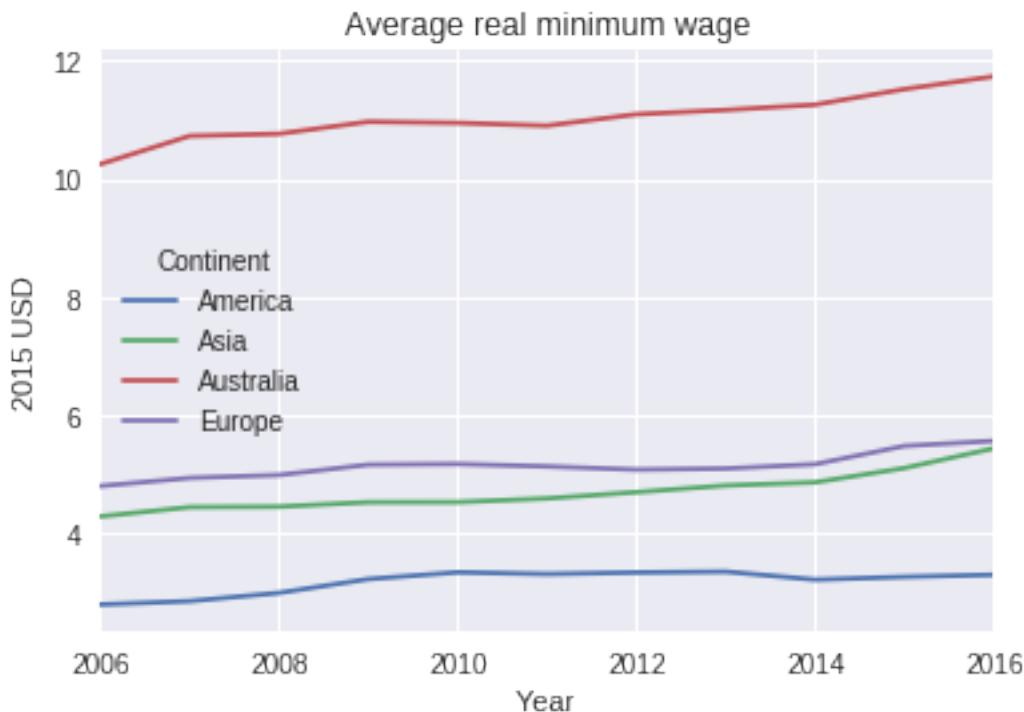
We can also specify a level of the `MultiIndex` (in the column axis) to aggregate over

```
In [30]: merged.mean(level='Continent', axis=1).head()
```

```
Out[30]: 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
```

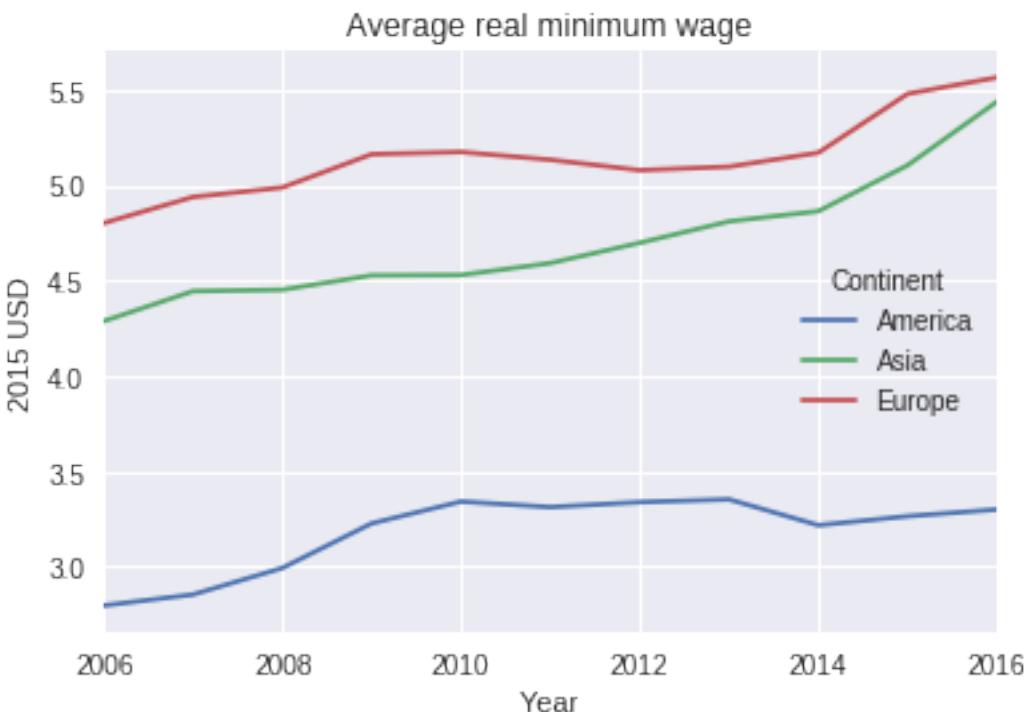
We can plot the average minimum wages in each continent as a time series

```
In [31]: merged.mean(level='Continent', axis=1).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

```
In [32]: merged = merged.drop('Australia', level='Continent', axis=1)
merged.mean(level='Continent', axis=1).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

In [33]: `merged.stack().describe()`

```
Out[33]: 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`

In [34]: `grouped = merged.groupby(level='Continent', axis=1)`
`grouped`

Out[34]: <pandas.core.groupby.generic.DataFrameGroupBy object at 0x7fa8c90f3630>

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`

In [35]: `grouped.size()`

```
Out[35]: 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.

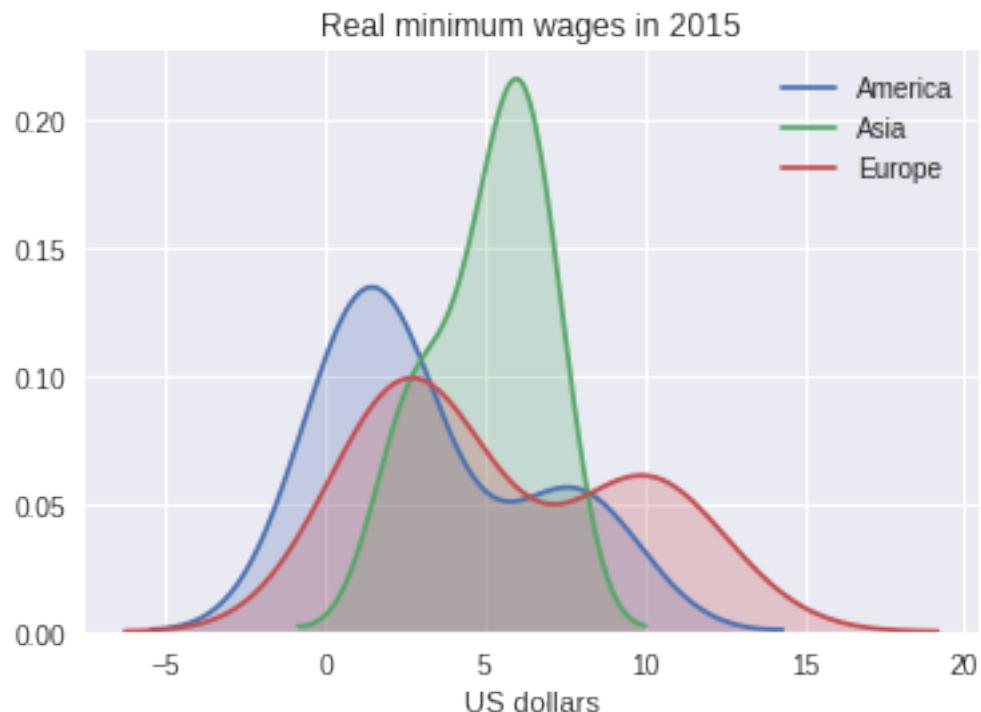
`grouped.groups.keys()` will return the keys from the `groupby` object

```
In [36]: import seaborn as sns
```

```
continents = grouped.groups.keys()

for continent in continents:
    sns.kdeplot(grouped.get_group(continent)[ '2015' ].unstack(), 
label=continent,
shade=True)

plt.title('Real minimum wages in 2015')
plt.xlabel('US dollars')
plt.legend()
plt.show()
```



55.6 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.

55.7 Exercises

55.7.1 Exercise 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:

```
In [37]: url3 = 'https://raw.githubusercontent.com/QuantEcon/lecture-
python/master/source/_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`.

55.7.2 Exercise 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.

55.8 Solutions

55.8.1 Exercise 1

```
In [38]: 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

In [39]: `employ.columns.names`

Out[39]: `FrozenList(['UNIT', 'AGE', 'SEX', 'INDIC_EM', 'GEO'])`

Variables within levels can be quickly retrieved with a loop

In [40]: `for name in employ.columns.names:
 print(name, employ.columns.get_level_values(name).unique())`

```

UNIT Index(['Percentage of total population', 'Thousands 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')
```

55.8.2 Exercise 2

To easily filter by country, swap `GEO` to the top level and sort the `MultiIndex`

```
In [41]: 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'

```
In [42]: 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()
```

```
Out[42]: 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

```
In [43]: employ_f = employ.xs(('Percentage of total population', 'Active population'),
                           level=('UNIT', 'INDIC_EM'),
                           axis=1)
employ_f.head()
```

GEO	AGE	SEX	DATE	Austria			United Kingdom			\
				From 15 to 24 years			From 55 to 64 years			
				Females	Males	Total	Females	Males		
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	DATE	Total						
				59.30						
				59.80						

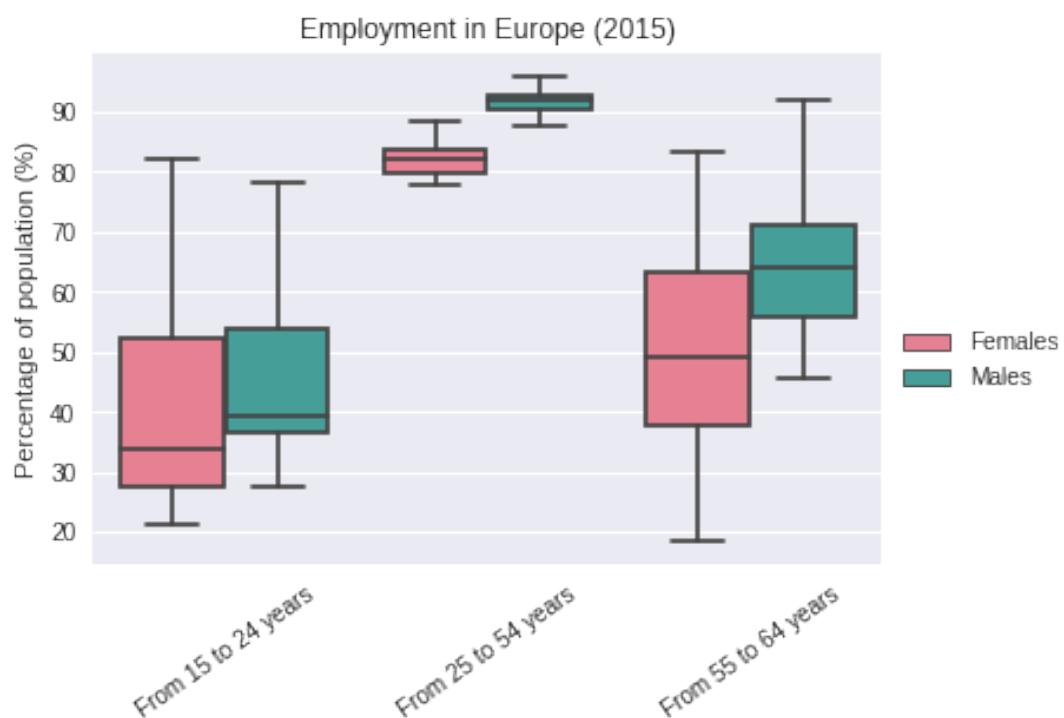
```
2009-01-01 60.30
2010-01-01 60.00
2011-01-01 59.70

[5 rows x 306 columns]
```

Drop the ‘Total’ value before creating the grouped boxplot

```
In [44]: employ_f = employ_f.drop('Total', level='SEX', axis=1)
```

```
In [45]: box = employ_f['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()
```



Chapter 56

Linear Regression in Python

56.1 Contents

- Overview 56.2
- Simple Linear Regression 56.3
- Extending the Linear Regression Model 56.4
- Endogeneity 56.5
- Summary 56.6
- Exercises 56.7
- Solutions 56.8

In addition to what's in Anaconda, this lecture will need the following libraries:

```
In [1]: !pip install linearmodels
```

56.2 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 [1].

- 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:

```
In [2]: import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import pandas as pd
import statsmodels.api as sm
from statsmodels.iolib.summary2 import summary_col
from linarmodels.iv import IV2SLS
```

56.2.1 Prerequisites

This lecture assumes you are familiar with basic econometrics.

For an introductory text covering these topics, see, for example, [111].

56.3 Simple Linear Regression

[1] 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

```
In [3]: 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
0	AFG	0.000000	1.0	NaN	NaN	1.0	2.0
1	AGO	8.000000	1.0	5.363636	7.770645	3.0	3.0
2	ARE	0.000000	1.0	7.181818	9.804219	NaN	NaN
3	ARG	60.000004	1.0	6.386364	9.133459	1.0	6.0
4	ARM	0.000000	0.0	NaN	7.682482	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		

```

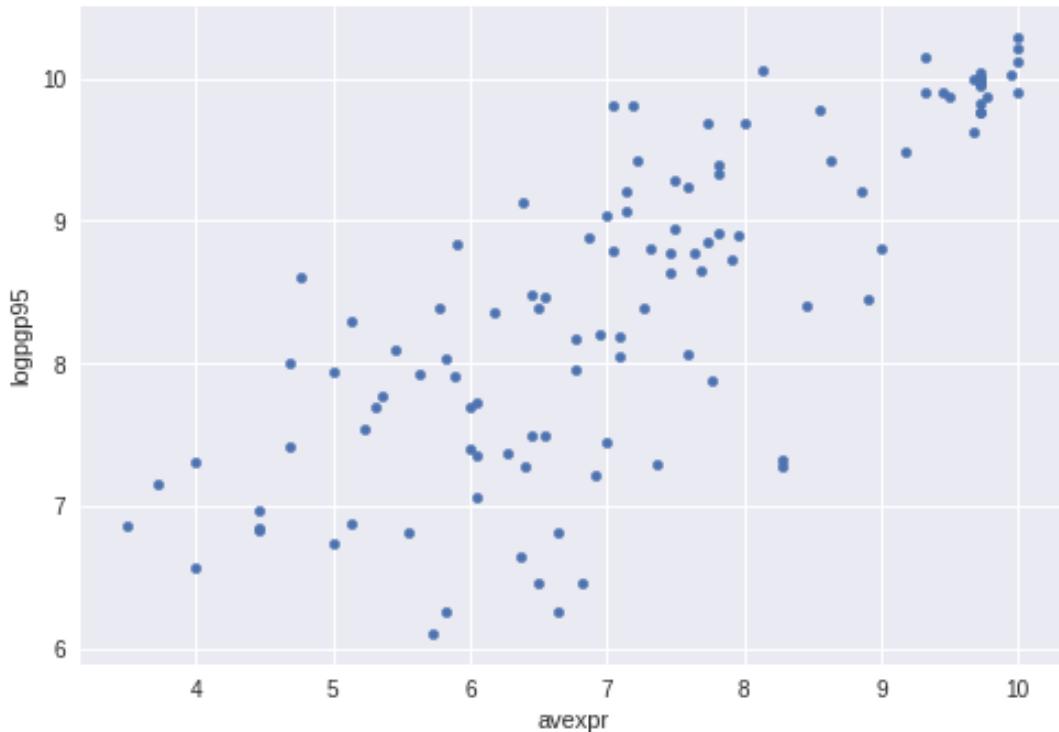
2      NaN      NaN      NaN      NaN      NaN
3      3.0   68.900002  4.232656 -0.872274  1.0
4      NaN      NaN      NaN      NaN      NaN

```

Let's use a scatterplot to see whether any obvious relationship exists between GDP per capita and the protection against expropriation index

```
In [4]: plt.style.use('seaborn')

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 [1])

```
In [5]: # Dropping NA's is required to use numpy's polyfit
df1_subset = df1.dropna(subset=['logpgp95', 'avexpr'])

# Use only 'base sample' for plotting purposes
df1_subset = df1_subset[df1_subset['baseco'] == 1]

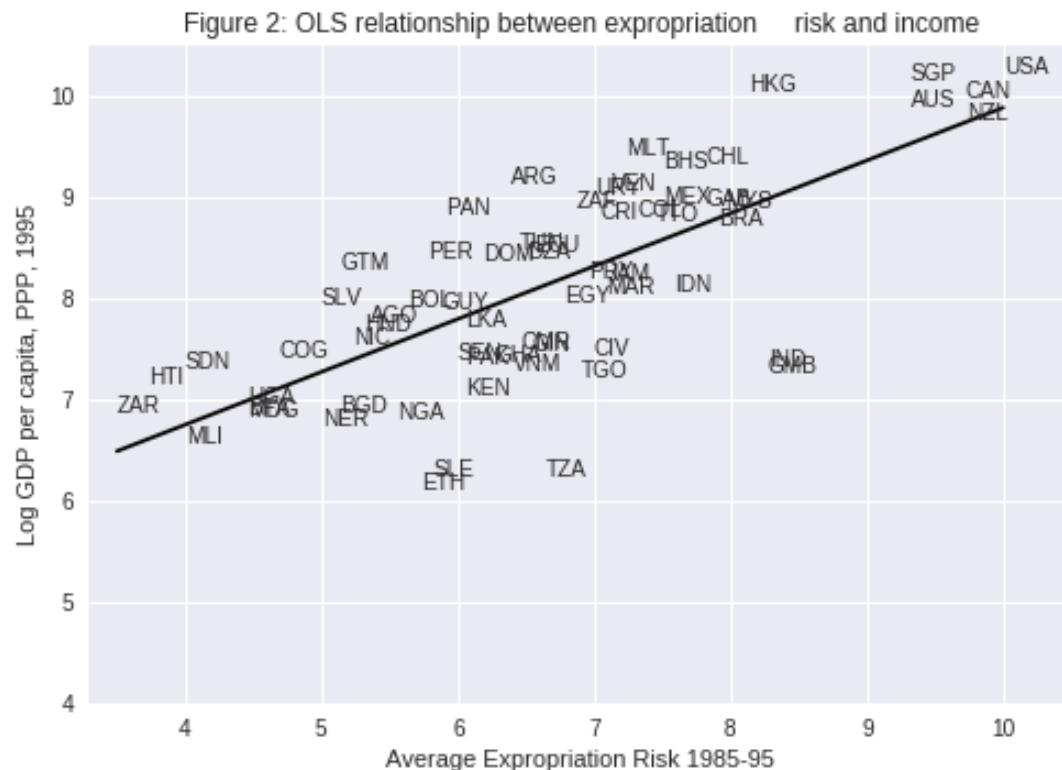
X = df1_subset['avexpr']
y = df1_subset['logpgp95']
labels = df1_subset['shortnam']

# 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 $x_i = 1$)

```
In [6]: 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

```
In [7]: reg1 = sm.OLS(endog=df1['logppg95'], exog=df1[['const', 'avexpr']], \
    missing='drop')
type(reg1)
```

```
Out[7]: 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 $\hat{\beta}_1$

```
In [8]: results = reg1.fit()
         type(results)
```

```
Out[8]: 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.

```
In [9]: 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:	Tue, 09 Mar 2021	Prob (F-statistic):	4.16e-24			
Time:	16:24:42	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			

Warnings:

[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

$$\widehat{\logpgp95}_i = 4.63 + 0.53 \text{ } avexpr_i$$

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.

```
In [10]: mean_expr = np.mean(df1_subset['avexpr'])
mean_expr
```

```
Out[10]: 6.515625
```

```
In [11]: predicted_logpdp95 = 4.63 + 0.53 * 7.07
predicted_logpdp95
```

```
Out[11]: 8.3771
```

An easier (and more accurate) way to obtain this result is to use `.predict()` and set `constant = 1` and $\text{avexpr}_i = \text{mean_expr}$

```
In [12]: results.predict(exog=[1, mean_expr])
```

```
Out[12]: 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

```
In [13]: # Drop missing observations from whole sample
```

```
df1_plot = df1.dropna(subset=['logpgp95', 'avexpr'])

# Plot predicted values

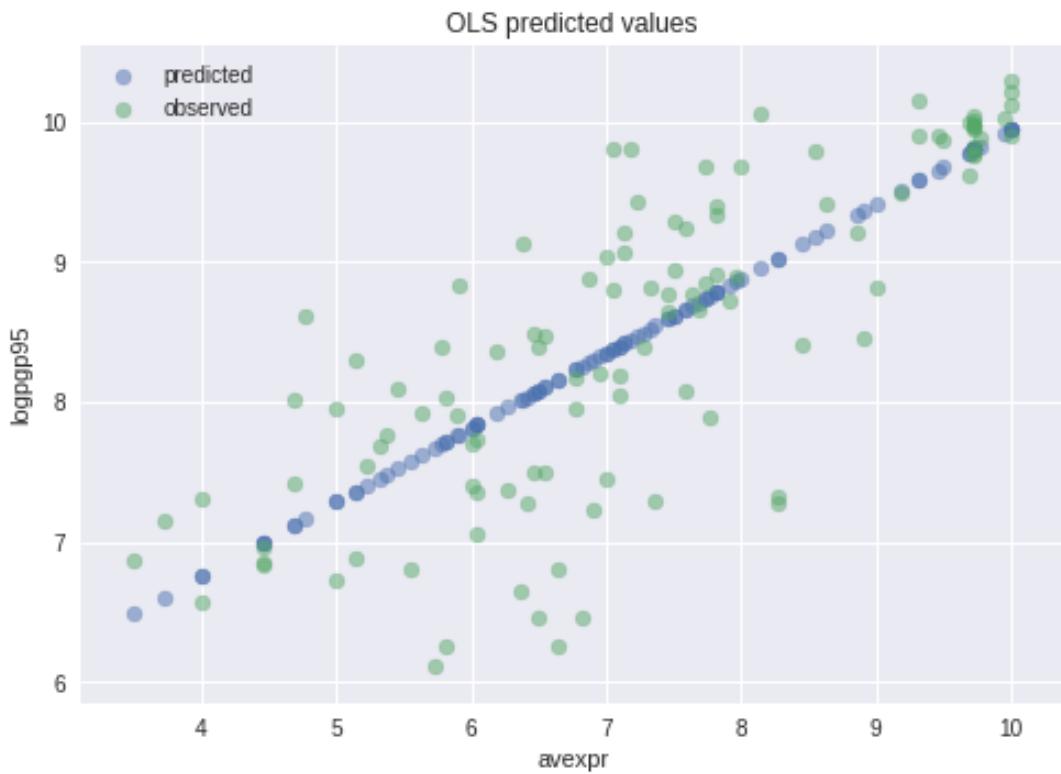
fix, ax = plt.subplots()
ax.scatter(df1_plot['avexpr'], results.predict(), alpha=0.5,
           label='predicted')

# 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()
```



56.4 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 $\logpgp95_i$.

[1] 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`

```
In [14]: 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

# 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)

```
In [15]: 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='%.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*** (0.30)	4.87*** (0.33)	5.85*** (0.34)
avexpr	0.53*** (0.04)	0.46*** (0.06)	0.39*** (0.05)
lat_abst		0.87* (0.49)	0.33 (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
No. observations	111	111	111

Standard errors in parentheses.

* p<.1, ** p<.05, ***p<.01

56.5 Endogeneity

As [1] 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 [1] 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 [1]), 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.

```
In [16]: # 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 \
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.

[1] 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

$\log em4_i$.

Therefore, we will estimate the first-stage regression as

$$avexpr_i = \delta_0 + \delta_1 \log em4_i + v_i$$

The data we need to estimate this equation is located in **maketable4.dta** (only complete data, indicated by **baseco = 1**, is used for estimation)

```
In [17]: # 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:	Tue, 09 Mar 2021	Prob (F-statistic):	1.08e-05			
Time:	16:24:45	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			

Warnings:

[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

$$\logpgp95_i = \beta_0 + \beta_1 \widehat{avexpr}_i + u_i$$

```
In [18]: 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:      Tue, 09 Mar 2021    Prob (F-statistic):   2.66e-10
Time:          16:24:45    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
=====

Warnings:
[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)

```
In [19]: 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: Tue, Mar 09 2021 P-value (F-stat) 0.0000
Time: 16:24:45 Distribution: chi2(1)
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.

[1] 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.

56.6 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.

56.7 Exercises

56.7.1 Exercise 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

$$\begin{aligned} H_0 : \text{Cov}(avexpr_i, u_i) &= 0 \quad (\text{no endogeneity}) \\ H_1 : \text{Cov}(avexpr_i, u_i) &\neq 0 \quad (\text{endogeneity}) \end{aligned}$$

This test is running in two stages.

First, we regress $avexpr_i$ on the instrument, $logem4_i$

$$avexpr_i = \pi_0 + \pi_1 logem4_i + v_i$$

Second, we retrieve the residuals \hat{v}_i and include them in the original equation

$$logpgp95_i = \beta_0 + \beta_1 avexpr_i + \alpha \hat{v}_i + u_i$$

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.

56.7.2 Exercise 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.

56.8 Solutions

56.8.1 Exercise 1

```
In [20]: # Load in data
df4 = pd.read_stata('https://github.com/QuantEcon/lecture-
python/blob/master/source/_static/lecture_specific/ols/maketable4.
dta?raw=true')

# 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:	Tue, 09 Mar 2021	Prob (F-statistic):	1.07e-17			
Time:	16:24:45	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			

Warnings:

[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.

56.8.2 Exercise 2

```
In [21]: # Load in data
df1 = pd.read_stata('https://github.com/QuantEcon/lecture-
python/blob/master/source/_static/lecture_specific/ols/maketable1.
dta?raw=true')
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 57

Maximum Likelihood Estimation

57.1 Contents

- Overview 57.2
- Set Up and Assumptions 57.3
- Conditional Distributions 57.4
- Maximum Likelihood Estimation 57.5
- MLE with Numerical Methods 57.6
- Maximum Likelihood Estimation with `statsmodels` 57.7
- Summary 57.8
- Exercises 57.9
- Solutions 57.10

57.2 Overview

In a [previous lecture](#), 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:

```
In [1]: import numpy as np
         from numpy import exp
         import matplotlib.pyplot as plt
```

```
%matplotlib inline
from scipy.special import factorial
import pandas as pd
from mpl_toolkits.mplot3d import Axes3D
import statsmodels.api as sm
from statsmodels.api import Poisson
from scipy import stats
from scipy.stats import norm
from statsmodels.iolib.summary2 import summary_col
```

57.2.1 Prerequisites

We assume familiarity with basic probability and multivariate calculus.

57.3 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.

57.3.1 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**.

57.3.2 Counting Billionaires

Treisman [106] 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

$$f(y) = \frac{\mu^y}{y!} e^{-\mu}, \quad y = 0, 1, 2, \dots, \infty$$

We can plot the Poisson distribution over y for different values of μ as follows

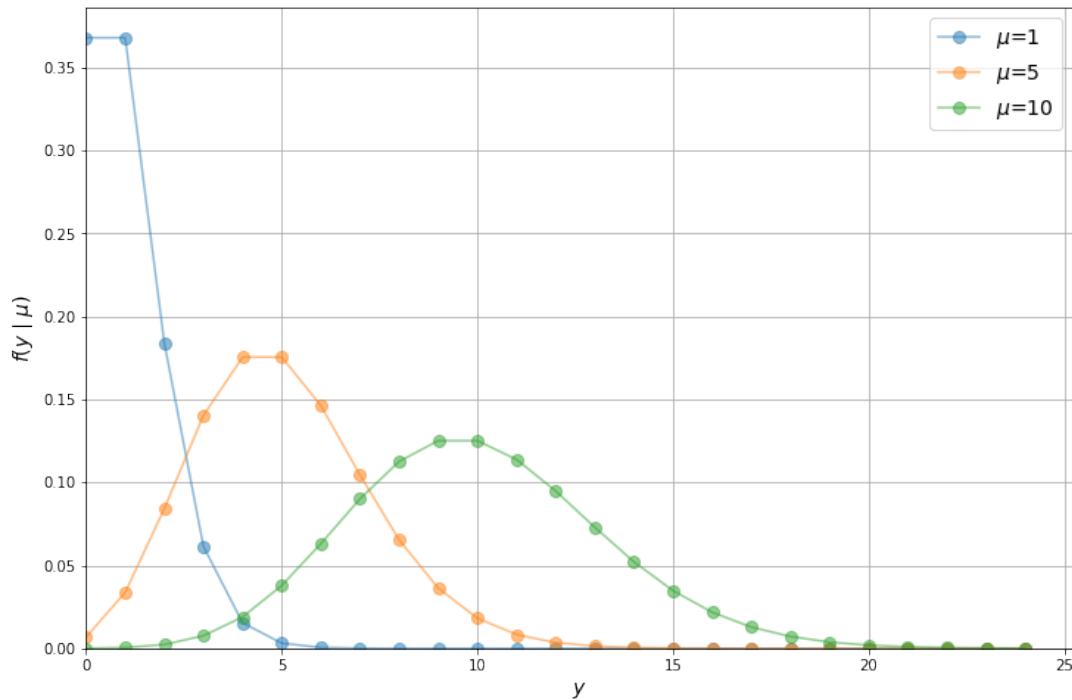
```
In [2]: poisson_pmf = lambda y, mu: mu**y / factorial(y) * exp(-mu)
y_values = range(0, 25)

fig, ax = plt.subplots(figsize=(12, 8))

for mu in [1, 5, 10]:
    distribution = []
    for y_i in y_values:
        distribution.append(poisson_pmf(y_i, mu))
    ax.plot(y_values,
            distribution,
            label=f'$\mu={mu}$',
            alpha=0.5,
            marker='o',
            markersize=8)

ax.grid()
ax.set_xlabel('y', fontsize=14)
ax.set_ylabel('f(y | \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 here or from its [AER page](#).

In [3]: `pd.options.display.max_columns = 10`

```
# Load in data and view
df = pd.read_stata('https://github.com/QuantEcon/lecture-
python/blob/master/source/_static/lecture_specific/mle/fp.dta?raw=true')
df.head()
```

Out[3]:

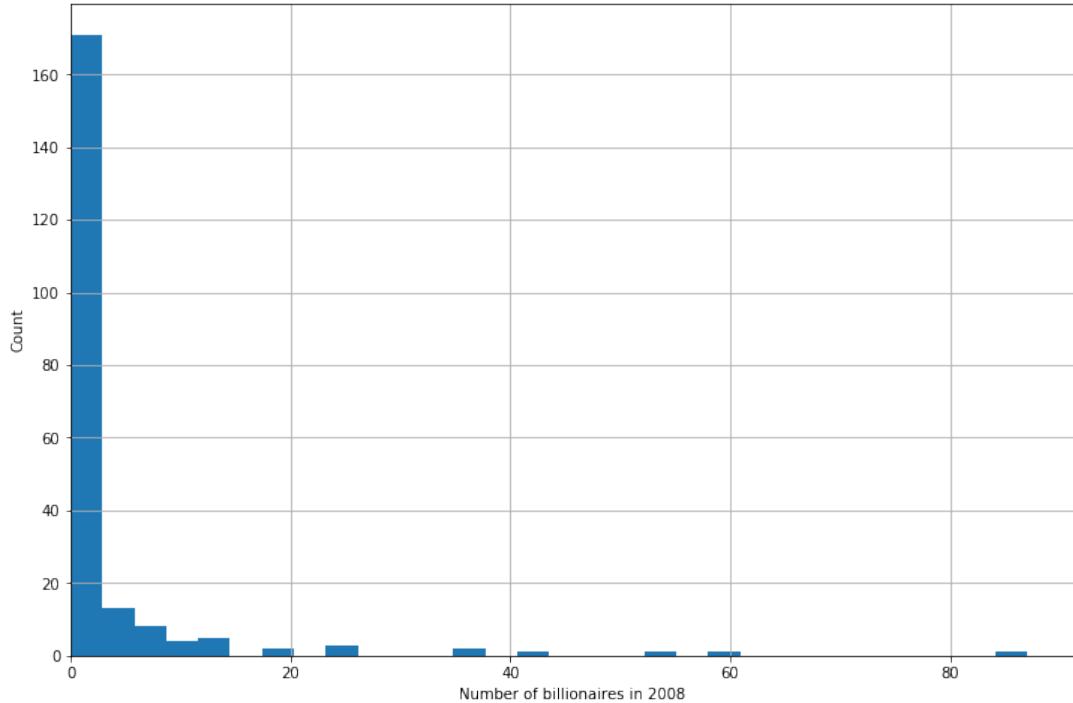
	country	ccode	year	cyear	numbil	...	topint08	rintr	\
0	United States	2.0	1990.0	21990.0	NaN	...	39.799999	4.988405	
1	United States	2.0	1991.0	21991.0	NaN	...	39.799999	4.988405	
2	United States	2.0	1992.0	21992.0	NaN	...	39.799999	4.988405	
3	United States	2.0	1993.0	21993.0	NaN	...	39.799999	4.988405	
4	United States	2.0	1994.0	21994.0	NaN	...	39.799999	4.988405	
	noyrs	roflaw	nrrents						
0	20.0	1.61	NaN						
1	20.0	1.61	NaN						
2	20.0	1.61	NaN						
3	20.0	1.61	NaN						
4	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)

In [4]: `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).

57.4 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:

$$f(y_i | \mathbf{x}_i) = \frac{\mu_i^{y_i}}{y_i!} e^{-\mu_i}; \quad y_i = 0, 1, 2, \dots, \infty. \quad (1)$$

where $\mu_i = \exp(\mathbf{x}'_i \beta) = \exp(\beta_0 + \beta_1 x_{i1} + \dots + \beta_k x_{ik})$

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

In [5]: `y_values = range(0, 20)`

```
# Define a parameter vector with estimates
β = np.array([0.26, 0.18, 0.25, -0.1, -0.22])

# Create some observations X
```

```

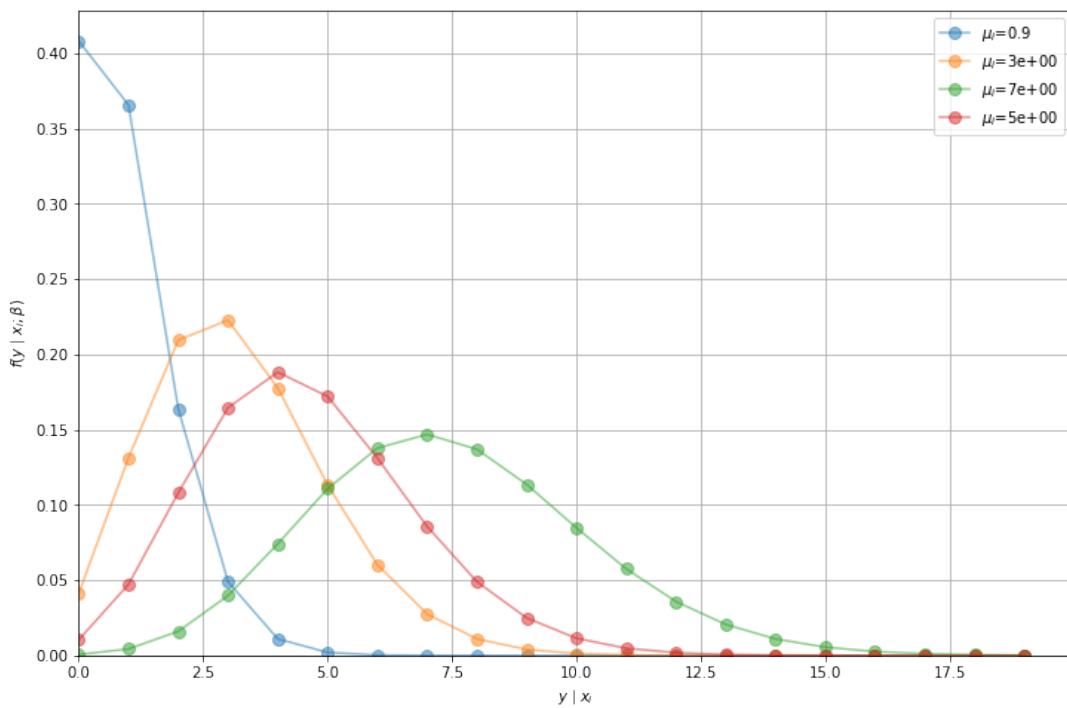
datasets = [np.array([0, 1, 1, 1, 2]),
            np.array([2, 3, 2, 4, 0]),
            np.array([3, 4, 5, 3, 2]),
            np.array([6, 5, 4, 4, 7])]

fig, ax = plt.subplots(figsize=(12, 8))

for X in datasets:
    mu = exp(X @ beta)
    distribution = []
    for y_i in y_values:
        distribution.append(poisson_pmf(y_i, mu))
    ax.plot(y_values,
            distribution,
            label=f'$\mu_i={mu:.1f}$',
            marker='o',
            markersize=8,
            alpha=0.5)

ax.grid()
ax.legend()
ax.set_xlabel('y | x_i')
ax.set_ylabel(r'$f(y | x_i; \beta)$')
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).

57.5 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

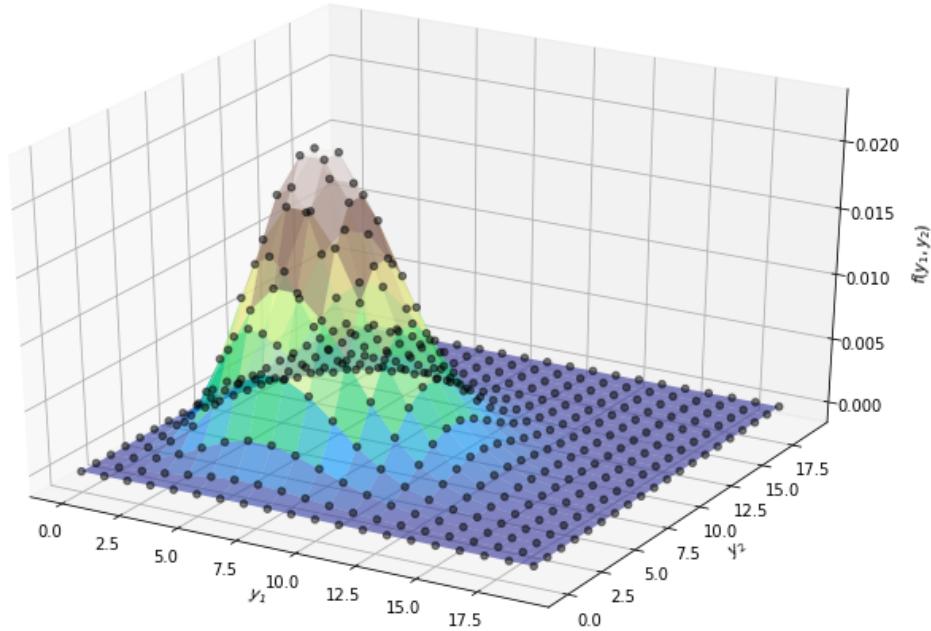
```
In [6]: def plot_joint_poisson(mu=7, y_n=20):
    yi_values = np.arange(0, y_n, 1)

    # Create coordinate points of X and Y
    X, Y = np.meshgrid(yi_values, yi_values)

    # Multiply distributions together
    Z = poisson_pmf(X, mu) * poisson_pmf(Y, mu)

    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('$y_1$', ylabel='$y_2$')
    ax.set_zlabel('$f(y_1, y_2)$', labelpad=10)
    plt.show()

plot_joint_poisson(mu=7, y_n=20)
```



Similarly, the joint pmf of our data (which is distributed as a conditional Poisson distribution) can be written as

$$f(y_1, y_2, \dots, y_n | \mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n; \beta) = \prod_{i=1}^n \frac{\mu_i^{y_i}}{y_i!} e^{-\mu_i}$$

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

$$\begin{aligned} \mathcal{L}(\beta | y_1, y_2, \dots, y_n ; \mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n) &= \prod_{i=1}^n \frac{\mu_i^{y_i}}{y_i!} e^{-\mu_i} \\ &= f(y_1, y_2, \dots, y_n | \mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n; \beta) \end{aligned}$$

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

$$\begin{aligned}
\log \mathcal{L}(\beta) &= \log \left(f(y_1; \beta) \cdot f(y_2; \beta) \cdot \dots \cdot f(y_n; \beta) \right) \\
&= \sum_{i=1}^n \log f(y_i; \beta) \\
&= \sum_{i=1}^n \log \left(\frac{\mu_i^{y_i}}{y_i!} e^{-\mu_i} \right) \\
&= \sum_{i=1}^n y_i \log \mu_i - \sum_{i=1}^n \mu_i - \sum_{i=1}^n \log y!
\end{aligned}$$

The MLE of the Poisson to the Poisson for $\hat{\beta}$ can be obtained by solving

$$\max_{\beta} \left(\sum_{i=1}^n y_i \log \mu_i - \sum_{i=1}^n \mu_i - \sum_{i=1}^n \log y! \right)$$

However, no analytical solution exists to the above problem – to find the MLE we need to use numerical methods.

57.6 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

$$\log \mathcal{L}(\beta) = -(\beta - 10)^2 - 10$$

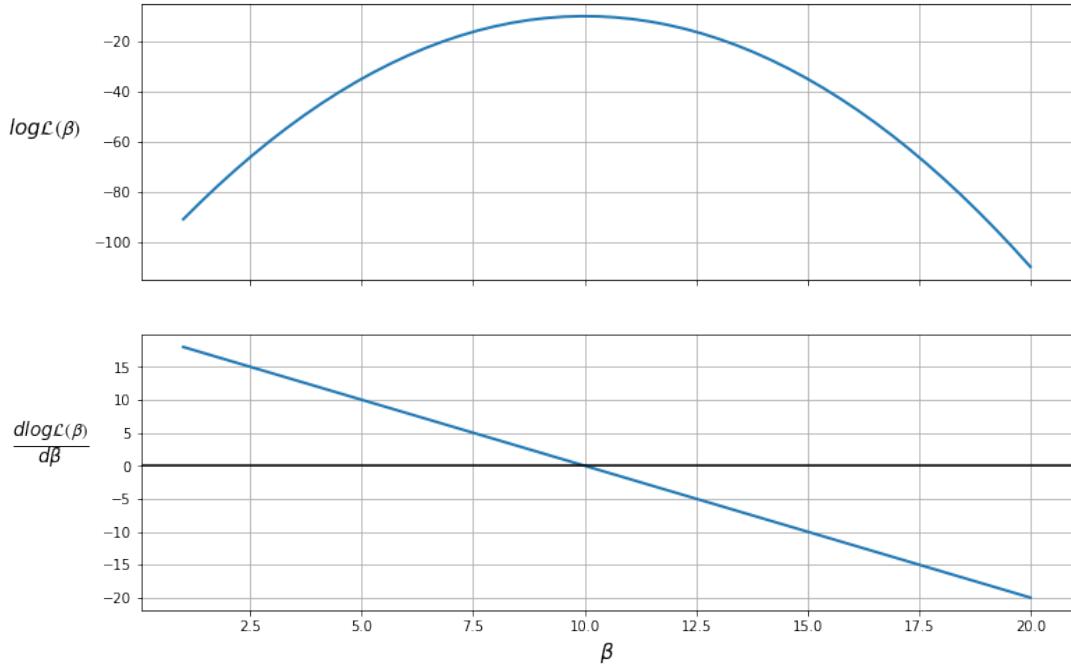
```
In [7]: β = np.linspace(1, 20)
logL = -(β - 10)**2 - 10
dlogL = -2 * β + 20

fig, (ax1, ax2) = plt.subplots(2, sharex=True, figsize=(12, 8))

ax1.plot(β, logL, lw=2)
ax2.plot(β, dlogL, lw=2)

ax1.set_ylabel(r'$\log \mathcal{L}(\beta)$',
               rotation=0,
               labelpad=35,
               fontsize=15)
ax2.set_ylabel(r'$\frac{d \log \mathcal{L}(\beta)}{d \beta}$',
               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:

$$G(\beta_{(k)}) = \frac{d \log \mathcal{L}(\beta_{(k)})}{d \beta_{(k)}}$$

$$H(\beta_{(k)}) = \frac{d^2 \log \mathcal{L}(\beta_{(k)})}{d \beta_{(k)} d \beta'_{(k)}}$$

1. 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 $G(\beta_{(k)}) = 0$ ie. 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

In [8]: `class PoissonRegression:`

```
    def __init__(self, y, X, β):
        self.X = X
        self.n, self.k = X.shape
        # Reshape y as a n_by_1 column vector
        self.y = y.reshape(self.n,1)
        # Reshape β as a k_by_1 column vector
        self.β = β.reshape(self.k,1)

    def μ(self):
        return np.exp(self.X @ self.β)

    def logL(self):
        y = self.y
        μ = self.μ()
        return np.sum(y * np.log(μ) - μ - np.log(factorial(y)))

    def G(self):
        y = self.y
        μ = self.μ()
        return X.T @ (y - μ)

    def H(self):
        X = self.X
        μ = self.μ()
        return -(X.T @ (μ * X))
```

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.

In [9]: `def newton_raphson(model, tol=1e-3, max_iter=1000, 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 np.any(error > tol) and i < max_iter:
    H, G = model.H(), model.G()
    β_new = model.β - (np.linalg.inv(H) @ G)
    error = β_new - model.β
    model.β = β_new

    # Print iterations
    if display:
        β_list = [f'{t:.3}' for t in list(model.β.flatten())]
        update = f'{i:<13}{model.logL():<16.8}{β_list}'
        print(update)

    i += 1

print(f'Number of iterations: {i}')
print(f'β_hat = {model.β.flatten()}')

# Return a flat array for β (instead of a k_by_1 column vector)
return model.β.flatten()

```

Let's try out our algorithm with a small dataset of 5 observations and 3 variables in \mathbf{X} .

```

In [10]: X = np.array([[1, 2, 5],
                      [1, 1, 3],
                      [1, 4, 2],
                      [1, 5, 2],
                      [1, 3, 1]])

y = np.array([1, 0, 1, 1, 0])

# Take a guess at initial βs
init_β = np.array([0.1, 0.1, 0.1])

# Create an object with Poisson model values
poi = PoissonRegression(y, X, β=init_β)

# Use newton_raphson to find the MLE
β_hat = newton_raphson(poi, display=True)

```

Iteration_k	Log-likelihood	θ

0	-4.3447622	[-1.49, '0.265', '0.244']
1	-3.5742413	[-3.38, '0.528', '0.474']
2	-3.3999526	[-5.06, '0.782', '0.702']

```

3           -3.3788646      ['-5.92', '0.909', '0.82']
4           -3.3783559      ['-6.07', '0.933', '0.843']
5           -3.3783555      ['-6.08', '0.933', '0.843']
Number of iterations: 6
β_hat = [-6.07848205  0.93340226  0.84329625]

```

As this was a simple model with few observations, the algorithm achieved convergence in only 6 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 $\hat{\beta}$

In [11]: poi.G()

```
Out[11]: array([[-3.95169226e-07],
 [-1.00114804e-06],
 [-7.73114559e-07]])
```

The iterative process can be visualized in the following diagram, where the maximum is found at $\beta = 10$

In [12]: `logL = lambda x: -(x - 10) ** 2 - 10`

```

def find_tangent(β, a=0.01):
    y1 = logL(β)
    y2 = logL(β+a)
    x = np.array([[β, 1], [β+a, 1]])
    m, c = np.linalg.lstsq(x, np.array([y1, y2]), rcond=None)[0]
    return m, c

β = np.linspace(2, 18)
fig, ax = plt.subplots(figsize=(12, 8))
ax.plot(β, logL(β), lw=2, c='black')

for β in [7, 8.5, 9.5, 10]:
    β_line = np.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], f'$G({\beta}) = {abs(m)}:{.0f}$', fontsize=12)
    ax.vlines(β, -24, logL(β), linestyles='--', alpha=0.5)
    ax.hlines(logL(β), 6, β, linestyles='--', alpha=0.5)

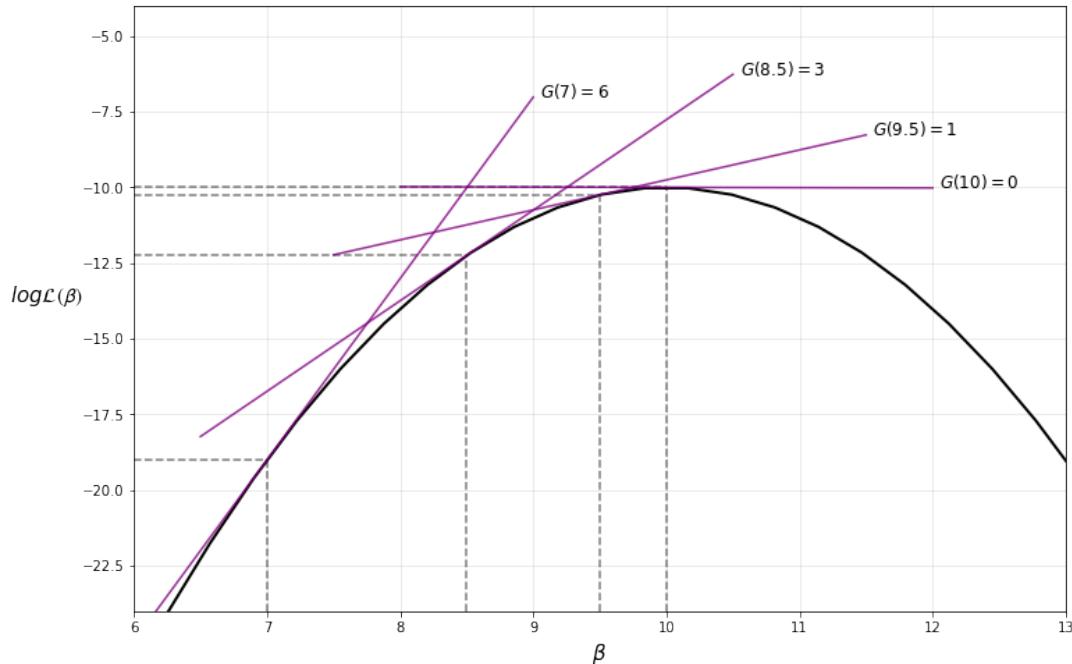
ax.set(ylim=(-24, -4), xlim=(6, 13))
ax.set_xlabel(r'$\beta$', fontsize=15)
ax.set_ylabel(r'$\log \mathcal{L}(\beta)$',
              rotation=0,

```

```

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](#).

57.7 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.

```

In [13]: X = np.array([[1, 2, 5],
                     [1, 1, 3],
                     [1, 4, 2],
                     [1, 5, 2],
                     [1, 3, 1]])

y = np.array([1, 0, 1, 1, 0])

stats_poisson = Poisson(y, X).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:              Tue, 09 Mar 2021      Pseudo R-squ.:            0.2546
Time:                 16:20:07      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 (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)

```
In [14]: # Keep only year 2008
df = df[df['year'] == 2008]

# Add a constant
df['const'] = 1

# Variable sets
reg1 = ['const', 'lndgppc', 'lnpop', 'gattwto08']
reg2 = ['const', 'lndgppc', 'lnpop',
        'gattwto08', 'lnmcap08', 'rintr', 'topint08']
reg3 = ['const', 'lndgppc', '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

```
In [15]: # Specify model
poisson_reg = sm.Poisson(df[['numbilo']], df[reg1],
                         missing='drop').fit(cov_type='HCO')
print(poisson_reg.summary())
```

```

Optimization terminated successfully.
    Current function value: 2.226090
    Iterations 9
                Poisson Regression Results
=====
Dep. Variable:      numbilo    No. Observations:                 197
Model:             Poisson     Df Residuals:                  193
Method:            MLE        Df Model:                      3
Date:      Tue, 09 Mar 2021   Pseudo R-squ.:               0.8574
Time:          16:20:07      Log-Likelihood:           -438.54
converged:       True       LL-Null:                  -3074.7
Covariance Type: HCO       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
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

```
In [16]: 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 = sm.Poisson(df[['numbilo']], df[reg],
                         missing='drop').fit(cov_type='HCO',
                                              maxiter=100, disp=0)
    results.append(result)

results_table = summary_col(results,
                            float_format='%.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*** (2.578)	-19.444*** (4.820)	-20.858*** (4.255)
lndgppc	1.084*** (0.138)	0.717*** (0.244)	0.737*** (0.233)
lnpop	1.171*** (0.097)	0.806*** (0.213)	0.929*** (0.195)
gattwto08	0.006 (0.007)	0.007 (0.006)	0.004 (0.006)
lnmcap08	0.399** (0.172)	0.286* (0.167)	
rintr	-0.010 (0.010)	-0.009 (0.010)	
topint08	-0.051*** (0.011)	-0.058*** (0.012)	
nrrents		-0.005 (0.010)	
roflaw		0.203 (0.372)	
Pseudo R-squared	0.86	0.90	0.90
No. observations	197	131	131

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

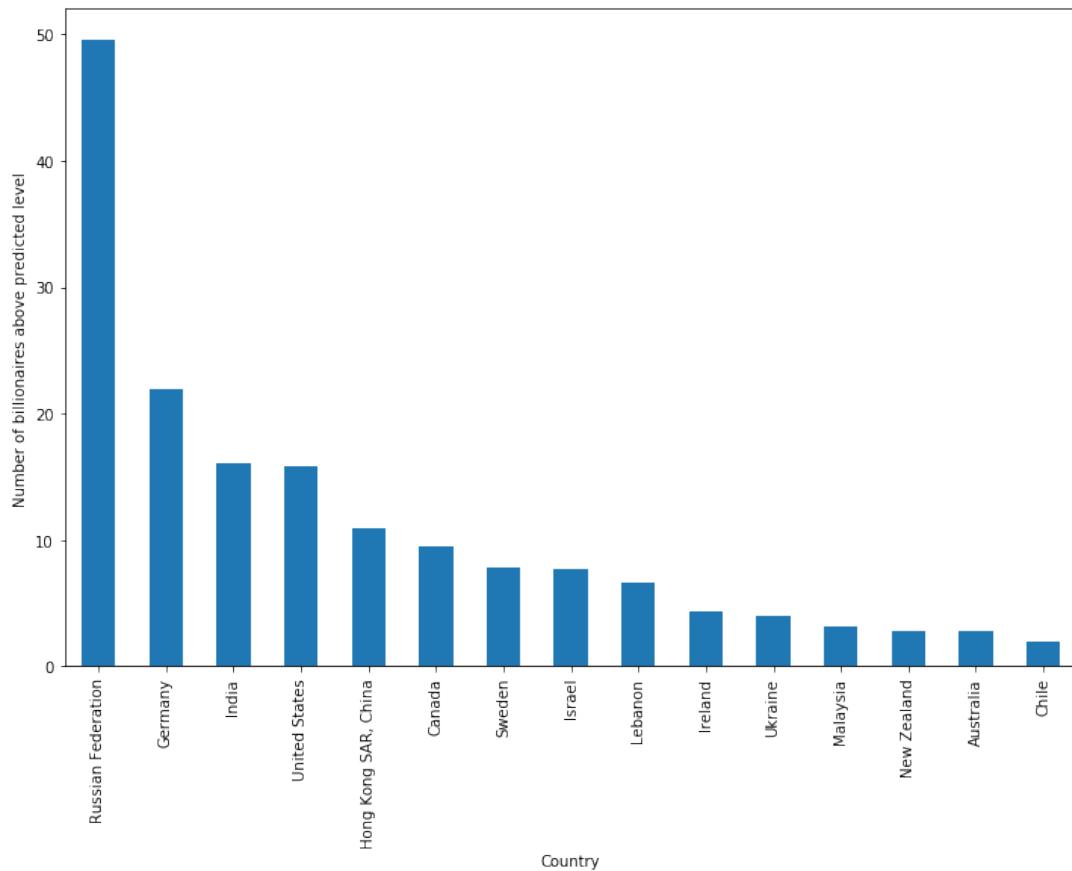
```
In [17]: data = ['const', 'lndgppc', 'lnpop', 'gattwto08', 'lnmcap08', 'rintr',
              'topint08', 'nrrents', 'roflaw', 'numbilo', 'country']
results_df = df[data].dropna()

# Use last model (model 3)
results_df['prediction'] = results[-1].predict()

# Calculate difference
results_df['difference'] = results_df['numbilo'] - 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.

57.8 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](#).

57.9 Exercises

57.9.1 Exercise 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

$$f(y_i; \beta) = \mu_i^{y_i} (1 - \mu_i)^{1-y_i}, \quad y_i = 0, 1$$

where $\mu_i = \Phi(\mathbf{x}'_i \beta)$

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 **scipy** module **stats.norm** contains the functions needed to compute the cmf and pmf of the normal distribution.

57.9.2 Exercise 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

```
In [18]: 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.

57.10 Solutions

57.10.1 Exercise 1

The log-likelihood can be written as

$$\log \mathcal{L} = \sum_{i=1}^n [y_i \log \Phi(\mathbf{x}'_i \beta) + (1 - y_i) \log(1 - \Phi(\mathbf{x}'_i \beta))]$$

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

In [19]: `class ProbitRegression:`

```
def __init__(self, y, X, beta):
    self.X, self.y, self.beta = X, y, beta
    self.n, self.k = X.shape

def mu(self):
    return norm.cdf(self.X @ self.beta.T)

def phi(self):
    return norm.pdf(self.X @ self.beta.T)

def logL(self):
    mu = self.mu()
    return np.sum(y * np.log(mu) + (1 - y) * np.log(1 - mu))

def G(self):
    mu = self.mu()
    phi = self.phi()
    return np.sum((X.T * y * phi / mu - X.T * (1 - y) * phi / (1 - mu)),
                 axis=1)

def H(self):
    X = self.X
    beta = self.beta
    mu = self.mu()
    phi = self.phi()
    a = (phi + (X @ beta.T) * mu) / mu**2
    b = (phi - (X @ beta.T) * (1 - mu)) / (1 - mu)**2
    return -(phi * (y * a + (1 - y) * b) * X.T) @ X
```

57.10.2 Exercise 2

In [20]: `X = np.array([[1, 2, 4], [1, 1, 1], [1, 4, 3], [1, 5, 6], [1, 3, 5]])`

`y = np.array([1, 0, 1, 1, 0])`

```
# Take a guess at initial β
β = np.array([0.1, 0.1, 0.1])

# Create instance of Probit regression class
prob = ProbitRegression(y, X, β)

# Run Newton-Raphson algorithm
newton_raphson(prob)

Iteration_k  Log-likelihood  θ
-----
---
---
0          -2.3796884      ['-1.34', '0.775', '-0.157']
1          -2.3687526      ['-1.53', '0.775', '-0.0981']
2          -2.3687294      ['-1.55', '0.778', '-0.0971']
3          -2.3687294      ['-1.55', '0.778', '-0.0971']
Number of iterations: 4
β_hat = [-1.54625858  0.77778952 -0.09709757]
```

Out[20]: array([-1.54625858, 0.77778952, -0.09709757])

In [21]: # Use statsmodels to verify results

```
print(Probit(y, X).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:              Tue, 09 Mar 2021   Pseudo R-squ.:  0.2961
Time:                16:20:07      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
=====
```


Bibliography

- [1] 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.
- [2] Daron Acemoglu and James A. Robinson. The political economy of the Kuznets curve. *Review of Development Economics*, 6(2):183–203, 2002.
- [3] SeHyoun Ahn, Greg Kaplan, Benjamin Moll, Thomas Winberry, and Christian Wolf. When inequality matters for macro and macro matters for inequality. *NBER Macroeconomics Annual*, 32(1):1–75, 2018.
- [4] S Rao Aiyagari. Uninsured Idiosyncratic Risk and Aggregate Saving. *The Quarterly Journal of Economics*, 109(3):659–684, 1994.
- [5] D. B. O. Anderson and J. B. Moore. *Optimal Filtering*. Dover Publications, 2005.
- [6] 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.
- [7] Robert L Axtell. Zipf distribution of us firm sizes. *science*, 293(5536):1818–1820, 2001.
- [8] Robert J Barro. On the Determination of the Public Debt. *Journal of Political Economy*, 87(5):940–971, 1979.
- [9] Jess Benhabib and Alberto Bisin. Skewed wealth distributions: Theory and empirics. *Journal of Economic Literature*, 56(4):1261–91, 2018.
- [10] 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.
- [11] 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.
- [12] Dmitri Bertsekas. *Dynamic Programming and Stochastic Control*. Academic Press, New York, 1975.
- [13] Truman Bewley. The permanent income hypothesis: A theoretical formulation. *Journal of Economic Theory*, 16(2):252–292, 1977.
- [14] Truman F Bewley. Stationary monetary equilibrium with a continuum of independently fluctuating consumers. In Werner Hildenbrand and Andreu Mas-Colell, editors, *Contributions to Mathematical Economics in Honor of Gerard Debreu*, pages 27–102. North-Holland, Amsterdam, 1986.

- [15] Anmol Bhandari, David Evans, Mikhail Golosov, and Thomas J Sargent. Inequality, business cycles, and monetary-fiscal policy. Technical report, National Bureau of Economic Research, 2018.
- [16] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.
- [17] Dariusz Buraczewski, Ewa Damek, Thomas Mikosch, et al. *Stochastic models with power-law tails*. Springer, 2016.
- [18] 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.
- [19] Andrew S Caplin. The variability of aggregate demand with (s, s) inventory policies. *Econometrica*, pages 1395–1409, 1985.
- [20] Christopher D Carroll. A Theory of the Consumption Function, with and without Liquidity Constraints. *Journal of Economic Perspectives*, 15(3):23–45, 2001.
- [21] Christopher D Carroll. The method of endogenous gridpoints for solving dynamic stochastic optimization problems. *Economics Letters*, 91(3):312–320, 2006.
- [22] David Cass. Optimum growth in an aggregative model of capital accumulation. *Review of Economic Studies*, 32(3):233–240, 1965.
- [23] Wilbur John Coleman. Solving the Stochastic Growth Model by Policy-Function Iteration. *Journal of Business & Economic Statistics*, 8(1):27–29, 1990.
- [24] 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.
- [25] Bruno de Finetti. La prévision: Ses lois logiques, ses sources subjectives. *Annales de l'Institut Henri Poincaré*, 7:1 – 68, 1937. English translation in Kyburg and Smokler (eds.), *Studies in Subjective Probability*, Wiley, New York, 1964.
- [26] Angus Deaton. Saving and Liquidity Constraints. *Econometrica*, 59(5):1221–1248, 1991.
- [27] Angus Deaton and Christina Paxson. Intertemporal Choice and Inequality. *Journal of Political Economy*, 102(3):437–467, 1994.
- [28] 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.
- [29] Ulrich Doraszelski and Mark Satterthwaite. Computable markov-perfect industry dynamics. *The RAND Journal of Economics*, 41(2):215–243, 2010.
- [30] Y E Du, Ehud Lehrer, and A D Y Pauzner. Competitive economy as a ranking device over networks. submitted, 2013.
- [31] R M Dudley. *Real Analysis and Probability*. Cambridge Studies in Advanced Mathematics. Cambridge University Press, 2002.
- [32] 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.
- [33] Robert F Engle and Clive W J Granger. Co-integration and Error Correction: Representation, Estimation, and Testing. *Econometrica*, 55(2):251–276, 1987.

- [34] Richard Ericson and Ariel Pakes. Markov-perfect industry dynamics: A framework for empirical work. *The Review of Economic Studies*, 62(1):53–82, 1995.
- [35] 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.
- [36] G W Evans and S Honkapohja. *Learning and Expectations in Macroeconomics*. Frontiers of Economic Research. Princeton University Press, 2001.
- [37] Pablo Fajgelbaum, Edouard Schaal, and Mathieu Taschereau-Dumouchel. Uncertainty traps. Technical report, National Bureau of Economic Research, 2015.
- [38] M. Friedman. *A Theory of the Consumption Function*. Princeton University Press, 1956.
- [39] Milton Friedman and Rose D Friedman. *Two Lucky People*. University of Chicago Press, 1998.
- [40] Yoshi Fujiwara, Corrado Di Guilmi, Hideaki Aoyama, Mauro Gallegati, and Wataru Souma. Do pareto–zipf and gibrat laws hold true? an analysis with european firms. *Physica A: Statistical Mechanics and its Applications*, 335(1-2):197–216, 2004.
- [41] Xavier Gabaix. Power laws in economics: An introduction. *Journal of Economic Perspectives*, 30(1):185–206, 2016.
- [42] Robert Gibrat. *Les inégalités économiques: Applications d'une loi nouvelle, la loi de l'effet proportionnel*. PhD thesis, Recueil Sirey, 1931.
- [43] Edward Glaeser, Jose Scheinkman, and Andrei Shleifer. The injustice of inequality. *Journal of Monetary Economics*, 50(1):199–222, 2003.
- [44] Geoffrey J Gordon. Stable function approximation in dynamic programming. In *Machine Learning Proceedings 1995*, pages 261–268. Elsevier, 1995.
- [45] Olle Häggström. *Finite Markov chains and algorithmic applications*, volume 52. Cambridge University Press, 2002.
- [46] 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.
- [47] Robert E Hall. Stochastic Implications of the Life Cycle-Permanent Income Hypothesis: Theory and Evidence. *Journal of Political Economy*, 86(6):971–987, 1978.
- [48] 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*, No. 505, 1982.
- [49] James D Hamilton. What's real about the business cycle? *Federal Reserve Bank of St. Louis Review*, (July-August):435–452, 2005.
- [50] L P Hansen and T J Sargent. *Robustness*. Princeton University Press, 2008.
- [51] L P Hansen and T J Sargent. *Recursive Models of Dynamic Linear Economies*. The Gorman Lectures in Economics. Princeton University Press, 2013.
- [52] Lars Peter Hansen and Scott F Richard. The role of conditioning information in deducing testable restrictions implied by dynamic asset pricing models. *Econometrica*, 55(3):587–613, May 1987.

- [53] 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.
- [54] J. Michael Harrison and David M. Kreps. Martingales and arbitrage in multiperiod securities markets. *Journal of Economic Theory*, 20(3):381–408, June 1979.
- [55] 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.
- [56] 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.
- [57] Hugo A Hopenhayn. Entry, exit, and firm dynamics in long run equilibrium. *Econometrica: Journal of the Econometric Society*, pages 1127–1150, 1992.
- [58] Hugo A Hopenhayn and Edward C Prescott. Stochastic Monotonicity and Stationary Distributions for Dynamic Economies. *Econometrica*, 60(6):1387–1406, 1992.
- [59] Mark Huggett. The risk-free rate in heterogeneous-agent incomplete-insurance economies. *Journal of Economic Dynamics and Control*, 17(5-6):953–969, 1993.
- [60] K Jänich. *Linear Algebra*. Springer Undergraduate Texts in Mathematics and Technology. Springer, 1994.
- [61] 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.
- [62] Boyan Jovanovic. Firm-specific capital and turnover. *Journal of Political Economy*, 87(6):1246–1260, 1979.
- [63] K L Judd. Cournot versus bertrand: A dynamic resolution. Technical report, Hoover Institution, Stanford University, 1990.
- [64] Takashi Kamihigashi. Elementary results on solutions to the bellman equation of dynamic programming: existence, uniqueness, and convergence. Technical report, Kobe University, 2012.
- [65] Illenin Kondo, Logan T Lewis, and Andrea Stella. On the us firm and establishment size distributions. Technical report, SSRN, 2018.
- [66] Tjalling C. Koopmans. On the concept of optimal economic growth. In Tjalling C. Koopmans, editor, *The Economic Approach to Development Planning*, page 225–287. Chicago, 1965.
- [67] David M. Kreps. *Notes on the Theory of Choice*. Westview Press, Boulder, Colorado, 1988.
- [68] Moritz Kuhn. Recursive Equilibria In An Aiyagari-Style Economy With Permanent Income Shocks. *International Economic Review*, 54:807–835, 2013.
- [69] Martin Lettau and Sydney Ludvigson. Consumption, Aggregate Wealth, and Expected Stock Returns. *Journal of Finance*, 56(3):815–849, 06 2001.

- [70] 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.
- [71] 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.
- [72] L Ljungqvist and T J Sargent. *Recursive Macroeconomic Theory*. MIT Press, 4 edition, 2018.
- [73] Robert E Lucas, Jr. Asset prices in an exchange economy. *Econometrica: Journal of the Econometric Society*, 46(6):1429–1445, 1978.
- [74] Robert E Lucas, Jr. and Edward C Prescott. Investment under uncertainty. *Econometrica: Journal of the Econometric Society*, pages 659–681, 1971.
- [75] Qingyin Ma, John Stachurski, and Alexis Akira Toda. The income fluctuation problem and the evolution of wealth. *Journal of Economic Theory*, 187:105003, 2020.
- [76] Benoit Mandelbrot. The variation of certain speculative prices. *The Journal of Business*, 36(4):394–419, 1963.
- [77] 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.
- [78] V Filipe Martins-da Rocha and Yiannis Vailakis. Existence and Uniqueness of a Fixed Point for Local Contractions. *Econometrica*, 78(3):1127–1141, 2010.
- [79] A Mas-Colell, M D Whinston, and J R Green. *Microeconomic Theory*, volume 1. Oxford University Press, 1995.
- [80] J J McCall. Economics of Information and Job Search. *The Quarterly Journal of Economics*, 84(1):113–126, 1970.
- [81] S P Meyn and R L Tweedie. *Markov Chains and Stochastic Stability*. Cambridge University Press, 2009.
- [82] 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.
- [83] Derek Neal. The Complexity of Job Mobility among Young Men. *Journal of Labor Economics*, 17(2):237–261, 1999.
- [84] Y Nishiyama, S Osada, and K Morimune. Estimation and testing for rank size rule regression under pareto distribution. In *Proceedings of the International Environmental Modelling and Software Society iEMSs 2004 International Conference*. Citeseer, 2004.
- [85] Jenő Pál and John Stachurski. Fitted value function iteration with probability one contractions. *Journal of Economic Dynamics and Control*, 37(1):251–264, 2013.
- [86] Jonathan A Parker. The Reaction of Household Consumption to Predictable Changes in Social Security Taxes. *American Economic Review*, 89(4):959–973, 1999.
- [87] 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.

- [88] Svetlozar Todorov Rachev. *Handbook of heavy tailed distributions in finance: Handbooks in finance*, volume 1. Elsevier, 2003.
- [89] Kevin L Reffett. Production-based asset pricing in monetary economies with transactions costs. *Economica*, pages 427–443, 1996.
- [90] Michael Reiter. Solving heterogeneous-agent models by projection and perturbation. *Journal of Economic Dynamics and Control*, 33(3):649–665, 2009.
- [91] Hernán D Rozenfeld, Diego Rybski, Xavier Gabaix, and Hernán A Makse. The area and population of cities: New insights from a different perspective on cities. *American Economic Review*, 101(5):2205–25, 2011.
- [92] Stephen P Ryan. The costs of environmental regulation in a concentrated industry. *Econometrica*, 80(3):1019–1061, 2012.
- [93] Paul A. Samuelson. Interactions between the multiplier analysis and the principle of acceleration. *Review of Economic Studies*, 21(2):75–78, 1939.
- [94] Thomas J Sargent. The Demand for Money During Hyperinflations under Rational Expectations: I. *International Economic Review*, 18(1):59–82, February 1977.
- [95] Thomas J Sargent. *Macroeconomic Theory*. Academic Press, New York, 2nd edition, 1987.
- [96] Jack Schechtman and Vera L S Escudero. Some results on an income fluctuation problem. *Journal of Economic Theory*, 16(2):151–166, 1977.
- [97] Jose A. Scheinkman. *Speculation, Trading, and Bubbles*. Columbia University Press, New York, 2014.
- [98] Thomas C Schelling. Models of Segregation. *American Economic Review*, 59(2):488–493, 1969.
- [99] Christian Schlüter and Mark Trede. Size distributions reconsidered. *Econometric Reviews*, 38(6):695–710, 2019.
- [100] John Stachurski. Continuous state dynamic programming via nonexpansive approximation. *Computational Economics*, 31(2):141–160, 2008.
- [101] 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.
- [102] N L Stokey, R E Lucas, and E C Prescott. *Recursive Methods in Economic Dynamics*. Harvard University Press, 1989.
- [103] 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.
- [104] R K Sundaram. *A First Course in Optimization Theory*. Cambridge University Press, 1996.
- [105] George Tauchen. Finite state markov-chain approximations to univariate and vector autoregressions. *Economics Letters*, 20(2):177–181, 1986.
- [106] Daniel Treisman. Russia’s billionaires. *The American Economic Review*, 106(5):236–241, 2016.

- [107] Ngo Van Long. Dynamic games in the economics of natural resources: a survey. *Dynamic Games and Applications*, 1(1):115–148, 2011.
- [108] Pareto Vilfredo. Cours d'économie politique. *Rouge, Lausanne*, 2, 1896.
- [109] Abraham Wald. *Sequential Analysis*. John Wiley and Sons, New York, 1947.
- [110] Charles Whiteman. *Linear Rational Expectations Models: A User's Guide*. University of Minnesota Press, Minneapolis, Minnesota, 1983.
- [111] Jeffrey M Wooldridge. *Introductory econometrics: A modern approach*. Nelson Education, 2015.
- [112] G Alastair Young and Richard L Smith. *Essentials of statistical inference*. Cambridge University Press, 2005.