Quantitative Economics with Python

Thomas J. Sargent & John Stachurski

Jul 06, 2022

CONTENTS

This website presents a set of lectures on quantitative economic modeling, designed and written by Thomas J. Sargent and John Stachurski.

For an overview of the series, see this page

Previous website

While this new site will receive all future updates, you may still view the old site here for the next month.

Part I

Tools and Techniques

CHAPTER

ONE

GEOMETRIC SERIES FOR ELEMENTARY ECONOMICS

Contents

1.1 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:

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
import sympy as sym
from sympy import init_printing, latex
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D
```

1.2 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.

1.2.1 Infinite Geometric Series

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

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

Where $\cdots$ means that the series continues without end.

The key formula is

$$1 + c + c^2 + c^3 + \cdots = \frac{1}{1 - c} \tag{1.1}$$

To prove key formula (1.1), multiply both sides by $(1 - c)$ and verify that if $c \in (-1, 1)$, then the outcome is the equation $1 = 1$.

1.2.2 Finite Geometric Series

The second series that interests us is the finite geometric series

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

where T is a positive integer.

The key formula here is

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

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.

1.3 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 (1.1) is at the heart of the classic model of the money creation process – one that leads us to the celebrated **money multiplier**.

1.3.1 A Simple Model

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

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

$$L_i + R_i = D_i \tag{1.2}$$

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

$$R_i = r D_i \tag{1.3}$$

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

$$D_{i+1} = L_i \tag{1.4}$$

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

$$D_0 \text{ is given exogenously}$$

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 (1.2) and (1.3) tells us that

$$L_i = (1 - r) D_i \tag{1.5}$$

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

Combining equation (1.5) with equation (1.4) tells us that

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

which implies that

$$D_i = (1-r)^i D_0 \ \text{ for } i \geq 0 \tag{1.6}$$

Equation (1.6) expresses D_i as the i th term in the product of D_0 and the geometric series

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

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

$$\sum_{i=0}^{\infty}(1-r)^i D_0 = \frac{D_0}{1-(1-r)} = \frac{D_0}{r} \tag{1.7}$$

1.3.2 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 (1.7) 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 $D_0 = \sum_{i=0}^{\infty} R_i$.

1.4 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

1.4.1 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.

1.4.2 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

$$y_1 = c_1 + i = by_0 + i = (1 + b)i$$

and

$$y_2 = c_2 + i = by_1 + i = (1 + b + b^2)i$$

and more generally

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

or

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

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

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

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

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

at times $0, 1, 2, \ldots$.

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.

1.5 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 $t = 0, 1, 2, \ldots$ 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 $t = 0, 1, 2, \ldots$.

Two important geometric sequences are

$$1, R, R^2, \cdots \tag{1.8}$$

and

$$1, R^{-1}, R^{-2}, \cdots \tag{1.9}$$

Sequence (1.8) tells us how dollar values of an investment **accumulate** through time.

Sequence (1.9) tells us how to **discount** future dollars to get their values in terms of today's dollars.

1.5.1 Accumulation

Geometric sequence (1.8) 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 time1
- 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

$$x, xR, xR^2, \cdots$$

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

1.5.2 Discounting

Geometric sequence (1.9) 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).

1.5.3 Application to Asset Pricing

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

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

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,

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

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 $t = 1, 2, \ldots, T$, 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:

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

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.

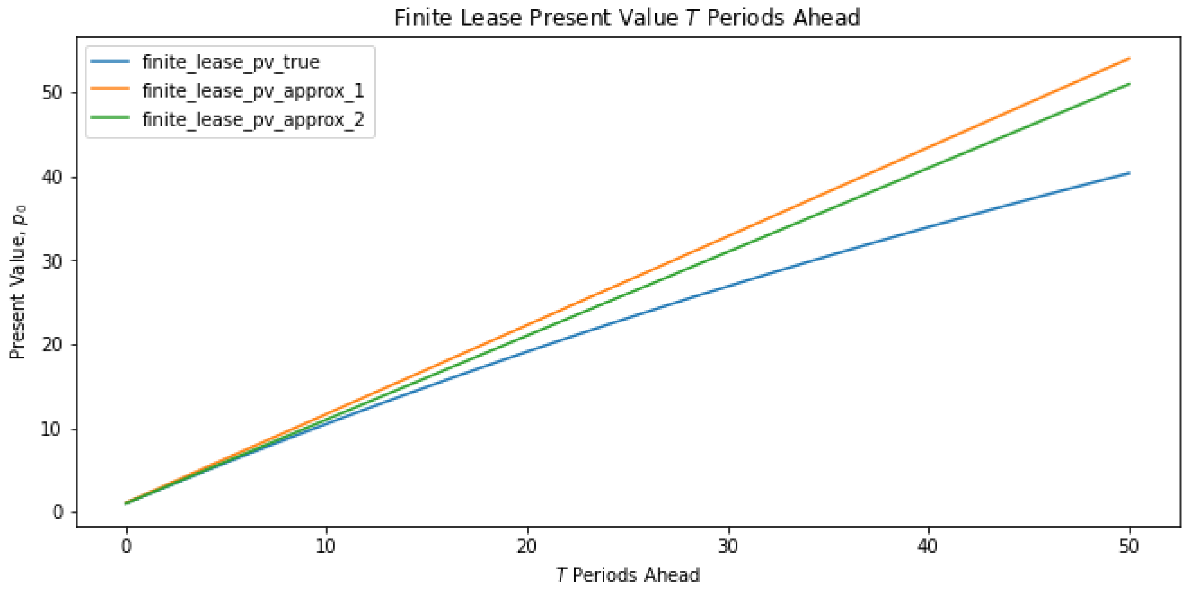
First we plot the true finite stream present-value after computing it below

```python
# 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

```
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.

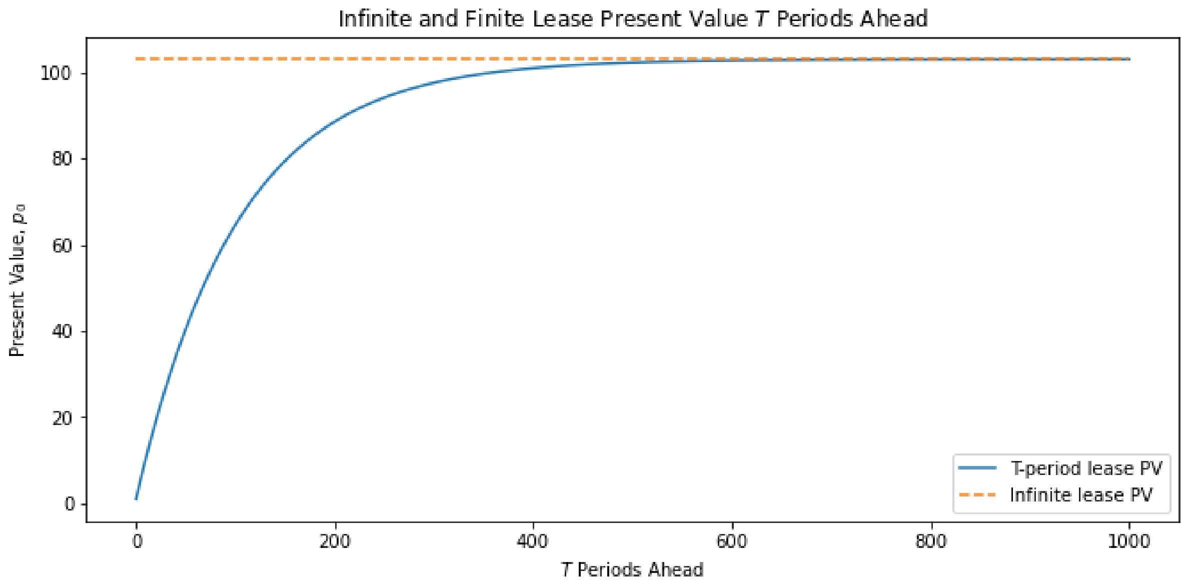
Next we compare the infinite and finite duration lease present values over different lease lengths $T$.

```
# 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')
```

(continues on next page)

(continued from previous page)

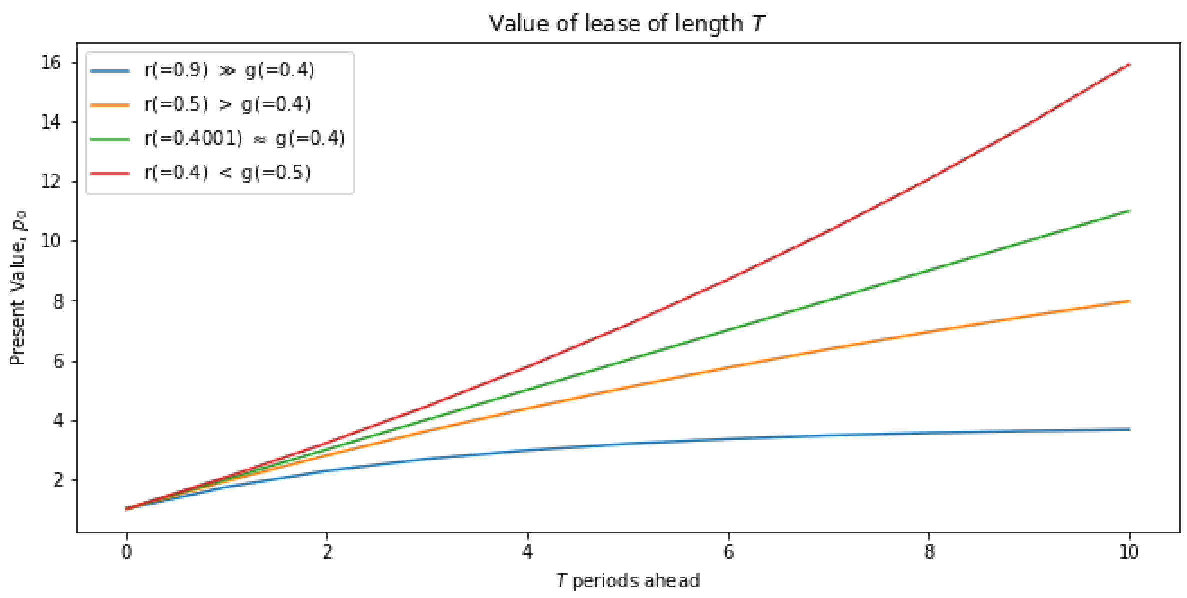
```
f_1 = finite_lease_pv_true(T, g, r, x_0)
f_2 = np.full(T_max+1, infinite_lease(g, r, x_0))
ax.plot(T, f_1, label='T-period lease PV')
ax.plot(T, f_2, '--', label='Infinite lease PV')
ax.set_xlabel('$T$ Periods Ahead')
ax.set_ylabel('Present Value, $p_0$')
ax.legend()
plt.show()
```



The graph above shows how as duration $T \to +\infty$, 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

```
# 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()
```



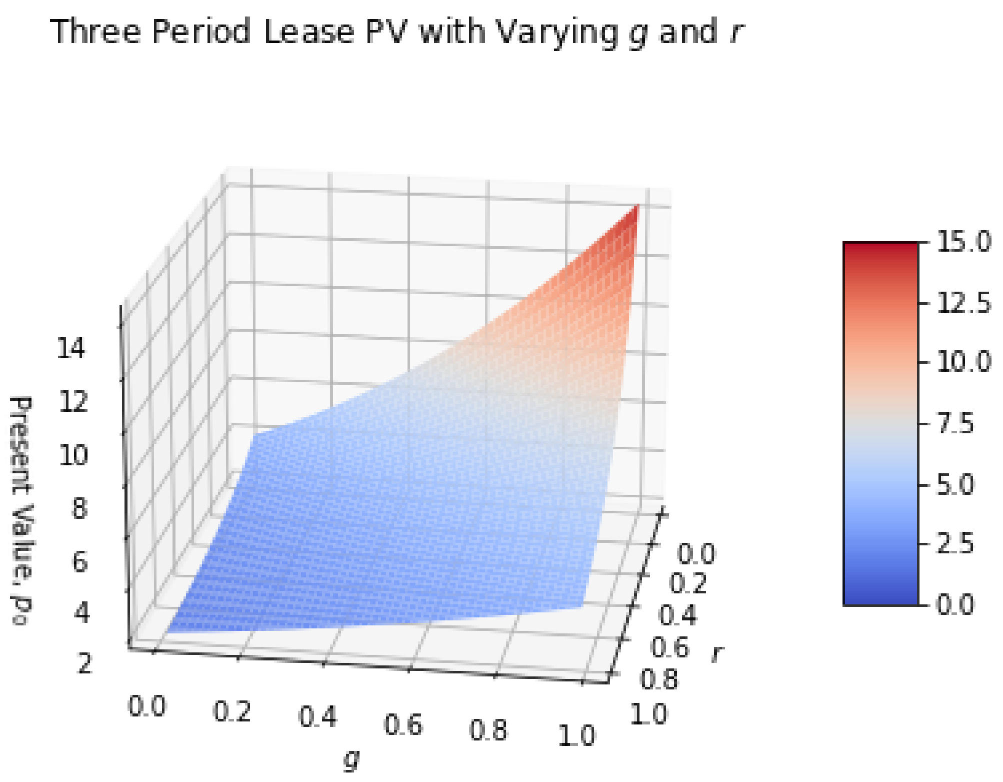
This graph gives a big hint for why the condition $r > g$ is necessary if a lease of length $T = +\infty$ is to have finite value.

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!

```
# 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()
```

```
/tmp/ipykernel_11195/2419678664.py:4: MatplotlibDeprecationWarning: Calling gca()␣
↪with keyword arguments was deprecated in Matplotlib 3.4. Starting two minor␣
↪releases later, gca() will take no keyword arguments. The gca() function should␣
↪only be used to get the current axes, or if no axes exist, create new axes with␣
↪default keyword arguments. To create a new axes with non-default arguments, use␣
↪plt.axes() or plt.subplot().
  ax = fig.gca(projection='3d')
```



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

```
# 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(use_latex='mathjax')
print('Our formula is:')
p0
```

```
Our formula is:
```

$$\frac{x_0}{-\frac{g+1}{r+1}+1}$$

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

```
dp0 / dg is:
```

$$\frac{x_0}{(r+1)\left(-\frac{g+1}{r+1}+1\right)^2}$$

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

```
dp0 / dr is:
```

$$-\frac{x_0\,(g+1)}{(r+1)^2\left(-\frac{g+1}{r+1}+1\right)^2}$$

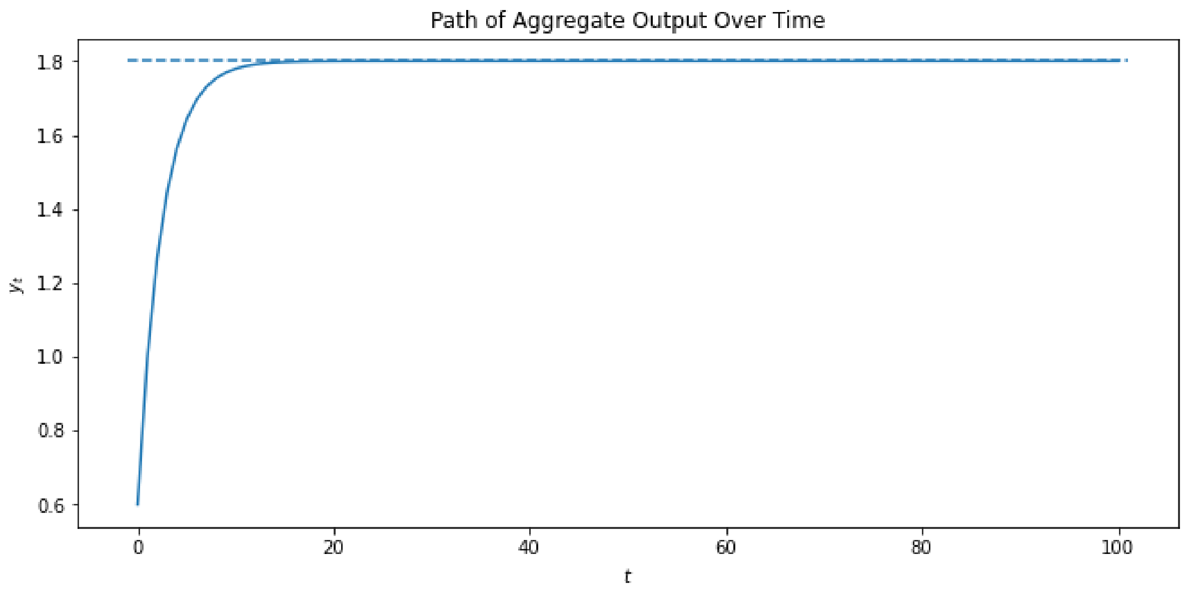
We can see that for $\frac{\partial p_0}{\partial r} < 0$ as long as $r > g$, $r > 0$ and $g > 0$ and x_0 is positive, so $\frac{\partial p_0}{\partial r}$ will always be negative.

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.

1.6 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.

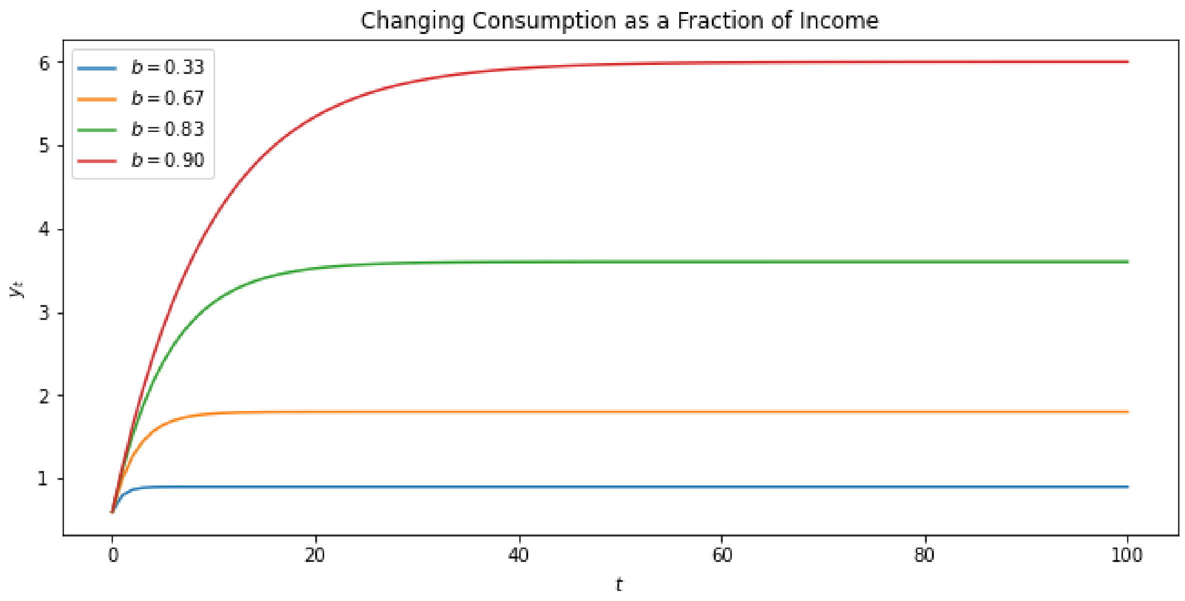
```
# Function that calculates a path of y
def calculate_y(i, b, g, T, y_init):
    y = np.zeros(T+1)
    y[0] = i + b * y_init + g
    for t in range(1, T+1):
        y[t] = b * y[t-1] + i + g
    return y

# 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

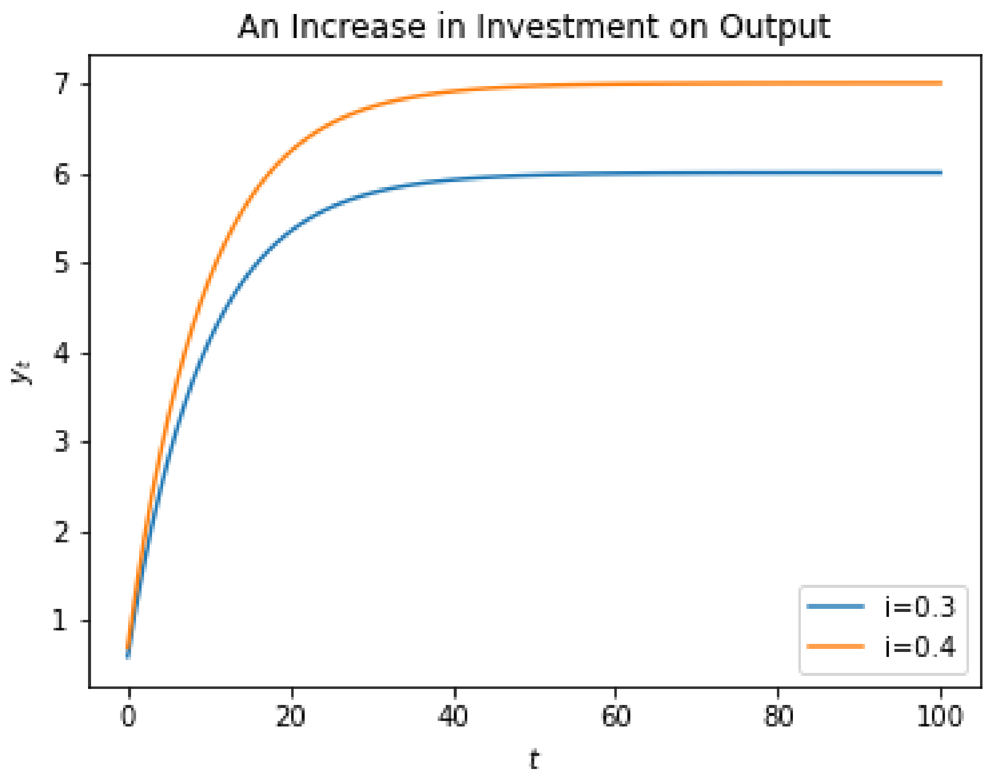
```python
bs = (1/3, 2/3, 5/6, 0.9)

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()
```

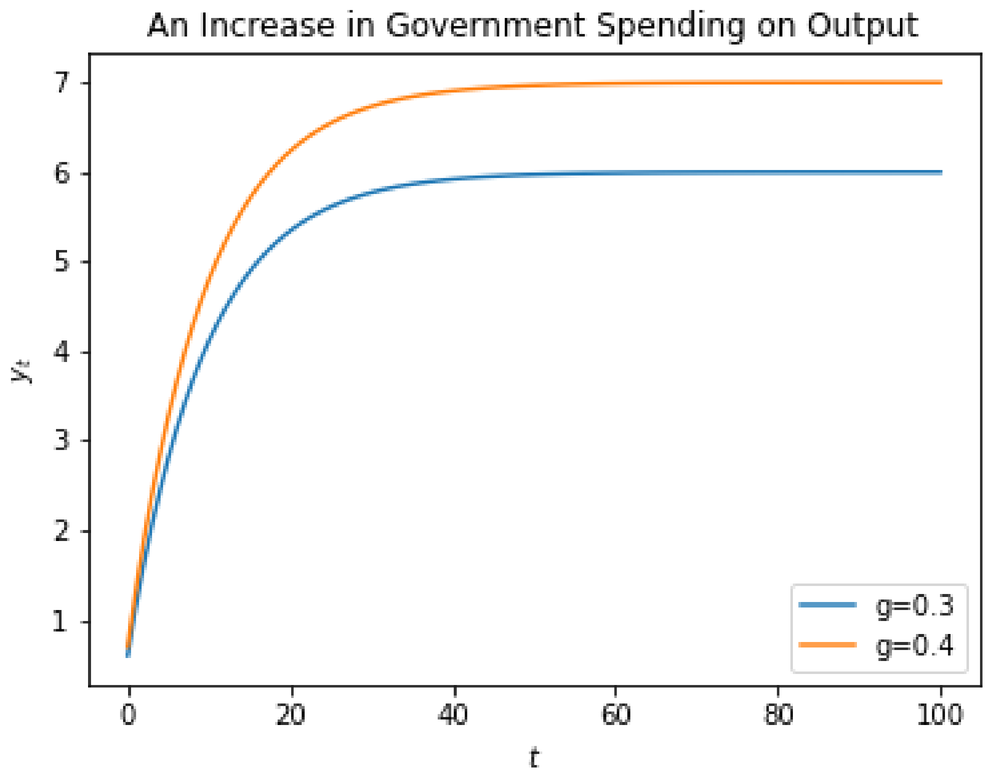


Increasing the marginal propensity to consume b increases the path of output over time.

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

```
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

TWO

MODELING COVID 19

Contents

2.1 Overview

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

See, in particular

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

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

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

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

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

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

We will use the following standard imports:

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from numpy import exp
```

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

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

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

2.2 The SIR Model

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

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

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

Comments:

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

2.2.1 Time Path

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

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

The interest is primarily in

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

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

$$\begin{aligned} \dot{s}(t) &= -\beta(t)\, s(t)\, i(t) \\ \dot{e}(t) &= \beta(t)\, s(t)\, i(t) - \sigma e(t) \\ \dot{i}(t) &= \sigma e(t) - \gamma i(t) \end{aligned} \tag{2.1}$$

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 (2.1) can be written in vector form as

$$\dot{x} = F(x, t), \qquad x := (s, e, i) \tag{2.2}$$

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

2.2.2 Parameters

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

As in Atkeson's note, we set

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

The transmission rate is modeled as

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

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

2.3 Implementation

First we set the population size to match the US.

```
pop_size = 3.3e8
```

Next we fix parameters as described above.

```
γ = 1 / 18
σ = 1 / 5.2
```

Now we construct a function that represents F in (2.2)

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

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

    """
    s, e, i = x

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

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

    return ds, de, di
```

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

The initial conditions are set to

```python
# 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

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

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

```python
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
```

2.4 Experiments

Let's run some experiments using this code.

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

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

2.4.1 Experiment 1: Constant R0 Case

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

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

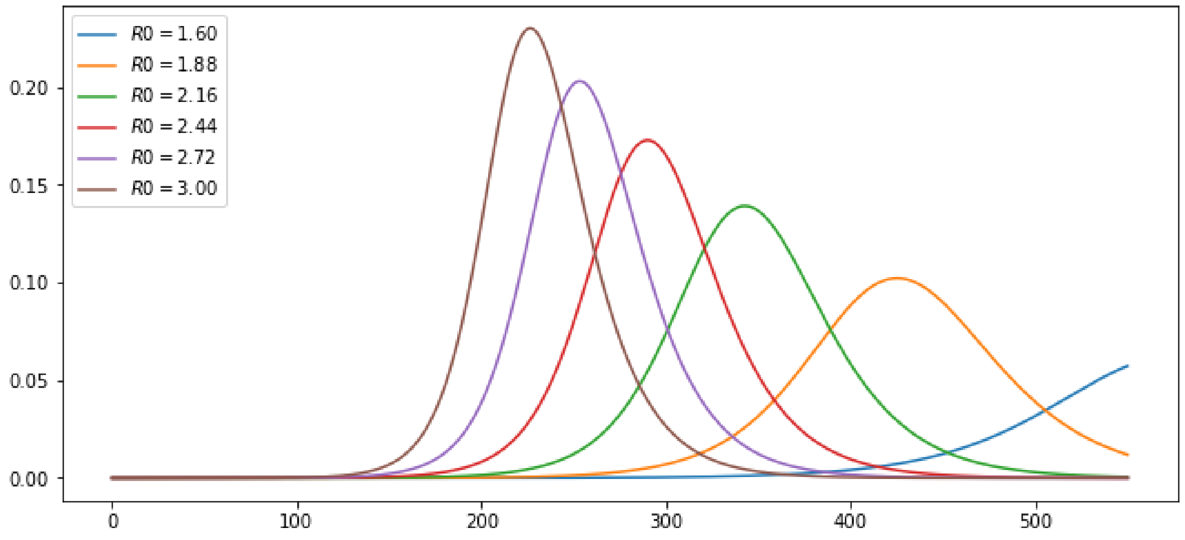
```python
R0_vals = np.linspace(1.6, 3.0, 6)
labels = [f'$R0 = {r:.2f}$' for r in R0_vals]
i_paths, c_paths = [], []

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.

```python
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.

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

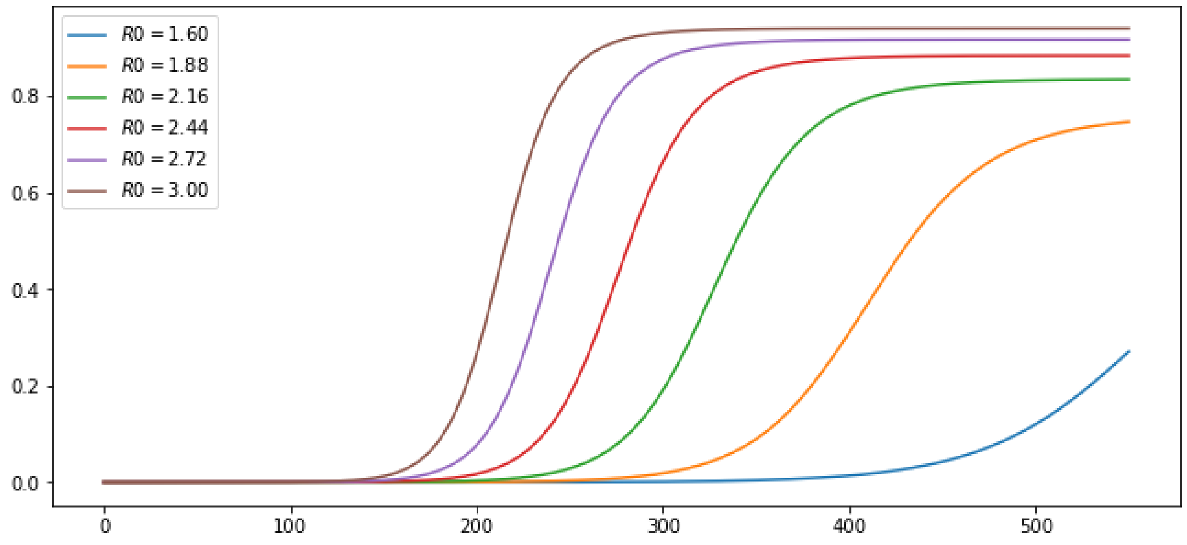


As expected, lower effective transmission rates defer the peak of infections.

They also lead to a lower peak in current cases.

Here are cumulative cases, as a fraction of population:

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



2.4.2 Experiment 2: Changing Mitigation

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

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

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

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

This is due to progressive adoption of stricter mitigation measures.

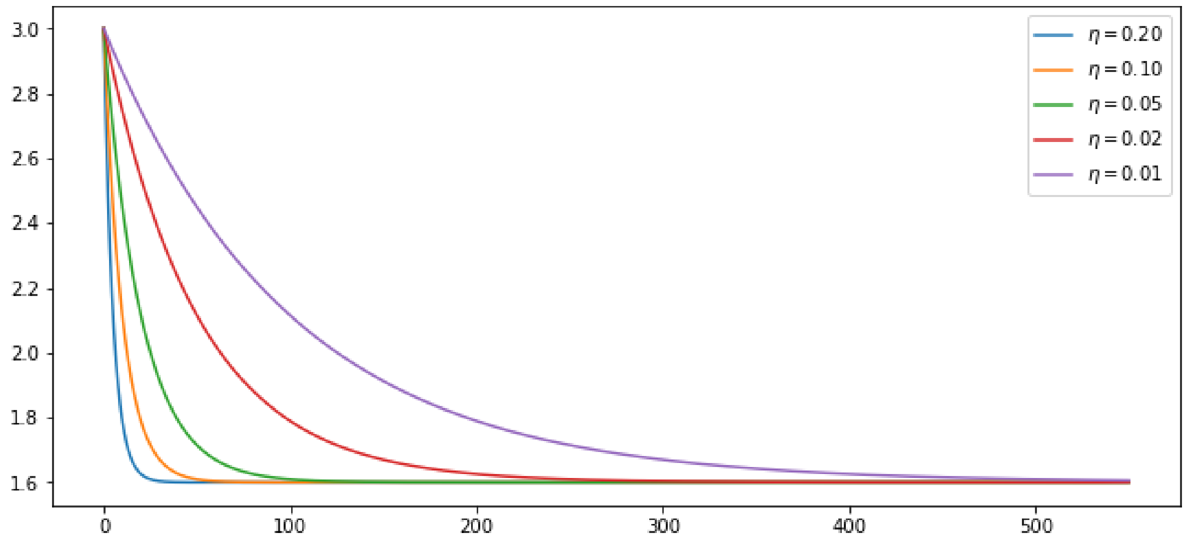
The parameter `η` controls the rate, or the speed at which restrictions are imposed.

We consider several different rates:

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

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

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

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

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

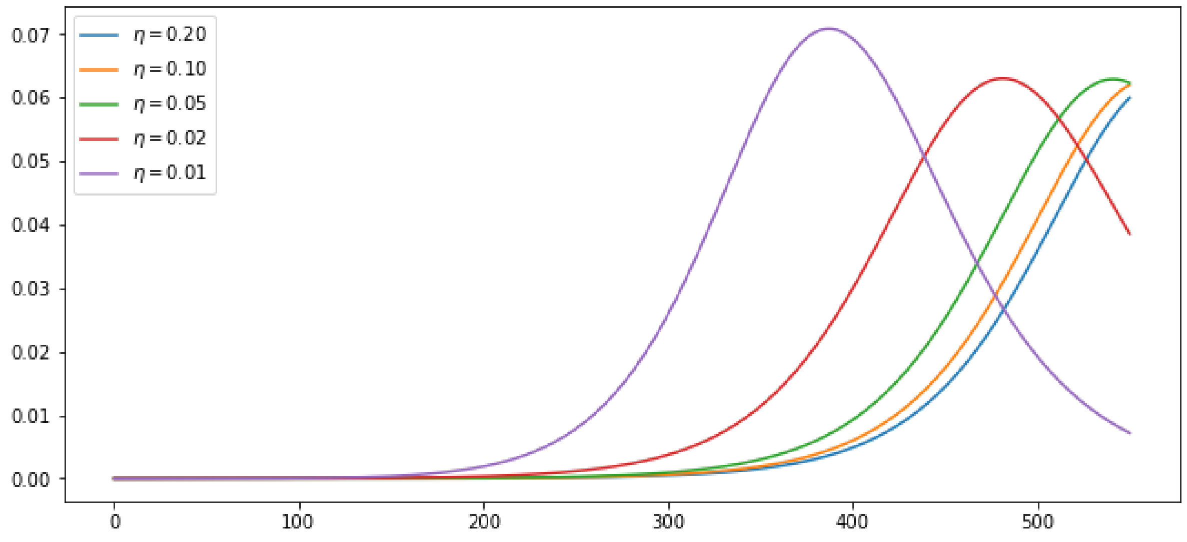


Let's calculate the time path of infected people:

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

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

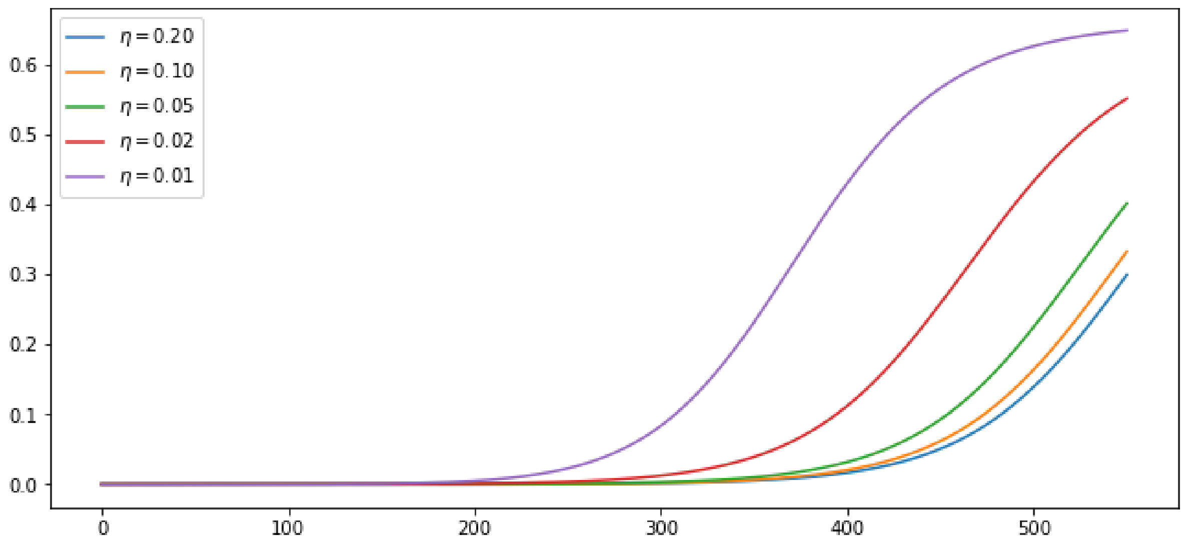
These are current cases under the different scenarios:

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



Here are cumulative cases, as a fraction of population:

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



2.5 Ending Lockdown

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

Consider these two mitigation scenarios:

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

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

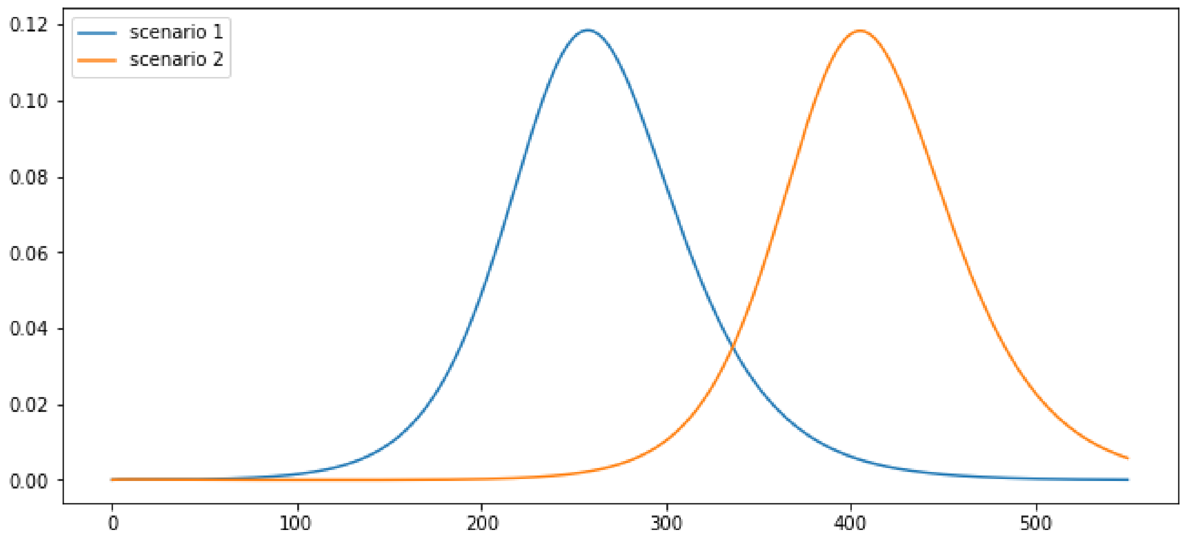
```python
# initial conditions
i_0 = 25_000 / pop_size
e_0 = 75_000 / pop_size
s_0 = 1 - i_0 - e_0
x_0 = s_0, e_0, i_0
```

Let's calculate the paths:

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

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

i_paths, c_paths = [], []

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

Here is the number of active infections:

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



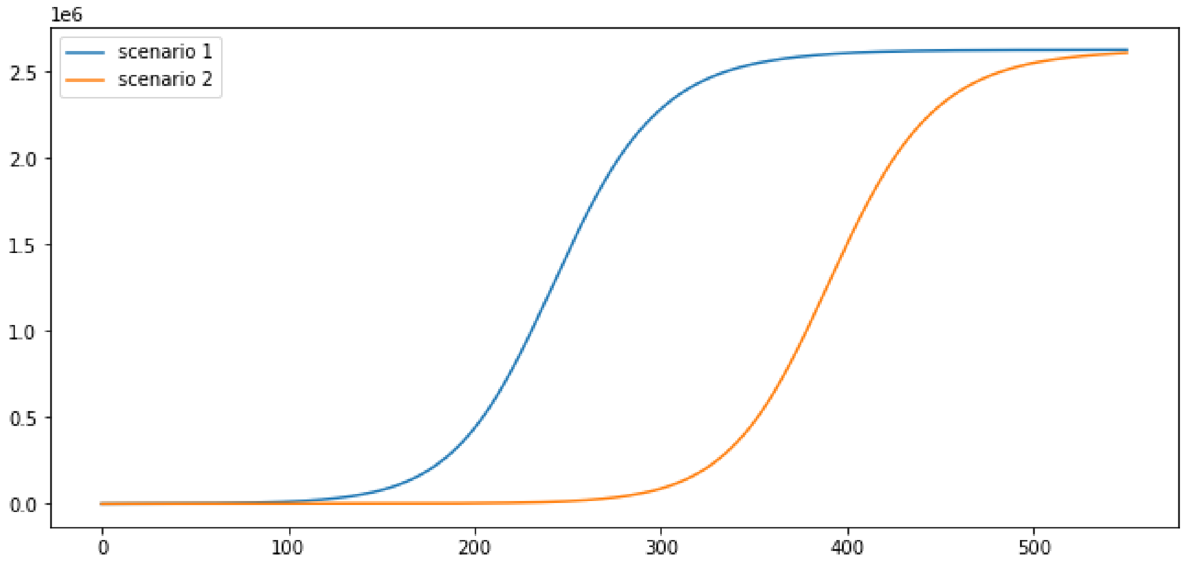
What kind of mortality can we expect under these scenarios?

Suppose that 1% of cases result in death

```
ν = 0.01
```

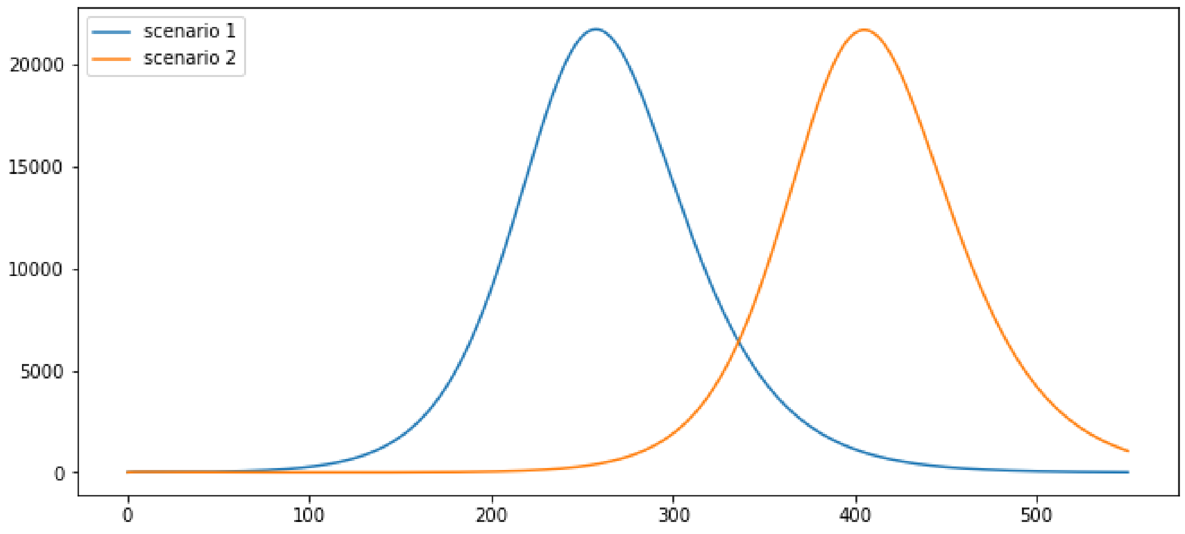
This is the cumulative number of deaths:

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



This is the daily death rate:

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



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

CHAPTER

THREE

LINEAR ALGEBRA

Contents

3.1 Overview

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

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

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

or, more generally,

$$\begin{aligned} y_1 &= a_{11}x_1 + a_{12}x_2 + \cdots + a_{1k}x_k \\ &\vdots \\ y_n &= a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nk}x_k \end{aligned} \tag{3.1}$$

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

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

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

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

These are the kinds of topics addressed by linear algebra.

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

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

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

Let's start with some imports:

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D
from scipy.interpolate import interp2d
from scipy.linalg import inv, solve, det, eig
```

3.2 Vectors

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

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

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

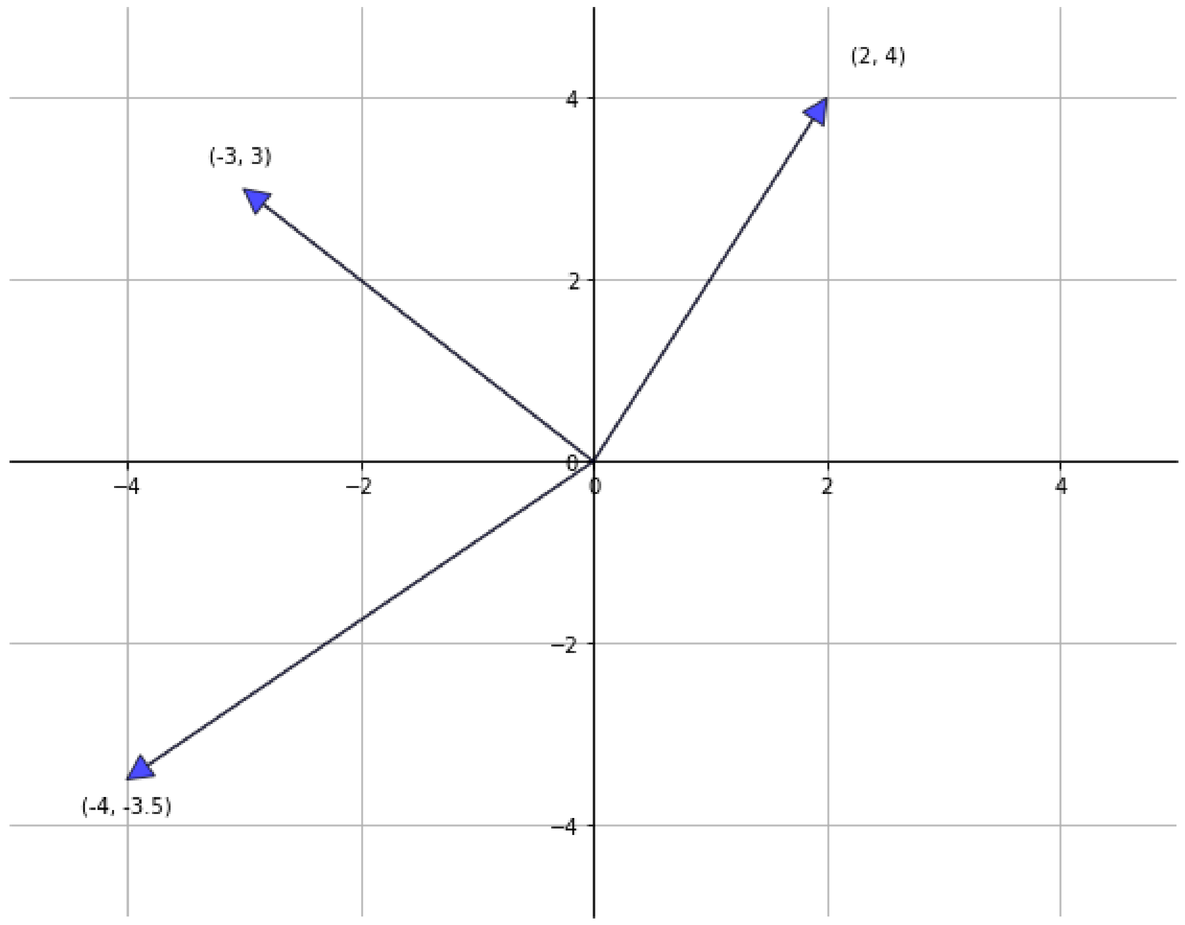
The set of all n-vectors is denoted by $\mathbb{R}^n$.

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

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

The following figure represents three vectors in this manner

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

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



3.2.1 Vector Operations

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

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

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

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

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

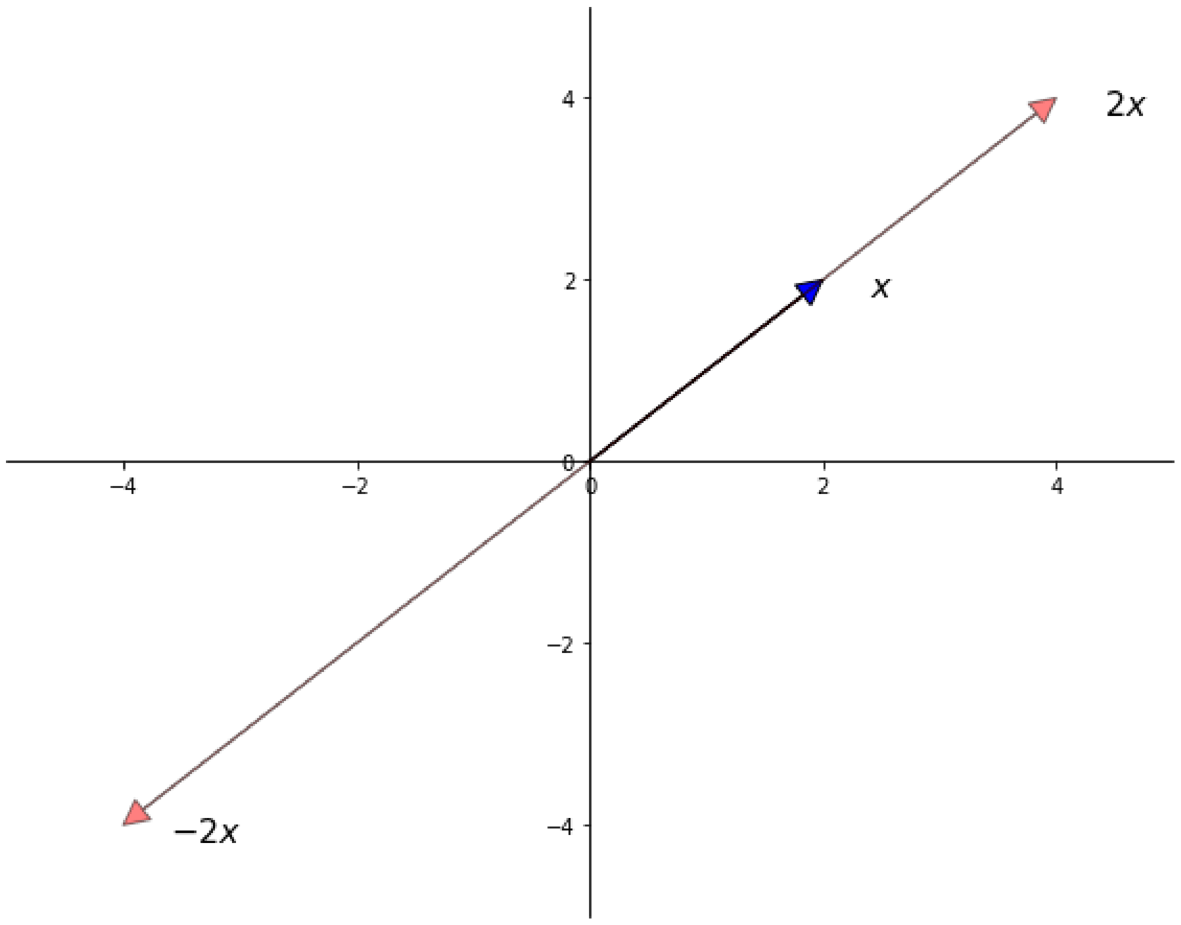
Scalar multiplication is illustrated in the next figure

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

(continues on next page)

(continued from previous page)

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

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


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

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



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

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

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

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

```
4 * x
```

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

3.2.2 Inner Product and Norm

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

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

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

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

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

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

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

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

```
12.0
```

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

```
1.7320508075688772
```

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

```
1.7320508075688772
```

3.2.3 Span

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

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

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

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

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

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

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

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

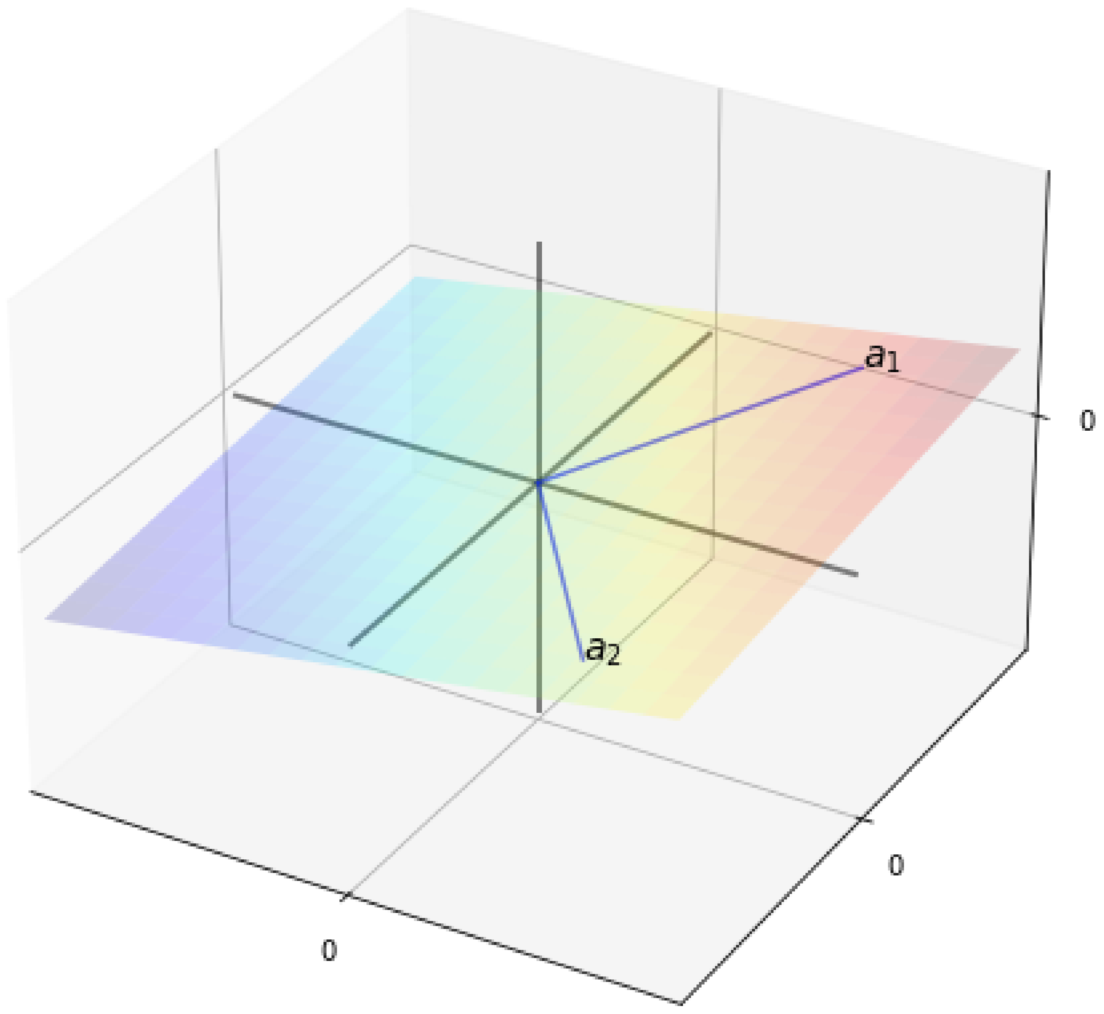
```
fig = plt.figure(figsize=(10, 8))
ax = fig.gca(projection='3d')

x_min, x_max = -5, 5
```

(continues on next page)

(continued from previous page)

```python
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()
```

```
/tmp/ipykernel_15617/266575435.py:2: MatplotlibDeprecationWarning: Calling gca()␣
↪with keyword arguments was deprecated in Matplotlib 3.4. Starting two minor␣
↪releases later, gca() will take no keyword arguments. The gca() function should␣
↪only be used to get the current axes, or if no axes exist, create new axes with␣
↪default keyword arguments. To create a new axes with non-default arguments, use␣
↪plt.axes() or plt.subplot().
  ax = fig.gca(projection='3d')
```



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 $\mathbb{R}^3$.

3.2.4 Linear Independence

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

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

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

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

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

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

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

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

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

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

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

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

3.2.5 Unique Representations

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

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

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

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

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

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

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

3.3 Matrices

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

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

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

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

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

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

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

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

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

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

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

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

3.3.1 Matrix Operations

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

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

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

and

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

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

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

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

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

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

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

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

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

$$Ax = \begin{bmatrix} a_{11} & \cdots & a_{1k} \\ \vdots & \vdots & \vdots \\ a_{n1} & \cdots & a_{nk} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_k \end{bmatrix} := \begin{bmatrix} a_{11}x_1 + \cdots + a_{1k}x_k \\ \vdots \\ a_{n1}x_1 + \cdots + a_{nk}x_k \end{bmatrix} \tag{3.2}$$

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 $IA = A$.

3.3.2 Matrices in NumPy

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

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

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

type(A)
```

```
tuple
```

```
A = np.array(A)

type(A)
```

```
numpy.ndarray
```

```
A.shape
```

```
(2, 2)
```

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

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

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

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

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

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

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

```
A + B
```

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

To multiply matrices we use the `@` symbol.

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

See here for more discussion.

3.3.3 Matrices as Maps

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

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

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

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

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

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

3.4 Solving Systems of Equations

Recall again the system of equations (3.1).

If we compare (3.1) and (3.2), we see that (3.1) can now be written more conveniently as

$$y = Ax \tag{3.3}$$

The problem we face is to determine a vector $x \in \mathbb{R}^k$ that solves (3.3), 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?

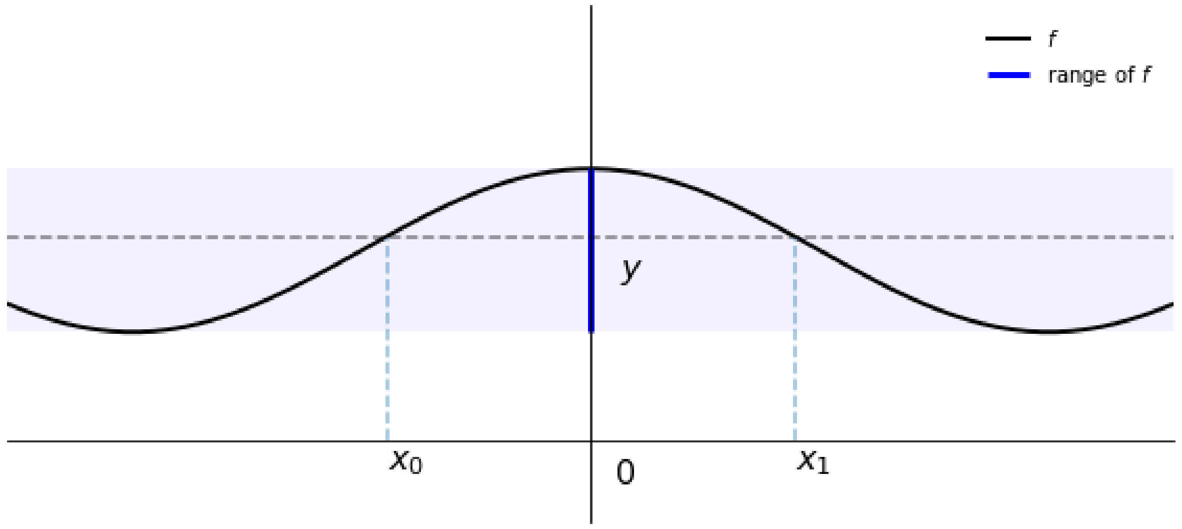
The answer to both these questions is negative, as the next figure shows

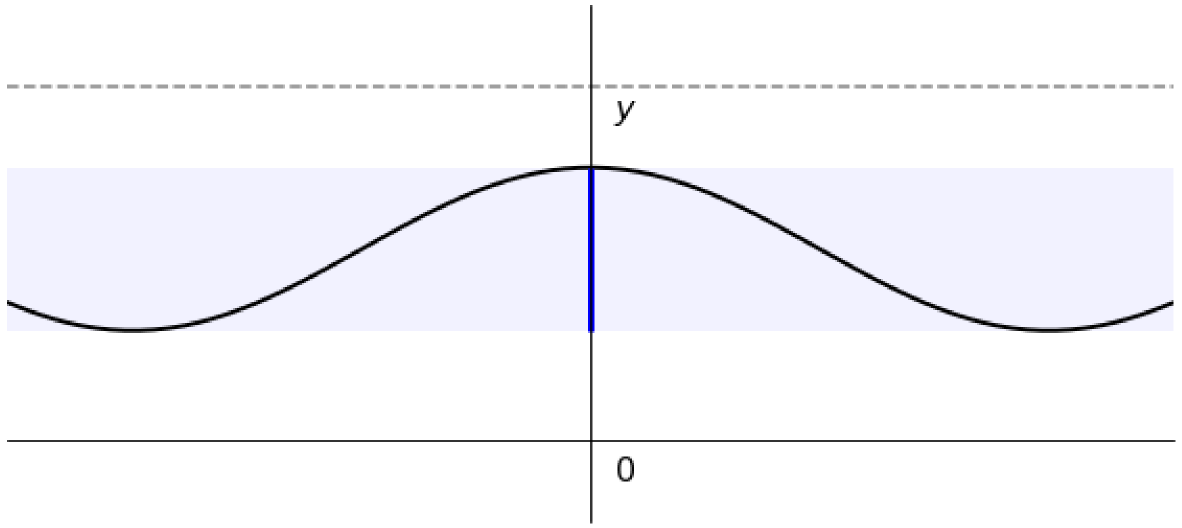
```python
def f(x):
    return 0.6 * np.cos(4 * x) + 1.4


xmin, xmax = -1, 1
x = np.linspace(xmin, xmax, 160)
```

(continues on next page)

(continued from previous page)

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

3.4.1 The Square Matrix Case

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

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

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

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

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

Moreover, the solution is unique.

In particular, the following are equivalent

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

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

Inverse Matrices

Can we give some sort of expression for the solution?

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

A similar expression is available in the matrix case.

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

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

This is the solution that we're looking for.

Determinants

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

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

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

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

3.4.2 More Rows than Columns

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.4.3 More Columns than Rows

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

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

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

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

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

(For example, use the canonical basis vectors)

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

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

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

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

In other words, uniqueness fails.

3.4.4 Linear Equations with SciPy

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

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

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

```
-2.0
```

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

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

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

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

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

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

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

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

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

3.5 Eigenvalues and Eigenvectors

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

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

$$Av = \lambda v$$

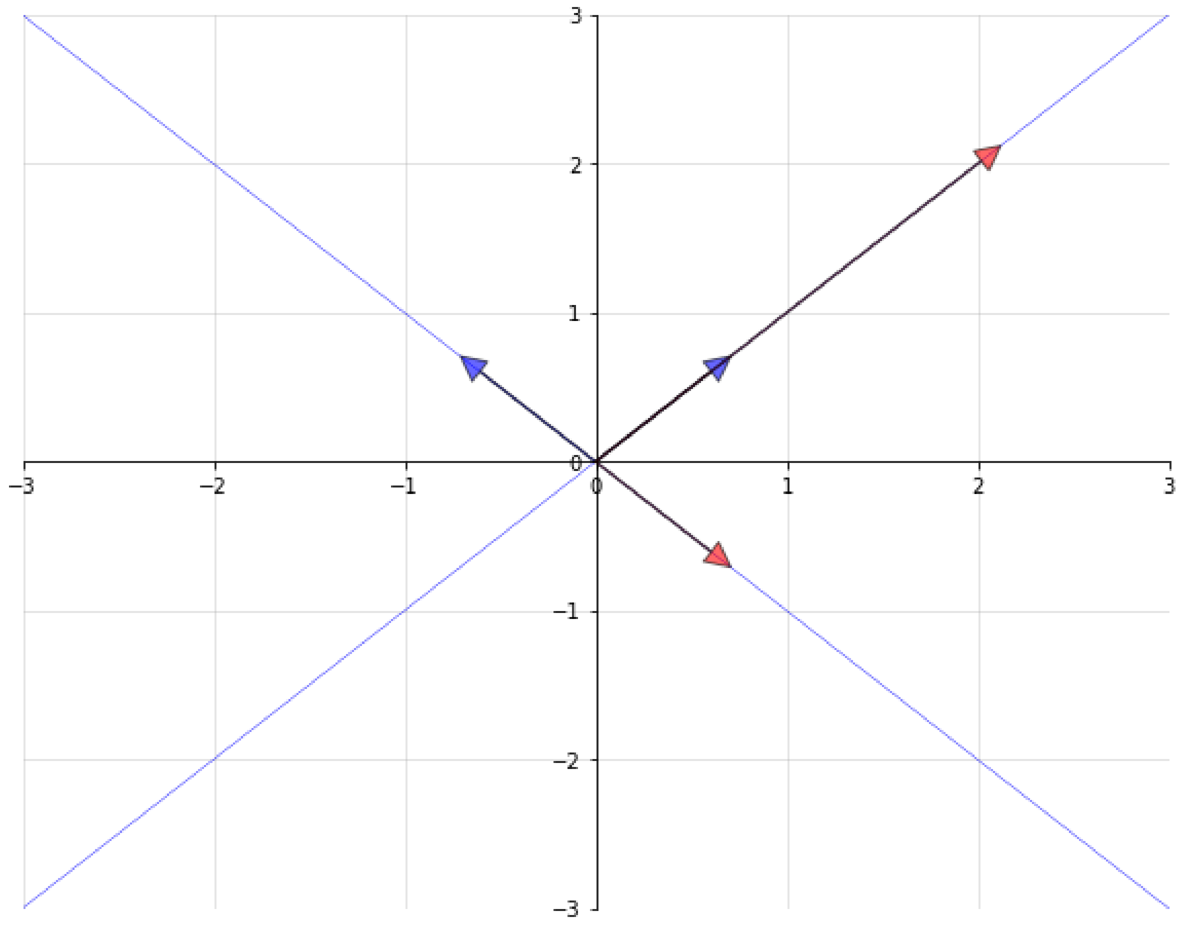
then we say that λ is an *eigenvalue* of A, and v is an *eigenvector*.

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

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

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

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

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

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

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

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

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

plt.show()
```



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

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

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

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

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

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

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

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

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

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

A = np.array(A)
```

(continues on next page)

(continued from previous page)

```
evals, evecs = eig(A)
evals
```

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

```
evecs
```

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

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

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

3.5.1 Generalized Eigenvalues

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

$$Av = \lambda Bv$$

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

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

3.6 Further Topics

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

3.6.1 Series Expansions

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

A generalization of this idea exists in the matrix setting.

Matrix Norms

Let A be a square matrix, and let

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

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

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

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

Neumann's Theorem

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

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

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

$$(I - A)^{-1} = \sum_{k=0}^{\infty} A^k \tag{3.4}$$

Spectral Radius

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

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

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

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

In which case (3.4) is valid.

3.6.2 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 $x \in \mathbb{R}^n \quad \{0\}$
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).

3.6.3 Differentiating Linear and Quadratic Forms

The following formulas are useful in many economic contexts. Let

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

Then

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

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

Exercise 3.7.1 below asks you to apply these formulas.

3.6.4 Further Reading

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

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

If you don't mind a slightly abstract approach, a nice intermediate-level text on linear algebra is [Janich94].

3.7 Exercises

Exercise 3.7.1

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

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

subject to the linear constraint

$$y = Ax + Bu$$

Here

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

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

One way to solve the problem is to form the Lagrangian

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

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

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

Show that these conditions imply that

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

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

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

3.8 Solutions

Solution to Exercise 3.7.1

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

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

To evaluate the function

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

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

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

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

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

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

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

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

This implies that

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

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

CHAPTER

FOUR

QR DECOMPOSITION

4.1 Overview

This lecture describes the QR decomposition and how it relates to

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

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

4.2 Matrix Factorization

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

A QR decomposition of a real matrix A takes the form

$$A = QR$$

where

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

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

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

4.3 Gram-Schmidt process

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

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

We'll deal with a rectangular matrix A later.

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

4.3.1 Gram-Schmidt process for square A

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

In particular, let

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

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

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

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

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

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

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

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

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

The Gram-Schmidt procedure continues iterating.

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

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

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

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

It can be verified that

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

Thus, we have constructed the decomposision

$$A = QR$$

where

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

and

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

4.3.2 A not square

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

Then a QR decomposition is

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

which implies that

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

4.4 Some Code

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

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

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

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

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

    for i in range(1, n):

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

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

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

    return Q, R
```

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

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

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

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

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

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

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

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

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

    return D

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

    D = diag_sign(Q)

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

    return Q, R
```

4.5 Example

Now let's do an example.

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

A
```

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

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

```
Q
```

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

```
R
```

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

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

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

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

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


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

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

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


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

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

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

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

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

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

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

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

4.6 Using QR Decomposition to Compute Eigenvalues

Now for a useful fact about the QR algorithm.

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

Here is the algorithm:

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

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

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

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

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

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

        A_new[:, :] = R @ Q

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

    eigvals = np.diag(A_new)

    return eigvals
```

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

Here goes

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

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

```
[0.3642660469068212, 0.49117719304268104, 1.1907905618674441]
```

Compare with the `scipy` package.

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

```
[0.3642660469071797, 0.49117719304232105, 1.1907905618674457]
```

4.7 QR and PCA

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

Here are some.

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

Let's verify conjecture 5 with some Python code.

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

```python
k = 5
n = 1000

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

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

```python
X.shape
```

```
(1000, 5)
```

Let's apply the QR decomposition to X'.

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

Check the shapes of Q and R.

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

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

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

```
RR = R @ R.T

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

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

```
XX = X.T @ X

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

Compare the eigenvalues which are on the diagnoals of Λ and $\hat{\Lambda}$.

```
□, □_hat
```

```
(array([1.46666089e+00, 1.56022587e+02, 5.03047121e+02, 1.01131984e+03,
        8.48616704e+03]),
 array([1.46666089e+00, 1.56022587e+02, 5.03047121e+02, 1.01131984e+03,
        8.48616704e+03]))
```

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

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

```
QP_tilde = Q @ P_tilde

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

```
4.385380947269368e-15
```

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

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

```
np.abs(QPΛPQ - XX).max()
```

```
5.4569682106375694e-12
```

CHAPTER

FIVE

COMPLEX NUMBERS AND TRIGONOMETRY

Contents

5.1 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) [Sam39] 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.

5.1.1 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,

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

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:

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from sympy import *
```

5.1.2 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 $\theta = \tan^{-1}(\sqrt{3}) = \frac{\pi}{3} = 60^o$.

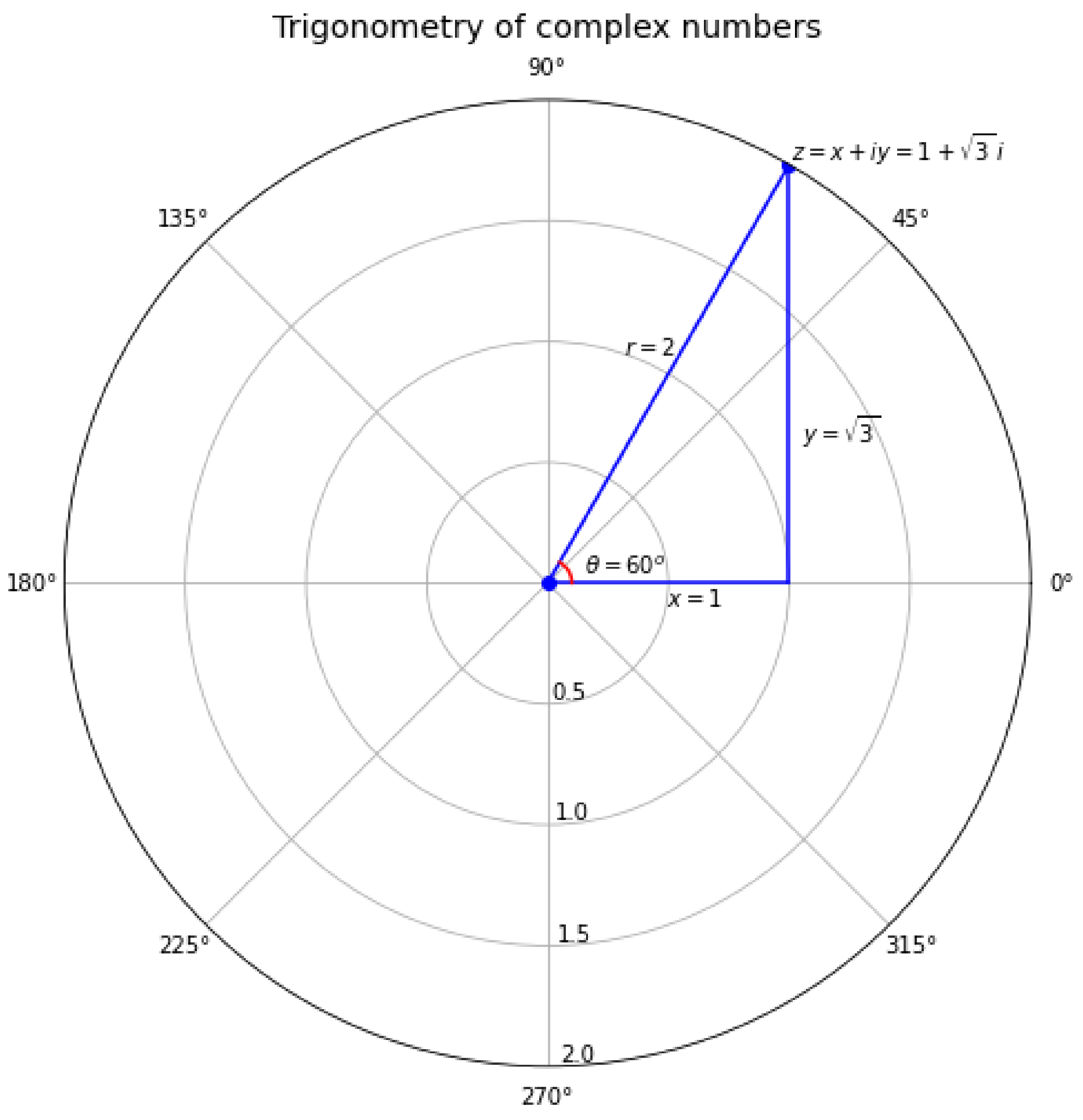
Let's use Python to plot the trigonometric form of the complex number $z = 1 + \sqrt{3}i$.

```
# Abbreviate useful values and functions
π = np.pi


# 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, θ), (0, r), marker='o', color='b')          # Plot r
ax.plot(np.zeros(x_range.shape), x_range, color='b')       # Plot x
ax.plot(θ_range, x / np.cos(θ_range), color='b')        # Plot y
```

(continues on next page)

(continued from previous page)

```
ax.plot(θ_range, np.full(θ_range.shape, 0.1), color='r')  # Plot θ

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

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



5.2 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

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

and compute.

5.3 Applications of de Moivre's Theorem

5.3.1 Example 1

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

We have

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

and thus

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

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

5.3.2 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 $n = 0, 1, 2, \ldots,$.

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

Thus,

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

5.3.3 Example 3

This example provides machinery that is at the heard of Samuelson's analysis of his multiplier-accelerator model [Sam39].

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_1.

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:

```
# 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 0
eq1 = Eq(x1/x0 - r * cos(ω+θ) / cos(ω), 0)
ω = nsolve(eq1, ω, 0)
ω = np.float(ω)
print(f'ω = {ω:1.3f}')

# Solve for p
eq2 = Eq(x0 - 2 * p * cos(ω), 0)
p = nsolve(eq2, p, 0)
p = np.float(p)
print(f'p = {p:1.3f}')
```

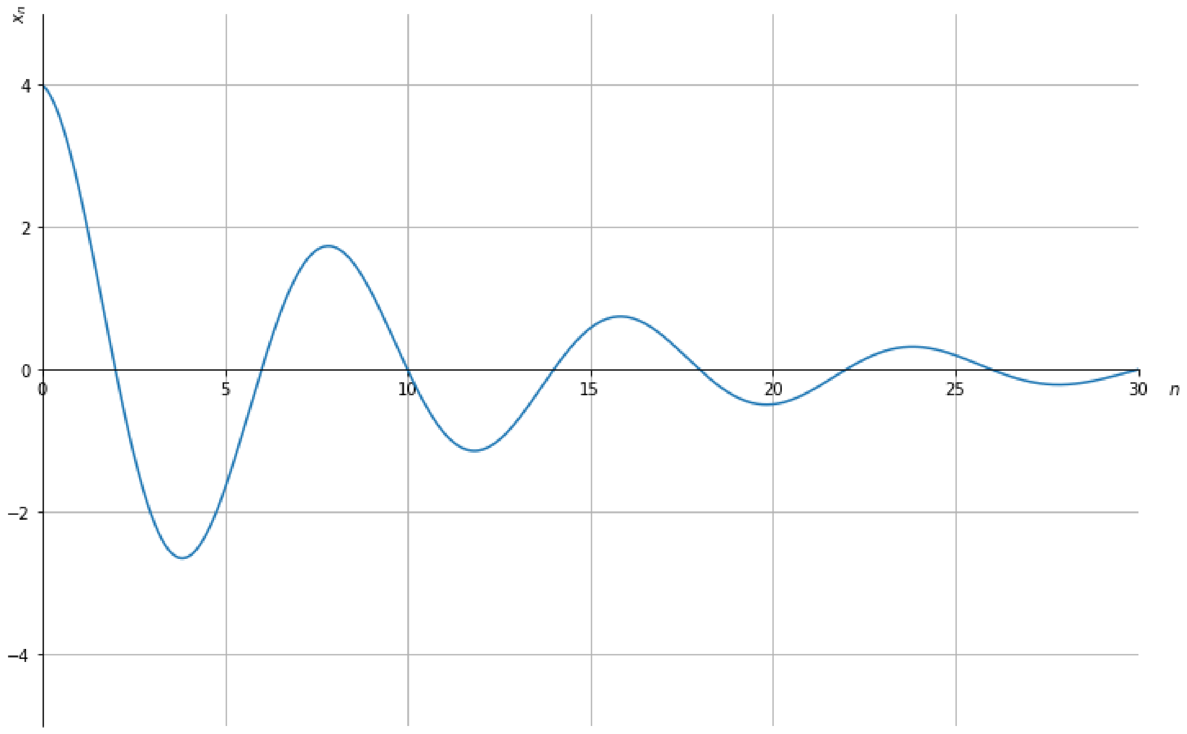
```
ω = 0.000
p = 2.000
```

```
/tmp/ipykernel_11335/2347233413.py:14: DeprecationWarning: `np.float` is a␣
↪deprecated alias for the builtin `float`. To silence this warning, use `float`␣
↪by itself. Doing this will not modify any behavior and is safe. If you␣
↪specifically wanted the numpy scalar type, use `np.float64` here.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/
↪release/1.20.0-notes.html#deprecations
  ω = np.float(ω)
/tmp/ipykernel_11335/2347233413.py:20: DeprecationWarning: `np.float` is a␣
↪deprecated alias for the builtin `float`. To silence this warning, use `float`␣
↪by itself. Doing this will not modify any behavior and is safe. If you␣
↪specifically wanted the numpy scalar type, use `np.float64` here.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/
↪release/1.20.0-notes.html#deprecations
  p = np.float(p)
```

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.

```python
# 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(ω + n * θ)

# 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()
```



5.3.4 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:

```python
# 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)
```

5.3.5 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:

```python
# 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(ω)sin(ω) is:
```

$\sin^2(\omega)/2$

```
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(ω)sin(ω) from -π to π is:
```

0

5.3.6 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

SIX

CIRCULANT MATRICES

6.1 Overview

This lecture describes circulant matrices and some of their properties.

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

- convolution
- Fourier transforms
- permutation matrices

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

We begin by importing some Python packages

```
import numpy as np
from numba import njit
import matplotlib.pyplot as plt
%matplotlib inline
```

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

6.2 Constructing a Circulant Matrix

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

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

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

$$C = \begin{bmatrix} c_0 & c_1 & c_2 & c_3 & c_4 & \cdots & c_{N-1} \\ c_{N-1} & c_0 & c_1 & c_2 & c_3 & \cdots & c_{N-2} \\ c_{N-2} & c_{N-1} & c_0 & c_1 & c_2 & \cdots & c_{N-3} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ c_3 & c_4 & c_5 & c_6 & c_7 & \cdots & c_2 \\ c_2 & c_3 & c_4 & c_5 & c_6 & \cdots & c_1 \\ c_1 & c_2 & c_3 & c_4 & c_5 & \cdots & c_0 \end{bmatrix} \tag{6.1}$$

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

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

```
@njit
def construct_cirlulant(row):

    N = row.size

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

    for i in range(N):

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

    return C
```

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

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

6.2.1 Some Properties of Circulant Matrices

Here are some useful properties:

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

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

Now consider a circulant matrix with first row

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

and consider a vector

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

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

$$b_k = \sum_{i=0}^{n-1} c_{k-i} a_i \tag{6.2}$$

We use $*$ to denote **convolution** via the calculation described in equation (6.2).

It can be verified that the vector b satisfies

$$b = C^T a$$

where C^T is the transpose of the circulant matrix defined in equation (6.1).

6.3 Connection to Permutation Matrix

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

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

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

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

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

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

Every permutation corresponds to a unique permutation matrix.

For example, the $N \times N$ matrix

$$P = \begin{bmatrix} 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 \\ 1 & 0 & 0 & 0 & \cdots & 0 \end{bmatrix} \tag{6.3}$$

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

Eigenvalues of the cyclic shift permutation matrix P defined in equation (6.3) can be computed by constructing

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

and solving

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

Eigenvalues λ_i can be complex.

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

Thus, **singular values** of the permutation matrix P defined in equation (6.3) all equal 1.

It can be verified that permutation matrices are orthogonal matrices:

$$PP' = I$$

6.4 Examples with Python

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

```
@njit
def construct_P(N):

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

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

    return P
```

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

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

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

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

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

□1 = 0.0+1.0j
vec1 = [ 0.5+0.j -0.5+0.j -0.5-0.j -0.5+0.j]

□2 = 0.0-1.0j
vec2 = [-0.5+0.j   0. -0.5j  0. +0.5j -0.5+0.j ]

□3 = 1.0+0.0j
vec3 = [ 0.5+0.j  0.5-0.j  0.5+0.j -0.5+0.j]
```

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

These eigenvalues are uniformly distributed along the unit circle.

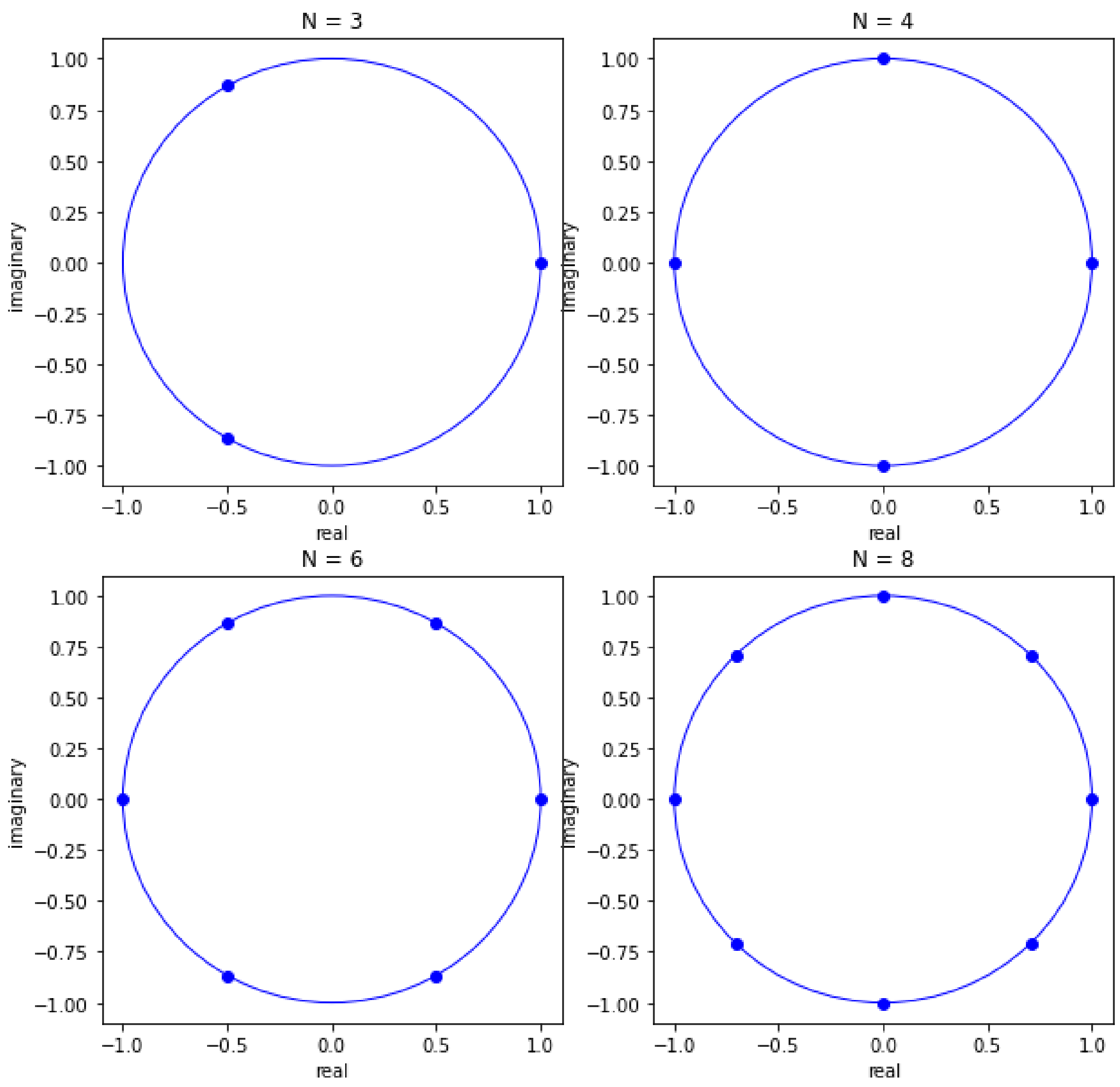
They are the n **roots of unity**, meaning they are the n numbers z that solve $z^n = 1$, where z is a complex number.

In particular, the n roots of unity are

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

where j denotes the purely imaginary unit number.

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

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

    row_i = i // 2
    col_i = i % 2

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

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

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

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

plt.show()
```



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

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

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

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

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

The matrix F_8 defines a Discete Fourier Transform.

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

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

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

```
def construct_F(N):

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

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

    return F, w
```

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

```
/tmp/ipykernel_11100/903011294.py:3: DeprecationWarning: `np.complex` is a␣
↪deprecated alias for the builtin `complex`. To silence this warning, use␣
↪`complex` by itself. Doing this will not modify any behavior and is safe. If you␣
↪specifically wanted the numpy scalar type, use `np.complex128` here.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/
↪release/1.20.0-notes.html#deprecations
  w = np.e ** (-np.complex(0, 2*np.pi/N))
/tmp/ipykernel_11100/903011294.py:5: DeprecationWarning: `np.complex` is a␣
↪deprecated alias for the builtin `complex`. To silence this warning, use␣
↪`complex` by itself. Doing this will not modify any behavior and is safe. If you␣
↪specifically wanted the numpy scalar type, use `np.complex128` here.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/
↪release/1.20.0-notes.html#deprecations
  F = np.ones((N, N), dtype=np.complex)
```

```
w
```

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

```
F8
```

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

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

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

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

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

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

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

```
/tmp/ipykernel_11100/646542455.py:1: DeprecationWarning: `np.complex` is a
 ↪deprecated alias for the builtin `complex`. To silence this warning, use
 ↪`complex` by itself. Doing this will not modify any behavior and is safe. If you
 ↪specifically wanted the numpy scalar type, use `np.complex128` here.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/
 ↪release/1.20.0-notes.html#deprecations
  diff_arr = np.empty(8, dtype=np.complex)
```

```
diff_arr
```

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

6.5 Associated Permutation Matrix

Next, we execute calculations to verify that the circulant matrix C defined in equation (6.1) can be written as

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

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

We illustrate this for $N = 8$ case.

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

```
c
```

```
array([0.873, 0.241, 0.701, 0.546, 0.32 , 0.88 , 0.282, 0.474])
```

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

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

```
N = 8

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

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

```
C
```

```
array([[0.873, 0.241, 0.701, 0.546, 0.32 , 0.88 , 0.282, 0.474],
       [0.474, 0.873, 0.241, 0.701, 0.546, 0.32 , 0.88 , 0.282],
       [0.282, 0.474, 0.873, 0.241, 0.701, 0.546, 0.32 , 0.88 ],
       [0.88 , 0.282, 0.474, 0.873, 0.241, 0.701, 0.546, 0.32 ],
       [0.32 , 0.88 , 0.282, 0.474, 0.873, 0.241, 0.701, 0.546],
       [0.546, 0.32 , 0.88 , 0.282, 0.474, 0.873, 0.241, 0.701],
       [0.701, 0.546, 0.32 , 0.88 , 0.282, 0.474, 0.873, 0.241],
       [0.241, 0.701, 0.546, 0.32 , 0.88 , 0.282, 0.474, 0.873]])
```

```
C8
```

```
array([[0.873, 0.241, 0.701, 0.546, 0.32 , 0.88 , 0.282, 0.474],
       [0.474, 0.873, 0.241, 0.701, 0.546, 0.32 , 0.88 , 0.282],
       [0.282, 0.474, 0.873, 0.241, 0.701, 0.546, 0.32 , 0.88 ],
       [0.88 , 0.282, 0.474, 0.873, 0.241, 0.701, 0.546, 0.32 ],
       [0.32 , 0.88 , 0.282, 0.474, 0.873, 0.241, 0.701, 0.546],
       [0.546, 0.32 , 0.88 , 0.282, 0.474, 0.873, 0.241, 0.701],
       [0.701, 0.546, 0.32 , 0.88 , 0.282, 0.474, 0.873, 0.241],
       [0.241, 0.701, 0.546, 0.32 , 0.88 , 0.282, 0.474, 0.873]])
```

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

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

```
0.0
```

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

```
?_C8 = np.zeros(8, dtype=np.complex)

for j in range(8):
    for k in range(8):
        ?_C8[j] += c[k] * w ** (j * k)
```

```
/tmp/ipykernel_11100/866898372.py:1: DeprecationWarning: `np.complex` is a␣
↪deprecated alias for the builtin `complex`. To silence this warning, use␣
↪`complex` by itself. Doing this will not modify any behavior and is safe. If you␣
↪specifically wanted the numpy scalar type, use `np.complex128` here.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/
↪release/1.20.0-notes.html#deprecations
  ?_C8 = np.zeros(8, dtype=np.complex)
```

```
?_C8
```

```
array([4.316+0.j   , 0.051-0.019j, 0.21 -0.1j  , 1.056+0.82j ,
       0.035-0.j   , 1.056-0.82j , 0.21 +0.1j  , 0.051+0.019j])
```

We can verify this by comparing `C8 @ Q8[:, j]` with `?_C8[j] * Q8[:, j]`.

```
# verify
for j in range(8):
    diff = C8 @ Q8[:, j] - ?_C8[j] * Q8[:, j]
    print(diff)
```

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

6.6 Discrete Fourier Transform

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

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

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

where

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

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

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

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

    return X
```

Consider the following example.

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

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

```
x
```

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

Apply a discrete Fourier transform.

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

```
/tmp/ipykernel_11100/1700622740.py:5: DeprecationWarning: `np.complex` is a␣
↪deprecated alias for the builtin `complex`. To silence this warning, use␣
↪`complex` by itself. Doing this will not modify any behavior and is safe. If you␣
↪specifically wanted the numpy scalar type, use `np.complex128` here.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/
↪release/1.20.0-notes.html#deprecations
  w = np.e ** (-np.complex(0, 2*np.pi/N))
/tmp/ipykernel_11100/1700622740.py:7: DeprecationWarning: `np.complex` is a␣
↪deprecated alias for the builtin `complex`. To silence this warning, use␣
↪`complex` by itself. Doing this will not modify any behavior and is safe. If you␣
↪specifically wanted the numpy scalar type, use `np.complex128` here.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/
↪release/1.20.0-notes.html#deprecations
  X = np.zeros(N, dtype=np.complex)
```

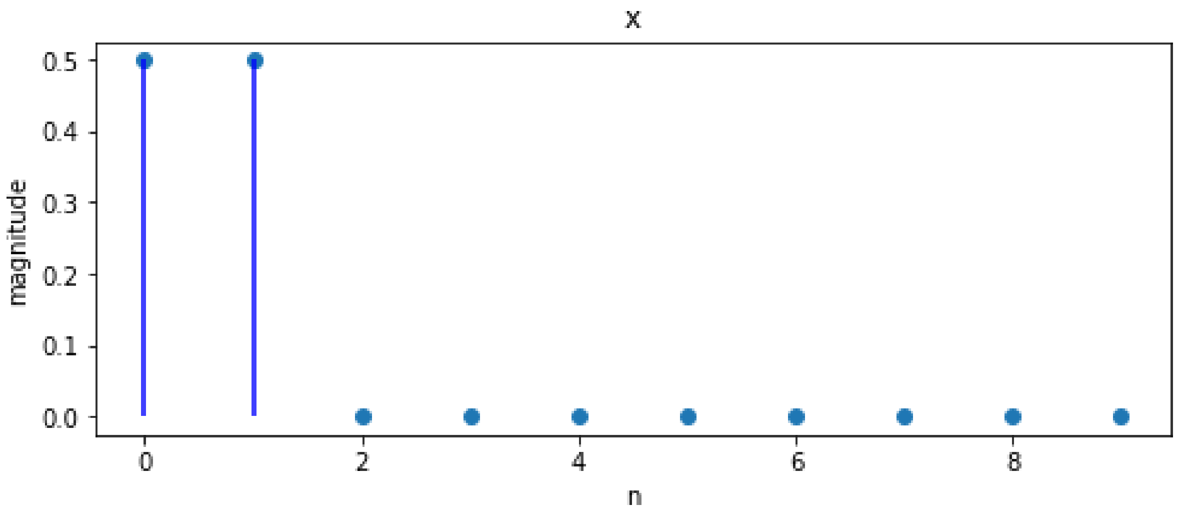
```
X
```

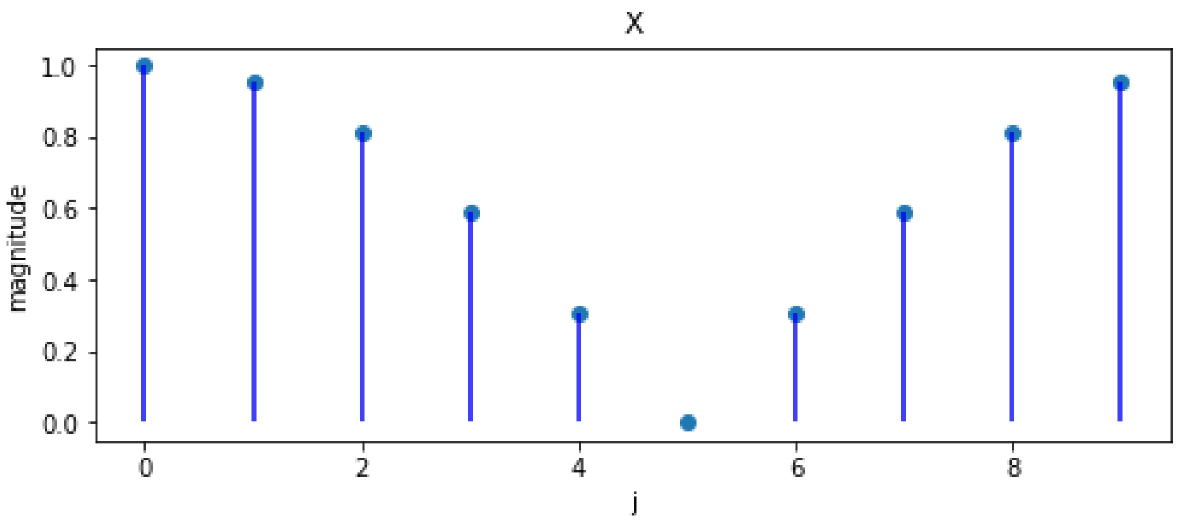
```
array([ 1.   +0.j   ,  0.905-0.294j,  0.655-0.476j,  0.345-0.476j,
        0.095-0.294j, -0.   +0.j   ,  0.095+0.294j,  0.345+0.476j,
        0.655+0.476j,  0.905+0.294j])
```

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

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

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

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

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

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





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

The inverse Fourier transform is defined as

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

```python
def inverse_transform(X):

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

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

    return x
```

```python
inverse_transform(X)
```

```
/tmp/ipykernel_11100/1761241726.py:4: DeprecationWarning: `np.complex` is a␣
↪deprecated alias for the builtin `complex`. To silence this warning, use␣
↪`complex` by itself. Doing this will not modify any behavior and is safe. If you␣
↪specifically wanted the numpy scalar type, use `np.complex128` here.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/
↪release/1.20.0-notes.html#deprecations
  w = np.e ** (np.complex(0, 2*np.pi/N))
/tmp/ipykernel_11100/1761241726.py:6: DeprecationWarning: `np.complex` is a␣
↪deprecated alias for the builtin `complex`. To silence this warning, use␣
↪`complex` by itself. Doing this will not modify any behavior and is safe. If you␣
↪specifically wanted the numpy scalar type, use `np.complex128` here.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/
↪release/1.20.0-notes.html#deprecations
  x = np.zeros(N, dtype=np.complex)
```

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

Another example is

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

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

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

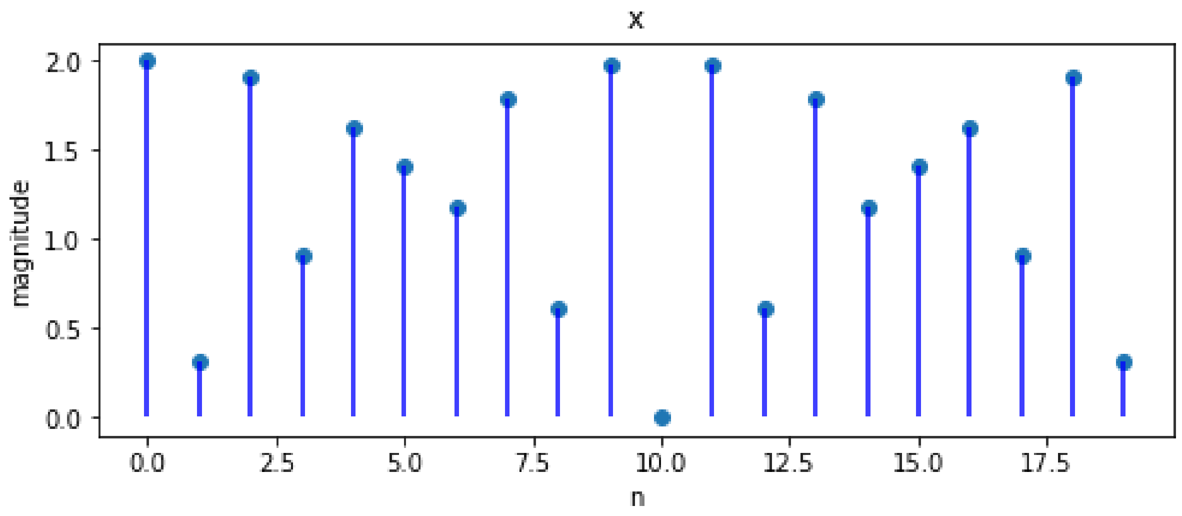
Since $\frac{11}{40}$ is in between $\frac{10}{40}$ and $\frac{12}{40}$ (each of which is an integer multiple of $\frac{1}{20}$), the complex coefficients in the DFT have their largest magnitudes at $k = 5, 6, 15, 16$, not just at a single frequency.

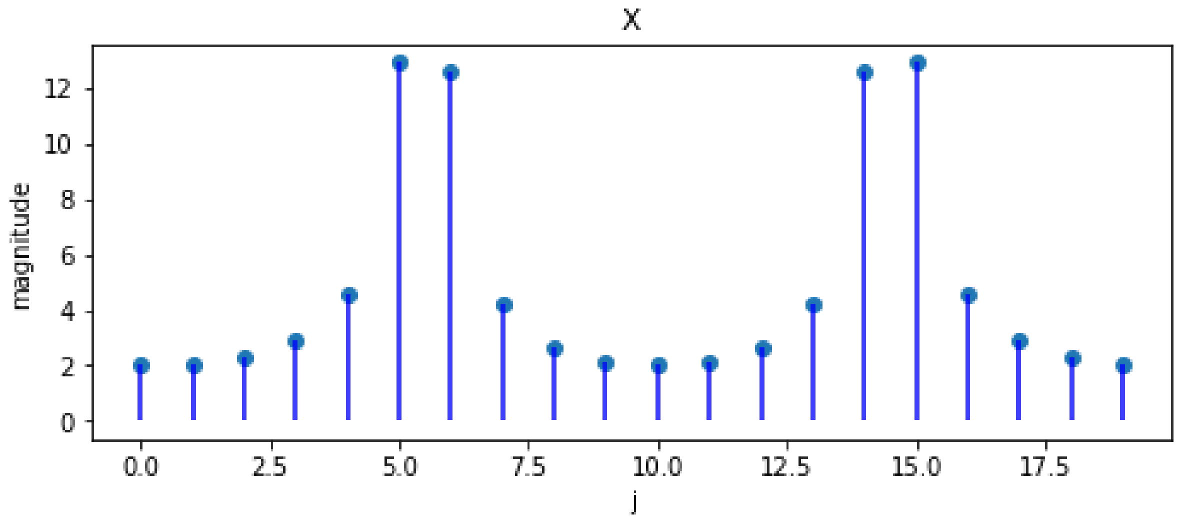
```python
N = 20
x = np.empty(N)

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

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

```
/tmp/ipykernel_11100/1700622740.py:5: DeprecationWarning: `np.complex` is a␣
 ↪deprecated alias for the builtin `complex`. To silence this warning, use␣
 ↪`complex` by itself. Doing this will not modify any behavior and is safe. If you␣
 ↪specifically wanted the numpy scalar type, use `np.complex128` here.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/
 ↪release/1.20.0-notes.html#deprecations
  w = np.e ** (-np.complex(0, 2*np.pi/N))
/tmp/ipykernel_11100/1700622740.py:7: DeprecationWarning: `np.complex` is a␣
 ↪deprecated alias for the builtin `complex`. To silence this warning, use␣
 ↪`complex` by itself. Doing this will not modify any behavior and is safe. If you␣
 ↪specifically wanted the numpy scalar type, use `np.complex128` here.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/
 ↪release/1.20.0-notes.html#deprecations
  X = np.zeros(N, dtype=np.complex)
```

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





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

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

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

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

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

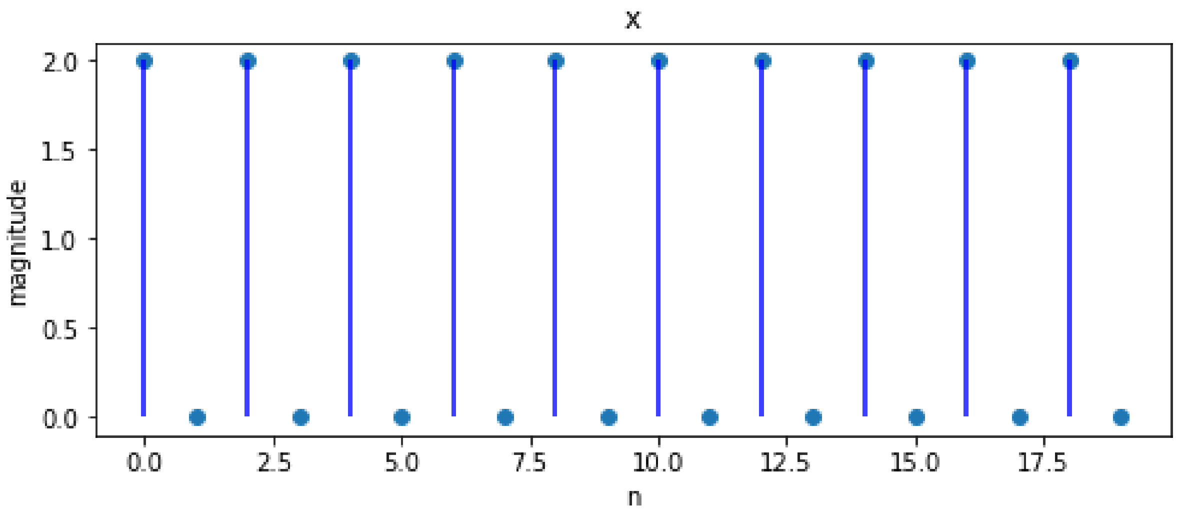
```
/tmp/ipykernel_11100/1700622740.py:5: DeprecationWarning: `np.complex` is a␣
↪deprecated alias for the builtin `complex`. To silence this warning, use␣
↪`complex` by itself. Doing this will not modify any behavior and is safe. If you␣
↪specifically wanted the numpy scalar type, use `np.complex128` here.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/
↪release/1.20.0-notes.html#deprecations
```

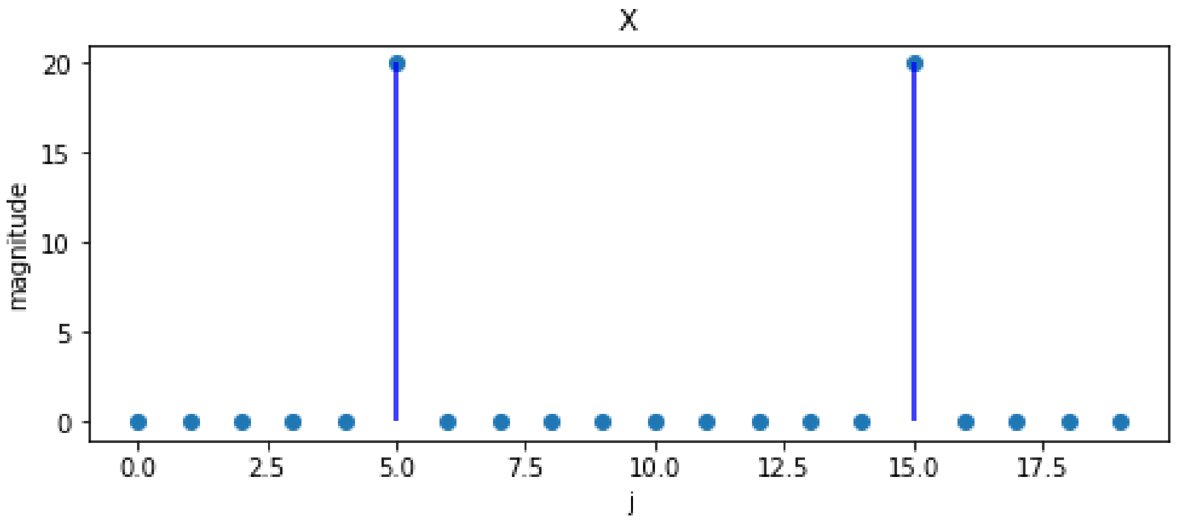
(continues on next page)

(continued from previous page)

```
  w = np.e ** (-np.complex(0, 2*np.pi/N))
/tmp/ipykernel_11100/1700622740.py:7: DeprecationWarning: `np.complex` is a␣
↪deprecated alias for the builtin `complex`. To silence this warning, use␣
↪`complex` by itself. Doing this will not modify any behavior and is safe. If you␣
↪specifically wanted the numpy scalar type, use `np.complex128` here.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/
↪release/1.20.0-notes.html#deprecations
  X = np.zeros(N, dtype=np.complex)
```

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





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

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

```
N = 20
```

(continues on next page)

(continued from previous page)

```
x = np.empty(N)

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

```
x
```

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

First use the summation formula to transform x to X.

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

```
/tmp/ipykernel_11100/1700622740.py:5: DeprecationWarning: `np.complex` is a␣
↪deprecated alias for the builtin `complex`. To silence this warning, use␣
↪`complex` by itself. Doing this will not modify any behavior and is safe. If you␣
↪specifically wanted the numpy scalar type, use `np.complex128` here.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/
↪release/1.20.0-notes.html#deprecations
  w = np.e ** (-np.complex(0, 2*np.pi/N))
/tmp/ipykernel_11100/1700622740.py:7: DeprecationWarning: `np.complex` is a␣
↪deprecated alias for the builtin `complex`. To silence this warning, use␣
↪`complex` by itself. Doing this will not modify any behavior and is safe. If you␣
↪specifically wanted the numpy scalar type, use `np.complex128` here.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/
↪release/1.20.0-notes.html#deprecations
  X = np.zeros(N, dtype=np.complex)
```

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

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

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

```
/tmp/ipykernel_11100/903011294.py:3: DeprecationWarning: `np.complex` is a␣
↪deprecated alias for the builtin `complex`. To silence this warning, use␣
↪`complex` by itself. Doing this will not modify any behavior and is safe. If you␣
↪specifically wanted the numpy scalar type, use `np.complex128` here.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/
↪release/1.20.0-notes.html#deprecations
  w = np.e ** (-np.complex(0, 2*np.pi/N))
/tmp/ipykernel_11100/903011294.py:5: DeprecationWarning: `np.complex` is a␣
↪deprecated alias for the builtin `complex`. To silence this warning, use␣
↪`complex` by itself. Doing this will not modify any behavior and is safe. If you␣
↪specifically wanted the numpy scalar type, use `np.complex128` here.
```

(continues on next page)

(continued from previous page)

```
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/
↪release/1.20.0-notes.html#deprecations
  F = np.ones((N, N), dtype=np.complex)
```

```
F20 @ x
```

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

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

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

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

CHAPTER

SEVEN

SINGULAR VALUE DECOMPOSITION (SVD)

In addition to regular packages contained in Anaconda by default, this lecture also requires:

```
!pip install quandl
```

```
import numpy as np
import numpy.linalg as LA
import matplotlib.pyplot as plt
%matplotlib inline
import quandl as ql
import pandas as pd
```

7.1 Overview

The **singular value decomposition** is a work-horse in applications of least squares projection that form a foundation for some important machine learning methods.

This lecture describes the singular value decomposition and two of its uses:

- principal components analysis (PCA)
- dynamic mode decomposition (DMD)

Each of these can be thought of as a data-reduction procedure designed to capture salient patterns by projecting data onto a limited set of factors.

7.2 The Setup

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

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

In this lecture, we'll think of X as a matrix of **data**.

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

We'll be interested in two cases

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

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

In the first case in which there are many more observations n than random variables m, we learn about a joint distribution by taking averages across observations of functions of the observations.

Here we'll look for **patterns** by using a **singular value decomposition** to do a **principal components analysis** (PCA).

In the second case in which there are many more random variables m than observations n, we'll proceed in a different way.

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

7.3 Singular Value Decomposition

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

$$X = U\Sigma V^T$$

where

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

where

- U is an $m \times m$ matrix whose columns are eigenvectors of $X^T X$
- V is an $n \times n$ matrix whose columns are eigenvectors of XX^T
- Σ is an $m \times n$ matrix in which the first r places on its main diagonal are positive numbers $\sigma_1, \sigma_2, \dots, \sigma_r$ called **singular values**; remaining entries of Σ are all zero
- The r singular values are square roots of the eigenvalues of the $m \times m$ matrix XX^T and the $n \times n$ matrix $X^T X$
- When U is a complex valued matrix, U^T denotes the **conjugate-transpose** or **Hermitian-transpose** of U, meaning that U_{ij}^T is the complex conjugate of U_{ji}.
- Similarly, when V is a complex valued matrix, V^T denotes the **conjugate-transpose** or **Hermitian-transpose** of V

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

There is also an alternative shape convention called an **economy** or **reduced** SVD .

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

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

Sometimes, we will use a **full** SVD in which U, Σ, and V have shapes (m, m), (m, n), (n, n)

Caveat: The properties

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

apply to a **full** SVD but not to a **reduced** SVD.

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

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

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

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

When we study Dynamic Mode Decomposition below, we shall want to remember this caveat because we'll be using reduced SVD's to compute key objects.

7.4 Reduced Versus Full SVD

Earlier, we mentioned **full** and **reduced** SVD's.

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

In a **full** SVD

- U is $m \times m$
- Σ is $m \times n$
- V is $n \times n$

In a **reduced** SVD

- U is $m \times r$
- Σ is $r \times r$
- V is $n \times r$

Let's do a some small exercise to compare **full** and **reduced** SVD's.

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

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

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

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

```
(None,
 array([[-0.37541622, -0.19584961, -0.67837623, -0.4401461 , -0.40839036],
        [-0.55849099, -0.35735004,  0.65235117, -0.3671743 , -0.00312385],
        [-0.42904804,  0.86421085,  0.11626597,  0.02008157, -0.23481127],
        [-0.48271863, -0.27745485, -0.10739446,  0.81450895, -0.12265046],
        [-0.36062582,  0.10051012, -0.2986508 , -0.08732897,  0.87351479]]),
 array([1.55734944, 0.39671477]),
 array([[-0.96609979, -0.25816894],
        [-0.25816894,  0.96609979]]))
```

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

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

```
(None,
 array([[-0.37541622, -0.19584961],
        [-0.55849099, -0.35735004],
        [-0.42904804,  0.86421085],
        [-0.48271863, -0.27745485],
        [-0.36062582,  0.10051012]]),
 array([1.55734944, 0.39671477]),
 array([[-0.96609979, -0.25816894],
        [-0.25816894,  0.96609979]]))
```

```
rr = np.linalg.matrix_rank(X)
print('rank of X - '), rr
```

```
rank of X -
```

```
(None, 2)
```

Properties:

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

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

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

```
UUT, UTU =
```

```
(None,
 array([[ 1.00000000e+00,  7.25986945e-17, -4.29313316e-17,
         -7.16514089e-20, -3.42603458e-17],
        [ 7.25986945e-17,  1.00000000e+00,  1.64603035e-16,
         -6.51859739e-17, -8.41387647e-18],
        [-4.29313316e-17,  1.64603035e-16,  1.00000000e+00,
          9.85891939e-18, -1.53238976e-16],
        [-7.16514089e-20, -6.51859739e-17,  9.85891939e-18,
          1.00000000e+00, -4.63853856e-17],
        [-3.42603458e-17, -8.41387647e-18, -1.53238976e-16,
         -4.63853856e-17,  1.00000000e+00]]),
 array([[ 1.00000000e+00,  6.09772859e-18,  1.03210775e-16,
          7.46020143e-17,  9.69958745e-18],
        [ 6.09772859e-18,  1.00000000e+00,  1.78870202e-16,
          1.51877577e-17,  3.14064924e-17],
        [ 1.03210775e-16,  1.78870202e-16,  1.00000000e+00,
          8.72751834e-17, -6.17786179e-17],
        [ 7.46020143e-17,  1.51877577e-17,  8.72751834e-17,
          1.00000000e+00,  3.04189949e-17],
        [ 9.69958745e-18,  3.14064924e-17, -6.17786179e-17,
          3.04189949e-17,  1.00000000e+00]]))
```

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

```
UhatUhatT, UhatTUhat=
```

```
(None,
 array([[ 0.17929441,  0.27965344, -0.00818377,  0.23555983,  0.11569991],
        [ 0.27965344,  0.43961124, -0.06920632,  0.36874251,  0.16548898],
        [-0.00818377, -0.06920632,  0.93094262, -0.03267001,  0.24158774],
        [ 0.23555983,  0.36874251, -0.03267001,  0.30999847,  0.14619378],
        [ 0.11569991,  0.16548898,  0.24158774,  0.14619378,  0.14015327]]),
 array([[1.00000000e+00, 6.09772859e-18],
        [6.09772859e-18, 1.00000000e+00]]))
```

Remark: The cells above illustrate application of the `fullmatrices=True` and `full-matrices=False` options. Using `full-matrices=False` returns a reduced singular value decomposition. This option implements an optimal reduced rank approximation of a matrix, in the sense of minimizing the Frobenius norm of the discrepancy between the approximating matrix and the matrix being approximated. Optimality in this sense is established in the celebrated Eckart–Young theorem. See https://en.wikipedia.org/wiki/Low-rank_approximation.

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

Let's do another exercise, but now we'll set $m = 2 < 5 = n$

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

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

```
(None,
 array([[ 0.75354378, -0.65739773],
        [ 0.65739773,  0.75354378]]),
 array([1.52371698, 0.78047845]),
 array([[ 0.24335076,  0.20429337,  0.41002952,  0.69252016,  0.50133447],
        [ 0.47340092,  0.16756132,  0.56260509, -0.076725  , -0.65222969],
        [-0.35798988, -0.64724402,  0.63701494, -0.16367735,  0.14261877],
        [-0.61776202,  0.05930924, -0.01976992,  0.58720849, -0.51927627],
        [-0.45484648,  0.71256229,  0.3304125 , -0.37805425,  0.18240677]]))
```

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

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

```
(None,
 array([[ 0.75354378, -0.65739773],
        [ 0.65739773,  0.75354378]]),
```

(continues on next page)

(continued from previous page)

```
array([1.52371698, 0.78047845]),
array([[ 0.24335076,  0.20429337,  0.41002952,  0.69252016,  0.50133447],
       [ 0.47340092,  0.16756132,  0.56260509, -0.076725  , -0.65222969]]))
```

```
rr = np.linalg.matrix_rank(X)
print('rank X = '), rr
```

```
rank X =
```

```
(None, 2)
```

7.5 Digression: Polar Decomposition

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

$$X = SQ$$

where

$$S = U\Sigma U^T$$
$$Q = UV^T$$

and S is evidently a symmetric matrix and Q is an orthogonal matrix.

7.6 Principle Components Analysis (PCA)

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

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

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

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

where for $j = 1, \dots, n$ the column vector $X_j = \begin{bmatrix} X_{1j} \\ X_{2j} \\ \vdots \\ X_{mj} \end{bmatrix}$ is a vector of observations on variables $\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}$.

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

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

The number of positive singular values equals the rank of matrix X.

Arrange the singular values in decreasing order.

Arrange the positive singular values on the main diagonal of the matrix Σ of into a vector σ_R.

Set all other entries of Σ to zero.

7.7 Relationship of PCA to SVD

To relate a SVD to a PCA (principal component analysis) of data set X, first construct the SVD of the data matrix X:

$$X = U\Sigma V^T = \sigma_1 U_1 V_1^T + \sigma_2 U_2 V_2^T + \cdots + \sigma_r U_r V_r^T \tag{7.1}$$

where

$$U = [U_1 | U_2 | \ldots | U_m]$$

$$V^T = \begin{bmatrix} V_1^T \\ V_2^T \\ \ldots \\ V_n^T \end{bmatrix}$$

In equation (7.1), each of the $m \times n$ matrices $U_j V_j^T$ is evidently of rank 1.

Thus, we have

$$X = \sigma_1 \begin{pmatrix} U_{11} V_1^T \\ U_{21} V_1^T \\ \cdots \\ U_{m1} V_1^T \end{pmatrix} + \sigma_2 \begin{pmatrix} U_{12} V_2^T \\ U_{22} V_2^T \\ \cdots \\ U_{m2} V_2^T \end{pmatrix} + \ldots + \sigma_r \begin{pmatrix} U_{1r} V_r^T \\ U_{2r} V_r^T \\ \cdots \\ U_{mr} V_r^T \end{pmatrix} \tag{7.2}$$

Here is how we would interpret the objects in the matrix equation (7.2) in a time series context:

- $V_k^T = \begin{bmatrix} V_{k1} & V_{k2} & \ldots & V_{kn} \end{bmatrix}$ for each $k = 1, \ldots, n$ is a time series $\{V_{kj}\}_{j=1}^n$ for the kth **principal component**
- $U_j = \begin{bmatrix} U_{1k} \\ U_{2k} \\ \ldots \\ U_{mk} \end{bmatrix}$ $k = 1, \ldots, m$ is a vector of **loadings** of variables X_i on the kth principle component, $i = 1, \ldots, m$
- σ_k for each $k = 1, \ldots, r$ is the strength of kth **principal component**

7.8 PCA with Eigenvalues and Eigenvectors

We now use an eigen decomposition of a sample covariance matrix to do PCA.

Let $X_{m\times n}$ be our $m \times n$ data matrix.

Let's assume that sample means of all variables are zero.

We can assure this by **pre-processing** the data by subtracting sample means.

Define the sample covariance matrix Ω as

$$\Omega = XX^T$$

Then use an eigen decomposition to represent Ω as follows:

$$\Omega = P\Lambda P^T$$

Here

- P is $m \times m$ matrix of eigenvectors of Ω
- Λ is a diagonal matrix of eigenvalues of Ω

We can then represent X as

$$X = P\epsilon$$

where

$$\epsilon\epsilon^T = \Lambda.$$

We can verify that

$$XX^T = P\Lambda P^T.$$

It follows that we can represent the data matrix as

$$X = [X_1|X_2|\dots|X_m] = [P_1|P_2|\dots|P_m]\begin{bmatrix}\epsilon_1\\ \epsilon_2\\ \dots\\ \epsilon_m\end{bmatrix} = P_1\epsilon_1 + P_2\epsilon_2 + \dots + P_m\epsilon_m$$

where

$$\epsilon\epsilon^T = \Lambda.$$

To reconcile the preceding representation with the PCA that we obtained through the SVD above, we first note that $\epsilon_j^2 = \lambda_j \equiv \sigma_j^2$.

Now define $\tilde{\epsilon_j} = \frac{\epsilon_j}{\sqrt{\lambda_j}}$, which evidently implies that $\tilde{\epsilon}_j\tilde{\epsilon}_j^T = 1$.

Therefore

$$\begin{aligned}X &= \sqrt{\lambda_1}P_1\tilde{\epsilon_1} + \sqrt{\lambda_2}P_2\tilde{\epsilon_2} + \dots + \sqrt{\lambda_m}P_m\tilde{\epsilon_m}\\ &= \sigma_1P_1\tilde{\epsilon_2} + \sigma_2P_2\tilde{\epsilon_2} + \dots + \sigma_mP_m\tilde{\epsilon_m},\end{aligned}$$

which evidently agrees with

$$X = \sigma_1U_1{V_1}^T + \sigma_2U_2{V_2}^T + \dots + \sigma_rU_r{V_r}^T$$

provided that we set

- $U_j = P_j$ (the loadings of variables on principal components)
- ${V_k}^T = \tilde{\epsilon_k}$ (the principal components)

Since there are several possible ways of computing P and U for given a data matrix X, depending on algorithms used, we might have sign differences or different orders between eigenvectors.

We can resolve such ambiguities about U and P by

1. sorting eigenvalues and singular values in descending order
2. imposing positive diagonals on P and U and adjusting signs in V^T accordingly

7.9 Connections

To pull things together, it is useful to assemble and compare some formulas presented above.

First, consider the following SVD of an $m \times n$ matrix:

$$X = U\Sigma V^T$$

Compute:

$$\begin{aligned} XX^T &= U\Sigma V^T V\Sigma^T U^T \\ &\equiv U\Sigma\Sigma^T U^T \\ &\equiv U\Lambda U^T \end{aligned}$$

Thus, U in the SVD is the matrix P of eigenvectors of XX^T and $\Sigma\Sigma^T$ is the matrix Λ of eigenvalues.

Second, let's compute

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

Thus, the matrix V in the SVD is the matrix of eigenvectors of $X^T X$

Summarizing and fitting things together, we have the eigen decomposition of the sample covariance matrix

$$XX^T = P\Lambda P^T$$

where P is an orthogonal matrix.

Further, from the SVD of X, we know that

$$XX^T = U\Sigma\Sigma^T U^T$$

where U is an orthonal matrix.

Thus, $P = U$ and we have the representation of X

$$X = P\epsilon = U\Sigma V^T$$

It follows that

$$U^T X = \Sigma V^T = \epsilon$$

Note that the preceding implies that

$$\epsilon\epsilon^T = \Sigma V^T V\Sigma^T = \Sigma\Sigma^T = \Lambda,$$

so that everything fits together.

Below we define a class `DecomAnalysis` that wraps PCA and SVD for a given a data matrix `X`.

```python
class DecomAnalysis:
    """
    A class for conducting PCA and SVD.
    """

    def __init__(self, X, n_component=None):

        self.X = X

        self.Ω = (X @ X.T)

        self.m, self.n = X.shape
        self.r = LA.matrix_rank(X)

        if n_component:
```

(continues on next page)

(continued from previous page)

```
            self.n_component = n_component
        else:
            self.n_component = self.m

    def pca(self):

        □, P = LA.eigh(self.Ω)    # columns of P are eigenvectors

        ind = sorted(range(□.size), key=lambda x: □[x], reverse=True)

        # sort by eigenvalues
        self.□ = □[ind]
        P = P[:, ind]
        self.P = P @ diag_sign(P)

        self.Λ = np.diag(self.□)

        self.explained_ratio_pca = np.cumsum(self.□) / self.□.sum()

        # compute the N by T matrix of principal components
        self.□ = self.P.T @ self.X

        P = self.P[:, :self.n_component]
        □ = self.□[:self.n_component, :]

        # transform data
        self.X_pca = P @ □

    def svd(self):

        U, □, VT = LA.svd(self.X)

        ind = sorted(range(□.size), key=lambda x: □[x], reverse=True)

        # sort by eigenvalues
        d = min(self.m, self.n)

        self.□ = □[ind]
        U = U[:, ind]
        D = diag_sign(U)
        self.U = U @ D
        VT[:d, :] = D @ VT[ind, :]
        self.VT = VT

        self.Σ = np.zeros((self.m, self.n))
        self.Σ[:d, :d] = np.diag(self.□)

        □_sq = self.□ ** 2
        self.explained_ratio_svd = np.cumsum(□_sq) / □_sq.sum()

        # slicing matrices by the number of components to use
        U = self.U[:, :self.n_component]
        Σ = self.Σ[:self.n_component, :self.n_component]
        VT = self.VT[:self.n_component, :]

        # transform data
```

(continues on next page)

(continued from previous page)

```
        self.X_svd = U @ Σ @ VT

    def fit(self, n_component):

        # pca
        P = self.P[:, :n_component]
        ⍰ = self.⍰[:n_component, :]

        # transform data
        self.X_pca = P @ ⍰

        # svd
        U = self.U[:, :n_component]
        Σ = self.Σ[:n_component, :n_component]
        VT = self.VT[:n_component, :]

        # transform data
        self.X_svd = U @ Σ @ VT

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

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

    return D
```

We also define a function that prints out information so that we can compare decompositions obtained by different algorithms.

```
def compare_pca_svd(da):
    """
    Compare the outcomes of PCA and SVD.
    """

    da.pca()
    da.svd()

    print('Eigenvalues and Singular values\n')
    print(f'λ = {da.λ}\n')
    print(f'σ^2 = {da.σ**2}\n')
    print('\n')

    # loading matrices
    fig, axs = plt.subplots(1, 2, figsize=(14, 5))
    plt.suptitle('loadings')
    axs[0].plot(da.P.T)
    axs[0].set_title('P')
    axs[0].set_xlabel('m')
    axs[1].plot(da.U.T)
    axs[1].set_title('U')
    axs[1].set_xlabel('m')
    plt.show()

    # principal components
    fig, axs = plt.subplots(1, 2, figsize=(14, 5))
    plt.suptitle('principal components')
```

(continues on next page)

(continued from previous page)

```
    axs[0].plot(da.ε.T)
    axs[0].set_title('ε')
    axs[0].set_xlabel('n')
    axs[1].plot(da.VT[:da.r, :].T * np.sqrt(da.λ))
    axs[1].set_title('$V^T*\sqrt{\lambda}$')
    axs[1].set_xlabel('n')
    plt.show()
```

For an example PCA applied to analyzing the structure of intelligence tests see this lecture *Multivariable Normal Distribution*.

Look at the parts of that lecture that describe and illustrate the classic factor analysis model.

7.10 Dynamic Mode Decomposition (DMD)

We turn to the case in which $m >> n$.

Here an $m \times n$ data matrix $\tilde{X}$ contains many more random variables m than observations n.

This **tall and skinny** case is associated with **Dynamic Mode Decomposition**.

Dynamic mode decomposition was introduced by [Sch10],

You can read more about Dynamic Mode Decomposition here [KBBWP16] and here [BK19] (section 7.2).

We want to fit a **first-order vector autoregression**

$$X_{t+1} = AX_t + C\epsilon_{t+1} \tag{7.3}$$

where ϵ_{t+1} is the time $t+1$ instance of an i.i.d. $m \times 1$ random vector with mean vector zero and identity covariance matrix and

where the $m \times 1$ vector X_t is

$$X_t = \begin{bmatrix} X_{1,t} & X_{2,t} & \cdots & X_{m,t} \end{bmatrix}^T \tag{7.4}$$

and where T again denotes complex transposition and $X_{i,t}$ is an observation on variable i at time t.

We want to fit equation (7.3).

Our data are organized in an $m \times (n+1)$ matrix $\tilde{X}$

$$\tilde{X} = [X_1 \mid X_2 \mid \cdots \mid X_n \mid X_{n+1}]$$

where for $t = 1, \ldots, n+1$, the $m \times 1$ vector X_t is given by (7.4).

Thus, we want to estimate a system (7.3) that consists of m least squares regressions of **everything** on one lagged value of **everything**.

The i'th equation of (7.3) is a regression of $X_{i,t+1}$ on the vector X_t.

We proceed as follows.

From $\tilde{X}$, we form two $m \times n$ matrices

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

and

$$X' = [X_2 \mid X_3 \mid \cdots \mid X_{n+1}]$$

Here $'$ does not indicate matrix transposition but instead is part of the name of the matrix X'.

In forming X and X', we have in each case dropped a column from $\tilde{X}$, the last column in the case of X, and the first column in the case of X'.

Evidently, X and X' are both $m \times n$ matrices.

We denote the rank of X as $p \leq \min(m, n)$.

Two possible cases are

- $n >> m$, so that we have many more time series observations n than variables m
- $m >> n$, so that we have many more variables m than time series observations n

At a general level that includes both of these special cases, a common formula describes the least squares estimator $\hat{A}$ of A for both cases, but important details differ.

The common formula is

$$\hat{A} = X'X^+ \tag{7.5}$$

where X^+ is the pseudo-inverse of X.

Formulas for the pseudo-inverse differ for our two cases.

When $n >> m$, so that we have many more time series observations n than variables m and when X has linearly independent **rows**, XX^T has an inverse and the pseudo-inverse X^+ is

$$X^+ = X^T(XX^T)^{-1}$$

Here X^+ is a **right-inverse** that verifies $XX^+ = I_{m\times m}$.

In this case, our formula (7.5) for the least-squares estimator of the population matrix of regression coefficients A becomes

$$\hat{A} = X'X^T(XX^T)^{-1}$$

This formula is widely used in economics to estimate vector autorgressions.

The right side is proportional to the empirical cross second moment matrix of X_{t+1} and X_t times the inverse of the second moment matrix of X_t.

This least-squares formula widely used in econometrics.

Tall-Skinny Case:

When $m >> n$, so that we have many more variables m than time series observations n and when X has linearly independent **columns**, X^TX has an inverse and the pseudo-inverse X^+ is

$$X^+ = (X^TX)^{-1}X^T$$

Here X^+ is a **left-inverse** that verifies $X^+X = I_{n\times n}$.

In this case, our formula (7.5) for a least-squares estimator of A becomes

$$\hat{A} = X'(X^TX)^{-1}X^T \tag{7.6}$$

This is the case that we are interested in here.

If we use formula (7.6) to calculate $\hat{A}X$ we find that

$$\hat{A}X = X'$$

so that the regression equation **fits perfectly**, the usual outcome in an **underdetermined least-squares** model.

Thus, we want to fit equation (7.3) in a situation in which we have a number n of observations that is small relative to the number m of variables that appear in the vector X_t.

To reiterate and offer an idea about how we can efficiently calculate the pseudo-inverse X^+, as our estimator $\hat{A}$ of A we form an $m \times m$ matrix that solves the least-squares best-fit problem

$$\hat{A} = \text{argmin}_{\check{A}} ||X' - \check{A}X||_F \tag{7.7}$$

where $|| \cdot ||_F$ denotes the Frobeneus norm of a matrix.

The minimizer of the right side of equation (7.7) is

$$\hat{A} = X'X^+ \tag{7.8}$$

where the (possibly huge) $n \times m$ matrix $X^+ = (X^T X)^{-1} X^T$ is again a pseudo-inverse of X.

The ith row of $\hat{A}$ is an $m \times 1$ vector of regression coefficients of $X_{i,t+1}$ on $X_{j,t}, j = 1, \dots, m$.

For some situations that we are interested in, $X^T X$ can be close to singular, a situation that can make some numerical algorithms be error-prone.

To confront that possibility, we'll use efficient algorithms for computing and for constructing reduced rank approximations of $\hat{A}$ in formula (7.6).

The ith row of $\hat{A}$ is an $m \times 1$ vector of regression coefficients of $X_{i,t+1}$ on $X_{j,t}, j = 1, \dots, m$.

An efficient way to compute the pseudo-inverse X^+ is to start with a singular value decomposition

$$X = U\Sigma V^T \tag{7.9}$$

We can use the singular value decomposition (7.9) efficiently to construct the pseudo-inverse X^+ by recognizing the following string of equalities.

$$\begin{aligned} X^+ &= (X^T X)^{-1} X^T \\ &= (V\Sigma U^T U \Sigma V^T)^{-1} V \Sigma U^T \\ &= (V\Sigma\Sigma V^T)^{-1} V\Sigma U^T \\ &= V\Sigma^{-1}\Sigma^{-1}V^T V\Sigma U^T \\ &= V\Sigma^{-1}U^T \end{aligned} \tag{7.10}$$

(Since we are in the $m >> n$ case in which $V^T V = I$ in a reduced SVD, we can use the preceding string of equalities for a reduced SVD as well as for a full SVD.)

Thus, we shall construct a pseudo-inverse X^+ of X by using a singular value decomposition of X in equation (7.9) to compute

$$X^+ = V\Sigma^{-1}U^T \tag{7.11}$$

where the matrix Σ^{-1} is constructed by replacing each non-zero element of Σ with σ_j^{-1}.

We can use formula (7.11) together with formula (7.8) to compute the matrix $\hat{A}$ of regression coefficients.

Thus, our estimator $\hat{A} = X'X^+$ of the $m \times m$ matrix of coefficients A is

$$\hat{A} = X'V\Sigma^{-1}U^T \tag{7.12}$$

In addition to doing that, we'll eventually use **dynamic mode decomposition** to compute a rank r approximation to $\hat{A}$, where $r < p$.

Remark: We described and illustrated a **reduced** singular value decomposition above, and compared it with a **full** singular value decomposition. In our Python code, we'll typically use a reduced SVD.

Next, we describe alternative representations of our first-order linear dynamic system.

7.11 Representation 1

In this representation, we shall use a **full** SVD of X.

We use the m columns of U, and thus the m rows of U^T, to define a $m \times 1$ vector $\tilde{b}_t$ as follows

$$\tilde{b}_t = U^T X_t \tag{7.13}$$

and

$$X_t = U \tilde{b}_t \tag{7.14}$$

(Here we use the notation b to remind ourselves that we are creating a **b**asis vector.)

Since we are using a **full** SVD, UU^T is an $m \times m$ identity matrix.

So it follows from equation (7.13) that we can reconstruct X_t from $\tilde{b}_t$ by using

- Equation (7.13) serves as an **encoder** that rotates the $m \times 1$ vector X_t to become an $m \times 1$ vector $\tilde{b}_t$
- Equation (7.14) serves as a **decoder** that recovers the $m \times 1$ vector X_t by rotating the $m \times 1$ vector $\tilde{b}_t$

Define a transition matrix for a rotated $m \times 1$ state $\tilde{b}_t$ by

$$\tilde{A} = U^T \hat{A} U \tag{7.15}$$

We can evidently recover $\hat{A}$ from

$$\hat{A} = U \tilde{A} U^T$$

Dynamics of the rotated $m \times 1$ state $\tilde{b}_t$ are governed by

$$\tilde{b}_{t+1} = \tilde{A} \tilde{b}_t$$

To construct forecasts $\overline{X}_t$ of future values of X_t conditional on X_1, we can apply decoders (i.e., rotators) to both sides of this equation and deduce

$$\overline{X}_{t+1} = U \tilde{A}^t U^T X_1$$

where we use $\overline{X}_t$ to denote a forecast.

7.12 Representation 2

This representation is related to one originally proposed by [Sch10].

It can be regarded as an intermediate step to a related and perhaps more useful representation 3.

As with Representation 1, we continue to

- use a **full** SVD and **not** a reduced SVD

As we observed and illustrated earlier in this lecture, for a full SVD UU^T and $U^T U$ are both identity matrices; but under a reduced SVD of X, $U^T U$ is not an identity matrix.

As we shall see, a full SVD is too confining for what we ultimately want to do, namely, situations in which $U^T U$ is **not** an identity matrix because we use a reduced SVD of X.

But for now, let's proceed under the assumption that both of the preceding two requirements are satisfied.

Form an eigendecomposition of the $m \times m$ matrix $\tilde{A} = U^T \hat{A} U$ defined in equation (7.15):

$$\tilde{A} = W \Lambda W^{-1} \tag{7.16}$$

where Λ is a diagonal matrix of eigenvalues and W is an $m \times m$ matrix whose columns are eigenvectors corresponding to rows (eigenvalues) in Λ.

When $UU^T = I_{m \times m}$, as is true with a full SVD of X, it follows that

$$\hat{A} = U \tilde{A} U^T = U W \Lambda W^{-1} U^T \tag{7.17}$$

Evidently, according to equation (7.17), the diagonal matrix Λ contains eigenvalues of $\hat{A}$ and corresponding eigenvectors of $\hat{A}$ are columns of the matrix UW.

Thus, the systematic (i.e., not random) parts of the X_t dynamics captured by our first-order vector autoregressions are described by

$$X_{t+1} = U W \Lambda W^{-1} U^T X_t$$

Multiplying both sides of the above equation by $W^{-1} U^T$ gives

$$W^{-1} U^T X_{t+1} = \Lambda W^{-1} U^T X_t$$

or

$$\hat{b}_{t+1} = \Lambda \hat{b}_t$$

where now our encoder is

$$\hat{b}_t = W^{-1} U^T X_t$$

and our decoder is

$$X_t = U W \hat{b}_t$$

We can use this representation to construct a predictor $\overline{X}_{t+1}$ of X_{t+1} conditional on X_1 via:

$$\overline{X}_{t+1} = U W \Lambda^t W^{-1} U^T X_1 \tag{7.18}$$

In effect, [Sch10] defined an $m \times m$ matrix Φ_s as

$$\Phi_s = UW \tag{7.19}$$

and represented equation (7.18) as

$$\overline{X}_{t+1} = \Phi_s \Lambda^t \Phi_s^+ X_1 \tag{7.20}$$

Components of the basis vector $\hat{b}_t = W^{-1} U^T X_t \equiv \Phi_s^+$ are often called DMD **modes**, or sometimes also DMD **projected nodes**.

We turn next to an alternative representation suggested by Tu et al. [TRL+14], one that is more appropriate to use when, as in practice is typically the case, we use a reduced SVD.

7.13 Representation 3

Departing from the procedures used to construct Representations 1 and 2, each of which deployed a **full** SVD, we now use a **reduced** SVD.

Again, we let $p \leq \min(m, n)$ be the rank of X.

Construct a **reduced** SVD

$$X = \tilde{U}\tilde{\Sigma}\tilde{V}^T,$$

where now U is $m \times p$ and Σ is $p \times p$ and V^T is $p \times n$.

Our minimum-norm least-squares estimator approximator of A now has representation

$$\hat{A} = X'\tilde{V}\tilde{\Sigma}^{-1}\tilde{U}^T$$

Paralleling a step in Representation 1, define a transition matrix for a rotated $p \times 1$ state $\tilde{b}_t$ by

$$\tilde{A} = \tilde{U}^T\hat{A}\tilde{U} \tag{7.21}$$

Because we are now working with a reduced SVD, so that $\tilde{U}\tilde{U}^T \neq I$, since $\hat{A} \neq \tilde{U}\tilde{A}\tilde{U}^T$, we can't simply recover $\hat{A}$ from $\tilde{A}$ and $\tilde{U}$.

Nevertheless, hoping for the best, we persist and construct an eigendecomposition of what is now a $p \times p$ matrix $\tilde{A}$:

$$\tilde{A} = W\Lambda W^{-1} \tag{7.22}$$

Mimicking our procedure in Representation 2, we cross our fingers and compute the $m \times p$ matrix

$$\tilde{\Phi}_s = \tilde{U}W \tag{7.23}$$

that corresponds to (7.19) for a full SVD.

At this point, it is interesting to compute $\hat{A}\tilde{\Phi}_s$:

$$\begin{aligned}
\hat{A}\tilde{\Phi}_s &= (X'\tilde{V}\tilde{\Sigma}^{-1}\tilde{U}^T)(\tilde{U}W) \\
&= X'\tilde{V}\tilde{\Sigma}^{-1}W \\
&\neq (\tilde{U}W)\Lambda \\
&= \tilde{\Phi}_s\Lambda
\end{aligned}$$

That $\hat{A}\tilde{\Phi}_s \neq \tilde{\Phi}_s\Lambda$ means, that unlike the corresponding situation in Representation 2, columns of $\tilde{\Phi}_s = \tilde{U}W$ are **not** eigenvectors of $\hat{A}$ corresponding to eigenvalues Λ.

But in a quest for eigenvectors of $\hat{A}$ that we *can* compute with a reduced SVD, let's define

$$\Phi \equiv \hat{A}\tilde{\Phi}_s = X'\tilde{V}\tilde{\Sigma}^{-1}W$$

It turns out that columns of Φ **are** eigenvectors of $\hat{A}$, a consequence of a result established by Tu et al. [TRL+14].

To present their result, for convenience we'll drop the tilde $\tilde{\cdot}$ above U, V, and Σ and adopt the understanding that each of them is computed with a reduced SVD.

Thus, we now use the notation that the $m \times p$ matrix Φ is defined as

$$\Phi = X'V\Sigma^{-1}W \tag{7.24}$$

Proposition The p columns of Φ are eigenvectors of $\check{A}$.

Proof: From formula (7.24) we have

$$\begin{aligned} \hat{A}\Phi &= (X'V\Sigma^{-1}U^T)(X'V\Sigma^{-1}W) \\ &= X'V\Sigma^{-1}\tilde{A}W \\ &= X'V\Sigma^{-1}W\Lambda \\ &= \Phi\Lambda \end{aligned}$$

Thus, we have deduced that

$$\hat{A}\Phi = \Phi\Lambda \tag{7.25}$$

Let ϕ_i be the the $i$the column of Φ and λ_i be the corresponding i eigenvalue of $\tilde{A}$ from decomposition (7.22).

Writing out the $m \times 1$ vectors on both sides of equation (7.25) and equating them gives

$$\hat{A}\phi_i = \lambda_i \phi_i.$$

Thus, ϕ_i is an eigenvector of $\hat{A}$ that corresponds to eigenvalue λ_i of $\tilde{A}$.

This concludes the proof.

Also see [BK19] (p. 238)

7.13.1 Decoder of X as linear projection

From eigendecomposition (7.25) we can represent $\hat{A}$ as

$$\hat{A} = \Phi\Lambda\Phi^+. \tag{7.26}$$

From formula (7.26) we can deduce the reduced dimension dynamics

$$\check{b}_{t+1} = \Lambda\check{b}_t$$

where

$$\check{b}_t = \Phi^+ X_t \tag{7.27}$$

Since Φ has p linearly independent columns, the generalized inverse of Φ is

$$\Phi^\dagger = (\Phi^T\Phi)^{-1}\Phi^T$$

and so

$$\check{b} = (\Phi^T\Phi)^{-1}\Phi^T X \tag{7.28}$$

$\check{b}$ is recognizable as the matrix of least squares regression coefficients of the matrix X on the matrix Φ and

$$\check{X} = \Phi\check{b}$$

is the least squares projection of X on Φ.

By virtue of least-squares projection theory discussed here https://python-advanced.quantecon.org/orth_proj.html, we can represent X as the sum of the projection $\check{X}$ of X on Φ plus a matrix of errors.

To verify this, note that the least squares projection $\check{X}$ is related to X by

$$X = \Phi \check{b} + \epsilon$$

where ϵ is an $m \times n$ matrix of least squares errors satisfying the least squares orthogonality conditions $\epsilon^T \Phi = 0$ or

$$(X - \Phi \check{b})^T \Phi = 0_{m \times p} \tag{7.29}$$

Rearranging the orthogonality conditions (7.29) gives $X^T \Phi = \check{b} \Phi^T \Phi$ which implies formula (7.28).

7.13.2 Alternative algorithm

There is a better way to compute the $p \times 1$ vector $\check{b}_t$ than provided by formula (7.27).

In particular, the following argument from [BK19] (page 240) provides a computationally efficient way to compute $\check{b}_t$.

For convenience, we'll do this first for time $t = 1$.

For $t = 1$, we have

$$X_1 = \Phi \check{b}_1 \tag{7.30}$$

where $\check{b}_1$ is an $r \times 1$ vector.

Recall from representation 1 above that $X_1 = U \tilde{b}_1$, where $\tilde{b}_1$ is the time 1 basis vector for representation 1.

It then follows from equation (7.24) that

$$U \tilde{b}_1 = X' V \Sigma^{-1} W \check{b}_1$$

and consequently

$$\tilde{b}_1 = U^T X' V \Sigma^{-1} W \check{b}_1$$

Recall that from equation (7.12), $\tilde{A} = U^T X' V \Sigma^{-1}$.

It then follows that

$$\tilde{b}_1 = \tilde{A} W \check{b}_1$$

and therefore, by the eigendecomposition (7.16) of $\tilde{A}$, we have

$$\tilde{b}_1 = W \Lambda \check{b}_1$$

Consequently,

$$\check{b}_1 = (W \Lambda)^{-1} \tilde{b}_1$$

or

$$\check{b}_1 = (W \Lambda)^{-1} U^T X_1, \tag{7.31}$$

which is computationally more efficient than the following instance of equation (7.27) for computing the initial vector $\check{b}_1$:

$$\check{b}_1 = \Phi^+ X_1 \tag{7.32}$$

Users of DMD sometimes call components of the basis vector $\check{b}_t = \Phi^+ X_t \equiv (W \Lambda)^{-1} U^T X_t$ the **exact** DMD modes.

Conditional on X_t, we can compute our decoded $\check{X}_{t+j}, j = 1, 2, \ldots$ from either

$$\check{X}_{t+j} = \Phi \Lambda^j \Phi^+ X_t \tag{7.33}$$

or

$$\check{X}_{t+j} = \Phi \Lambda^j (W \Lambda)^{-1} U^T X_t. \tag{7.34}$$

We can then use $\check{X}_{t+j}$ to forcast X_{t+j}.

7.14 Using Fewer Modes

Some of the preceding formulas assume that we have retained all p modes associated with the positive singular values of X.

We can adjust our formulas to describe a situation in which we instead retain only the $r < p$ largest singular values.

In that case, we simply replace Σ with the appropriate $r \times r$ matrix of singular values, U with the $m \times r$ matrix of whose columns correspond to the r largest singular values, and V with the $n \times r$ matrix whose columns correspond to the r largest singular values.

Counterparts of all of the salient formulas above then apply.

7.15 Source for Some Python Code

You can find a Python implementation of DMD here:

https://mathlab.github.io/PyDMD/

Part II

Elementary Statistics

CHAPTER

EIGHT

ELEMENTARY PROBABILITY WITH MATRICES

This lecture uses matrix algebra to illustrate some basic ideas about probability theory.

After providing somewhat informal definitions of the underlying objects, we'll use matrices and vectors to describe probability distributions.

Among concepts that we'll be studying include

- a joint probability distribution
- marginal distributions associated with a given joint distribution
- conditional probability distributions
- statistical independence of two random variables
- joint distributions associated with a prescribed set of marginal distributions
  - couplings
  - copulas
- the probability distribution of a sum of two independent random variables
  - convolution of marginal distributions
- parameters that define a probability distribution
- sufficient statistics as data summaries

We'll use a matrix to represent a bivariate probability distribution and a vector to represent a univariate probability distribution

As usual, we'll start with some imports

```
# !pip install prettytable
```

```
import numpy as np
import matplotlib.pyplot as plt
import prettytable as pt
from mpl_toolkits.mplot3d import Axes3D
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('retina')
%matplotlib inline
```

```
/tmp/ipykernel_17182/4202758366.py:6: DeprecationWarning: `set_matplotlib_formats`␣
↪is deprecated since IPython 7.23, directly use `matplotlib_inline.backend_inline.
↪set_matplotlib_formats()`
  set_matplotlib_formats('retina')
```

8.1 Sketch of Basic Concepts

We'll briefly define what we mean by a **probability space**, a **probability measure**, and a **random variable**.

For most of this lecture, we sweep these objects into the background, but they are there underlying the other objects that we'll mainly focus on.

Let Ω be a set of possible underlying outcomes and let $\omega \in \Omega$ be a particular underlying outcomes.

Let $\mathcal{G} \subset \Omega$ be a subset of Ω.

Let $\mathcal{F}$ be a collection of such subsets $\mathcal{G} \subset \Omega$.

The pair $\Omega, \mathcal{F}$ forms our **probability space** on which we want to put a probability measure.

A **probability measure** μ maps a set of possible underlying outcomes $\mathcal{G} \in \mathcal{F}$ into a scalar number between 0 and 1

- this is the "probability" that X belongs to A, denoted by Prob$\{X \in A\}$.

A **random variable** $X(\omega)$ is a function of the underlying outcome $\omega \in \Omega$.

The random variable $X(\omega)$ has a **probability distribution** that is induced by the underlying probability measure μ and the function $X(\omega)$:

$$\text{Prob}(X \in A) = \int_{\mathcal{G}} \mu(\omega) d\omega \tag{8.1}$$

where $\mathcal{G}$ is the subset of Ω for which $X(\omega) \in A$.

We call this the induced probability distribution of random variable X.

8.2 Digression: What Does Probability Mean?

Before diving in, we'll say a few words about what probability theory means and how it connects to statistics.

These are topics that are also touched on in the quantecon lectures https://python.quantecon.org/prob_meaning.html and https://python.quantecon.org/navy_captain.html.

For much of this lecture we'll be discussing fixed "population" probabilities.

These are purely mathematical objects.

To appreciate how statisticians connect probabilities to data, the key is to understand the following concepts:

- A single draw from a probability distribution
- Repeated independently and identically distributed (i.i.d.) draws of "samples" or "realizations" from the same probability distribution
- A **statistic** defined as a function of a sequence of samples
- An **empirical distribution** or **histogram** (a binned empirical distribution) that records observed **relative frequencies**
- The idea that a population probability distribution is what we anticipate **relative frequencies** will be in a long sequence of i.i.d. draws. Here the following mathematical machinery makes precise what is meant by **anticipated relative frequencies**
  - **Law of Large Numbers (LLN)**
  - **Central Limit Theorem (CLT)**

Scalar example

Consider the following discrete distribution

$$X \sim \{f_i\}_{i=0}^{I-1}, \quad f_i \geqslant 0, \quad \sum_i f_i = 1$$

Draw a sample $x_0, x_1, \dots, x_{N-1}$, N draws of X from $\{f_i\}_{i=1}^I$.

What do the "identical" and "independent" mean in IID or iid ("identically and independently distributed)?

- "identical" means that each draw is from the same distribution.
- "independent" means that the joint distribution equal tthe product of marginal distributions, i.e.,

$$\begin{aligned}\text{Prob}\{x_0 = i_0, x_1 = i_1, \dots, x_{N-1} = i_{N-1}\} &= \text{Prob}\{x_0 = i_0\} \cdot \dots \cdot \text{Prob}\{x_{I-1} = i_{I-1}\} \\ &= f_{i_0} f_{i_1} \cdot \dots \cdot f_{i_{N-1}}\end{aligned}$$

Consider the **empirical distribution**:

$$\begin{aligned} i &= 0, \dots, I-1, \\ N_i &= \text{number of times } X = i, \\ N &= \sum_{i=0}^{I-1} N_i \quad \text{total number of draws}, \\ \tilde{f}_i &= \frac{N_i}{N} \sim \text{ frequency of draws for which } X = i \end{aligned}$$

Key ideas that justify connecting probability theory with statistics are laws of large numbers and central limit theorems

LLN:

- A Law of Large Numbers (LLN) states that $\tilde{f}_i \to f_i$ as $N \to \infty$

CLT:

- A Central Limit Theorem (CLT) describes a **rate** at which $\tilde{f}_i \to f_i$

Remarks

- For "frequentist" statisticians, **anticipated relative frequency** is **all** that a probability distribution means.
- But for a Bayesian it means something more or different.

8.3 Representing Probability Distributions

A probability distribution $\text{Prob}(X \in A)$ can be described by its **cumulative distribution function (CDF)**

$$F_X(x) = \text{Prob}\{X \leq x\}.$$

Sometimes, but not always, a random variable can also be described by **density function** $f(x)$ that is related to its CDF by

$$\text{Prob}\{X \in B\} = \int_{t \in B} f(t) dt$$

$$F(x) = \int_{-\infty}^{x} f(t) dt$$

Here B is a set of possible X's whose probability we want to compute.

When a probability density exists, a probability distribution can be characterized either by its CDF or by its density.

For a **discrete-valued** random variable

- the number of possible values of X is finite or countably infinite
- we replace a **density** with a **probability mass function**, a non-negative sequence that sums to one
- we replace integration with summation in the formula like (8.1) that relates a CDF to a probability mass function

In this lecture, we mostly discuss discrete random variables.

Doing this enables us to confine our tool set basically to linear algebra.

Later we'll briefly discuss how to approximate a continuous random variable with a discrete random variable.

8.4 Univariate Probability Distributions

We'll devote most of this lecture to discrete-valued random variables, but we'll say a few things about continuous-valued random variables.

8.4.1 Discrete random variable

Let X be a discrete random variable that takes possible values: $i = 0, 1, \dots, I - 1 = \bar{X}$.

Here, we choose the maximum index $I - 1$ because of how this aligns nicely with Python's index convention.

Define $f_i \equiv \text{Prob}\{X = i\}$ and assemble the non-negative vector

$$f = \begin{bmatrix} f_0 \\ f_1 \\ \vdots \\ f_{I-1} \end{bmatrix} \tag{8.2}$$

for which $f_i \in [0, 1]$ for each i and $\sum_{i=0}^{I-1} f_i = 1$.

This vector defines a **probability mass function**.

The distribution (8.2) has **parameters** $\{f_i\}_{i=0,1,\cdots,I-2}$ since $f_{I-1} = 1 - \sum_{i=0}^{I-2} f_i$.

These parameters pin down the shape of the distribution.

(Sometimes $I = \infty$.)

Such a "non-parametric" distribution has as many "parameters" as there are possible values of the random variable.

We often work with special distributions that are characterized by a small number parameters.

In these special parametric distributions,

$$f_i = g(i; \theta)$$

where θ is a vector of parameters that is of much smaller dimension than I.

Remarks:

- The concept of **parameter** is intimately related to the notion of **sufficient statistic**.
- Sufficient statistic are nonlinear function of a data set.
- Sufficient statistics are designed to summarize all **information** about the parameters that is contained in the big data set.
- They are important tools that AI uses to reduce the size of a **big data** set

- R. A. Fisher provided a sharp definition of **information** – see https://en.wikipedia.org/wiki/Fisher_information

An example of a parametric probability distribution is a **geometric distribution**.

It is described by

$$f_i = \text{Prob}\{X = i\} = (1-\lambda)\lambda^i, \quad \lambda \in [0,1], \quad i = 0, 1, 2, \ldots$$

Evidently, $\sum_{i=0}^{\infty} f_i = 1$.

Let θ be a vector of parameters of the distribution described by f, then

$$f_i(\theta) \geq 0, \sum_{i=0}^{\infty} f_i(\theta) = 1$$

8.4.2 Continuous random variable

Let X be a continous random variable that takes values $X \in \tilde{X} \equiv [X_U, X_L]$ whose distributions have parameters θ.

$$\text{Prob}\{X \in A\} = \int_{x \in A} f(x;\theta)\, dx; \quad f(x;\theta) \geq 0$$

where A is a subset of $\tilde{X}$ and

$$\text{Prob}\{X \in \tilde{X}\} = 1$$

8.5 Bivariate Probability Distributions

We'll now discuss a bivariate **joint distribution**.

To begin, we restrict ourselves to two discrete random variables.

Let X, Y be two discrete random variables that take values:

$$X \in \{0, \ldots, J-1\}$$

$$Y \in \{0, \ldots, J-1\}$$

Then their **joint distribution** is described by a matrix

$$F_{I \times J} = [f_{ij}]_{i \in \{0,\ldots,J-1\}, j \in \{0,\ldots,J-1\}}$$

whose elements are

$$f_{ij} = \text{Prob}\{X = i, Y = j\} \geq 0$$

where

$$\sum_i \sum_j f_{ij} = 1$$

8.6 Marginal Probability Distributions

The joint distribution induce marginal distributions

$$
\textrm{Prob}\{X=i\}=\sum_{j=0}^{J-1}f_{ij}=\mu_i, i=0,\dots,I-1,
$$

$$
\textrm{Prob}\{Y=j\}=\sum_{i=0}^{I-1}f_{ij}=\nu_i, i=0,\dots,J-1
$$

For example, let the joint distribution over (X, Y) be

$$
F=\begin{bmatrix} .25 & .1 \\ .15 & .5 \end{bmatrix} \tag{8.3}
$$

Then marginal distributions are:

$$
\begin{aligned}
\textrm{Prob}\{X=0\} &= .25+.1=.35 \\
\textrm{Prob}\{X=1\} &= .15+.5=.65 \\
\textrm{Prob}\{Y=0\} &= .25+.15=.4 \\
\textrm{Prob}\{Y=1\} &= .1+.5=.6
\end{aligned}
$$

Digression: If two random variables X, Y are continuous and have joint density $f(x, y)$, then marginal distributions can be computed by

$$
\begin{aligned}
f(x) &= \int_{\mathbb{R}} f(x,y)dy \\
f(y) &= \int_{\mathbb{R}} f(x,y)dx
\end{aligned}
$$

8.7 Conditional Probability Distributions

Conditional probabilities are defined according to

$$
\textrm{Prob}\{A \mid B\}=\frac{\textrm{Prob}\{A\cap B\}}{\textrm{Prob}\{B\}}
$$

where A, B are two events.

For a pair of discrete random variables, we have the **conditional distribution**

$$
\textrm{Prob}\{X=i|Y=j\}=\frac{f_{ij}}{\sum_i f_{ij}}=\frac{\textrm{Prob}\{X=i,Y=j\}}{\textrm{Prob}\{Y=j\}}
$$

where $i=0,\dots,I-1, \quad j=0,\dots,J-1$.

Note that

$$
\sum_i \textrm{Prob}\{X_i=i|Y_j=j\}=\frac{\sum_i f_{ij}}{\sum_i f_{ij}}=1
$$

Remark: The mathematics of conditional probability implies **Bayes' Law**:

$$
\textrm{Prob}\{X=i|Y=j\}=\frac{\textrm{Prob}\{X=i,Y=j\}}{\textrm{Prob}\{Y=j\}}=\frac{\textrm{Prob}\{Y=j|X=i\}\textrm{Prob}\{X=i\}}{\textrm{Prob}\{Y=j\}}
$$

For the joint distribution (8.3)

$$
\textrm{Prob}\{X=0|Y=1\}=\frac{.1}{.1+.5}=\frac{.1}{.6}
$$

8.8 Statistical Independence

Random variables X and Y are statistically **independent** if

$$\text{Prob}\{X = i, Y = j\} = f_i g_i$$

where

$$\begin{aligned}\text{Prob}\{X = i\} = f_i \geq 0 \sum f_i = 1 \\ \text{Prob}\{Y = j\} = g_j \geq 0 \sum g_j = 1\end{aligned}$$

Conditional distributions are

$$\begin{aligned}\text{Prob}\{X = i | Y = j\} = \frac{f_i g_i}{\sum_i f_i g_j} = \frac{f_i g_i}{g_i} = f_i \\ \text{Prob}\{Y = j | X = i\} = \frac{f_i g_i}{\sum_j f_i g_j} = \frac{f_i g_i}{f_i} = g_i\end{aligned}$$

8.9 Means and Variances

The mean and variance of a discrete random variable X are

$$\begin{aligned}\mu_X \equiv \mathbb{E}[X] = \sum_k k\text{Prob}\{X = k\} \\ \sigma_X^2 \equiv \mathbb{D}[X] = \sum_k (k - \mathbb{E}[X])^2 \text{Prob}\{X = k\}\end{aligned}$$

A continuous random variable having density $f_X(x)$) has mean and variance

$$\begin{aligned}\mu_X \equiv \mathbb{E}[X] = \int_{-\infty}^{\infty} x f_X(x) dx \\ \sigma_X^2 \equiv \mathbb{D}[X] = \text{E}\left[(X - \mu_X)^2\right] = \int_{-\infty}^{\infty} (x - \mu_X)^2 f_X(x) dx\end{aligned}$$

8.10 Classic Trick for Generating Random Numbers

Suppose we have at our disposal a pseudo random number that draws a uniform random variable, i.e., one with probability distribution

$$\text{Prob}\{\tilde{X} = i\} = \frac{1}{I}, \quad i = 0, \ldots, I - 1$$

How can we transform $\tilde{X}$ to get a random variable X for which $\text{Prob}\{X = i\} = f_i, \quad i = 0, \ldots, I - 1$, where f_i is an arbitary discrete probability distribution on $i = 0, 1, \ldots, I - 1$?

The key tool is the inverse of a cumulative distribution function (CDF).

Observe that the CDF of a distribution is monotone and non-decreasing, taking values between 0 and 1.

We can draw a sample of a random variable X with a known CDF as follows:

- draw a random variable u from a uniform distribution on $[0, 1]$
- pass the sample value of u into the **"inverse"** target CDF for X

- X has the target CDF

Thus, knowing the **"inverse"** CDF of a distribution is enough to simulate from this distribution.

NOTE: The "inverse" CDF needs to exist for this method to work.

The inverse CDF is

$$F^{-1}(u) \equiv \inf\{x \in \mathbb{R} : F(x) \geq u\} \quad (0 < u < 1)$$

Here we use infimum because a CDF is a non-decreasing and right-continuous function.

Thus, suppose that

- U is a uniform random variable $U \in [0, 1]$
- We want to sample a random variable X whose CDF is F.

It turns out that if we use draw uniform random numbers U and then compute X from

$$X = F^{-1}(U),$$

then X ia a random variable with CDF $F_X(x) = F(x) = \text{Prob}\{X \leq x\}$.

We'll verify this in the special case in which F is continuous and bijective so that its inverse function exists andcan be denoted by F^{-1}.

Note that

$$\begin{aligned} F_X(x) &= \text{Prob}\{X \leq x\} \\ &= \text{Prob}\{F^{-1}(U) \leq x\} \\ &= \text{Prob}\{U \leq F(x)\} \\ &= F(x) \end{aligned}$$

where the last equality occurs because U is distributed uniformly on $[0, 1]$ while $F(x)$ is a constant given x that also lies on $[0, 1]$.

Let's use `numpy` to compute some examples.

Example: A continuous geometric (exponential) distribution

Let X follow a geometric distribution, with parameter $\lambda > 0$.

Its density function is

$$f(x) = \lambda e^{-\lambda x}$$

Its CDF is

$$F(x) = \int_0^\infty \lambda e^{-\lambda x} = 1 - e^{-\lambda x}$$

Let U follow a uniform distribution on $[0, 1]$.

X is a random variable such that $U = F(X)$.

The distribution X can be deduced from

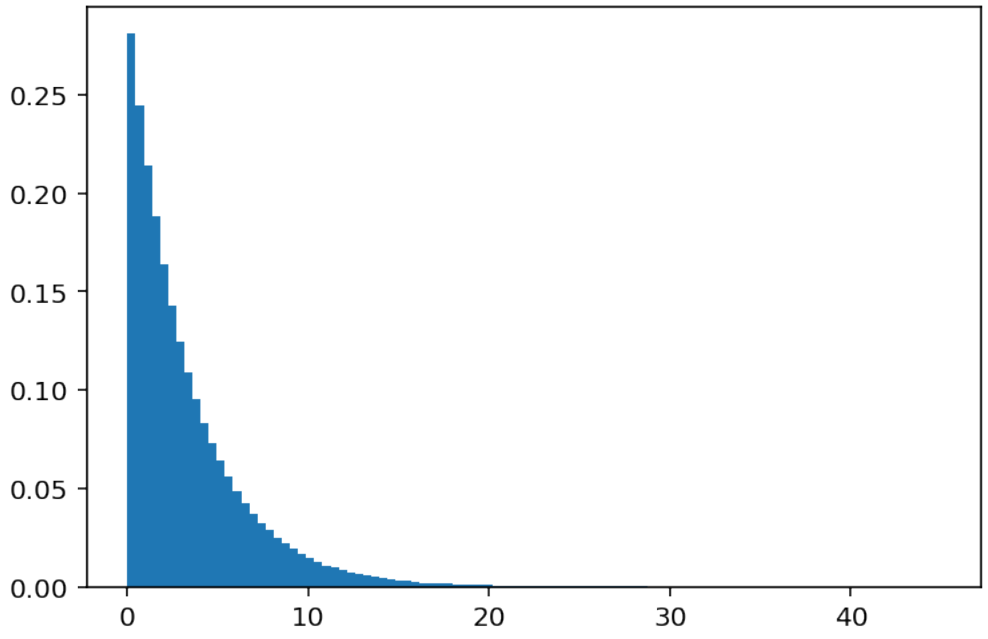
$$\begin{aligned} & U = F(X) = 1 - e^{-\lambda X} \\ \implies & \quad -U = e^{-\lambda X} \\ \implies & \quad \log(1 - U) = -\lambda X \\ \implies & \quad X = \frac{(1 - U)}{-\lambda} \end{aligned}$$

Let's draw u from $U[0,1]$ and calculate $x = \frac{log(1-U)}{-\lambda}$.

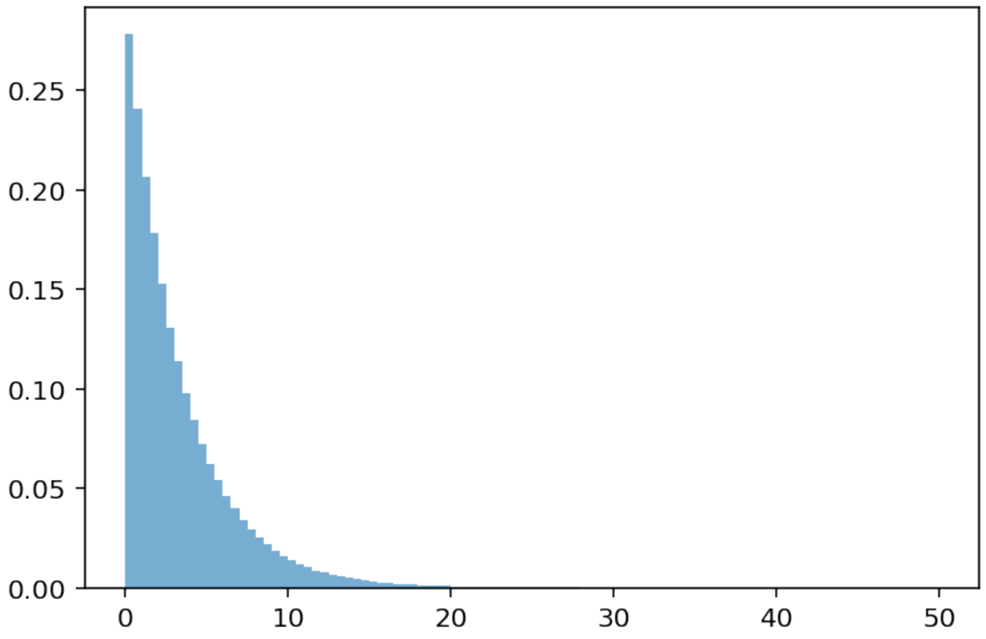
We'll check whether X seems to follow a **continuous geometric** (exponential) distribution.

Let's check with `numpy`.

```
n, λ = 1_000_000, 0.3

# draw uniform numbers
u = np.random.rand(n)

# transform
x = -np.log(1-u)/λ

# draw geometric distributions
x_g = np.random.exponential(1 / λ, n)

# plot and compare
plt.hist(x, bins=100, density=True)
plt.show()
```



```
plt.hist(x_g, bins=100, density=True, alpha=0.6)
plt.show()
```



Geometric distribution

Let X distributed geometrically, that is

$$\text{Prob}(X = i) = (1-\lambda)\lambda^i, \quad \lambda \in (0,1), \quad i = 0, 1, ...$$

$$\sum_{i=0}^{\infty} \text{Prob}(X = i) = 1 \longleftrightarrow (1-\lambda)\sum_{i=0}^{\infty}\lambda^i = \frac{1-\lambda}{1-\lambda} = 1$$

Its CDF is given by

$$\begin{aligned}\text{Prob}(X \le i) &= (1-\lambda)\sum_{j=0}^{i}\lambda^i \\ &= (1-\lambda)[\frac{1-\lambda^{i+1}}{1-\lambda}] \\ &= 1-\lambda^{i+1} \\ &= F(X) = F_i\end{aligned}$$

Again, let $\tilde{U}$ follow a uniform distribution and we want to find X such that $F(X) = \tilde{U}$.

Let's deduce the distribution of X from

$$\begin{aligned}\tilde{U} &= F(X) = 1-\lambda^{x+1} \\ 1-\tilde{U} &= \lambda^{x+1} \\ log(1-\tilde{U}) &= (x+1)\log\lambda \\ \frac{\log(1-\tilde{U})}{\log\lambda} &= x+1 \\ \frac{\log(1-\tilde{U})}{\log\lambda} - 1 &= x\end{aligned}$$

However, $\tilde{U} = F^{-1}(X)$ may not be an integer for any $x \geq 0$.

So let

$$x = \lceil \frac{\log(1-\tilde{U})}{\log \lambda} - 1 \rceil$$

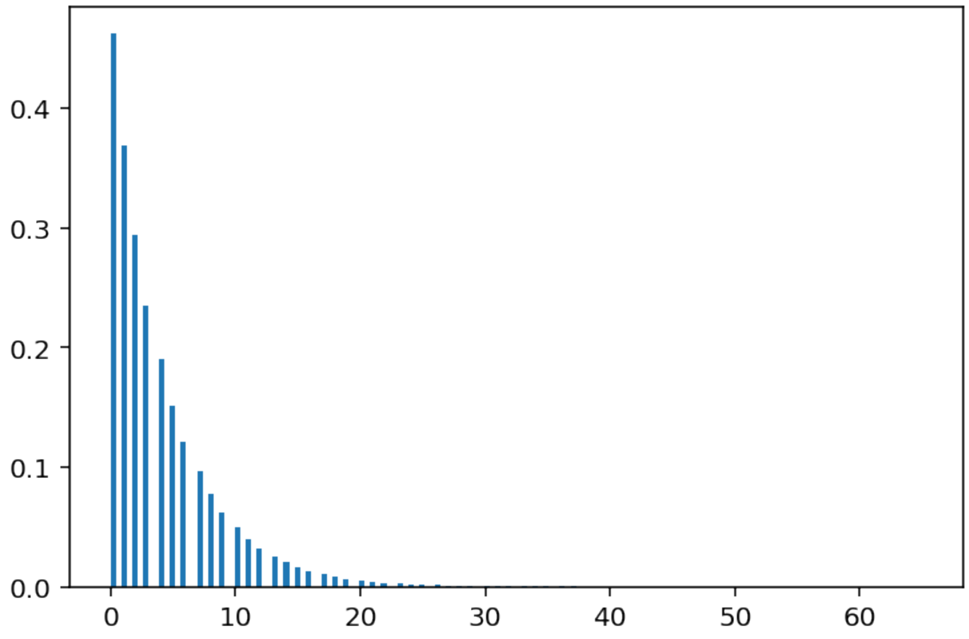
where $\lceil . \rceil$ is the ceiling function.

Thus x is the smallest integer such that the discrete geometric CDF is greater than or equal to $\tilde{U}$.

We can verify that x is indeed geometrically distributed by the following `numpy` program.

Note: The exponential distribution is the continuous analog of geometric distribution.

```
n, λ = 1_000_000, 0.8

# draw uniform numbers
u = np.random.rand(n)

# transform
x = np.ceil(np.log(1-u)/np.log(λ) - 1)

# draw geometric distributions
x_g = np.random.geometric(1-λ, n)

# plot and compare
plt.hist(x, bins=150, density=True)
plt.show()
```



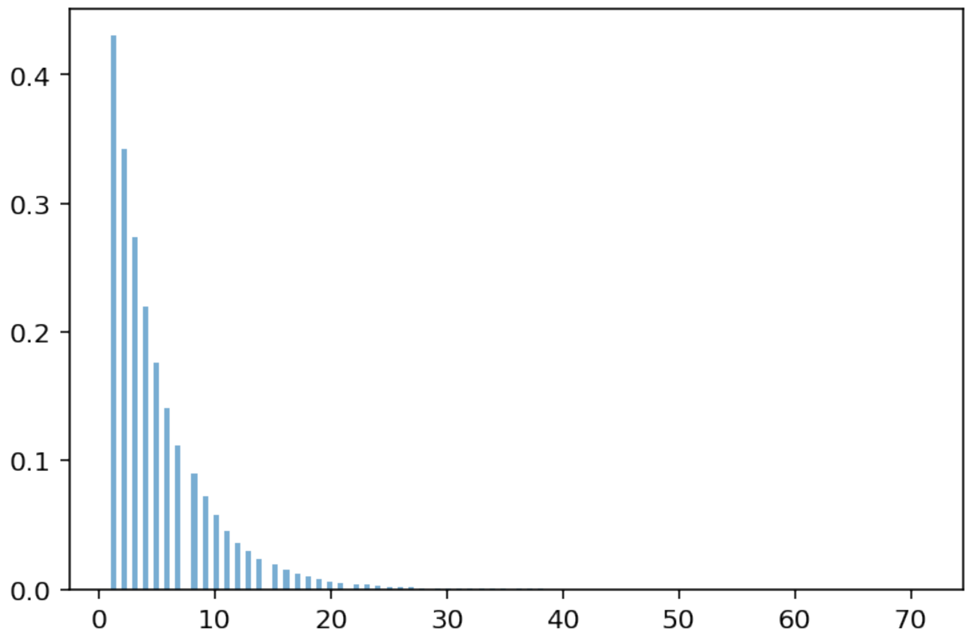
```
np.random.geometric(1-λ, n).max()
```

```
56
```

```
np.log(0.4)/np.log(0.3)
```

```
0.7610560044063083
```

```
plt.hist(x_g, bins=150, density=True, alpha=0.6)
plt.show()
```



8.11 Some Discrete Probability Distributions

Let's write some Python code to compute means and variances of some univariate random variables.

We'll use our code to

- compute population means and variances from the probability distribution
- generate a sample of N independently and identically distributed draws and compute sample means and variances
- compare population and sample means and variances

8.12 Geometric distribution

$$\text{Prob}(X = k) = (1-p)^{k-1}p, k = 1, 2, \ldots$$

$\implies$

$$\mathbb{E}(X) = \frac{1}{p}$$
$$\mathbb{D}(X) = \frac{1-p}{p^2}$$

We draw observations from the distribution and compare the sample mean and variance with the theoretical results.

```
# specify parameters
p, n = 0.3, 1_000_000

# draw observations from the distribution
x = np.random.geometric(p, n)

# compute sample mean and variance
μ_hat = np.mean(x)
σ2_hat = np.var(x)

print("The sample mean is: ", μ_hat, "\nThe sample variance is: ", σ2_hat)

# compare with theoretical results
print("\nThe population mean is: ", 1/p)
print("The population variance is: ", (1-p)/(p**2))
```

```
The sample mean is:  3.333938
The sample variance is:  7.759151412156005

The population mean is:  3.3333333333333335
The population variance is:  7.777777777777778
```

8.12.1 Newcomb–Benford distribution

The **Newcomb–Benford law** fits many data sets, e.g., reports of incomes to tax authorities, in which the leading digit is more likely to be small than large.

See https://en.wikipedia.org/wiki/Benford%27s_law

A Benford probability distribution is

$$\text{Prob}\{X=d\} = \log_{10}(d+1) - \log_{10}(d) = \log_{10}\left(1+\frac{1}{d}\right)$$

where $d \in \{1, 2, \cdots, 9\}$ can be thought of as a **first digit** in a sequence of digits.

This is a well defined discrete distribution since we can verify that probabilities are nonnegative and sum to 1.

$$\log_{10}\left(1+\frac{1}{d}\right) \geq 0, \quad \sum_{d=1}^{9} \log_{10}\left(1+\frac{1}{d}\right) = 1$$

The mean and variance of a Benford distribution are

$$\mathbb{E}\left[X\right] = \sum_{d=1}^{9} d \log_{10}\left(1+\frac{1}{d}\right) \simeq 3.4402$$

$$\mathbb{V}\left[X\right] = \sum_{d=1}^{9} \left(d - \mathbb{E}\left[X\right]\right)^2 \log_{10}\left(1+\frac{1}{d}\right) \simeq 6.0565$$

We verify the above and compute the mean and variance using `numpy`.

```
Benford_pmf = np.array([np.log10(1+1/d) for d in range(1,10)])
k = np.array(range(1,10))

# mean
```

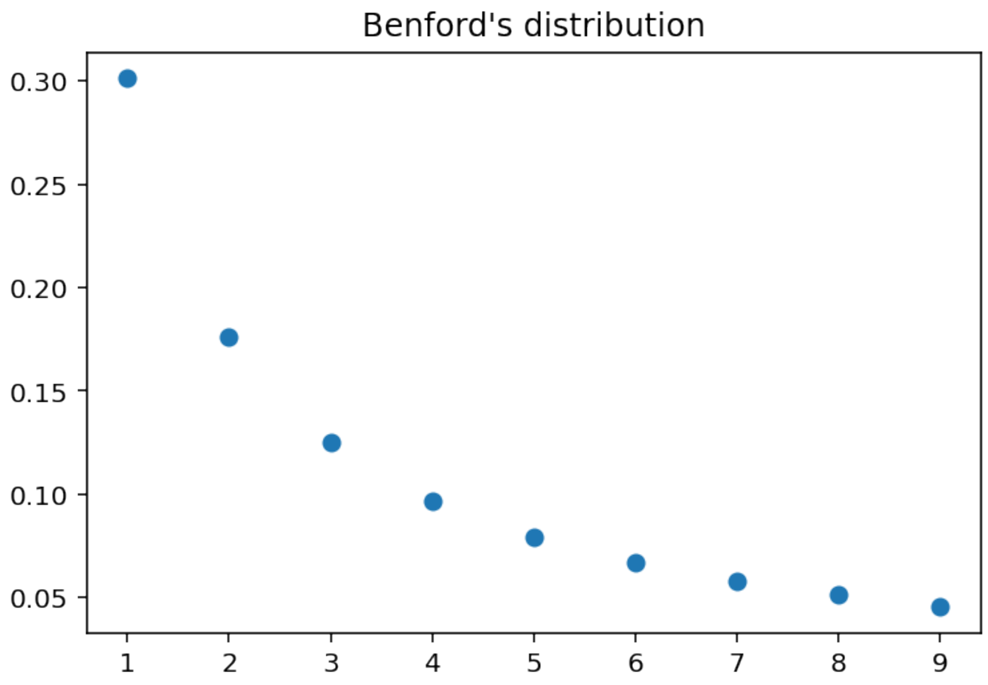
(continues on next page)

(continued from previous page)

```
mean = np.sum(Benford_pmf * k)

# variance
var = np.sum([(k-mean)**2 * Benford_pmf])

# verify sum to 1
print(np.sum(Benford_pmf))
print(mean)
print(var)
```

```
0.9999999999999999
3.440236967123206
6.056512631375667
```

```
# plot distribution
plt.plot(range(1,10), Benford_pmf, 'o')
plt.title('Benford\'s distribution')
plt.show()
```



8.12.2 Pascal (negative binomial) distribution

Consider a sequence of independent Bernoulli trials.

Let p be the probability of success.

Let X be a random variable that represents the number of failures before we get r success.

Its distribution is

$$X \sim NB(r, p)$$
$$\text{Prob}(X = k; r, p) = \binom{k+r-1}{r-1} p^r (1-p)^k$$

Here, we choose from among $k + r - 1$ possible outcomes because the last draw is by definition a success.

We compute the mean and variance to be

$$\mathbb{E}(X) = \frac{k(1-p)}{p}$$
$$\mathbb{V}(X) = \frac{k(1-p)}{p^2}$$

```
# specify parameters
r, p, n = 10, 0.3, 1_000_000

# draw observations from the distribution
x = np.random.negative_binomial(r, p, n)

# compute sample mean and variance
μ_hat = np.mean(x)
σ2_hat = np.var(x)

print("The sample mean is: ", μ_hat, "\nThe sample variance is: ", σ2_hat)
print("\nThe population mean is: ", r*(1-p)/p)
print("The population variance is: ", r*(1-p)/p**2)
```

```
The sample mean is:  23.33375
The sample variance is:  77.91651893750002

The population mean is:  23.333333333333336
The population variance is:  77.77777777777779
```

8.13 Continuous Random Variables

8.13.1 Univariate Gaussian distribution

We write

$$X \sim N(\mu, \sigma^2)$$

to indicate the probability distribution

$$f(x|u, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{[-\frac{1}{2\sigma^2}(x-u)^2]}$$

In the below example, we set $\mu = 0, \sigma = 0.1$.

```
# specify parameters
μ, σ = 0, 0.1

# specify number of draws
n = 1_000_000

# draw observations from the distribution
x = np.random.normal(μ, σ, n)

# compute sample mean and variance
μ_hat = np.mean(x)
σ_hat = np.std(x)

print("The sample mean is: ", μ_hat)
print("The sample standard deviation is: ", σ_hat)
```

```
The sample mean is:  -2.6699866495693146e-06
The sample standard deviation is:  0.09988310440282286
```

```
# compare
print(μ-μ_hat < 1e-3)
print(σ-σ_hat < 1e-3)
```

```
True
True
```

8.13.2 Uniform Distribution

$$X \sim U[a,b]$$
$$f(x) = \begin{cases} \frac{1}{b-a}, & a \le x \le b \\ 0, & \text{otherwise} \end{cases}$$

The population mean and variance are

$$\mathbb{E}(X) = \frac{a+b}{2}$$
$$\mathbb{V}(X) = \frac{(b-a)^2}{12}$$

```
# specify parameters
a, b = 10, 20

# specify number of draws
n = 1_000_000

# draw observations from the distribution
x = a + (b-a)*np.random.rand(n)

# compute sample mean and variance
μ_hat = np.mean(x)
σ2_hat = np.var(x)
```

(continues on next page)

(continued from previous page)

```python
print("The sample mean is: ", μ_hat, "\nThe sample variance is: ", σ2_hat)
print("\nThe population mean is: ", (a+b)/2)
print("The population variance is: ", (b-a)**2/12)
```

```
The sample mean is:  15.00222274370156
The sample variance is:  8.339607328148443

The population mean is:  15.0
The population variance is:  8.333333333333334
```

8.14 A Mixed Discrete-Continuous Distribution

We'll motivate this example with a little story.

Suppose that to apply for a job you take an interview and either pass or fail it.

You have 5% chance to pass an interview and you know your salary will uniformly distributed in the interval 300~400 a day only if you pass.

We can describe your daily salary as a discrete-continuous variable with the following probabilities:

$$P(X=0)=0.95$$

$$P(300\le X\le 400)=\int_{300}^{400} f(x)\,dx=0.05$$

$$f(x)=0.0005$$

Let's start by generating a random sample and computing sample moments.

```python
x = np.random.rand(1_000_000)
# x[x > 0.95] = 100*x[x > 0.95]+300
x[x > 0.95] = 100*np.random.rand(len(x[x > 0.95]))+300
x[x <= 0.95] = 0

μ_hat = np.mean(x)
σ2_hat = np.var(x)

print("The sample mean is: ", μ_hat, "\nThe sample variance is: ", σ2_hat)
```

```
The sample mean is:  17.548232806538643
The sample variance is:  5877.121811309432
```

The analytical mean and variance can be computed:

$$\begin{aligned}\mu &= \int_{300}^{400} xf(x)dx \\ &= 0.0005\int_{300}^{400} xdx \\ &= 0.0005\times \frac{1}{2}x^2\bigg|_{300}^{400}\end{aligned}$$

$$\begin{aligned}\sigma^2 &= 0.95 \times (0-17.5)^2 + \int_{300}^{400} (x-17.5)^2 f(x) dx \\ &= 0.95 \times 17.5^2 + 0.0005 \int_{300}^{400} (x-17.5)^2 dx \\ &= 0.95 \times 17.5^2 + 0.0005 \times \frac{1}{3}(x-17.5)^3 \bigg|_{300}^{400}\end{aligned}$$

```
mean = 0.0005*0.5*(400**2 - 300**2)
var = 0.95*17.5**2+0.0005/3*((400-17.5)**3-(300-17.5)**3)
print("mean: ", mean)
print("variance: ", var)
```

```
mean:  17.5
variance:  5860.416666666666
```

8.15 Matrix Representation of Some Bivariate Distributions

Let's use matrices to represent a joint distribution, conditional distribution, marginal distribution, and the mean and variance of a bivariate random variable.

The table below illustrates a probability distribution for a bivariate random variable.

$$F = [f_{ij}] = \begin{bmatrix} 0.3 & 0.2 \\ 0.1 & 0.4 \end{bmatrix}$$

Marginal distributions are

$$\text{Prob}(X = i) = \sum_j f_{ij} = u_i$$

$$\text{Prob}(Y = j) = \sum_i f_{ij} = v_j$$

Below we draw some samples confirm that the "sampling" distribution agrees well with the "population" distribution.

Sample results:

```
# specify parameters
xs = np.array([0, 1])
ys = np.array([10, 20])
f = np.array([[0.3, 0.2], [0.1, 0.4]])
f_cum = np.cumsum(f)

# draw random numbers
p = np.random.rand(1_000_000)
x = np.vstack([xs[1]*np.ones(p.shape), ys[1]*np.ones(p.shape)])
# map to the bivariate distribution

x[0, p < f_cum[2]] = xs[1]
x[1, p < f_cum[2]] = ys[0]

x[0, p < f_cum[1]] = xs[0]
x[1, p < f_cum[1]] = ys[1]
```

(continues on next page)

(continued from previous page)

```
x[0, p < f_cum[0]] = xs[0]
x[1, p < f_cum[0]] = ys[0]
print(x)
```

```
[[ 0.  0.  0. ...  0.  1.  0.]
 [10. 20. 10. ... 10. 20. 10.]]
```

Here, we use exactly the inverse CDF technique to generate sample from the joint distribution F.

```
# marginal distribution
xp = np.sum(x[0, :] == xs[0])/1_000_000
yp = np.sum(x[1, :] == ys[0])/1_000_000

# print output
print("marginal distribution for x")
xmtb = pt.PrettyTable()
xmtb.field_names = ['x_value', 'x_prob']
xmtb.add_row([xs[0], xp])
xmtb.add_row([xs[1], 1-xp])
print(xmtb)

print("\nmarginal distribution for y")
ymtb = pt.PrettyTable()
ymtb.field_names = ['y_value', 'y_prob']
ymtb.add_row([ys[0], yp])
ymtb.add_row([ys[1], 1-yp])
print(ymtb)
```

```
marginal distribution for x
+---------+----------+
| x_value |  x_prob  |
+---------+----------+
|    0    | 0.499959 |
|    1    | 0.500041 |
+---------+----------+

marginal distribution for y
+---------+----------+
| y_value |  y_prob  |
+---------+----------+
|    10   | 0.399779 |
|    20   | 0.600221 |
+---------+----------+
```

```
# conditional distributions
xc1 = x[0, x[1, :] == ys[0]]
xc2 = x[0, x[1, :] == ys[1]]
yc1 = x[1, x[0, :] == xs[0]]
yc2 = x[1, x[0, :] == xs[1]]

xc1p = np.sum(xc1 == xs[0])/len(xc1)
xc2p = np.sum(xc2 == xs[0])/len(xc2)
yc1p = np.sum(yc1 == ys[0])/len(yc1)
yc2p = np.sum(yc2 == ys[0])/len(yc2)
```

(continues on next page)

(continued from previous page)

```python
# print output
print("conditional distribution for x")
xctb = pt.PrettyTable()
xctb.field_names = ['y_value', 'prob(x=0)', 'prob(x=1)']
xctb.add_row([ys[0], xc1p, 1-xc1p])
xctb.add_row([ys[1], xc2p, 1-xc2p])
print(xctb)

print("\nconditional distribution for y")
yctb = pt.PrettyTable()
yctb.field_names = ['x_value',  'prob(y=10)', 'prob(y=20)']
yctb.add_row([xs[0], yc1p, 1-yc1p])
yctb.add_row([xs[1], yc2p, 1-yc2p])
print(yctb)
```

```
conditional distribution for x
+---------+--------------------+---------------------+
| y_value |     prob(x=0)      |      prob(x=1)      |
+---------+--------------------+---------------------+
|    10   | 0.7501469561932967 | 0.24985304380670326 |
|    20   | 0.3333205602603041 |  0.6666794397396959 |
+---------+--------------------+---------------------+

conditional distribution for y
+---------+---------------------+---------------------+
| x_value |      prob(y=10)     |      prob(y=20)     |
+---------+---------------------+---------------------+
|    0    |  0.5998351864852918 | 0.40016481351470823 |
|    1    | 0.19975562003915678 |  0.8002443799608432 |
+---------+---------------------+---------------------+
```

Let's calculate population marginal and conditional probabilities using matrix algebra.

$$\left[\begin{array}{ccccccc} & \vdots & y_1 & y_2 & \vdots & x \\ \cdots & \vdots & \cdots & \cdots & \vdots & \cdots \\ x_1 & \vdots & 0.3 & 0.2 & \vdots & 0.5 \\ x_2 & \vdots & 0.1 & 0.4 & \vdots & 0.5 \\ \cdots & \vdots & \cdots & \cdots & \vdots & \cdots \\ y & \vdots & 0.4 & 0.6 & \vdots & 1 \end{array}\right]$$

$\Longrightarrow$

(1) Marginal distribution:

$$\left[\begin{array}{cccc} var & \vdots & var_1 & var_2 \\ \cdots & \vdots & \cdots & \cdots \\ x & \vdots & 0.5 & 0.5 \\ \cdots & \vdots & \cdots & \cdots \\ y & \vdots & 0.4 & 0.6 \end{array}\right]$$

(2) Conditional distribution:

$$\left[\begin{array}{cccc} x & \vdots & x_1 & x_2 \\ \cdots\cdots\cdots & \vdots & \cdots\cdots\cdots & \cdots\cdots\cdots \\ y=y_1 & \vdots & \frac{0.3}{0.4}=0.75 & \frac{0.1}{0.4}=0.25 \\ \cdots\cdots\cdots & \vdots & \cdots\cdots\cdots & \cdots\cdots\cdots \\ y=y_2 & \vdots & \frac{0.2}{0.6}\approx 0.33 & \frac{0.4}{0.6}\approx 0.67 \end{array}\right]$$

$$\left[\begin{array}{ccccc} y & \vdots & y_1 & & y_2 \\ \cdots\cdots\cdots & \vdots & \cdots\cdots\cdots & & \cdots\cdots\cdots \\ x=x_1 & \vdots & \frac{0.3}{0.5}=0.6 & & \frac{0.2}{0.5}=0.4 \\ \cdots\cdots\cdots & \vdots & \cdots\cdots\cdots & & \cdots\cdots\cdots \\ x=x_2 & \vdots & \frac{0.1}{0.5}=0.2 & & \frac{0.4}{0.5}=0.8 \end{array}\right]$$

These population objects closely resemble sample counterparts computed above.

Let's wrap some of the functions we have used in a Python class for a general discrete bivariate joint distribution.

```python
class discrete_bijoint:

    def __init__(self, f, xs, ys):
        '''initialization
        -----------------
        parameters:
        f: the bivariate joint probability matrix
        xs: values of x vector
        ys: values of y vector
        '''
        self.f, self.xs, self.ys = f, xs, ys

    def joint_tb(self):
        '''print the joint distribution table'''
        xs = self.xs
        ys = self.ys
        f = self.f
        jtb = pt.PrettyTable()
        jtb.field_names = ['x_value/y_value', *ys, 'marginal sum for x']
        for i in range(len(xs)):
            jtb.add_row([xs[i], *f[i, :], np.sum(f[i, :])])
        jtb.add_row(['marginal_sum for y', *np.sum(f, 0), np.sum(f)])
        print("\nThe joint probability distribution for x and y\n", jtb)
        self.jtb = jtb

    def draw(self, n):
        '''draw random numbers
        ----------------------
        parameters:
        n: number of random numbers to draw
        '''
        xs = self.xs
        ys = self.ys
        f_cum = np.cumsum(self.f)
        p = np.random.rand(n)
        x = np.empty([2, p.shape[0]])
        lf = len(f_cum)
        lx = len(xs)-1
        ly = len(ys)-1
        for i in range(lf):
            x[0, p < f_cum[lf-1-i]] = xs[lx]
            x[1, p < f_cum[lf-1-i]] = ys[ly]
            if ly == 0:
                lx -= 1
                ly = len(ys)-1
            else:
                ly -= 1
        self.x = x
        self.n = n
```

(continues on next page)

(continued from previous page)

```
    def marg_dist(self):
        '''marginal distribution'''
        x = self.x
        xs = self.xs
        ys = self.ys
        n = self.n
        xmp = [np.sum(x[0, :] == xs[i])/n for i in range(len(xs))]
        ymp = [np.sum(x[1, :] == ys[i])/n for i in range(len(ys))]

        # print output
        xmtb = pt.PrettyTable()
        ymtb = pt.PrettyTable()
        xmtb.field_names = ['x_value', 'x_prob']
        ymtb.field_names = ['y_value', 'y_prob']
        for i in range(max(len(xs), len(ys))):
            if i < len(xs):
                xmtb.add_row([xs[i], xmp[i]])
            if i < len(ys):
                ymtb.add_row([ys[i], ymp[i]])
        xmtb.add_row(['sum', np.sum(xmp)])
        ymtb.add_row(['sum', np.sum(ymp)])
        print("\nmarginal distribution for x\n", xmtb)
        print("\nmarginal distribution for y\n", ymtb)

        self.xmp = xmp
        self.ymp = ymp

    def cond_dist(self):
        '''conditional distribution'''
        x = self.x
        xs = self.xs
        ys = self.ys
        n = self.n
        xcp = np.empty([len(ys), len(xs)])
        ycp = np.empty([len(xs), len(ys)])
        for i in range(max(len(ys), len(xs))):
            if i < len(ys):
                xi = x[0, x[1, :] == ys[i]]
                idx = xi.reshape(len(xi), 1) == xs.reshape(1, len(xs))
                xcp[i, :] = np.sum(idx, 0)/len(xi)
            if i < len(xs):
                yi = x[1, x[0, :] == xs[i]]
                idy = yi.reshape(len(yi), 1) == ys.reshape(1, len(ys))
                ycp[i, :] = np.sum(idy, 0)/len(yi)

        # print output
        xctb = pt.PrettyTable()
        yctb = pt.PrettyTable()
        xctb.field_names = ['x_value', *xs, 'sum']
        yctb.field_names = ['y_value', *ys, 'sum']
        for i in range(max(len(xs), len(ys))):
            if i < len(ys):
                xctb.add_row([ys[i], *xcp[i], np.sum(xcp[i])])
            if i < len(xs):
                yctb.add_row([xs[i], *ycp[i], np.sum(ycp[i])])
```

(continues on next page)

(continued from previous page)

```
        print("\nconditional distribution for x\n", xctb)
        print("\nconditional distribution for y\n", yctb)

        self.xcp = xcp
        self.xyp = ycp
```

Let's apply our code to some examples.

Example 1

```
# joint
d = discrete_bijoint(f, xs, ys)
d.joint_tb()
```

```
The joint probability distribution for x and y
 +--------------------+-----+--------------------+--------------------+
|  x_value/y_value   |  10 |         20         | marginal sum for x |
+--------------------+-----+--------------------+--------------------+
|         0          | 0.3 |        0.2         |        0.5         |
|         1          | 0.1 |        0.4         |        0.5         |
| marginal_sum for y | 0.4 | 0.6000000000000001 |        1.0         |
+--------------------+-----+--------------------+--------------------+
```

```
# sample marginal
d.draw(1_000_000)
d.marg_dist()
```

```
marginal distribution for x
 +---------+----------+
| x_value |  x_prob  |
+---------+----------+
|    0    | 0.499154 |
|    1    | 0.500846 |
|   sum   |   1.0    |
+---------+----------+
```

```
marginal distribution for y
 +---------+----------+
| y_value |  y_prob  |
+---------+----------+
|    10   | 0.398873 |
|    20   | 0.601127 |
|   sum   |   1.0    |
+---------+----------+
```

```
# sample conditional
d.cond_dist()
```

```
conditional distribution for x
 +---------+--------------------+---------------------+-----+
| x_value |         0          |          1          | sum |
+---------+--------------------+---------------------+-----+
```

(continues on next page)

(continued from previous page)

```
|    10    | 0.7498677523923655 | 0.25013224760763453 | 1.0 |
|    20    | 0.3327949002457051 |  0.6672050997542949 | 1.0 |
+----------+--------------------+---------------------+-----+

conditional distribution for y
 +----------+---------------------+--------------------+-----+
| y_value |          10         |         20         | sum |
+----------+---------------------+--------------------+-----+
|    0    |  0.5992178766472872 | 0.4007821233527128 | 1.0 |
|    1    | 0.19920494523266633 | 0.8007950547673337 | 1.0 |
+----------+---------------------+--------------------+-----+
```

Example 2

```
xs_new = np.array([10, 20, 30])
ys_new = np.array([1, 2])
f_new = np.array([[0.2, 0.1], [0.1, 0.3], [0.15, 0.15]])
d_new = discrete_bijoint(f_new, xs_new, ys_new)
d_new.joint_tb()
```

```
The joint probability distribution for x and y
 +--------------------+---------------------+------+---------------------+
|  x_value/y_value   |          1          |  2   |  marginal sum for x |
+--------------------+---------------------+------+---------------------+
|         10         |         0.2         | 0.1  | 0.30000000000000004 |
|         20         |         0.1         | 0.3  |         0.4         |
|         30         |         0.15        | 0.15 |         0.3         |
| marginal_sum for y | 0.45000000000000007 | 0.55 |         1.0         |
+--------------------+---------------------+------+---------------------+
```

```
d_new.draw(1_000_000)
d_new.marg_dist()
```

```
marginal distribution for x
 +----------+----------+
| x_value |  x_prob  |
+----------+----------+
|    10    | 0.29917  |
|    20    | 0.400588 |
|    30    | 0.300242 |
|   sum    |   1.0    |
+----------+----------+

marginal distribution for y
 +----------+----------+
| y_value |  y_prob  |
+----------+----------+
|    1     | 0.448634 |
|    2     | 0.551366 |
|   sum    |   1.0    |
+----------+----------+
```

```
d_new.cond_dist()
```

```
conditional distribution for x
 +---------+--------------------+---------------------+---------------------+-----+
| x_value |         10         |         20          |          30         | sum |
+---------+--------------------+---------------------+---------------------+-----+
|    1    | 0.4433257399127128 | 0.22248648118510858 |  0.3341877789021786 | 1.0 |
|    2    | 0.1818737462955641 |  0.5455051635392825 | 0.27262109016515346 | 1.0 |
+---------+--------------------+---------------------+---------------------+-----+

conditional distribution for y
 +---------+---------------------+--------------------+-----+
| y_value |          1          |         2          | sum |
+---------+---------------------+--------------------+-----+
|    10   |  0.664809305745897  | 0.335190694254103  | 1.0 |
|    20   | 0.24917121830908565 | 0.7508287816909144 | 1.0 |
|    30   |  0.4993571852039355 | 0.5006428147960645 | 1.0 |
+---------+---------------------+--------------------+-----+
```

8.16 A Continuous Bivariate Random Vector

A two-dimensional Gaussian distribution has joint density

$$f(x,y) = (2\pi\sigma_1\sigma_2\sqrt{1-\rho^2})^{-1}\exp\left[-\frac{1}{2(1-\rho^2)}\left(\frac{(x-\mu_1)^2}{\sigma_1^2}-\frac{2\rho(x-\mu_1)(y-\mu_2)}{\sigma_1\sigma_2}+\frac{(y-\mu_2)^2}{\sigma_2^2}\right)\right]$$

$$\frac{1}{2\pi\sigma_1\sigma_2\sqrt{1-\rho^2}}\exp\left[-\frac{1}{2(1-\rho^2)}\left(\frac{(x-\mu_1)^2}{\sigma_1^2}-\frac{2\rho(x-\mu_1)(y-\mu_2)}{\sigma_1\sigma_2}+\frac{(y-\mu_2)^2}{\sigma_2^2}\right)\right]$$

We start with a bivariate normal distribution pinned down by

$$\mu = \begin{bmatrix} 0 \\ 5 \end{bmatrix}, \quad \Sigma = \begin{bmatrix} 5 & .2 \\ .2 & 1 \end{bmatrix}$$

```python
# define the joint probability density function
def func(x, y, μ1=0, μ2=5, σ1=np.sqrt(5), σ2=np.sqrt(1), ρ=.2/np.sqrt(5*1)):
    A = (2 * np.pi * σ1 * σ2 * np.sqrt(1 - ρ**2))**(-1)
    B = -1 / 2 / (1 - ρ**2)
    C1 = (x - μ1)**2 / σ1**2
    C2 = 2 * ρ * (x - μ1) * (y - μ2) / σ1 / σ2
    C3 = (y - μ2)**2 / σ2**2
    return A * np.exp(B * (C1 - C2 + C3))
```

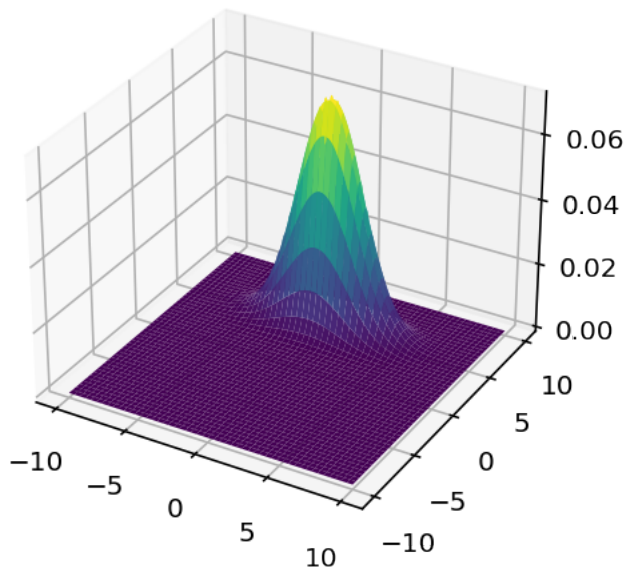
```python
μ1 = 0
μ2 = 5
σ1 = np.sqrt(5)
σ2 = np.sqrt(1)
ρ = .2 / np.sqrt(5 * 1)
```

```python
x = np.linspace(-10, 10, 1_000)
y = np.linspace(-10, 10, 1_000)
x_mesh, y_mesh = np.meshgrid(x, y, indexing="ij")
```

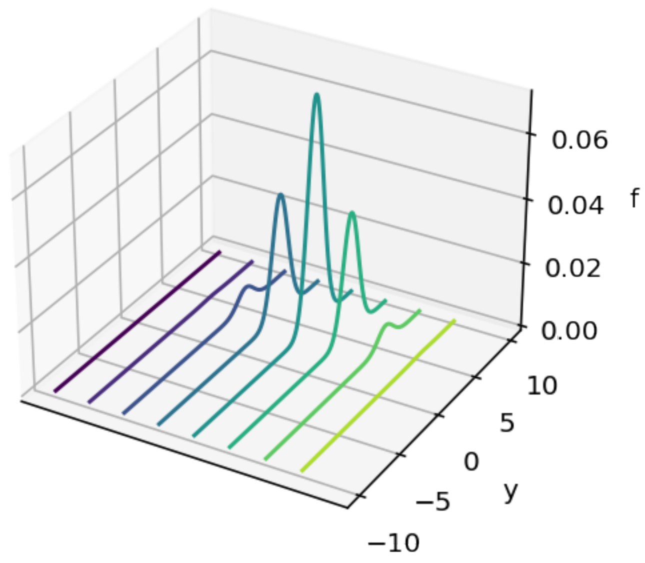
Joint Distribution

Let's plot the **population** joint density.

```
# %matplotlib notebook

fig = plt.figure()
ax = plt.axes(projection='3d')

surf = ax.plot_surface(x_mesh, y_mesh, func(x_mesh, y_mesh), cmap='viridis')
plt.show()
```



```
# %matplotlib notebook

fig = plt.figure()
ax = plt.axes(projection='3d')

curve = ax.contour(x_mesh, y_mesh, func(x_mesh, y_mesh), zdir='x')
plt.ylabel('y')
ax.set_zlabel('f')
ax.set_xticks([])
plt.show()
```



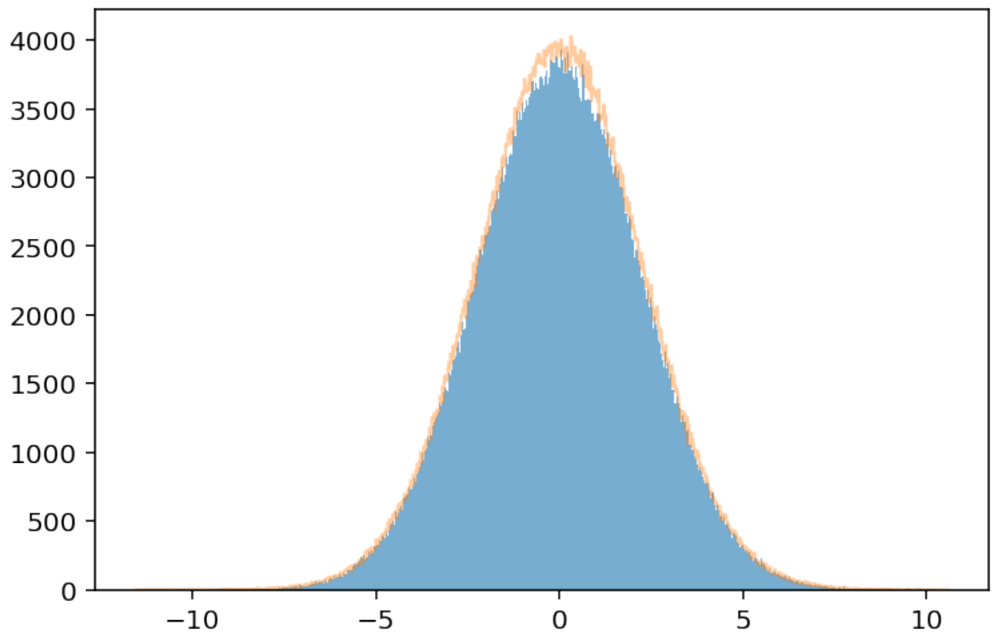
Next we can simulate from a built-in `numpy` function and calculate a **sample** marginal distribution from the sample mean and variance.

```
μ= np.array([0, 5])
σ= np.array([[5, .2], [.2, 1]])
n = 1_000_000
data = np.random.multivariate_normal(μ, σ, n)
x = data[:, 0]
y = data[:, 1]
```

Marginal distribution

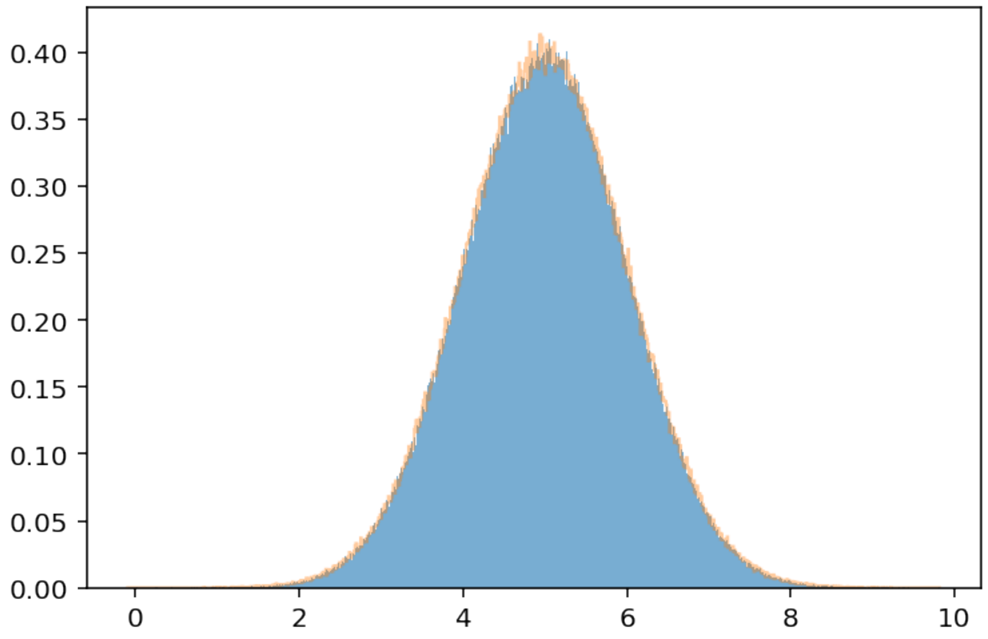
```
plt.hist(x, bins=1_000, alpha=0.6)
μx_hat, σx_hat = np.mean(x), np.std(x)
print(μx_hat, σx_hat)
x_sim = np.random.normal(μx_hat, σx_hat, 1_000_000)
plt.hist(x_sim, bins=1_000, alpha=0.4, histtype="step")
plt.show()
```

```
-0.00012943396296686 53 2.2338665663818036
```



```
plt.hist(y, bins=1_000, density=True, alpha=0.6)
μy_hat, σy_hat = np.mean(y), np.std(y)
print(μy_hat, σy_hat)
y_sim = np.random.normal(μy_hat, σy_hat, 1_000_000)
plt.hist(y_sim, bins=1_000, density=True, alpha=0.4, histtype="step")
plt.show()
```

```
4.998845851883271 1.0005847916711021
```



Conditional distribution

The population conditional distribution is

$$
\begin{aligned}
[X|Y=y] &\sim \mathbb{N}\left[\mu_X + \rho\sigma_X\frac{y-\mu_Y}{\sigma_Y}, \sigma_X^2(1-\rho^2)\right] \\
[Y|X=x] &\sim \mathbb{N}\left[\mu_Y + \rho\sigma_Y\frac{x-\mu_X}{\sigma_X}, \sigma_Y^2(1-\rho^2)\right]
\end{aligned}
$$

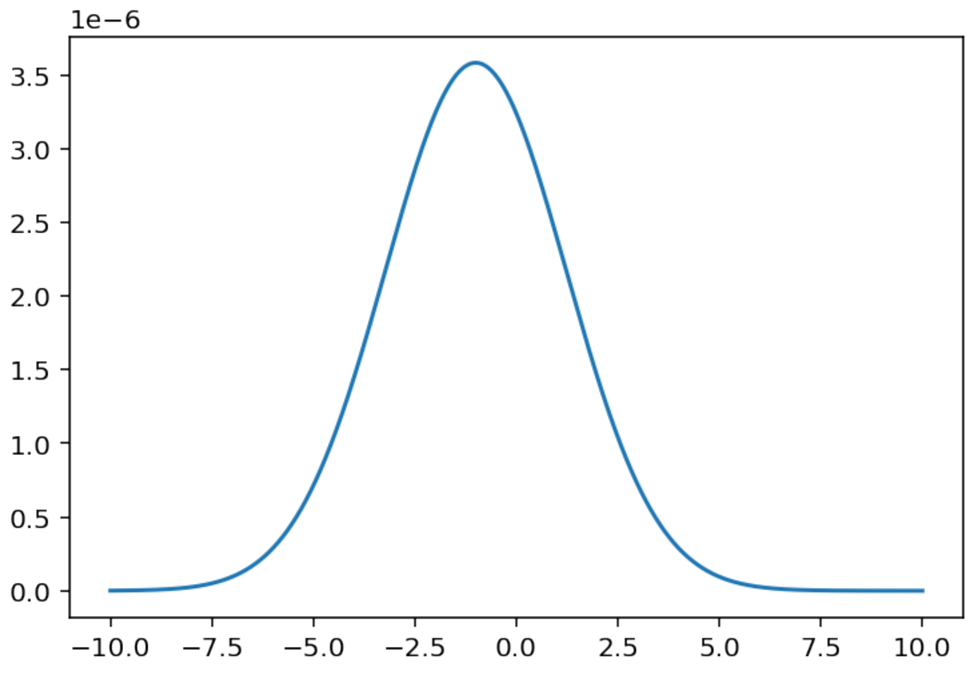
Let's approximate the joint density by discretizing and mapping the approximating joint density into a matrix.

We can compute the discretized marginal density by just using matrix algebra and noting that

$$
\text{Prob}\{X=i|Y=j\} = \frac{f_{ij}}{\sum_i f_{ij}}
$$

Fix $y=0$.

```python
# discretized marginal density
x = np.linspace(-10, 10, 1_000_000)
z = func(x, y=0) / np.sum(func(x, y=0))
plt.plot(x, z)
plt.show()
```

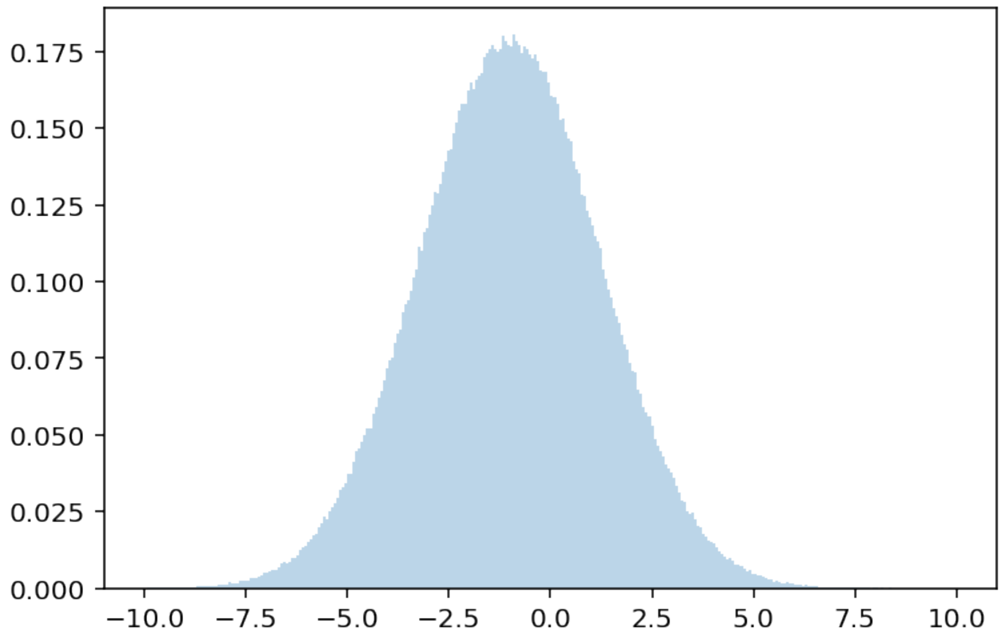


The mean and variance are computed by

$$
\begin{aligned}
\mathbb{E}\left[X|Y=j\right] &= \sum_i iProb\{X=i|Y=j\} = \sum_i i\frac{f_{ij}}{\sum_i f_{ij}} \\
\mathbb{D}\left[X|Y=j\right] &= \sum_i \left(i-\mu_{X|Y=j}\right)^2 \frac{f_{ij}}{\sum_i f_{ij}}
\end{aligned}
$$

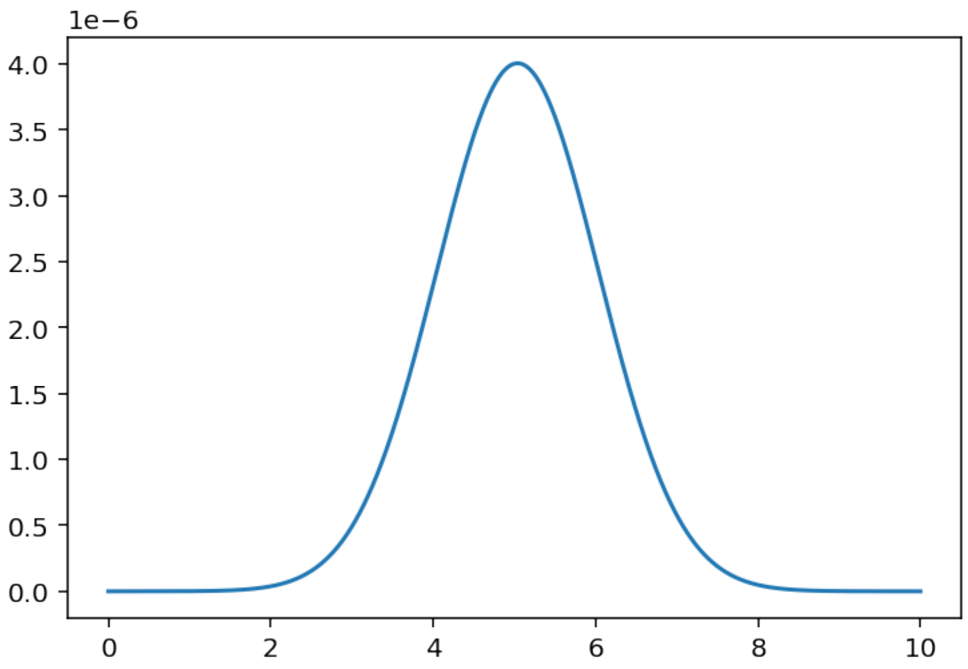
Let's draw from a normal distribution with above mean and variance and check how accurate our approximation is.

```
# discretized mean
μx = np.dot(x, z)

# discretized standard deviation
σx = np.sqrt(np.dot((x - μx)**2, z))

# sample
zz = np.random.normal(μx, σx, 1_000_000)
plt.hist(zz, bins=300, density=True, alpha=0.3, range=[-10, 10])
plt.show()
```

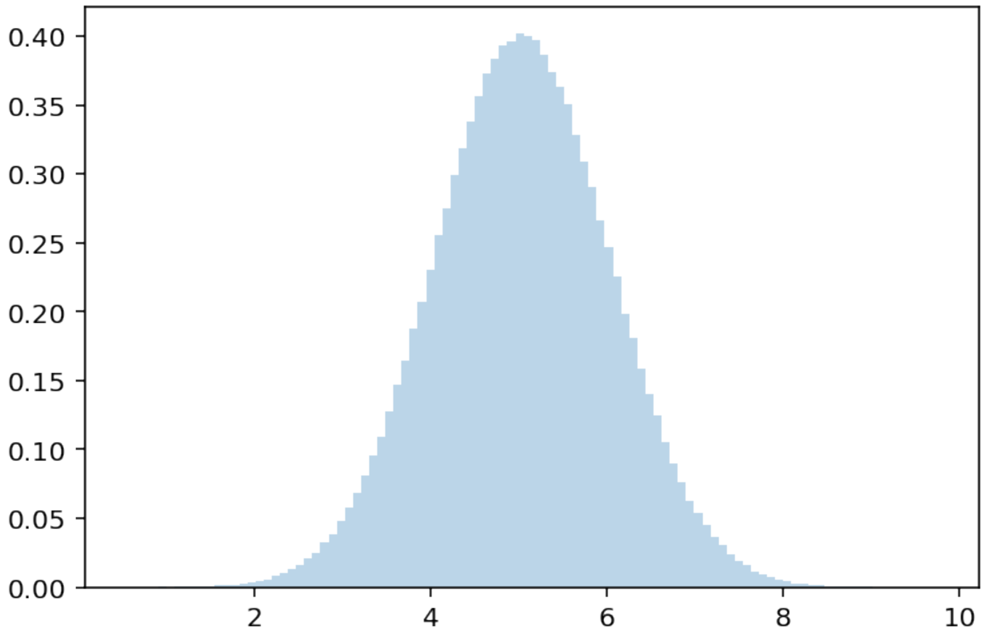


Fix $x = 1$.

```
y = np.linspace(0, 10, 1_000_000)
z = func(x=1, y=y) / np.sum(func(x=1, y=y))
plt.plot(y,z)
plt.show()
```



```
# discretized mean and standard deviation
μy = np.dot(y,z)
σy = np.sqrt(np.dot((y - μy)**2, z))

# sample
zz = np.random.normal(μy,σy,1_000_000)
plt.hist(zz, bins=100, density=True, alpha=0.3)
plt.show()
```



We compare with the analytically computed parameters and note that they are close.

```python
print(μx, σx)
print(μ1 + ρ * σ1 * (0 - μ2) / σ2, np.sqrt(σ1**2 * (1 - ρ**2)))

print(μy, σy)
print(μ2 + ρ * σ2 * (1 - μ1) / σ1, np.sqrt(σ2**2 * (1 - ρ**2)))
```

```
-0.9997518414498433 2.22658413316977
-1.0 2.227105745132009
5.039999456960771 0.9959851265795592
5.04 0.9959919678390986
```

8.17 Sum of Two Independently Distributed Random Variables

Let X, Y be two independent discrete random variables that take values in $\bar{X}, \bar{Y}$, respectively.

Define a new random variable $Z = X + Y$.

Evidently, Z takes values from $\bar{Z}$ defined as follows:

$$\begin{aligned} \bar{X} &= \{0, 1, \dots, I-1\}; \qquad f_i = \text{Prob}\{X = i\} \\ \bar{Y} &= \{0, 1, \dots, J-1\}; \qquad g_j = \text{Prob}\{Y = j\} \\ \bar{Z} &= \{0, 1, \dots, I+J-2\}; \qquad h_k = \text{Prob}\{X + Y = k\} \end{aligned}$$

Independence of X and Y implies that

$$\begin{aligned} h_k &= \text{Prob}\{X = 0, Y = k\} + \text{Prob}\{X = 1, Y = k-1\} + \dots + \text{Prob}\{X = k, Y = 0\} \\ h_k &= f_0 g_k + f_1 g_{k-1} + \dots + f_{k-1} g_1 + f_k g_0 \qquad \text{for} \quad k = 0, 1, \dots, I+J-2 \end{aligned}$$

Thus, we have:

$$h_k = \sum_{i=0}^{k} f_i g_{k-i} \equiv f * g$$

where $f * g$ denotes the **convolution** of the f and g sequences.

Similarly, for two random variables X, Y with densities f_X, g_Y, the density of $Z = X + Y$ is

$$f_Z(z) = \int_{-\infty}^{\infty} f_X(x) f_Y(z - x) dx \equiv f_X * g_Y$$

where $f_X * g_Y$ denotes the **convolution** of the f_X and g_Y functions.

8.18 Transition Probability Matrix

Consider the following joint probability distribution of two random variables.

Let X, Y be discrete random variables with joint distribution

$$\text{Prob}\{X = i, Y = j\} = \rho_{ij}$$

where $i = 0, \dots, I-1; j = 0, \dots, J-1$ and

$$\sum_i \sum_j \rho_{ij} = 1, \qquad \rho_{ij} \geqslant 0.$$

An associated conditional distribution is

$$\text{Prob}\{Y=i|X=j\}=\frac{\rho_{ij}}{\sum_i \rho_{ij}}=\frac{\text{Prob}\{Y=j,X=i\}}{\text{Prob}\{X=i\}}$$

We can define a transition probability matrix

$$p_{ij}=\text{Prob}\{Y=j|X=i\}=\frac{\rho_{ij}}{\sum_j \rho_{ij}}$$

where

$$\begin{bmatrix} p_{11} & p_{12} \\ p_{21} & p_{22} \end{bmatrix}$$

The first row is the probability of $Y=j, j=0,1$ conditional on $X=0$.

The second row is the probability of $Y=j, j=0,1$ conditional on $X=1$.

Note that

- $\sum_j \rho_{ij}=\frac{\sum_j \rho_{ij}}{\sum_j \rho_{ij}}=1$, so each row of ρ is a probability distribution (not so for each column.

8.19 Coupling

Start with a joint distribution

$$\begin{aligned} f_{ij} &= \text{Prob}\{X=i,Y=j\} \\ i &= 0,\cdots\ I-1 \\ j &= 0,\cdots\ J-1 \\ &\text{stacked to an } I\times J \text{ matrix} \\ &e.g.\quad I=1,J=1 \end{aligned}$$

where

$$\begin{bmatrix} f_{11} & f_{12} \\ f_{21} & f_{22} \end{bmatrix}$$

From the joint distribution, we have shown above that we obtain **unique** marginal distributions.

Now we'll try to go in a reverse direction.

We'll find that from two marginal distributions, can we usually construct more than one joint distribution that verifies these marginals.

Each of these joint distributions is called a **coupling** of the two martingal distributions.

Let's start with marginal distributions

$$\begin{aligned} \text{Prob}\{X=i\} &= \sum_j f_{ij}=\mu_i, i=0,\cdots,I-1 \\ \text{Prob}\{Y=j\} &= \sum_j f_{ij}=\nu_j, j=0,\cdots,J-1 \end{aligned}$$

Given two marginal distribution, μ for X and ν for Y, a joint distribution f_{ij} is said to be a **coupling** of μ and ν.

Example:

Consider the following bivariate example.

$$\begin{aligned}
\text{Prob}\{X=0\} =& 1-q=\mu_0 \\
\text{Prob}\{X=1\} =& q=\mu_1 \\
\text{Prob}\{Y=0\} =& 1-r=\nu_0 \\
\text{Prob}\{Y=1\} =& r=\nu_1 \\
\text{where } 0\leq q<r\leq 1
\end{aligned}$$

We construct two couplings.

The first coupling if our two marginal distributions is the joint distribution

$$f_{ij}=\begin{bmatrix} (1-q)(1-r) & (1-q)r \\ q(1-r) & qr \end{bmatrix}$$

To verify that it is a coupling, we check that

$$\begin{aligned}
(1-q)(1-r)+(1-q)r+q(1-r)+qr &= 1 \\
\mu_0 = (1-q)(1-r)+(1-q)r &= 1-q \\
\mu_1 = q(1-r)+qr &= q \\
\nu_0 = (1-q)(1-r)+(1-r)q &= 1-r \\
\mu_1 = r(1-q)+qr &= r
\end{aligned}$$

A second coupling of our two marginal distributions is the joint distribution

$$f_{ij}=\begin{bmatrix} (1-r) & r-q \\ 0 & q \end{bmatrix}$$

The verify that this is a coupling, note that

$$\begin{aligned}
1-r+r-q+q &= 1 \\
\mu_0 &= 1-q \\
\mu_1 &= q \\
\nu_0 &= 1-r \\
\nu_1 &= r
\end{aligned}$$

Thus, our two proposed joint distributions have the same marginal distributions.

But the joint distributions differ.

Thus, multiple joint distributions $[f_{ij}]$ can have the same marginals.

Remark:

- Couplings are important in optimal transport problems and in Markov processes.

8.20 Copula Functions

Suppose that $X_1, X_2, \dots, X_n$ are N random variables and that

- their marginal distributions are $F_1(x_1), F_2(x_2), \dots, F_N(x_N)$, and
- their joint distribution is $H(x_1, x_2, \dots, x_N)$

Then there exists a **copula function** $C(\cdot)$ that verifies

$$H(x_1, x_2, \dots, x_N) = C(F_1(x_1), F_2(x_2), \dots, F_N(x_N)).$$

We can obtain

$$C(u_1, u_2, \dots, u_n) = H[F_1^{-1}(u_1), F_2^{-1}(u_2), \dots, F_N^{-1}(u_N)]$$

In a reverse direction of logic, given univariate **marginal distributions** $F_1(x_1), F_2(x_2), \dots, F_N(x_N)$ and a copula function $C(\cdot)$, the function $H(x_1, x_2, \dots, x_N) = C(F_1(x_1), F_2(x_2), \dots, F_N(x_N))$ is a **coupling** of $F_1(x_1), F_2(x_2), \dots, F_N(x_N)$.

Thus, for given marginal distributions, we can use a copula function to determine a joint distribution when the associated univariate random variables are not independent.

Copula functions are often used to characterize **dependence** of random variables.

Discrete marginal distribution

As mentioned above, for two given marginal distributions there can be more than one coupling.

For example, consider two random variables X, Y with distributions

$$\begin{aligned}
\text{Prob}(X = 0) &= 0.6, \\
\text{Prob}(X = 1) &= 0.4, \\
\text{Prob}(Y = 0) &= 0.3, \\
\text{Prob}(Y = 1) &= 0.7,
\end{aligned}$$

For these two random variables there can be more than one coupling.

Let's first generate X and Y.

```
# define parameters
mu = np.array([0.6, 0.4])
nu = np.array([0.3, 0.7])

# number of draws
draws = 1_000_000

# generate draws from uniform distribution
p = np.random.rand(draws)

# generate draws of X and Y via uniform distribution
x = np.ones(draws)
y = np.ones(draws)
x[p <= mu[0]] = 0
x[p > mu[0]] = 1
y[p <= nu[0]] = 0
y[p > nu[0]] = 1
```

```
# calculate parameters from draws
q_hat = sum(x[x == 1])/draws
r_hat = sum(y[y == 1])/draws

# print output
print("distribution for x")
xmtb = pt.PrettyTable()
xmtb.field_names = ['x_value', 'x_prob']
```

(continues on next page)

(continued from previous page)

```
xmtb.add_row([0, 1-q_hat])
xmtb.add_row([1, q_hat])
print(xmtb)

print("distribution for y")
ymtb = pt.PrettyTable()
ymtb.field_names = ['y_value', 'y_prob']
ymtb.add_row([0, 1-r_hat])
ymtb.add_row([1, r_hat])
print(ymtb)
```

```
distribution for x
+---------+----------+
| x_value |  x_prob  |
+---------+----------+
|    0    | 0.600175 |
|    1    | 0.399825 |
+---------+----------+
distribution for y
+---------+----------+
| y_value |  y_prob  |
+---------+----------+
|    0    | 0.300562 |
|    1    | 0.699438 |
+---------+----------+
```

Let's now take our two marginal distributions, one for X, the other for Y, and construct two distinct couplings.

For the first joint distribution:

$$\text{Prob}(X = i, Y = j) = f_{ij}$$

where

$$[f_{ij}] = \begin{bmatrix} 0.18 & 0.42 \\ 0.12 & 0.28 \end{bmatrix}$$

Let's use Python to construct this joint distribution and then verify that its marginal distributions are what we want.

```
# define parameters
f1 = np.array([[0.18, 0.42], [0.12, 0.28]])
f1_cum = np.cumsum(f1)

# number of draws
draws1 = 1_000_000

# generate draws from uniform distribution
p = np.random.rand(draws1)

# generate draws of first copuling via uniform distribution
c1 = np.vstack([np.ones(draws1), np.ones(draws1)])
# X=0, Y=0
c1[0, p <= f1_cum[0]] = 0
c1[1, p <= f1_cum[0]] = 0
# X=0, Y=1
c1[0, (p > f1_cum[0])*(p <= f1_cum[1])] = 0
```

(continues on next page)

(continued from previous page)

```
c1[1, (p > f1_cum[0])*(p <= f1_cum[1])] = 1
# X=1, Y=0
c1[0, (p > f1_cum[1])*(p <= f1_cum[2])] = 1
c1[1, (p > f1_cum[1])*(p <= f1_cum[2])] = 0
# X=1, Y=1
c1[0, (p > f1_cum[2])*(p <= f1_cum[3])] = 1
c1[1, (p > f1_cum[2])*(p <= f1_cum[3])] = 1
```

```
# calculate parameters from draws
f1_00 = sum((c1[0, :] == 0)*(c1[1, :] == 0))/draws1
f1_01 = sum((c1[0, :] == 0)*(c1[1, :] == 1))/draws1
f1_10 = sum((c1[0, :] == 1)*(c1[1, :] == 0))/draws1
f1_11 = sum((c1[0, :] == 1)*(c1[1, :] == 1))/draws1

# print output of first joint distribution
print("first joint distribution for c1")
c1_mtb = pt.PrettyTable()
c1_mtb.field_names = ['c1_x_value', 'c1_y_value', 'c1_prob']
c1_mtb.add_row([0, 0, f1_00])
c1_mtb.add_row([0, 1, f1_01])
c1_mtb.add_row([1, 0, f1_10])
c1_mtb.add_row([1, 1, f1_11])
print(c1_mtb)
```

```
first joint distribution for c1
+------------+------------+----------+
| c1_x_value | c1_y_value | c1_prob  |
+------------+------------+----------+
|     0      |     0      | 0.179646 |
|     0      |     1      | 0.420357 |
|     1      |     0      | 0.120022 |
|     1      |     1      | 0.279975 |
+------------+------------+----------+
```

```
# calculate parameters from draws
c1_q_hat = sum(c1[0, :] == 1)/draws1
c1_r_hat = sum(c1[1, :] == 1)/draws1

# print output
print("marginal distribution for x")
c1_x_mtb = pt.PrettyTable()
c1_x_mtb.field_names = ['c1_x_value', 'c1_x_prob']
c1_x_mtb.add_row([0, 1-c1_q_hat])
c1_x_mtb.add_row([1, c1_q_hat])
print(c1_x_mtb)

print("marginal distribution for y")
c1_ymtb = pt.PrettyTable()
c1_ymtb.field_names = ['c1_y_value', 'c1_y_prob']
c1_ymtb.add_row([0, 1-c1_r_hat])
c1_ymtb.add_row([1, c1_r_hat])
print(c1_ymtb)
```

```
marginal distribution for x
+------------+--------------------+
| c1_x_value |     c1_x_prob      |
+------------+--------------------+
|     0      | 0.6000030000000001 |
|     1      |      0.399997      |
+------------+--------------------+
marginal distribution for y
+------------+---------------------+
| c1_y_value |      c1_y_prob      |
+------------+---------------------+
|     0      | 0.29966800000000005 |
|     1      |       0.700332      |
+------------+---------------------+
```

Now, let's construct another joint distribution that is also a coupling of X and Y

$$[f_{ij}] = \begin{bmatrix} 0.3 & 0.3 \\ 0 & 0.4 \end{bmatrix}$$

```
# define parameters
f2 = np.array([[0.3, 0.3], [0, 0.4]])
f2_cum = np.cumsum(f2)

# number of draws
draws2 = 1_000_000

# generate draws from uniform distribution
p = np.random.rand(draws2)

# generate draws of first coupling via uniform distribution
c2 = np.vstack([np.ones(draws2), np.ones(draws2)])
# X=0, Y=0
c2[0, p <= f2_cum[0]] = 0
c2[1, p <= f2_cum[0]] = 0
# X=0, Y=1
c2[0, (p > f2_cum[0])*(p <= f2_cum[1])] = 0
c2[1, (p > f2_cum[0])*(p <= f2_cum[1])] = 1
# X=1, Y=0
c2[0, (p > f2_cum[1])*(p <= f2_cum[2])] = 1
c2[1, (p > f2_cum[1])*(p <= f2_cum[2])] = 0
# X=1, Y=1
c2[0, (p > f2_cum[2])*(p <= f2_cum[3])] = 1
c2[1, (p > f2_cum[2])*(p <= f2_cum[3])] = 1
```

```
# calculate parameters from draws
f2_00 = sum((c2[0, :] == 0)*(c2[1, :] == 0))/draws2
f2_01 = sum((c2[0, :] == 0)*(c2[1, :] == 1))/draws2
f2_10 = sum((c2[0, :] == 1)*(c2[1, :] == 0))/draws2
f2_11 = sum((c2[0, :] == 1)*(c2[1, :] == 1))/draws2

# print output of second joint distribution
print("first joint distribution for c2")
c2_mtb = pt.PrettyTable()
c2_mtb.field_names = ['c2_x_value', 'c2_y_value', 'c2_prob']
c2_mtb.add_row([0, 0, f2_00])
```

(continues on next page)

(continued from previous page)

```
c2_mtb.add_row([0, 1, f2_01])
c2_mtb.add_row([1, 0, f2_10])
c2_mtb.add_row([1, 1, f2_11])
print(c2_mtb)
```

```
first joint distribution for c2
+------------+------------+----------+
| c2_x_value | c2_y_value | c2_prob  |
+------------+------------+----------+
|     0      |     0      | 0.300074 |
|     0      |     1      | 0.299807 |
|     1      |     0      |   0.0    |
|     1      |     1      | 0.400119 |
+------------+------------+----------+
```

```
# calculate parameters from draws
c2_q_hat = sum(c2[0, :] == 1)/draws2
c2_r_hat = sum(c2[1, :] == 1)/draws2

# print output
print("marginal distribution for x")
c2_x_mtb = pt.PrettyTable()
c2_x_mtb.field_names = ['c2_x_value', 'c2_x_prob']
c2_x_mtb.add_row([0, 1-c2_q_hat])
c2_x_mtb.add_row([1, c2_q_hat])
print(c2_x_mtb)

print("marginal distribution for y")
c2_ymtb = pt.PrettyTable()
c2_ymtb.field_names = ['c2_y_value', 'c2_y_prob']
c2_ymtb.add_row([0, 1-c2_r_hat])
c2_ymtb.add_row([1, c2_r_hat])
print(c2_ymtb)
```

```
marginal distribution for x
+------------+-----------+
| c2_x_value | c2_x_prob |
+------------+-----------+
|     0      |  0.599881 |
|     1      |  0.400119 |
+------------+-----------+
marginal distribution for y
+------------+---------------------+
| c2_y_value |      c2_y_prob      |
+------------+---------------------+
|     0      | 0.30007399999999995 |
|     1      |       0.699926      |
+------------+---------------------+
```

We have verified that both joint distributions, c_1 and c_2, have identical marginal distributions of X and Y, respectively.

So they are both couplings of X and Y.

8.21 Time Series

Suppose that there are two time periods.

- $t = 0$ "today"
- $t = 1$ "tomorrow"

Let $X(0)$ be a random variable to be realized at $t = 0$, $X(1)$ be a random variable to be realized at $t = 1$.

Suppose that

$$\text{Prob}\{X(0) = i, X(1) = j\} = f_{ij} \geq 0 \; i = 0, \cdots, I - 1$$
$$\sum_i \sum_j f_{ij} = 1$$

f_{ij} is a joint distribution over $[X(0), X(1)]$.

A conditional distribution is

$$\text{Prob}\{X(1) = j | X(0) = i\} = \frac{f_{ij}}{\sum_j f_{ij}}$$

Remark:

- This is a key formula for a theory of optimally predicting a time series.

CHAPTER

NINE

UNIVARIATE TIME SERIES WITH MATRIX ALGEBRA

Contents

9.1 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 [Sam39] 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:

```
import numpy as np
%matplotlib inline
import matplotlib.pyplot as plt
from matplotlib import cm
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
```

9.2 Samuelson's model

Let $t = 0, \pm 1, \pm 2, \ldots$ index time.

For $t = 1, 2, 3, \ldots, T$ suppose that

$$y_t = \alpha_0 + \alpha_1 y_{t-1} + \alpha_2 y_{t-2} \tag{9.1}$$

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 (9.1) is called a **second-order linear difference equation**.

But actually, it is a collection of T simultaneous linear equations in the T variables $y_1, y_2, \ldots, y_T$.

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

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ -\alpha_1 & 1 & 0 & 0 & \cdots & 0 & 0 & 0 \\ -\alpha_2 & -\alpha_1 & 1 & 0 & \cdots & 0 & 0 & 0 \\ 0 & -\alpha_2 & -\alpha_1 & 1 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & -\alpha_2 & -\alpha_1 & 1 \end{bmatrix}}_{\equiv A} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ \vdots \\ y_T \end{bmatrix} = \underbrace{\begin{bmatrix} \alpha_0 + \alpha_1 y_0 + \alpha_2 y_{-1} \\ \alpha_0 + \alpha_2 y_0 \\ \alpha_0 \\ \alpha_0 \\ \vdots \\ \alpha_0 \end{bmatrix}}_{\equiv b}$$

or

$$Ay = b$$

where

$$y = \begin{bmatrix} y_1 \\ y_2 \\ \cdots \\ y_T \end{bmatrix}$$

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*.

```
T = 80

# parameters
□0 = 10.0
□1 = 1.53
□2 = -.9

y_1 = 28. # y_{-1}
y0 = 24.
```

Now we construct A and b.

```
A = np.identity(T)  # The T x T identity matrix

for i in range(T):

    if i-1 >= 0:
        A[i, i-1] = -□1

    if i-2 >= 0:
        A[i, i-2] = -□2

b = np.full(T, □0)
b[0] = □0 + □1 * y0 + □2 * y_1
b[1] = □0 + □2 * y0
```

Let's look at the matrix A and the vector b for our example.

```
A, b
```

```
(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.  ,  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.

To solve $y = A^{-1}b$ we can either invert A directly, as in

```
A_inv = np.linalg.inv(A)

y = A_inv @ b
```

or we can use `np.linalg.solve`:

```
y_second_method = np.linalg.solve(A, b)
```

Here make sure the two methods give the same result, at least up to floating point precision:

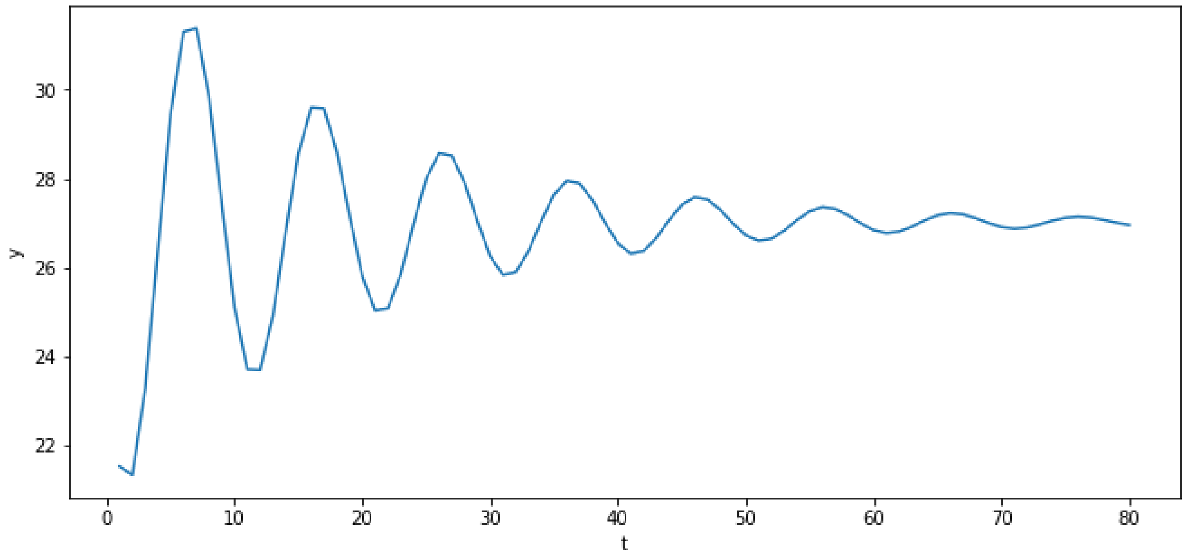
```
np.allclose(y, y_second_method)
```

```
True
```

Note: In general, `np.linalg.solve` is more numerically stable than using `np.linalg.inv` directly. However, stability is not an issue for this small example. Moreover, we will repeatedly use `A_inv` in what follows, so there is added value in computing it directly.

Now we can plot.

```
plt.plot(np.arange(T)+1, y)
plt.xlabel('t')
plt.ylabel('y')

plt.show()
```



The **steady state** value y^* of y_t is obtained by setting $y_t = y_{t-1} = y_{t-2} = y^*$ in (9.1), which yields

$$y^* = \frac{\alpha_0}{1 - \alpha_1 - \alpha_2}$$

If we set the initial values to $y_0 = y_{-1} = y^*$, then y_t will be constant:

```
y_star = ⍰0 / (1 - ⍰1 - ⍰2)
y_1_steady = y_star # y_{-1}
y0_steady = y_star

b_steady = np.full(T, ⍰0)
b_steady[0] = ⍰0 + ⍰1 * y0_steady + ⍰2 * y_1_steady
b_steady[1] = ⍰0 + ⍰2 * y0_steady
```

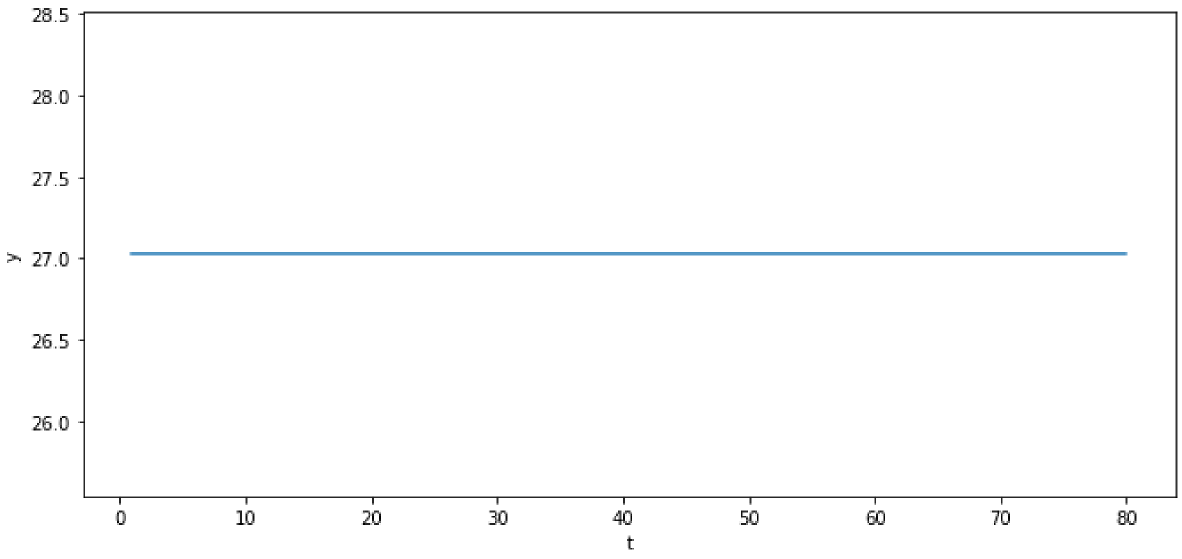
```
y_steady = A_inv @ b_steady
```

```
plt.plot(np.arange(T)+1, y_steady)
plt.xlabel('t')
```

(continues on next page)

(continued from previous page)

```
plt.ylabel('y')

plt.show()
```



9.3 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**:

$$y_t = \alpha_0 + \alpha_1 y_{t-1} + \alpha_2 y_{t-2} + u_t \tag{9.2}$$

where $u_t \sim N\left(0, \sigma_u^2\right)$ 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

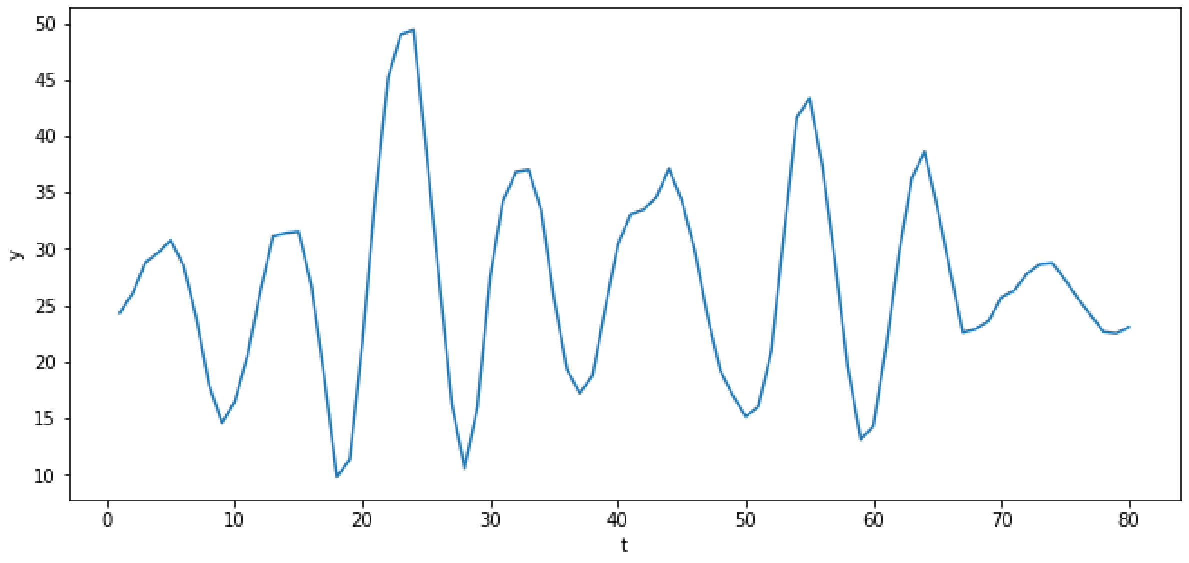
$$y = A^{-1}\left(b + u\right)$$

Let's try it out in Python.

```
□u = 2.
```

```
u = np.random.normal(0, □u, size=T)
y = A_inv @ (b + u)
```

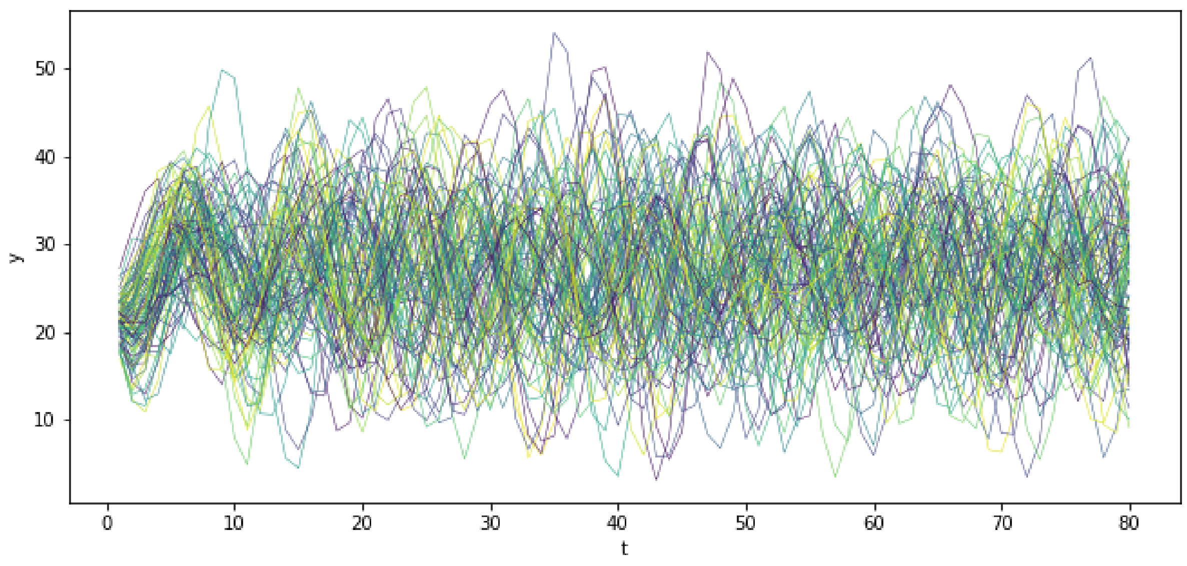
```
plt.plot(np.arange(T)+1, y)
plt.xlabel('t')
plt.ylabel('y')

plt.show()
```



The above time series looks a lot like (detrended) GDP series for a number of advanced countries in recent decades.

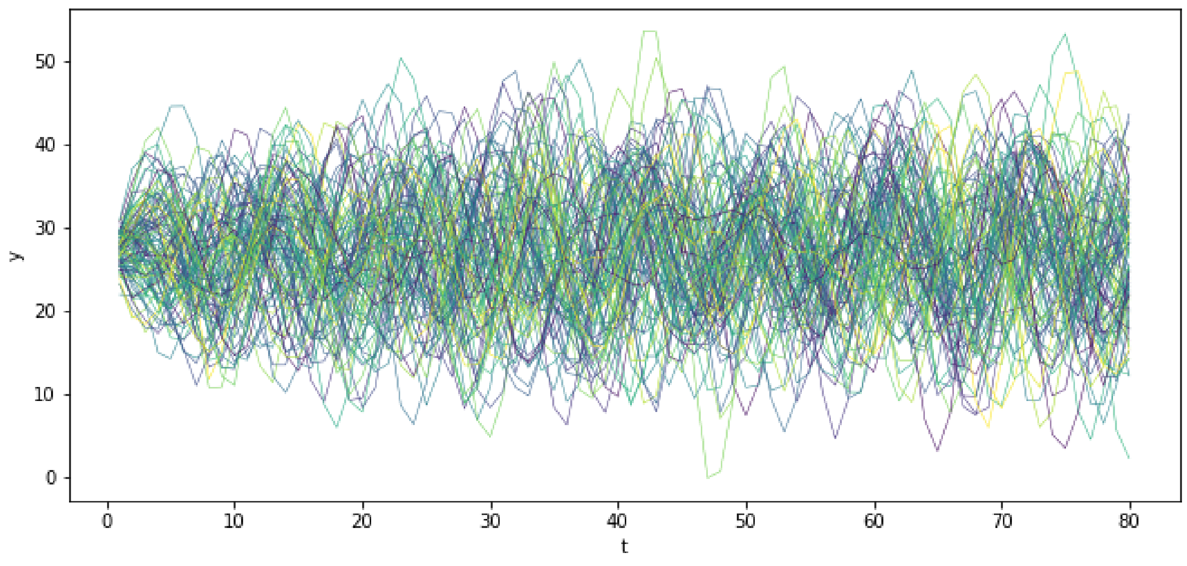
We can simulate N paths.

```
N = 100

for i in range(N):
    col = cm.viridis(np.random.rand())  # Choose a random color from viridis
    u = np.random.normal(0, □u, size=T)
    y = A_inv @ (b + u)
    plt.plot(np.arange(T)+1, y, lw=0.5, color=col)

plt.xlabel('t')
plt.ylabel('y')

plt.show()
```



Also consider the case when y_0 and y_{-1} are at steady state.

```
N = 100

for i in range(N):
    col = cm.viridis(np.random.rand())  # Choose a random color from viridis
    u = np.random.normal(0, σu, size=T)
    y_steady = A_inv @ (b_steady + u)
    plt.plot(np.arange(T)+1, y_steady, lw=0.5, color=col)

plt.xlabel('t')
plt.ylabel('y')

plt.show()
```



9.4 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

$$\underbrace{\begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ \vdots \\ p_T \end{bmatrix}}_{\equiv p} = \underbrace{\begin{bmatrix} 1 & \beta & \beta^2 & \cdots & \beta^{T-1} \\ 0 & 1 & \beta & \cdots & \beta^{T-2} \\ 0 & 0 & 1 & \cdots & \beta^{T-3} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix}}_{\equiv B} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_T \end{bmatrix}$$

```
𝛽 = .96
```

```
# construct B
B = np.zeros((T, T))

for i in range(T):
    B[i, i:] = 𝛽 ** np.arange(0, T-i)
```

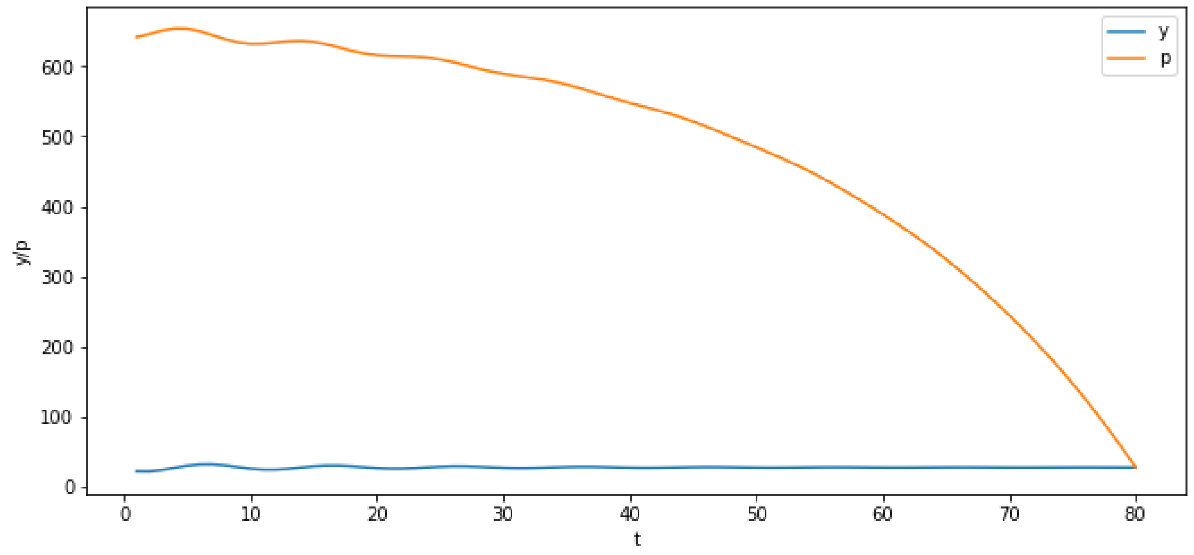
```
B
```

```
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.        ]])
```

```
□u = 0.
u = np.random.normal(0, □u, size=T)
y = A_inv @ (b + u)
y_steady = A_inv @ (b_steady + u)
```

```
p = B @ y
```

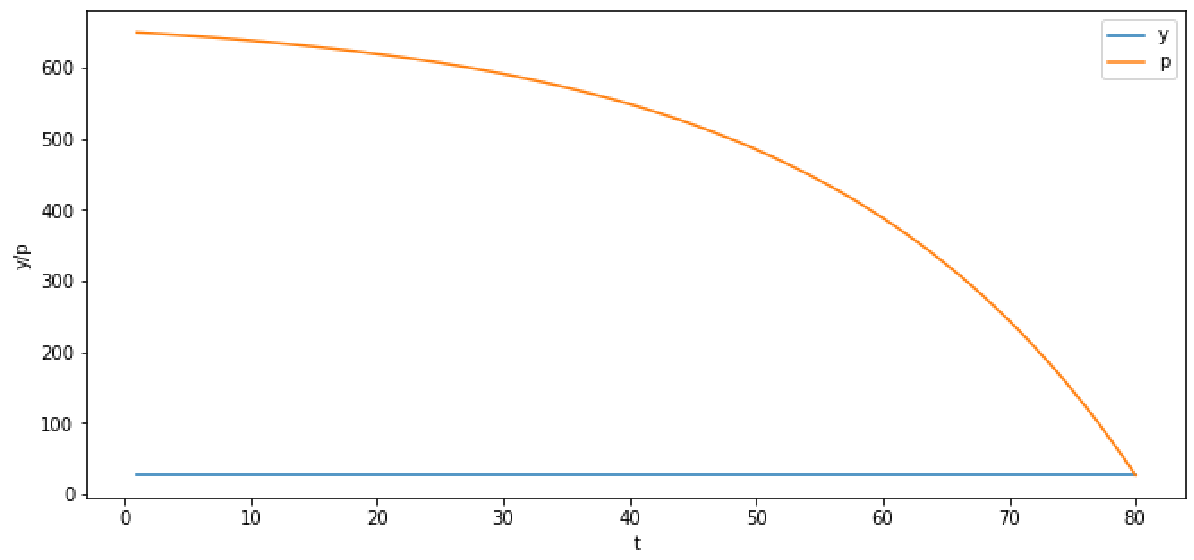
```
plt.plot(np.arange(0, T)+1, y, label='y')
plt.plot(np.arange(0, T)+1, p, label='p')
plt.xlabel('t')
plt.ylabel('y/p')
plt.legend()

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.

```
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()
```


y
p
y/p
t
0
100
200
300
400
500
600
0
10
20
30
40
50
60
70
80

CHAPTER

TEN

LLN AND CLT

Contents

10.1 Overview

This lecture illustrates two of the most important theorems of probability and statistics: The law of large numbers (LLN) and the central limit theorem (CLT).

These beautiful theorems lie behind many of the most fundamental results in econometrics and quantitative economic modeling.

The lecture is based around simulations that show the LLN and CLT in action.

We also demonstrate how the LLN and CLT break down when the assumptions they are based on do not hold.

In addition, we examine several useful extensions of the classical theorems, such as

- The delta method, for smooth functions of random variables.
- The multivariate case.

Some of these extensions are presented as exercises.

We'll need the following imports:

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import random
import numpy as np
from scipy.stats import t, beta, lognorm, expon, gamma, uniform, cauchy
```

(continues on next page)

(continued from previous page)

```
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
```

10.2 Relationships

The CLT refines the LLN.

The LLN gives conditions under which sample moments converge to population moments as sample size increases.

The CLT provides information about the rate at which sample moments converge to population moments as sample size increases.

10.3 LLN

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

10.3.1 The Classical LLN

The classical law of large numbers concerns independent and identically distributed (IID) random variables.

Here is the strongest version of the classical LLN, known as *Kolmogorov's strong law*.

Let $X_1, \ldots, X_n$ be independent and identically distributed scalar random variables, with common distribution F.

When it exists, let μ denote the common mean of this sample:

$$\mu := \mathbb{E} X = \int x F(dx)$$

In addition, let

$$\bar{X}_n := \frac{1}{n} \sum_{i=1}^n X_i$$

Kolmogorov's strong law states that, if $\mathbb{E}|X|$ is finite, then

$$\mathbb{P} \left\{ \bar{X}_n \to \mu \text{ as } n \to \infty \right\} = 1 \tag{10.1}$$

What does this last expression mean?

Let's think about it from a simulation perspective, imagining for a moment that our computer can generate perfect random samples (which of course it can't).

Let's also imagine that we can generate infinite sequences so that the statement $\bar{X}_n \to \mu$ can be evaluated.

In this setting, (10.1) should be interpreted as meaning that the probability of the computer producing a sequence where $\bar{X}_n \to \mu$ fails to occur is zero.

10.3.2 Proof

The proof of Kolmogorov's strong law is nontrivial – see, for example, theorem 8.3.5 of [Dud02].

On the other hand, we can prove a weaker version of the LLN very easily and still get most of the intuition.

The version we prove is as follows: If $X_1, \ldots, X_n$ is IID with $\mathbb{E}X_i^2 < \infty$, then, for any $\epsilon > 0$, we have

$$\mathbb{P}\left\{|\bar{X}_n - \mu| \geq \epsilon\right\} \to 0 \quad \text{as} \quad n \to \infty \tag{10.2}$$

(This version is weaker because we claim only convergence in probability rather than almost sure convergence, and assume a finite second moment)

To see that this is so, fix $\epsilon > 0$, and let σ^2 be the variance of each X_i.

Recall the Chebyshev inequality, which tells us that

$$\mathbb{P}\left\{|\bar{X}_n - \mu| \geq \epsilon\right\} \leq \frac{\mathbb{E}[(\bar{X}_n - \mu)^2]}{\epsilon^2} \tag{10.3}$$

Now observe that

$$\begin{aligned}
\mathbb{E}[(\bar{X}_n - \mu)^2] &= \mathbb{E}\left\{\left[\frac{1}{n}\sum_{i=1}^n (X_i - \mu)\right]^2\right\} \\
&= \frac{1}{n^2}\sum_{i=1}^n \sum_{j=1}^n \mathbb{E}(X_i - \mu)(X_j - \mu) \\
&= \frac{1}{n^2}\sum_{i=1}^n \mathbb{E}(X_i - \mu)^2 \\
&= \frac{\sigma^2}{n}
\end{aligned}$$

Here the crucial step is at the third equality, which follows from independence.

Independence means that if $i \neq j$, then the covariance term $\mathbb{E}(X_i - \mu)(X_j - \mu)$ drops out.

As a result, $n^2 - n$ terms vanish, leading us to a final expression that goes to zero in n.

Combining our last result with (10.3), we come to the estimate

$$\mathbb{P}\left\{|\bar{X}_n - \mu| \geq \epsilon\right\} \leq \frac{\sigma^2}{n\epsilon^2} \tag{10.4}$$

The claim in (10.2) is now clear.

Of course, if the sequence $X_1, \ldots, X_n$ is correlated, then the cross-product terms $\mathbb{E}(X_i - \mu)(X_j - \mu)$ are not necessarily zero.

While this doesn't mean that the same line of argument is impossible, it does mean that if we want a similar result then the covariances should be "almost zero" for "most" of these terms.

In a long sequence, this would be true if, for example, $\mathbb{E}(X_i - \mu)(X_j - \mu)$ approached zero when the difference between i and j became large.

In other words, the LLN can still work if the sequence $X_1, \ldots, X_n$ has a kind of "asymptotic independence", in the sense that correlation falls to zero as variables become further apart in the sequence.

This idea is very important in time series analysis, and we'll come across it again soon enough.

10.3.3 Illustration

Let's now illustrate the classical IID law of large numbers using simulation.

In particular, we aim to generate some sequences of IID random variables and plot the evolution of $\bar{X}_n$ as n increases.

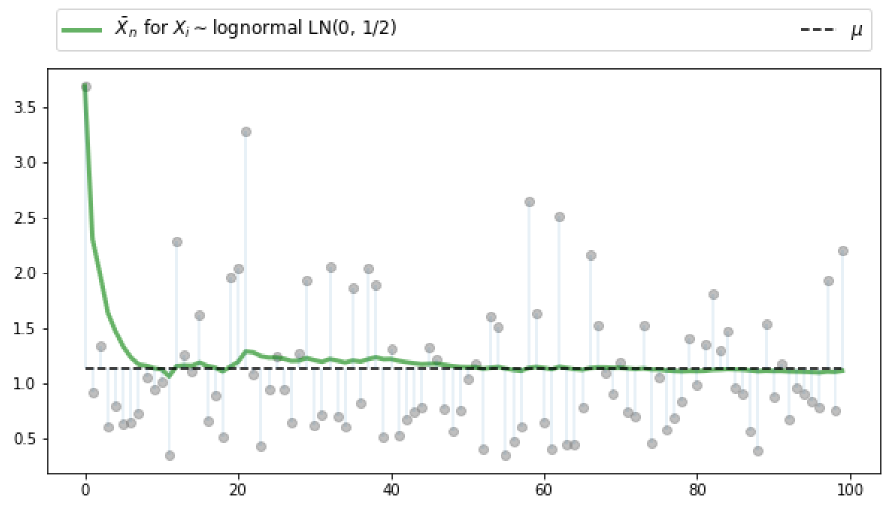
Below is a figure that does just this (as usual, you can click on it to expand it).

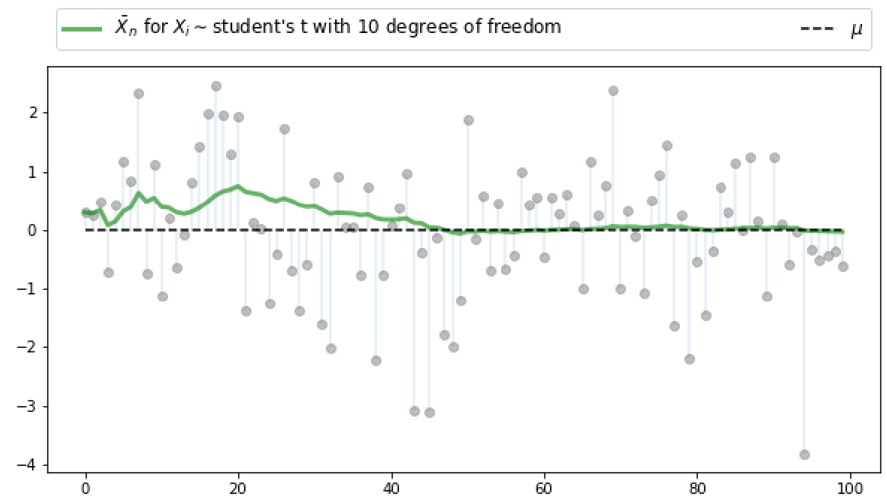
It shows IID observations from three different distributions and plots $\bar{X}_n$ against n in each case.

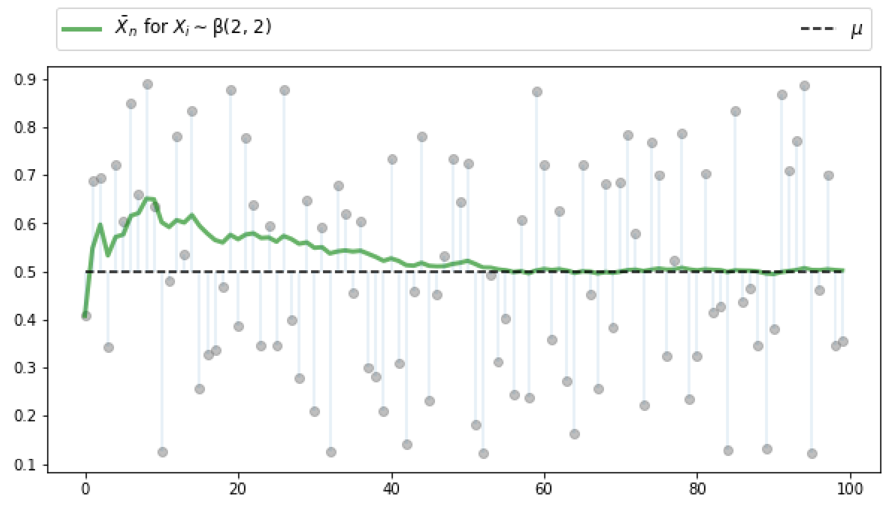
The dots represent the underlying observations X_i for $i = 1, \ldots, 100$.

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

```python
n = 100

# Arbitrary collection of distributions
distributions = {"student's t with 10 degrees of freedom": t(10),
                 "β(2, 2)": beta(2, 2),
                 "lognormal LN(0, 1/2)": lognorm(0.5),
                 "γ(5, 1/2)": gamma(5, scale=2),
                 "poisson(4)": poisson(4),
                 "exponential with λ = 1": expon(1)}

# Create a figure and some axes
num_plots = 3
fig, axes = plt.subplots(num_plots, 1, figsize=(10, 20))

# Set some plotting parameters to improve layout
bbox = (0., 1.02, 1., .102)
legend_args = {'ncol': 2,
               'bbox_to_anchor': bbox,
               'loc': 3,
               'mode': 'expand'}
plt.subplots_adjust(hspace=0.5)

for ax in axes:
    # Choose a randomly selected distribution
    name = random.choice(list(distributions.keys()))
    distribution = distributions.pop(name)

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

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

    # Plot
    ax.plot(list(range(n)), data, 'o', color='grey', alpha=0.5)
    axlabel = '$\\bar X_n$ for $X_i \sim$' + name
    ax.plot(list(range(n)), sample_mean, 'g-', lw=3, alpha=0.6, label=axlabel)
    m = distribution.mean()
    ax.plot(list(range(n)), [m] * n, 'k--', lw=1.5, label='$\mu$')
    ax.vlines(list(range(n)), m, data, lw=0.2)
    ax.legend(**legend_args, fontsize=12)

plt.show()
```


$\bar{X}_n$ for $X_i \sim$ lognormal LN(0, 1/2)
μ
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0
20
40
60
80
100


$\bar{X}_n$ for $X_i \sim$ student's t with 10 degrees of freedom
μ
2
1
0
−1
−2
−3
−4
0
20
40
60
80
100


$\bar{X}_n$ for $X_i \sim$ β(2, 2)
μ
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
20
40
60
80
100

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

10.4 CLT

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

10.4.1 Statement of the Theorem

The central limit theorem is one of the most remarkable results in all of mathematics.

In the classical IID setting, it tells us the following:

If the sequence $X_1, \ldots, X_n$ is IID, with common mean μ and common variance $\sigma^2 \in (0, \infty)$, then

$$\sqrt{n}(\bar{X}_n - \mu) \stackrel{d}{\to} N(0, \sigma^2) \quad \text{as} \quad n \to \infty \tag{10.5}$$

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

10.4.2 Intuition

The striking implication of the CLT is that for **any** distribution with finite second moment, the simple operation of adding independent copies **always** leads to a Gaussian curve.

A relatively simple proof of the central limit theorem can be obtained by working with characteristic functions (see, e.g., theorem 9.5.6 of [Dud02]).

The proof is elegant but almost anticlimactic, and it provides surprisingly little intuition.

In fact, all of the proofs of the CLT that we know are similar in this respect.

Why does adding independent copies produce a bell-shaped distribution?

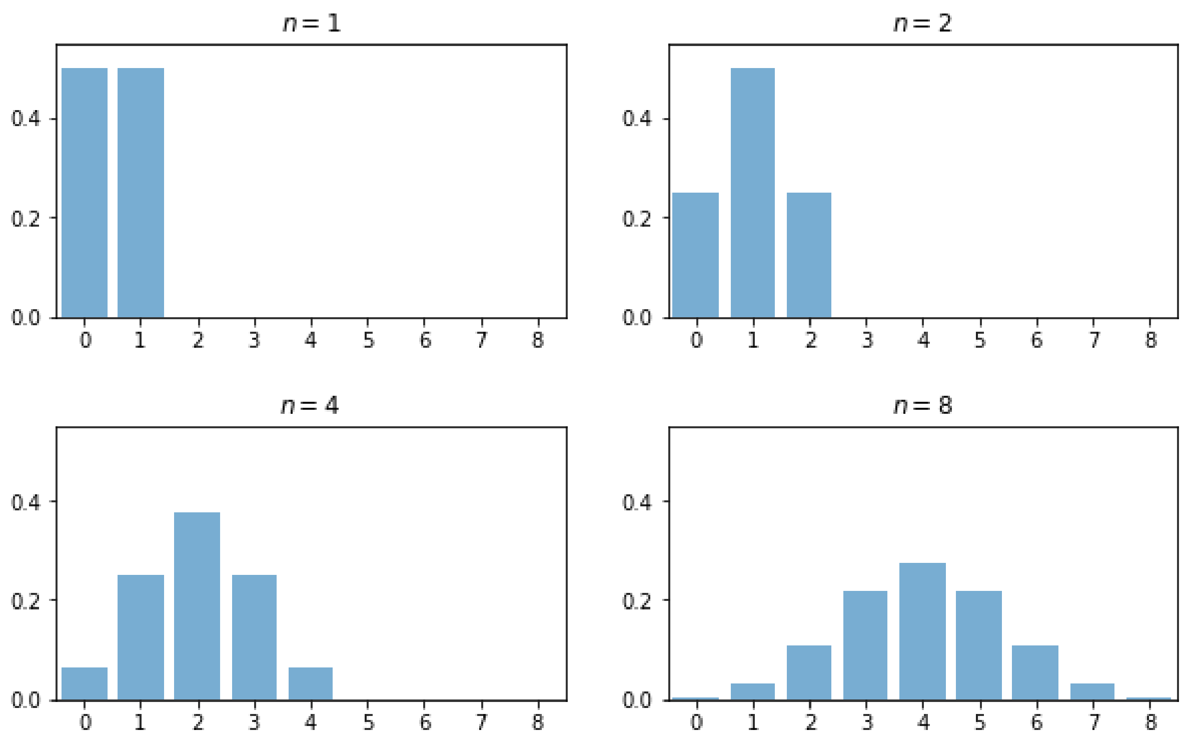
Part of the answer can be obtained by investigating the addition of independent Bernoulli random variables.

In particular, let X_i be binary, with $\mathbb{P}\{X_i = 0\} = \mathbb{P}\{X_i = 1\} = 0.5$, and let $X_1, \ldots, X_n$ be independent.

Think of $X_i = 1$ as a "success", so that $Y_n = \sum_{i=1}^n X_i$ is the number of successes in n trials.

The next figure plots the probability mass function of Y_n for $n = 1, 2, 4, 8$

```python
fig, axes = plt.subplots(2, 2, figsize=(10, 6))
plt.subplots_adjust(hspace=0.4)
axes = axes.flatten()
ns = [1, 2, 4, 8]
dom = list(range(9))

for ax, n in zip(axes, ns):
    b = binom(n, 0.5)
    ax.bar(dom, b.pmf(dom), alpha=0.6, align='center')
    ax.set(xlim=(-0.5, 8.5), ylim=(0, 0.55),
           xticks=list(range(9)), yticks=(0, 0.2, 0.4),
           title=f'$n = {n}$')

plt.show()
```



When $n = 1$, the distribution is flat — one success or no successes have the same probability.

When $n = 2$ we can either have 0, 1 or 2 successes.

Notice the peak in probability mass at the mid-point $k = 1$.

The reason is that there are more ways to get 1 success (“fail then succeed” or “succeed then fail”) than to get zero or two successes.

Moreover, the two trials are independent, so the outcomes “fail then succeed” and “succeed then fail” are just as likely as the outcomes “fail then fail” and “succeed then succeed”.

(If there was positive correlation, say, then “succeed then fail” would be less likely than “succeed then succeed”)

Here, already we have the essence of the CLT: addition under independence leads probability mass to pile up in the middle and thin out at the tails.

For $n = 4$ and $n = 8$ we again get a peak at the “middle” value (halfway between the minimum and the maximum possible value).

The intuition is the same — there are simply more ways to get these middle outcomes.

If we continue, the bell-shaped curve becomes even more pronounced.

We are witnessing the binomial approximation of the normal distribution.

10.4.3 Simulation 1

Since the CLT seems almost magical, running simulations that verify its implications is one good way to build intuition.

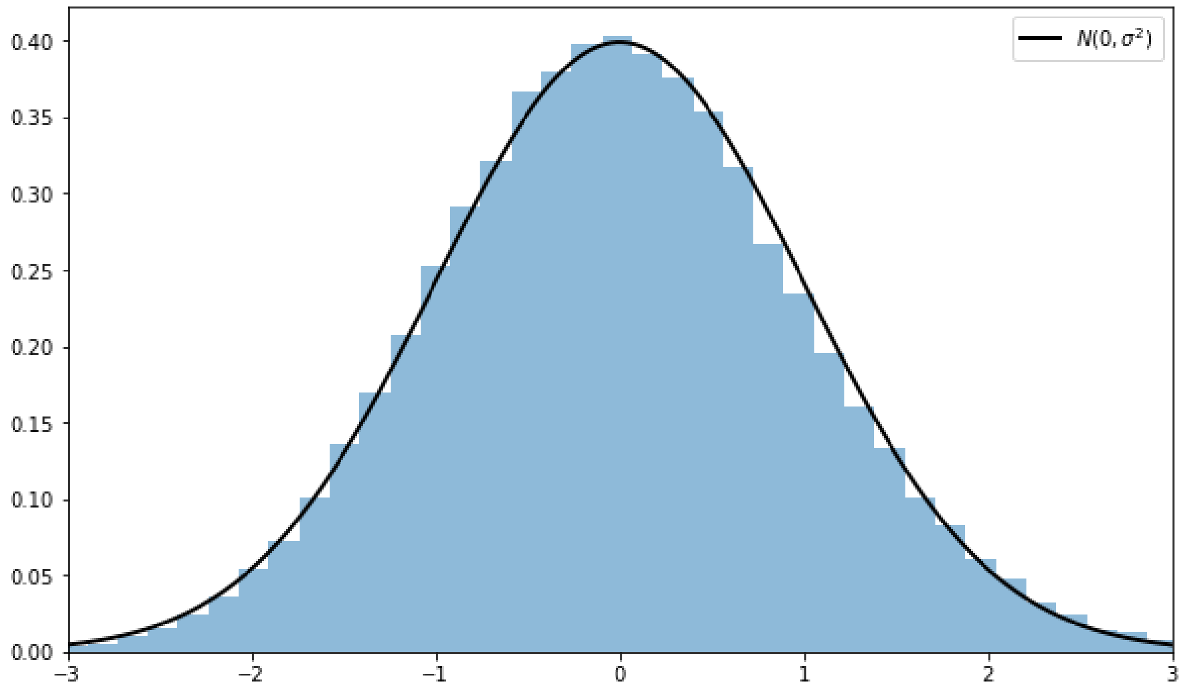
To this end, we now perform the following simulation

1. Choose an arbitrary distribution F for the underlying observations X_i.
2. Generate independent draws of $Y_n := \sqrt{n}(\bar{X}_n - \mu)$.
3. Use these draws to compute some measure of their distribution — such as a histogram.
4. Compare the latter to $N(0, \sigma^2)$.

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

(Please experiment with other choices of F, but remember that, to conform with the conditions of the CLT, the distribution must have a finite second moment.)

```python
# 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='$N(0, \sigma^2)$')
ax.legend()

plt.show()
```



Notice the absence of for loops — every operation is vectorized, meaning that the major calculations are all shifted to highly optimized C code.

The fit to the normal density is already tight and can be further improved by increasing `n`.

You can also experiment with other specifications of F.

10.4.4 Simulation 2

Our next simulation is somewhat like the first, except that we aim to track the distribution of $Y_n := \sqrt{n}(\bar{X}_n - \mu)$ as n increases.

In the simulation, we'll be working with random variables having $\mu = 0$.

Thus, when $n = 1$, we have $Y_1 = X_1$, so the first distribution is just the distribution of the underlying random variable.

For $n = 2$, the distribution of Y_2 is that of $(X_1 + X_2)/\sqrt{2}$, and so on.

What we expect is that, regardless of the distribution of the underlying random variable, the distribution of Y_n will smooth out into a bell-shaped curve.

The next figure shows this process for $X_i \sim f$, where f was specified as the convex combination of three different beta densities.

(Taking a convex combination is an easy way to produce an irregular shape for f.)

In the figure, the closest density is that of Y_1, while the furthest is that of Y_5

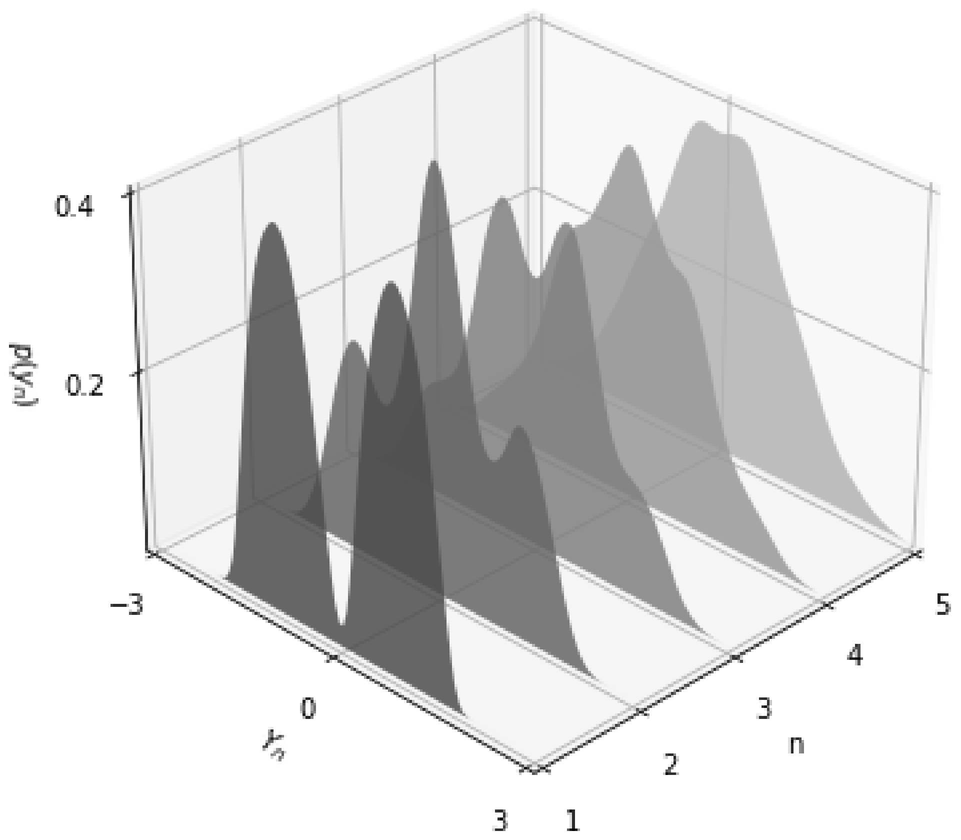
```python
beta_dist = beta(2, 2)

def gen_x_draws(k):
    """
    Returns a flat array containing k independent draws from the
    distribution of X, the underlying random variable.  This distribution
```

(continues on next page)

(continued from previous page)

```python
    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), ylabel='$Y_n$', zlabel='$p(y_n)$',
       xlabel=("n"), yticks=((-3, 0, 3)), ylim3d=(a, b),
       zlim3d=(0, 0.4), zticks=((0.2, 0.4)))
ax.invert_xaxis()
# Rotates the plot 30 deg on z axis and 45 deg on x axis
ax.view_init(30, 45)
plt.show()
```

```
/tmp/ipykernel_15567/392210314.py:36: MatplotlibDeprecationWarning: Calling gca()␣
↪with keyword arguments was deprecated in Matplotlib 3.4. Starting two minor␣
↪releases later, gca() will take no keyword arguments. The gca() function should␣
↪only be used to get the current axes, or if no axes exist, create new axes with␣
↪default keyword arguments. To create a new axes with non-default arguments, use␣
↪plt.axes() or plt.subplot().
  ax = fig.gca(projection='3d')
```



As expected, the distribution smooths out into a bell curve as n increases.

We leave you to investigate its contents if you wish to know more.

If you run the file from the ordinary IPython shell, the figure should pop up in a window that you can rotate with your mouse, giving different views on the density sequence.

10.4.5 The Multivariate Case

The law of large numbers and central limit theorem work just as nicely in multidimensional settings.

To state the results, let's recall some elementary facts about random vectors.

A random vector $\mathbf{X}$ is just a sequence of k random variables $(X_1, \ldots, X_k)$.

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

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

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

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

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

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

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

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

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

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 $\mathbf{X}_1, \ldots, \mathbf{X}_n$ be a sequence of independent and identically distributed random vectors, each one taking values in $\mathbb{R}^k$.

Let μ be the vector $\mathbb{E}[\mathbf{X}_i]$, and let Σ be the variance-covariance matrix of $\mathbf{X}_i$.

Interpreting vector addition and scalar multiplication in the usual way (i.e., pointwise), let

$$\bar{\mathbf{X}}_n := \frac{1}{n} \sum_{i=1}^{n} \mathbf{X}_i$$

In this setting, the LLN tells us that

$$\mathbb{P}\left\{\bar{\mathbf{X}}_n \to \mu \text{ as } n \to \infty\right\} = 1 \tag{10.6}$$

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

The CLT tells us that, provided Σ is finite,

$$\sqrt{n}(\bar{\mathbf{X}}_n - \mu) \xrightarrow{d} N(\mathbf{0}, \Sigma) \quad \text{as} \quad n \to \infty \tag{10.7}$$

10.5 Exercises

Exercise 10.5.1

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

Assume the conditions of the CLT as *stated above*.

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

$$\sqrt{n}\{g(\bar{X}_n) - g(\mu)\} \stackrel{d}{\to} N(0, g'(\mu)^2\sigma^2) \quad \text{as} \quad n \to \infty \tag{10.8}$$

This theorem is used frequently in statistics to obtain the asymptotic distribution of estimators — many of which can be expressed as functions of sample means.

(These kinds of results are often said to use the "delta method".)

The proof is based on a Taylor expansion of g around the point μ.

Taking the result as given, let the distribution F of each X_i be uniform on $[0, \pi/2]$ and let $g(x) = \sin(x)$.

Derive the asymptotic distribution of $\sqrt{n}\{g(\bar{X}_n) - g(\mu)\}$ and illustrate convergence in the same spirit as the program discussed *above*.

What happens when you replace $[0, \pi/2]$ with $[0, \pi]$?

What is the source of the problem?

Exercise 10.5.2

Here's a result that's often used in developing statistical tests, and is connected to the multivariate central limit theorem.

If you study econometric theory, you will see this result used again and again.

Assume the setting of the multivariate CLT *discussed above*, so that

1. $\mathbf{X}_1, \ldots, \mathbf{X}_n$ is a sequence of IID random vectors, each taking values in $\mathbb{R}^k$.
2. $\mu := \mathbb{E}[\mathbf{X}_i]$, and Σ is the variance-covariance matrix of $\mathbf{X}_i$.
3. The convergence

$$\sqrt{n}(\bar{\mathbf{X}}_n - \mu) \stackrel{d}{\to} N(\mathbf{0}, \Sigma) \tag{10.9}$$

is valid.

In a statistical setting, one often wants the right-hand side to be **standard** normal so that confidence intervals are easily computed.

This normalization can be achieved on the basis of three observations.

First, if $\mathbf{X}$ is a random vector in $\mathbb{R}^k$ and $\mathbf{A}$ is constant and $k \times k$, then

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

Second, by the continuous mapping theorem, if $\mathbf{Z}_n \stackrel{d}{\to} \mathbf{Z}$ in $\mathbb{R}^k$ and $\mathbf{A}$ is constant and $k \times k$, then

$$\mathbf{A}\mathbf{Z}_n \stackrel{d}{\to} \mathbf{A}\mathbf{Z}$$

Third, if $\mathbf{S}$ is a $k \times k$ symmetric positive definite matrix, then there exists a symmetric positive definite matrix $\mathbf{Q}$, called the inverse square root of $\mathbf{S}$, such that

$$\mathbf{Q}\mathbf{S}\mathbf{Q}' = \mathbf{I}$$

Here $\mathbf{I}$ is the $k \times k$ identity matrix.

Putting these things together, your first exercise is to show that if $\mathbf{Q}$ is the inverse square root of ⍰, then

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

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

$$n\|\mathbf{Q}(\bar{\mathbf{X}}_n - \mu)\|^2 \xrightarrow{d} \chi^2(k) \tag{10.10}$$

where $\chi^2(k)$ is the chi-squared distribution with k degrees of freedom.

(Recall that k is the dimension of $\mathbf{X}_i$, the underlying random vectors.)

Your second exercise is to illustrate the convergence in (10.10) with a simulation.

In doing so, let

$$\mathbf{X}_i := \begin{pmatrix} W_i \\ U_i + W_i \end{pmatrix}$$

where

- each W_i is an IID draw from the uniform distribution on $[-1, 1]$.
- each U_i is an IID draw from the uniform distribution on $[-2, 2]$.
- U_i and W_i are independent of each other.

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.

10.6 Solutions

Solution to Exercise 10.5.1

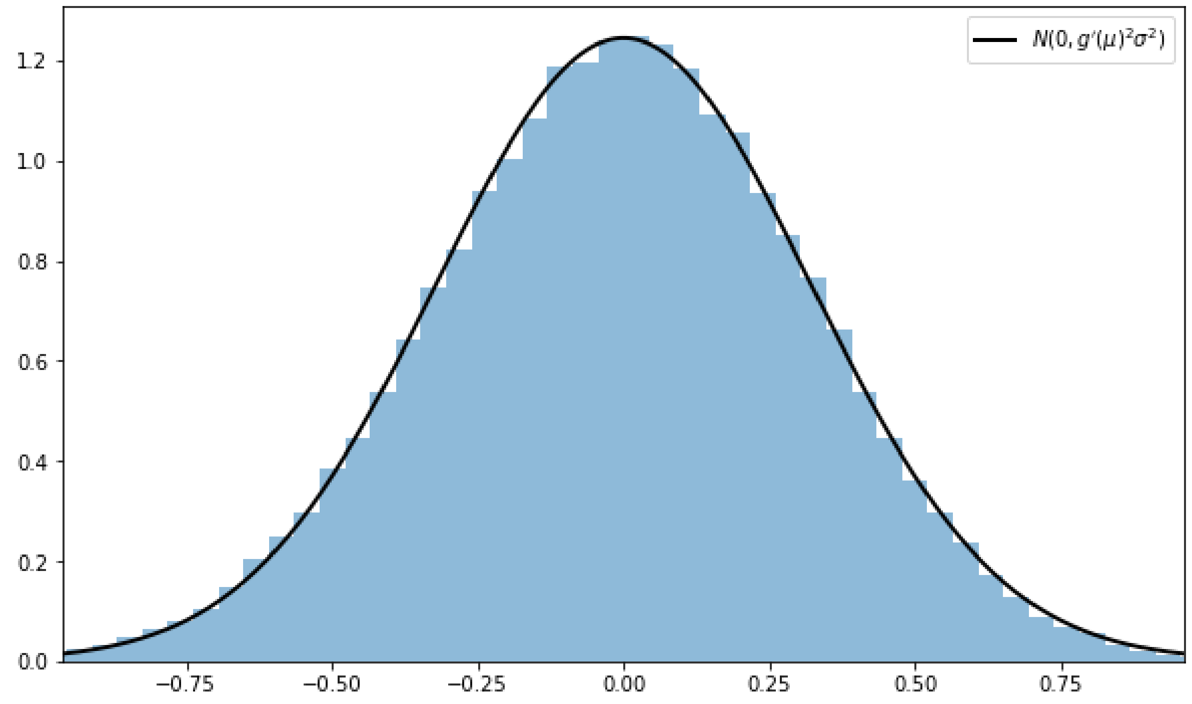
Here is one solution

```
"""
Illustrates the delta method, a consequence of the central limit theorem.
"""

# Set parameters
n = 250
replications = 100000
```

(continues on next page)

(continued from previous page)

```
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.

Solution to Exercise 10.5.2

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 $\mathbf{QY}$ is also normal.

Its mean is clearly $\mathbf{0}$, and its variance-covariance matrix is

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

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

Now we turn to the simulation exercise.

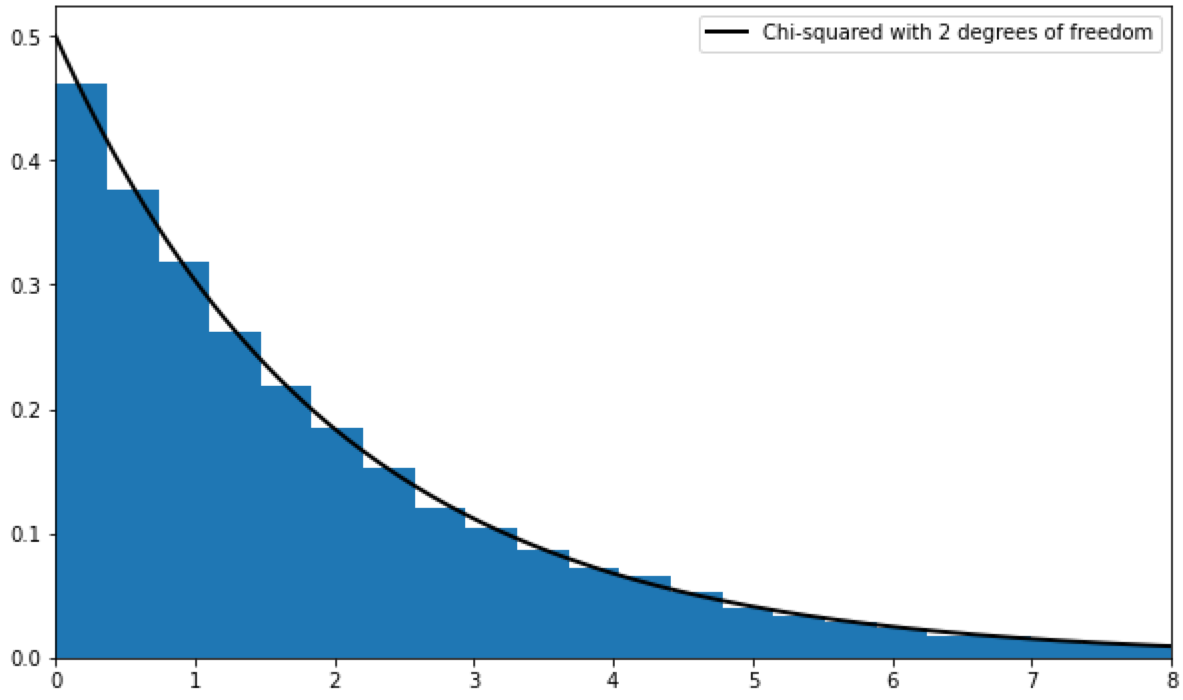
Our solution is as follows

```
# Set parameters
n = 250
replications = 50000
dw = uniform(loc=-1, scale=2)  # Uniform(-1, 1)
du = uniform(loc=-2, scale=4)  # Uniform(-2, 2)
sw, su = dw.std(), du.std()
vw, vu = sw**2, su**2
Σ = ((vw, vw), (vw, vw + vu))
Σ = np.array(Σ)

# Compute Σ^{-1/2}
Q = inv(sqrtm(Σ))

# Generate observations of the normalized sample mean
error_obs = np.empty((2, replications))
for i in range(replications):
    # Generate one sequence of bivariate shocks
    X = np.empty((2, n))
    W = dw.rvs(n)
    U = du.rvs(n)
    # Construct the n observations of the random vector
    X[0, :] = W
    X[1, :] = W + U
    # Construct the i-th observation of Y_n
    error_obs[:, i] = np.sqrt(n) * X.mean(axis=1)

# Premultiply by Q and then take the squared norm
temp = Q @ error_obs
chisq_obs = np.sum(temp**2, axis=0)

# Plot
fig, ax = plt.subplots(figsize=(10, 6))
```

(continues on next page)

(continued from previous page)

```
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

ELEVEN

TWO MEANINGS OF PROBABILITY

11.1 Overview

This lecture illustrates two distinct interpretations of a **probability distribution**

- A frequentist interpretation as **relative frequencies** anticipated to occur in a large i.i.d. sample
- A Bayesian interpretation as a **personal opinion** (about a parameter or list of parameters) after seeing a collection of observations

We recommend watching this video about **hypothesis testing** within the frequentist approach

https://youtu.be/8JIe_cz6qGA

After you watch that video, please watch the following video on the Bayesian approach to constructing **coverage intervals**

https://youtu.be/Pahyv9i_X2k

After you are familiar with the material in these videos, this lecture uses the Socratic method to to help consolidate your understanding of the different questions that are answered by

- a frequentist confidence interval
- a Bayesian coverage interval

We do this by inviting you to write some Python code.

It would be especially useful if you tried doing this after each question that we pose for you, before proceeding to read the rest of the lecture.

We provide our own answers as the lecture unfolds, but you'll learn more if you try writing your own code before reading and running ours.

Code for answering questions:

In addition to what's in Anaconda, this lecture will deploy the following library:

```
pip install prettytable
```

To answer our coding questions, we'll start with some imports

```
import numpy as np
import pandas as pd
import prettytable as pt
import matplotlib.pyplot as plt
from scipy.stats import binom
```

(continues on next page)

(continued from previous page)

```
import scipy.stats as st
%matplotlib inline
```

Empowered with these Python tools, we'll now explore the two meanings described above.

11.2 Frequentist Interpretation

Consider the following classic example.

The random variable X takes on possible values $k = 0, 1, 2, \ldots, n$ with probabilties

$$\text{Prob}(X = k|\theta) = \left(\frac{n!}{k!(n-k)!}\right) \theta^k (1-\theta)^{n-k}$$

where the fixed parameter $\theta \in (0, 1)$.

This is called the **binomial distribution**.

Here

- θ is the probability that one toss of a coin will be a head, an outcome that we encode as $Y = 1$.
- $1 - \theta$ is the probability that one toss of the coin will be a tail, an outcome that we denote $Y = 0$.
- X is the total number of heads that came up after flipping the coin n times.

Consider the following experiment:

Take I **independent** sequences of n **independent** flips of the coin

Notice the repeated use of the adjective **independent**:

- we use it once to describe that we are drawing n independent times from a **Bernoulli** distribution with parameter θ to arrive at one draw from a **Binomial** distribution with parameters θ, n.
- we use it again to describe that we are then drawing I sequences of n coin draws.

Let $y_h^i \in \{0, 1\}$ be the realized value of Y on the hth flip during the ith sequence of flips.

Let $\sum_{h=1}^n y_h^i$ denote the total number of times heads come up during the ith sequence of n independent coin flips.

Let f_k record the fraction of samples of length n for which $\sum_{h=1}^n y_h^i = k$:

$$f_k^I = \frac{\text{number of samples of length n for which } \sum_{h=1}^n y_h^i = k}{I}$$

The probability $\text{Prob}(X = k|\theta)$ answers the following question:

- As I becomes large, in what fraction of I independent draws of n coin flips should we anticipate k heads to occur?

As usual, a law of large numbers justifies this answer.

Exercise 11.2.1

1. Please write a Python class to compute f_k^I
2. Please use your code to compute $f_k^I, k = 0, \ldots, n$ and compare them to $\text{Prob}(X = k|\theta)$ for various values of θ, n and I
3. With the Law of Large numbers in mind, use your code to say something

Solution to Exercise 11.2.1

```python
class frequentist:

    def __init__(self, θ, n, I):

        '''
        initialization
        -----------------
        parameters:
        θ : probability that one toss of a coin will be a head with Y = 1
        n : number of independent flips in each independent sequence of draws
        I : number of independent sequence of draws

        '''

        self.θ, self.n, self.I = θ, n, I

    def binomial(self, k):

        '''compute the theoretical probability for specific input k'''

        θ, n = self.θ, self.n
        self.k = k
        self.P = binom.pmf(k, n, θ)

    def draw(self):

        '''draw n independent flips for I independent sequences'''

        θ, n, I = self.θ, self.n, self.I
        sample = np.random.rand(I, n)
        Y = (sample <= θ) * 1
        self.Y = Y

    def compute_fk(self, kk):

        '''compute f_{k}^I for specific input k'''

        Y, I = self.Y, self.I
        K = np.sum(Y, 1)
        f_kI = np.sum(K == kk) / I
        self.f_kI = f_kI
        self.kk = kk

    def compare(self):

        '''compute and print the comparison'''

        n = self.n
        comp = pt.PrettyTable()
        comp.field_names = ['k', 'Theoretical', 'Frequentist']
        self.draw()
        for i in range(n):
            self.binomial(i+1)
```

(continues on next page)

(continued from previous page)

```
            self.compute_fk(i+1)
            comp.add_row([i+1, self.P, self.f_kI])
        print(comp)
```

```
θ, n, k, I = 0.7, 20, 10, 1_000_000

freq = frequentist(θ, n, I)

freq.compare()
```

```
+----+------------------------+-------------+
| k  |      Theoretical       | Frequentist |
+----+------------------------+-------------+
| 1  | 1.6271660538000033e-09 |     0.0     |
| 2  | 3.606884752589999e-08  |     0.0     |
| 3  |  5.04963865362601e-07  |    1e-06    |
| 4  | 5.007558331512455e-06  |    6e-06    |
| 5  | 3.7389768875293014e-05 |   2.8e-05   |
| 6  | 0.00021810698510587546 |   0.000214  |
| 7  |  0.001017832597160754  |   0.000992  |
| 8  |  0.003859281930901185  |   0.003738  |
| 9  |  0.012006654896137007  |   0.011793  |
| 10 |  0.030817080900085007  |   0.030752  |
| 11 |   0.065369565545635    |   0.06523   |
| 12 |  0.11439673970486108   |   0.114338  |
| 13 |   0.1642619852172365   |   0.165208  |
| 14 |  0.19163898275344246   |   0.191348  |
| 15 |  0.17886305056987967   |   0.178536  |
| 16 |   0.1304209743738704   |   0.130838  |
| 17 |  0.07160367220526209   |   0.071312  |
| 18 |  0.027845872524268643  |   0.027988  |
| 19 |  0.006839337111223871  |   0.006858  |
| 20 | 0.0007979226629761189  |   0.00082   |
+----+------------------------+-------------+
```

From the table above, can you see the law of large numbers at work?

Let's do some more calculations.

Comparison with different θ

Now we fix

$$n = 20, k = 10, I = 1,000,000$$

We'll vary θ from 0.01 to 0.99 and plot outcomes against θ.

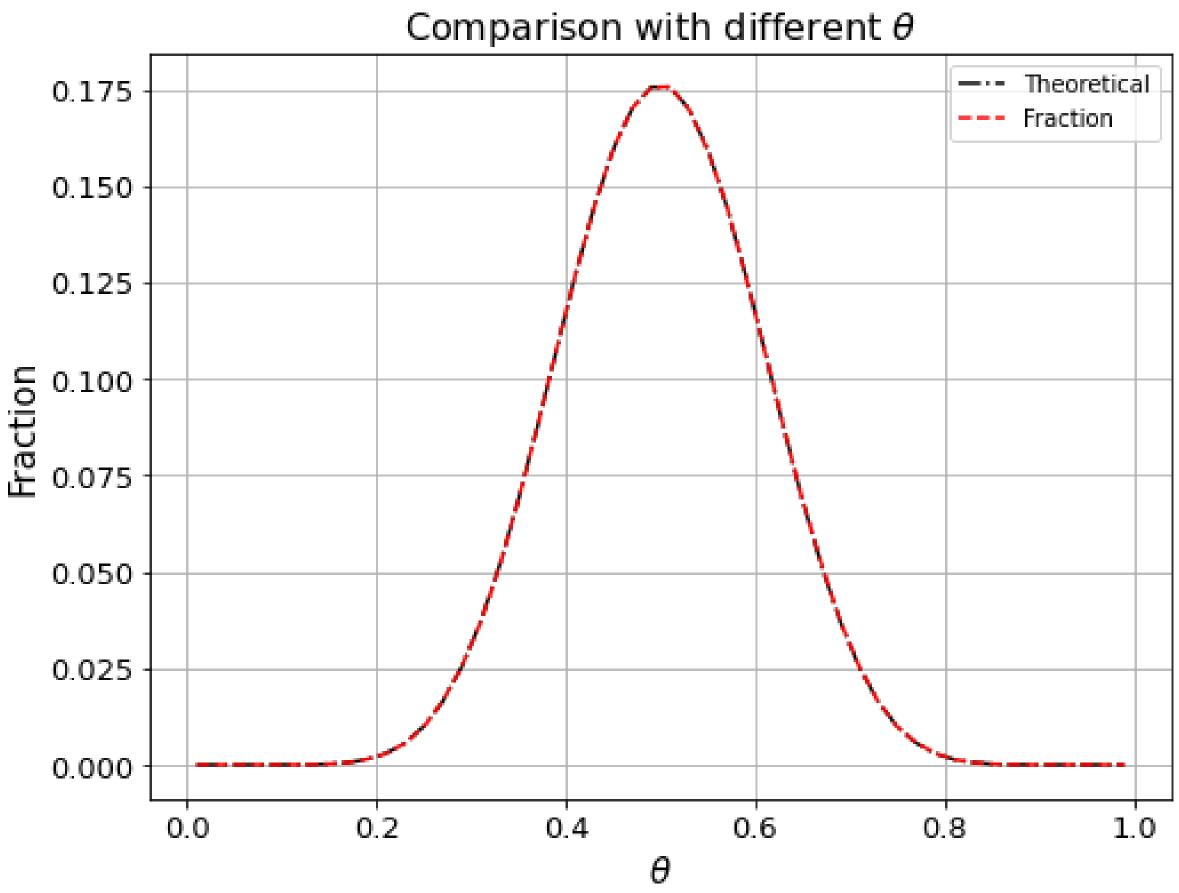
```
θ_low, θ_high, npt = 0.01, 0.99, 50
thetas = np.linspace(θ_low, θ_high, npt)
P = []
f_kI = []
for i in range(npt):
    freq = frequentist(thetas[i], n, I)
    freq.binomial(k)
```

(continues on next page)

(continued from previous page)

```
    freq.draw()
    freq.compute_fk(k)
    P.append(freq.P)
    f_kI.append(freq.f_kI)
```

```
fig, ax = plt.subplots(figsize=(8, 6))
ax.grid()
ax.plot(thetas, P, 'k-.', label='Theoretical')
ax.plot(thetas, f_kI, 'r--', label='Fraction')
plt.title(r'Comparison with different $\theta$', fontsize=16)
plt.xlabel(r'$\theta$', fontsize=15)
plt.ylabel('Fraction', fontsize=15)
plt.tick_params(labelsize=13)
plt.legend()
plt.show()
```



Comparison with different n

Now we fix $\theta = 0.7, k = 10, I = 1,000,000$ and vary n from 1 to 100.

Then we'll plot outcomes.

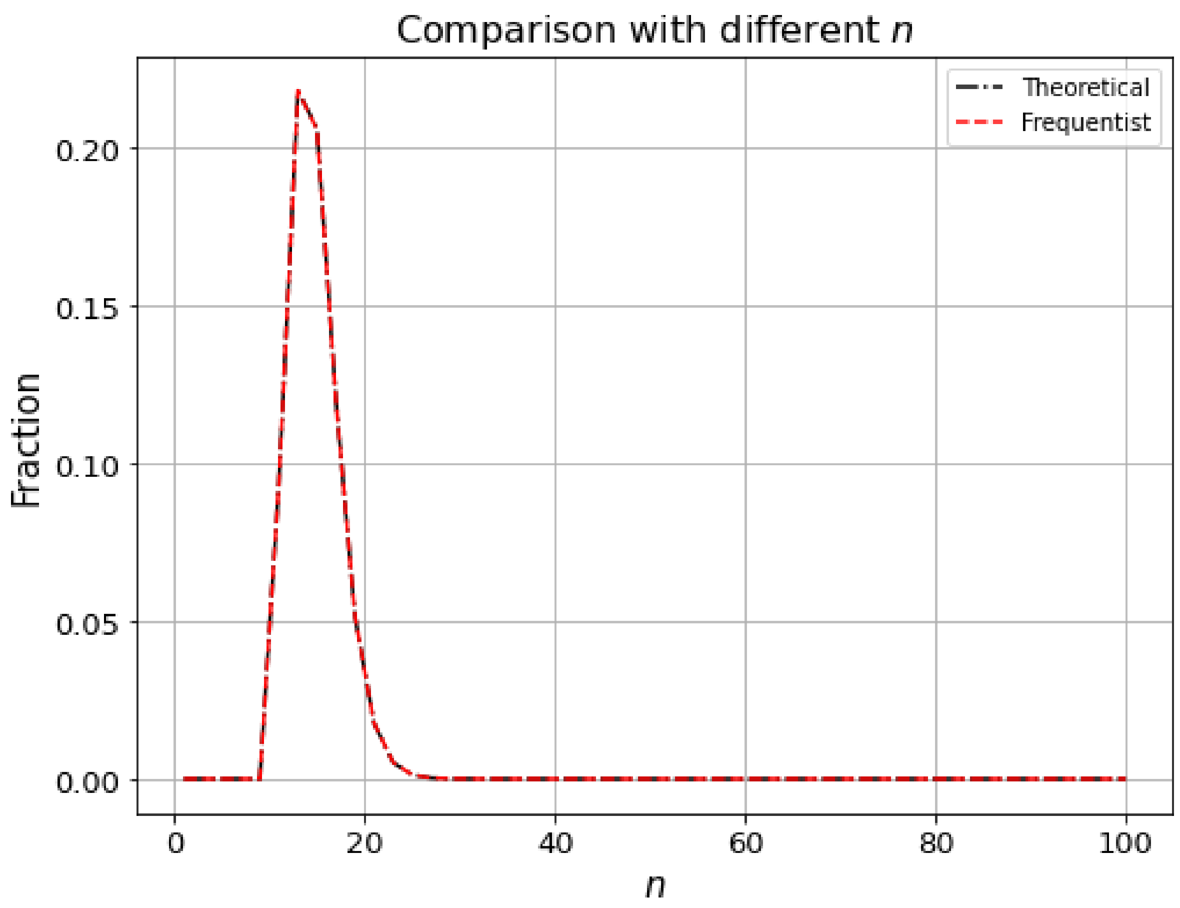
```
n_low, n_high, nn = 1, 100, 50
ns = np.linspace(n_low, n_high, nn, dtype='int')
```

(continues on next page)

(continued from previous page)

```python
P = []
f_kI = []
for i in range(nn):
    freq = frequentist(θ, ns[i], I)
    freq.binomial(k)
    freq.draw()
    freq.compute_fk(k)
    P.append(freq.P)
    f_kI.append(freq.f_kI)
```

```python
fig, ax = plt.subplots(figsize=(8, 6))
ax.grid()
ax.plot(ns, P, 'k-.', label='Theoretical')
ax.plot(ns, f_kI, 'r--', label='Frequentist')
plt.title(r'Comparison with different $n$', fontsize=16)
plt.xlabel(r'$n$', fontsize=15)
plt.ylabel('Fraction', fontsize=15)
plt.tick_params(labelsize=13)
plt.legend()
plt.show()
```

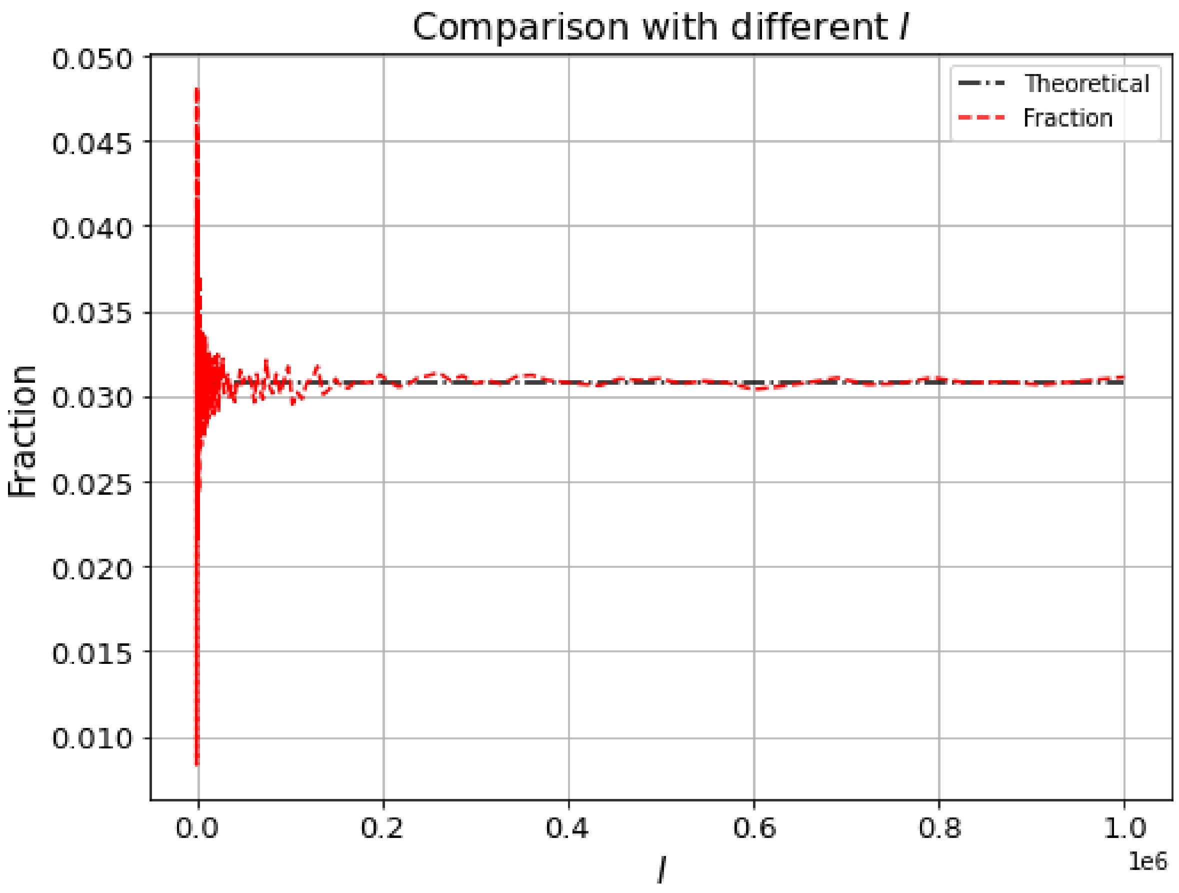


Comparison with different I

Now we fix $\theta = 0.7, n = 20, k = 10$ and vary $\log(I)$ from 2 to 7.

```
I_log_low, I_log_high, nI = 2, 6, 200
log_Is = np.linspace(I_log_low, I_log_high, nI)
Is = np.power(10, log_Is).astype(int)
P = []
f_kI = []
for i in range(nI):
    freq = frequentist(θ, n, Is[i])
    freq.binomial(k)
    freq.draw()
    freq.compute_fk(k)
    P.append(freq.P)
    f_kI.append(freq.f_kI)
```

```
fig, ax = plt.subplots(figsize=(8, 6))
ax.grid()
ax.plot(Is, P, 'k-.', label='Theoretical')
ax.plot(Is, f_kI, 'r--', label='Fraction')
plt.title(r'Comparison with different $I$', fontsize=16)
plt.xlabel(r'$I$', fontsize=15)
plt.ylabel('Fraction', fontsize=15)
plt.tick_params(labelsize=13)
plt.legend()
plt.show()
```



From the above graphs, we can see that I, **the number of independent sequences,** plays an important role.

When I becomes larger, the difference between theoretical probability and frequentist estimate becomes smaller.

Also, as long as I is large enough, changing θ or n does not substantially change the accuracy of the observed fraction as an approximation of θ.

The Law of Large Numbers is at work here.

For each draw of an independent sequence, $\textrm{Prob}(X_i = k|\theta)$ is the same, so aggregating all draws forms an i.i.d sequence of a binary random variable $\rho_{k,i}, i = 1, 2, ...I$, with a mean of $\textrm{Prob}(X = k|\theta)$ and a variance of

$$n \cdot \textrm{Prob}(X = k|\theta) \cdot (1 - \textrm{Prob}(X = k|\theta)).$$

So, by the LLN, the average of $P_{k,i}$ converges to:

$$E[\rho_{k,i}] = \textrm{Prob}(X = k|\theta) = \left(\frac{n!}{k!(n-k)!}\right)\theta^k(1-\theta)^{n-k}$$

as I goes to infinity.

11.3 Bayesian Interpretation

We again use a binomial distribution.

But now we don't regard θ as being a fixed number.

Instead, we think of it as a **random variable**.

θ is described by a probability distribution.

But now this probability distribution means something different than a relative frequency that we can anticipate to occur in a large i.i.d. sample.

Instead, the probability distribution of θ is now a summary of our views about likely values of θ either

- **before** we have seen **any** data at all, or
- **before** we have seen **more** data, after we have seen **some** data

Thus, suppose that, before seeing any data, you have a personal prior probability distribution saying that

$$P(\theta) = \frac{\theta^{\alpha-1}(1-\theta)^{\beta-1}}{B(\alpha, \beta)}$$

where $B(\alpha, \beta)$ is a **beta function** , so that $P(\theta)$ is a **beta distribution** with parameters α, β.

Exercise 11.3.1

a) Please write down the **likelihood function** for a sample of length n from a binomial distribution with parameter θ.

b) Please write down the **posterior** distribution for θ after observing one flip of the coin.

c) Please pretend that the true value of $\theta = .4$ and that someone who doesn't know this has a beta prior distribution with parameters with $\beta = \alpha = .5$.

d) Please write a Python class to simulate this person's personal posterior distribution for θ for a *single* sequence of n draws.

e) Please plot the posterior distribution for θ as a function of θ as n grows as $1, 2, ...$

f) For various n's, please describe and compute a Bayesian coverage interval for the interval $[.45, .55]$.

g) Please tell what question a Bayesian coverage interval answers.

h) Please compute the Posterior probabililty that $\theta \in [.45, .55]$ for various values of sample size n.

i) Please use your Python class to study what happens to the posterior distribution as $n \to +\infty$, again assuming that the true value of $\theta = .4$, though it is unknown to the person doing the updating via Bayes' Law.

Solution to Exercise 11.3.1

a) Please write down the **likelihood function** and the **posterior** distribution for θ after observing one flip of our coin.

Suppose the outcome is **Y**.

The likelihood function is:

$$L(Y|\theta) = \text{Prob}(X = Y|\theta) = \theta^Y(1-\theta)^{1-Y}$$

b) Please write the **posterior** distribution for θ after observing one flip of our coin.

The prior distribution is

$$\text{Prob}(\theta) = \frac{\theta^{\alpha-1}(1-\theta)^{\beta-1}}{B(\alpha,\beta)}$$

We can derive the posterior distribution for θ via

$$\begin{aligned}
\text{Prob}(\theta|Y) &= \frac{\text{Prob}(Y|\theta)\text{Prob}(\theta)}{\text{Prob}(Y)} \\
&= \frac{\text{Prob}(Y|\theta)\text{Prob}(\theta)}{\int_0^1 \text{Prob}(Y|\theta)\text{Prob}(\theta)d\theta} \\
&= \frac{\theta^Y(1-\theta)^{1-Y}\frac{\theta^{\alpha-1}(1-\theta)^{\beta-1}}{B(\alpha,\beta)}}{\int_0^1 \theta^Y(1-\theta)^{1-Y}\frac{\theta^{\alpha-1}(1-\theta)^{\beta-1}}{B(\alpha,\beta)}d\theta} \\
&= \frac{\theta^{Y+\alpha-1}(1-\theta)^{1-Y+\beta-1}}{\int_0^1 \theta^{Y+\alpha-1}(1-\theta)^{1-Y+\beta-1}d\theta}
\end{aligned}$$

which means that

$$\text{Prob}(\theta|Y) \sim \text{Beta}(\alpha+Y, \beta+(1-Y))$$

c) Please pretend that the true value of $\theta = .4$ and that someone who doesn't know this has a beta prior with $\beta = \alpha = .5$.

d) Please write a Python class to simulate this person's personal posterior distribution for θ for a *single* sequence of n draws.

```python
class Bayesian:

    def __init__(self, θ=0.4, n=1_000_000, α=0.5, β=0.5):
        """
        Parameters:
        ----------
        θ : float, ranging from [0,1].
           probability that one toss of a coin will be a head with Y = 1

        n : int.
           number of independent flips in an independent sequence of draws
```

(continues on next page)

(continued from previous page)

```
        α&β : int or float.
             parameters of the prior distribution on ϑ

        """
        self.θ, self.n, self.α, self.β = θ, n, α, β
        self.prior = st.beta(α, β)

    def draw(self):
        """
        simulate a single sequence of draws of length n, given probability ϑ

        """
        array = np.random.rand(self.n)
        self.draws = (array < self.θ).astype(int)

    def form_single_posterior(self, step_num):
        """
        form a posterior distribution after observing the first step_num elements of␣
↪the draws

        Parameters
        ----------
        step_num: int.
               number of steps observed to form a posterior distribution

        Returns
        -------
        the posterior distribution for sake of plotting in the subsequent steps

        """
        heads_num = self.draws[:step_num].sum()
        tails_num = step_num - heads_num

        return st.beta(self.α+heads_num, self.β+tails_num)

    def form_posterior_series(self,num_obs_list):
        """
        form a series of posterior distributions that form after observing different␣
↪number of draws.

        Parameters
        ----------
        num_obs_list: a list of int.
               a list of the number of observations used to form a series of␣
↪posterior distributions.

        """
        self.posterior_list = []
        for num in num_obs_list:
            self.posterior_list.append(self.form_single_posterior(num))
```

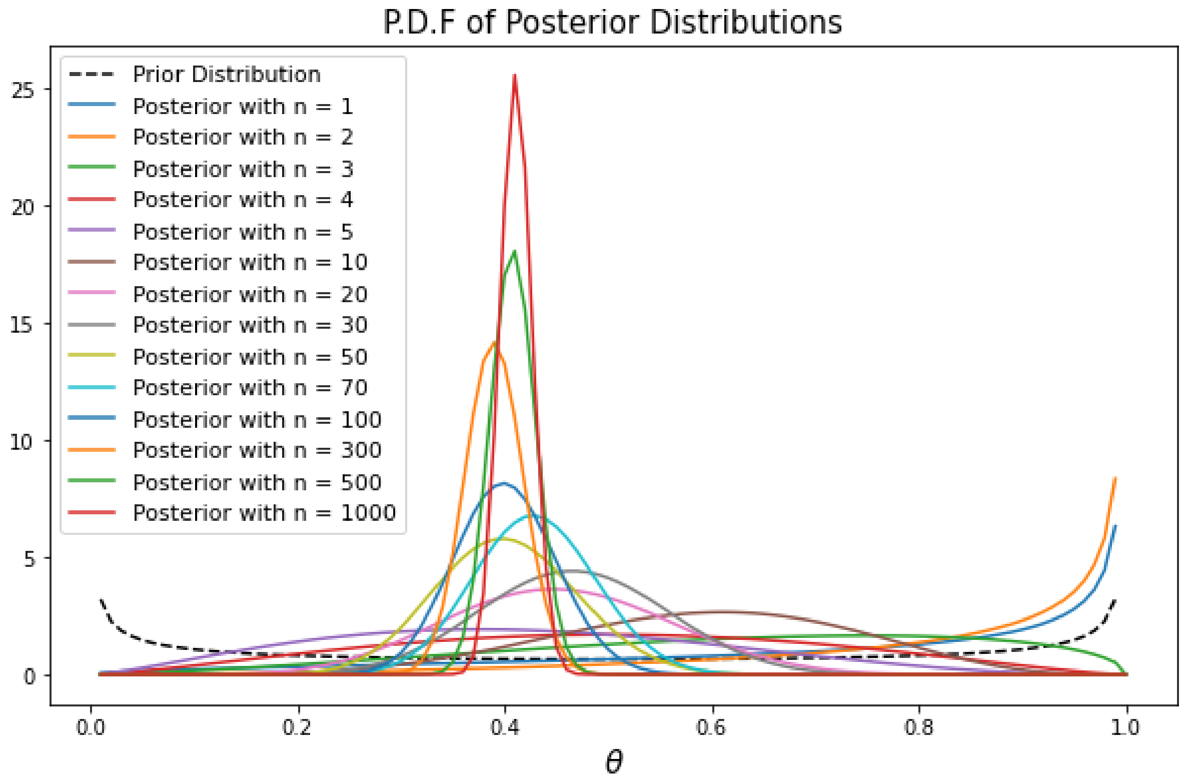
e) Please plot the posterior distribution for θ as a function of θ as n grows from $1, 2, \ldots$.

```
Bay_stat = Bayesian()
Bay_stat.draw()
```

(continues on next page)

(continued from previous page)

```
num_list = [1, 2, 3, 4, 5, 10, 20, 30, 50, 70, 100, 300, 500, 1000, # this line for␣
↪finite n
            5000, 10_000, 50_000, 100_000, 200_000, 300_000]  # this line for␣
↪approximately infinite n

Bay_stat.form_posterior_series(num_list)

θ_values = np.linspace(0.01, 1, 100)

fig, ax = plt.subplots(figsize=(10, 6))

ax.plot(θ_values, Bay_stat.prior.pdf(θ_values), label='Prior Distribution', color='k',
↪ linestyle='--')

for ii, num in enumerate(num_list[:14]):
    ax.plot(θ_values, Bay_stat.posterior_list[ii].pdf(θ_values), label='Posterior␣
↪with n = %d' % num)

ax.set_title('P.D.F of Posterior Distributions', fontsize=15)
ax.set_xlabel(r"$\theta$", fontsize=15)

ax.legend(fontsize=11)
plt.show()
```



f) For various n's, please describe and compute .05 and .95 quantiles for posterior probabilities.

```
upper_bound = [ii.ppf(0.05) for ii in Bay_stat.posterior_list[:14]]
lower_bound = [ii.ppf(0.95) for ii in Bay_stat.posterior_list[:14]]
```

(continues on next page)

(continued from previous page)

```
interval_df = pd.DataFrame()
interval_df['upper'] = upper_bound
interval_df['lower'] = lower_bound
interval_df.index = num_list[:14]
interval_df = interval_df.T
interval_df
```

```
             1         2         3        4         5        10        20  \
upper  0.228520  0.430741  0.235534  0.16528  0.127776  0.347322  0.280091
lower  0.998457  0.999132  0.937587  0.83472  0.739366  0.814884  0.629953

            30        50        70       100       300       500      1000
upper  0.32360  0.292234  0.334679  0.322252  0.344599  0.372306  0.385621
lower  0.61429  0.516104  0.526749  0.481969  0.436961  0.444479  0.436759
```

As n increases, we can see that Bayesian coverage intervals narrow and move toward 0.4.

g) Please tell what question a Bayesian coverage interval answers.

The Bayesian coverage interval tells the range of θ that corresponds to the $[p_1, p_2]$ quantiles of the cumulative probability distribution (CDF) of the posterior distribution.

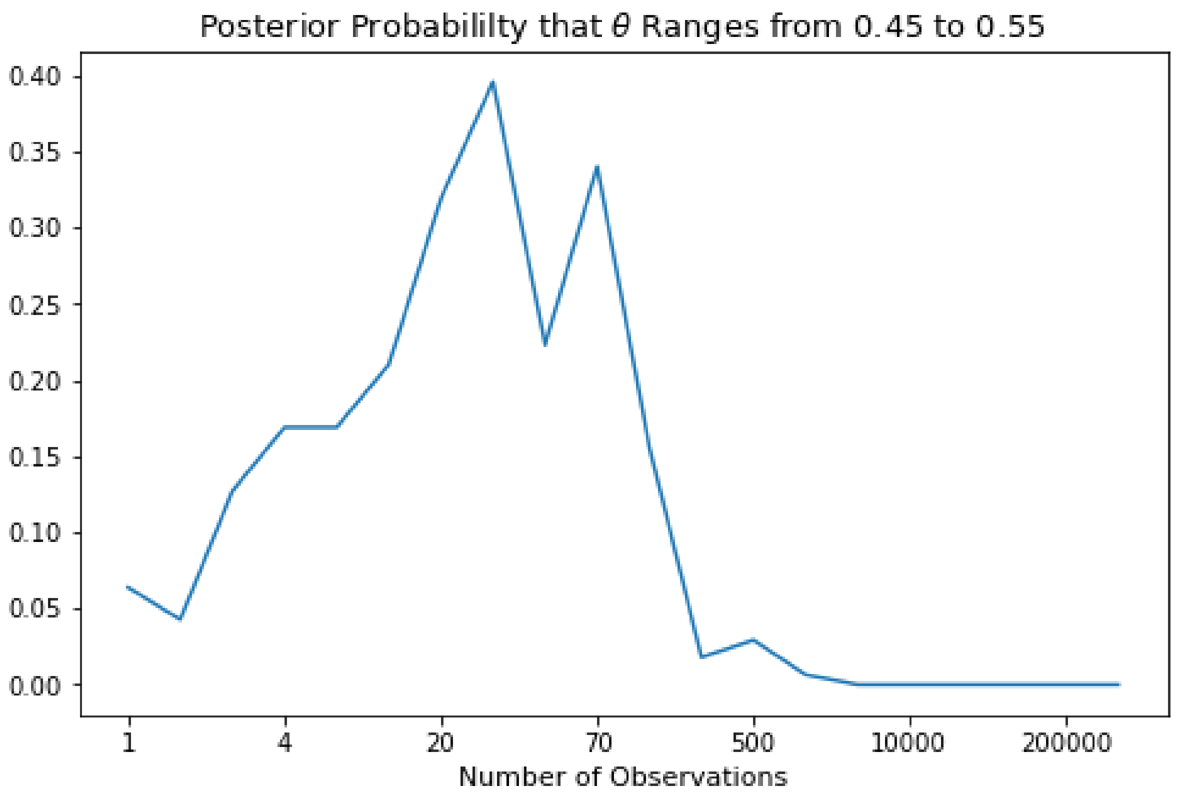
To construct the coverage interval we first compute a posterior distribution of the unknown parameter θ.

If the CDF is $F(\theta)$, then the Bayesian coverage interval $[a, b]$ for the interval $[p_1, p_2]$ is described by

$$F(a) = p_1, F(b) = p_2$$

h) Please compute the Posterior probabililty that $\theta \in [.45, .55]$ for various values of sample size n.

```
left_value, right_value = 0.45, 0.55

posterior_prob_list=[ii.cdf(right_value)-ii.cdf(left_value) for ii in Bay_stat.
↪posterior_list]

fig, ax = plt.subplots(figsize=(8, 5))
ax.plot(posterior_prob_list)
ax.set_title('Posterior Probabililty that '+ r"$\theta$" +' Ranges from %.2f to %.2f'
↪%(left_value, right_value),
             fontsize=13)
ax.set_xticks(np.arange(0, len(posterior_prob_list), 3))
ax.set_xticklabels(num_list[::3])
ax.set_xlabel('Number of Observations', fontsize=11)

plt.show()
```



Notice that in the graph above the posterior probabililty that $\theta \in [.45, .55]$ typically exhibits a hump shape as n increases.

Two opposing forces are at work.

The first force is that the individual adjusts his belief as he observes new outcomes, so his posterior probability distribution becomes more and more realistic, which explains the rise of the posterior probabililty.

However, $[.45, .55]$ actually excludes the true $\theta = .4$ that generates the data.

As a result, the posterior probabililty drops as larger and larger samples refine his posterior probability distribution of θ.

The descent seems precipitous only because of the scale of the graph that has the number of observations increasing disproportionately.

When the number of observations becomes large enough, our Bayesian becomes so confident about θ that he considers $\theta \in [.45, .55]$ very unlikely.

That is why we see a nearly horizontal line when the number of observations exceeds 500.

i) Please use your Python class to study what happens to the posterior distribution as $n \to +\infty$, again assuming that the true value of $\theta = .4$, though it is unknown to the person doing the updating via Bayes' Law.

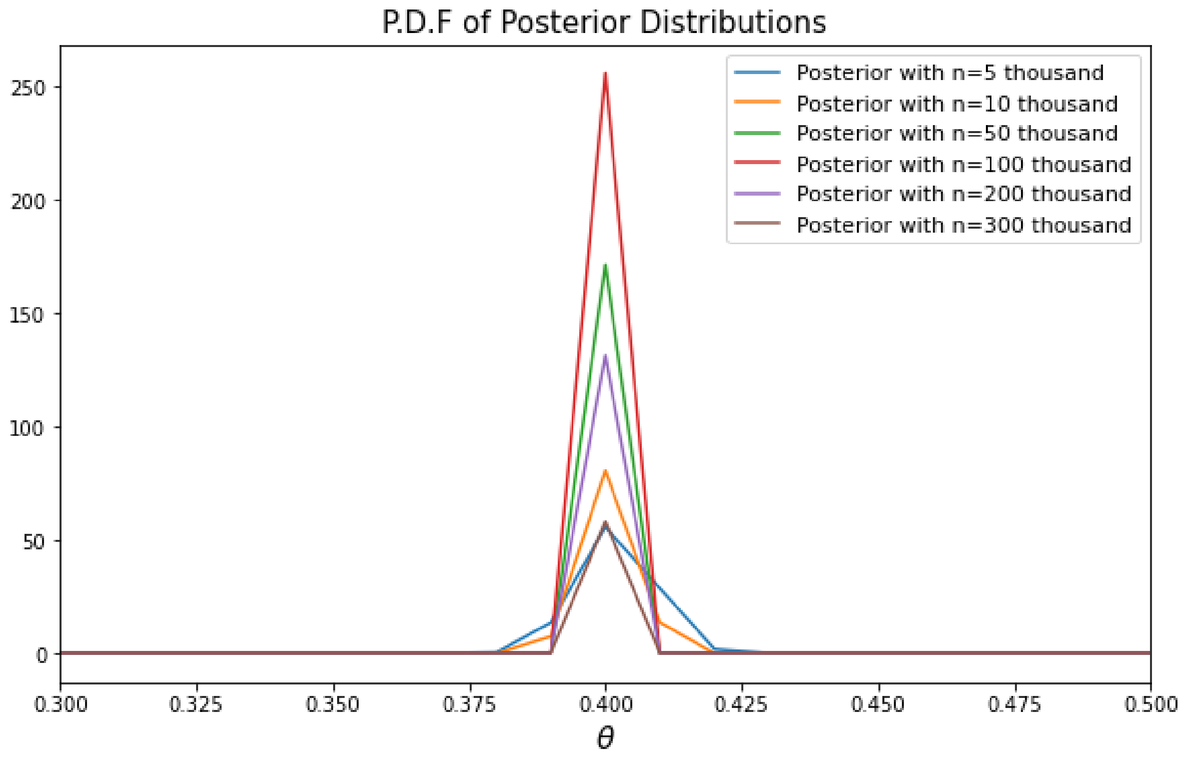
Using the Python class we made above, we can see the evolution of posterior distributions as n approaches infinity.

```
fig, ax = plt.subplots(figsize=(10, 6))

for ii, num in enumerate(num_list[14:]):
    ii += 14
    ax.plot(θ_values, Bay_stat.posterior_list[ii].pdf(θ_values),
            label='Posterior with n=%d thousand' % (num/1000))

ax.set_title('P.D.F of Posterior Distributions', fontsize=15)
```

(continues on next page)

(continued from previous page)

```
ax.set_xlabel(r"$\theta$", fontsize=15)
ax.set_xlim(0.3, 0.5)

ax.legend(fontsize=11)
plt.show()
```

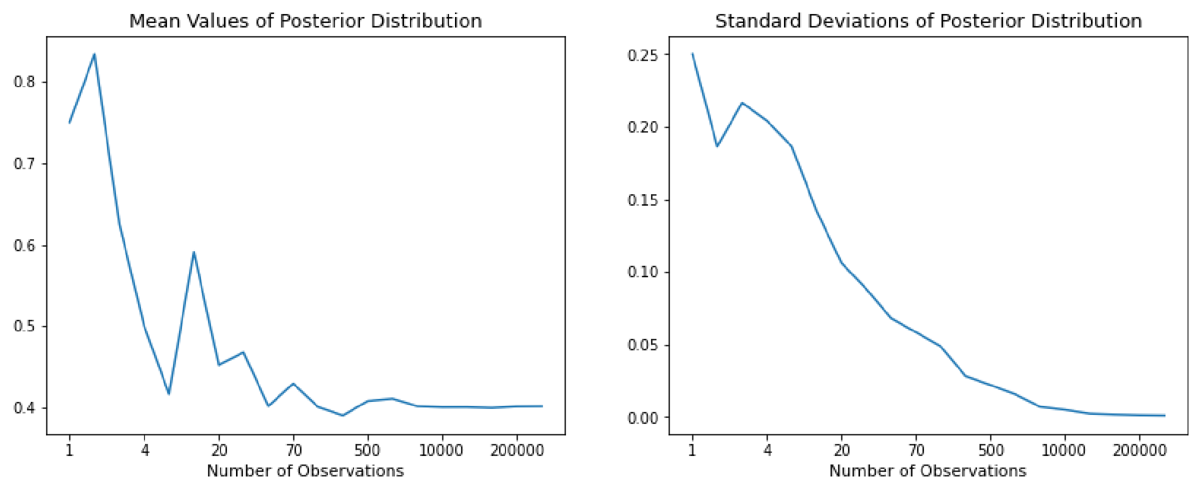


As n increases, we can see that the probability density functions *concentrate* on 0.4, the true value of θ.

Here the posterior means converges to 0.4 while the posterior standard deviations converges to 0 from above.

To show this, we compute the means and variances statistics of the posterior distributions.

```
mean_list = [ii.mean() for ii in Bay_stat.posterior_list]
std_list = [ii.std() for ii in Bay_stat.posterior_list]

fig, ax = plt.subplots(1, 2, figsize=(14, 5))

ax[0].plot(mean_list)
ax[0].set_title('Mean Values of Posterior Distribution', fontsize=13)
ax[0].set_xticks(np.arange(0, len(mean_list), 3))
ax[0].set_xticklabels(num_list[::3])
ax[0].set_xlabel('Number of Observations', fontsize=11)

ax[1].plot(std_list)
ax[1].set_title('Standard Deviations of Posterior Distribution', fontsize=13)
ax[1].set_xticks(np.arange(0, len(std_list), 3))
ax[1].set_xticklabels(num_list[::3])
ax[1].set_xlabel('Number of Observations', fontsize=11)

plt.show()
```



How shall we interpret the patterns above?

The answer is encoded in the Bayesian updating formulas.

It is natural to extend the one-step Bayesian update to an n-step Bayesian update.

$$\begin{aligned}
\text{Prob}(\theta|k) &= \frac{\text{Prob}(\theta,k)}{\text{Prob}(k)} = \frac{\text{Prob}(k|\theta)*\text{Prob}(\theta)}{\text{Prob}(k)} = \frac{\text{Prob}(k|\theta)*\text{Prob}(\theta)}{\int_0^1 \text{Prob}(k|\theta)*\text{Prob}(\theta)d\theta} \\
&= \frac{\binom{N}{k}(1-\theta)^{N-k}\theta^k * \frac{\theta^{\alpha-1}(1-\theta)^{\beta-1}}{B(\alpha,\beta)}}{\int_0^1 \binom{N}{k}(1-\theta)^{N-k}\theta^k * \frac{\theta^{\alpha-1}(1-\theta)^{\beta-1}}{B(\alpha,\beta)}d\theta} \\
&= \frac{(1-\theta)^{\beta+N-k-1}*\theta^{\alpha+k-1}}{\int_0^1 (1-\theta)^{\beta+N-k-1}*\theta^{\alpha+k-1}d\theta} \\
&= Beta(\alpha+k,\beta+N-k)
\end{aligned}$$

A beta distribution with α and β has the following mean and variance.

The mean is $\frac{\alpha}{\alpha+\beta}$

The variance is $\frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)}$

- α can be viewed as the number of successes
- β can be viewed as the number of failures

The random variables k and $N-k$ are governed by Binomial Distribution with $\theta = 0.4$.

Call this the true data generating process.

According to the Law of Large Numbers, for a large number of observations, observed frequencies of k and $N-k$ will be described by the true data generating process, i.e., the population probability distribution that we assumed when generating the observations on the computer. (See Exercise 11.2.1).

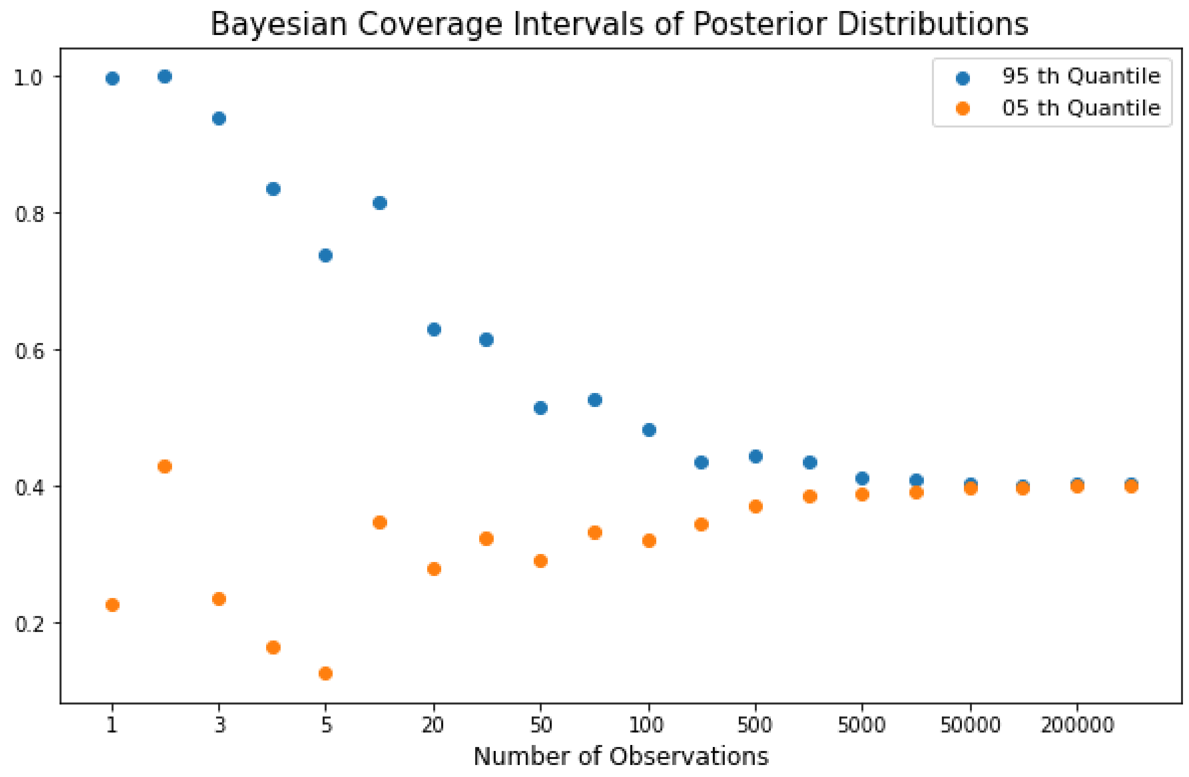
Consequently, the mean of the posterior distribution converges to 0.4 and the variance withers to zero.

```
upper_bound = [ii.ppf(0.95) for ii in Bay_stat.posterior_list]
lower_bound = [ii.ppf(0.05) for ii in Bay_stat.posterior_list]

fig, ax = plt.subplots(figsize=(10, 6))
```

(continues on next page)

(continued from previous page)

```
ax.scatter(np.arange(len(upper_bound)), upper_bound, label='95 th Quantile')
ax.scatter(np.arange(len(lower_bound)), lower_bound, label='05 th Quantile')

ax.set_xticks(np.arange(0, len(upper_bound), 2))
ax.set_xticklabels(num_list[::2])
ax.set_xlabel('Number of Observations', fontsize=12)
ax.set_title('Bayesian Coverage Intervals of Posterior Distributions', fontsize=15)

ax.legend(fontsize=11)
plt.show()
```



After observing a large number of outcomes, the posterior distribution collapses around 0.4.

Thus, the Bayesian statististian comes to believe that θ is near $.4$.

As shown in the figure above, as the number of observations grows, the Bayesian coverage intervals (BCIs) become narrower and narrower around 0.4.

However, if you take a closer look, you will find that the centers of the BCIs are not exactly 0.4, due to the persistent influence of the prior distribution and the randomness of the simulation path.

CHAPTER

TWELVE

MULTIVARIATE HYPERGEOMETRIC DISTRIBUTION

Contents

12.1 Overview

This lecture describes how an administrator deployed a **multivariate hypergeometric distribution** in order to access the fairness of a procedure for awarding research grants.

In the lecture we'll learn about

- properties of the multivariate hypergeometric distribution
- first and second moments of a multivariate hypergeometric distribution
- using a Monte Carlo simulation of a multivariate normal distribution to evaluate the quality of a normal approximation
- the administrator's problem and why the multivariate hypergeometric distribution is the right tool

12.2 The Administrator's Problem

An administrator in charge of allocating research grants is in the following situation.

To help us forget details that are none of our business here and to protect the anonymity of the administrator and the subjects, we call research proposals **balls** and continents of residence of authors of a proposal a **color**.

There are K_i balls (proposals) of color i.

There are c distinct colors (continents of residence).

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

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.

12.2.1 Details of the Awards Procedure Under Study

Let k_i be the number of balls of color i that are drawn.

Things have to add up so $\sum_{i=1}^{c} k_i = n$.

Under the hypothesis that the selection process judges proposals on their quality and that quality is independent of continent of the author's continent of residence, the administrator views the outcome of the selection procedure as a random vector

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

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**.

12.2.2 Multivariate Hypergeometric Distribution

Let's start with some imports.

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import matplotlib.cm as cm
import numpy as np
from scipy.special import comb
from scipy.stats import normaltest
from numba import njit, prange
```

To recapitulate, we assume there are in total c types of objects in an urn.

If there are K_i type i object in the urn and we take n draws at random without replacement, then the numbers of type i objects in the sample $(k_1, k_2, \dots, k_c)$ has the multivariate hypergeometric distribution.

Note again that $N = \sum_{i=1}^{c} K_i$ is the total number of objects in the urn and $n = \sum_{i=1}^{c} k_i$.

Notation

We use the following notation for **binomial coefficients**: $\binom{m}{q} = \frac{m!}{(m-q)!}$.

The multivariate hypergeometric distribution has the following properties:

Probability mass function:

$$\Pr\{X_i = k_i \ \forall i\} = \frac{\prod_{i=1}^{c} \binom{K_i}{k_i}}{\binom{N}{n}}$$

Mean:

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

Variances and covariances:

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

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

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

```
class Urn:

    def __init__(self, K_arr):
        """
        Initialization given the number of each type i object in the urn.

        Parameters
        ----------
        K_arr: ndarray(int)
            number of each type i object.
        """

        self.K_arr = np.array(K_arr)
        self.N = np.sum(K_arr)
```

(continues on next page)

(continued from previous page)

```
        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
        μ = n * K_arr / N

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

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

        return μ, Σ

    def simulate(self, n, size=1, seed=None):
        """
        Simulate a sample from multivariate hypergeometric
        distribution where at each draw we take n objects
```

(continues on next page)

(continued from previous page)

```
        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
```

12.3 Usage

12.3.1 First example

Apply this to an example from wiki:

Suppose there are 5 black, 10 white, and 15 red marbles in an urn. If six marbles are chosen without replacement, the probability that exactly two of each color are chosen is

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

```
# construct the urn
K_arr = [5, 10, 15]
urn = Urn(K_arr)
```

Now use the Urn Class method `pmf` to compute the probability of the outcome $X = \begin{pmatrix} 2 & 2 & 2 \end{pmatrix}$

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

```
array([0.0795756])
```

We can use the code to compute probabilities of a list of possible outcomes by constructing a 2-dimensional array `k_arr` and `pmf` will return an array of probabilities for observing each case.

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

```
array([0.0795756, 0.1061008])
```

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

```
n = 6
μ, Σ = urn.moments(n)
```

```
μ
```

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

```
Σ
```

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

12.3.2 Back to The Administrator's Problem

Now let's turn to the grant administrator's problem.

Here the array of numbers of i objects in the urn is $(157, 11, 46, 24)$.

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

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

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

```
array([0.01547738])
```

We can compute probabilities of three possible outcomes by constructing a 3-dimensional arrays `k_arr` and utilizing the method `pmf` of the `Urn` class.

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

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

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

```
n = 6 # number of draws
μ, Σ = urn.moments(n)
```

```
# mean
μ
```

```
array([3.95798319, 0.27731092, 1.15966387, 0.60504202])
```

```
# variance-covariance matrix
Σ
```

```
array([[ 1.31862604, -0.17907267, -0.74884935, -0.39070401],
       [-0.17907267,  0.25891399, -0.05246715, -0.02737417],
       [-0.74884935, -0.05246715,  0.91579029, -0.11447379],
       [-0.39070401, -0.02737417, -0.11447379,  0.53255196]])
```

We can simulate a large sample and verify that sample means and covariances closely approximate the population means and covariances.

```
size = 10_000_000
sample = urn.simulate(n, size=size)
```

```
# mean
np.mean(sample, 0)
```

```
array([3.9575241, 0.2772789, 1.1598499, 0.6053471])
```

```
# variance covariance matrix
np.cov(sample.T)
```

```
array([[ 1.31769043, -0.17888995, -0.74840691, -0.39039358],
       [-0.17888995,  0.25887754, -0.05253011, -0.02745748],
       [-0.74840691, -0.05253011,  0.9158426 , -0.11490558],
       [-0.39039358, -0.02745748, -0.11490558,  0.53275664]])
```

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

12.3.3 Quality of Normal Approximation

To judge the quality of a multivariate normal approximation to the multivariate hypergeometric distribution, we draw a large sample from a multivariate normal distribution with the mean vector and covariance matrix for the corresponding multivariate hypergeometric distribution and compare the simulated distribution with the population multivariate hypergeometric distribution.

```
sample_normal = np.random.multivariate_normal(μ, Σ, size=size)
```

```
def bivariate_normal(x, y, μ, Σ, i, j):

    μ_x, μ_y = μ[i], μ[j]
    σ_x, σ_y = np.sqrt(Σ[i, i]), np.sqrt(Σ[j, j])
    σ_xy = Σ[i, j]

    x_μ = x - μ_x
    y_μ = y - μ_y

    ρ = σ_xy / (σ_x * σ_y)
    z = x_μ**2 / σ_x**2 + y_μ**2 / σ_y**2 - 2 * ρ * x_μ * y_μ / (σ_x * σ_y)
    denom = 2 * np.pi * σ_x * σ_y * np.sqrt(1 - ρ**2)
```

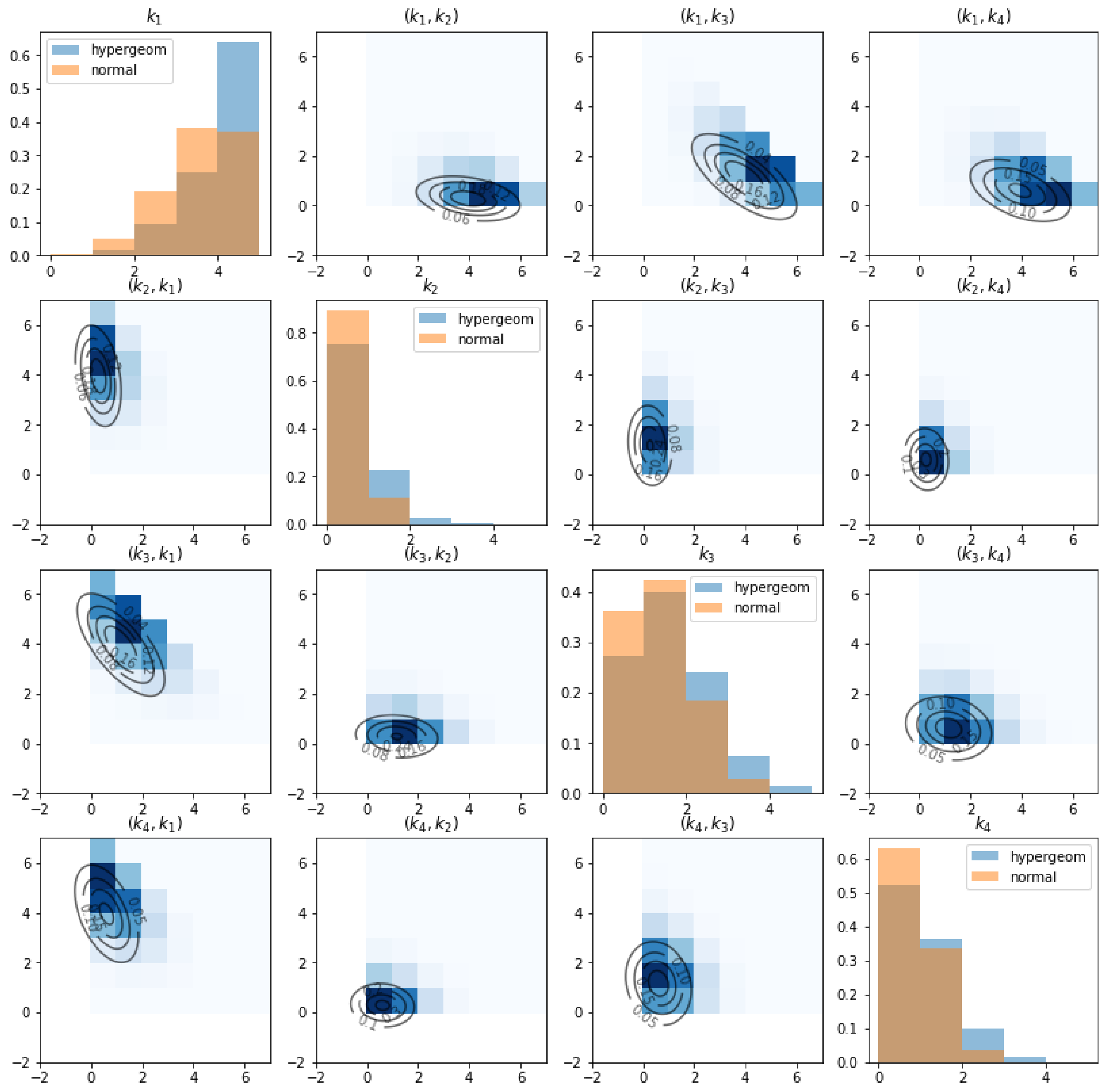
(continues on next page)

(continued from previous page)

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

```
@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
```

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

# grids for ploting the bivariate Gaussian
x_grid = np.linspace(-2, n+1, 100)
y_grid = np.linspace(-2, n+1, 100)
X, Y = np.meshgrid(x_grid, y_grid)

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

        # bivariate Gaussian density function
        Z = bivariate_normal(X, Y, μ, Σ, i, j)
        cs = axs[i, j].contour(X, Y, Z, 4, colors="black", alpha=0.6)
        axs[i, j].clabel(cs, inline=1, fontsize=10)

        # empirical multivariate hypergeometric distrbution
        count_mat = count(sample[:, i], sample[:, j], n)
        axs[i, j].pcolor(count_mat.T/size, cmap='Blues')
        axs[i, j].set_title('$(k_{' +str(i+1) +'}, k_{' + str(j+1) + '})$')

plt.show()
```



The diagonal graphs plot the marginal distributions of k_i for each i using histograms.

Note the substantial differences between hypergeometric distribution and the approximating normal distribution.

The off-diagonal graphs plot the empirical joint distribution of k_i and k_j for each pair (i, j).

The darker the blue, the more data points are contained in the corresponding cell. (Note that k_i is on the x-axis and k_j is on the y-axis).

The contour maps plot the bivariate Gaussian density function of (k_i, k_j) with the population mean and covariance given by slices of μ and Σ that we computed above.

Let's also test the normality for each k_i using `scipy.stats.normaltest` that implements D'Agostino and Pearson's test that combines skew and kurtosis to form an omnibus test of normality.

The null hypothesis is that the sample follows normal distribution.

> `normaltest` returns an array of p-values associated with tests for each k_i sample.

```
test_multihyper = normaltest(sample)
test_multihyper.pvalue
```

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

As we can see, all the p-values are almost 0 and the null hypothesis is soundly rejected.

By contrast, the sample from normal distribution does not reject the null hypothesis.

```
test_normal = normaltest(sample_normal)
test_normal.pvalue
```

```
array([0.20342918, 0.37411736, 0.32851877, 0.44077682])
```

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

CHAPTER

THIRTEEN

MULTIVARIATE NORMAL DISTRIBUTION

Contents

13.1 Overview

This lecture describes a workhorse in probability theory, statistics, and economics, namely, the **multivariate normal distribution**.

In this lecture, you will learn formulas for

- the joint distribution of a random vector x of length N
- marginal distributions for all subvectors of x
- conditional distributions for subvectors of x conditional on other subvectors of x

We will use the multivariate normal distribution to formulate some useful models:

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

13.2 The Multivariate Normal Distribution

This lecture defines a Python class `MultivariateNormal` to be used to generate **marginal** and **conditional** distributions associated with a multivariate normal distribution.

For a multivariate normal distribution it is very convenient that

- conditional expectations equal linear least squares projections
- conditional distributions are characterized by multivariate linear regressions

We apply our Python class to some examples.

We use the following imports:

```python
%matplotlib inline
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from numba import njit
import statsmodels.api as sm
```

Assume that an $N \times 1$ random vector z has a multivariate normal probability density.

This means that the probability density takes the form

$$f(z; \mu, \Sigma) = (2\pi)^{-\left(\frac{N}{2}\right)} \det(\Sigma)^{-\frac{1}{2}} \exp\left(-.5(z-\mu)' \Sigma^{-1}(z-\mu)\right)$$

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

The covariance matrix Σ is symmetric and positive definite.

```python
@njit
def f(z, μ, Σ):
    """
    The density function of multivariate normal distribution.

    Parameters
    ---------------
    z: ndarray(float, dim=2)
        random vector, N by 1
    μ: ndarray(float, dim=1 or 2)
        the mean of z, N by 1
    Σ: ndarray(float, dim=2)
        the covarianece matrix of z, N by 1
    """

```

(continues on next page)

(continued from previous page)

```
    z = np.atleast_2d(z)
    μ = np.atleast_2d(μ)
    Σ = np.atleast_2d(Σ)

    N = z.size

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

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

For some integer $k \in \{1, \dots, N-1\}$, partition z as

$$z = \left[\begin{array}{c} z_1 \\ z_2 \end{array} \right],$$

where z_1 is an $(N-k) \times 1$ vector and z_2 is a $k \times 1$ vector.

Let

$$\mu = \left[\begin{array}{c} \mu_1 \\ \mu_2 \end{array} \right], \quad \Sigma = \left[\begin{array}{cc} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{array} \right]$$

be corresponding partitions of μ and Σ.

The **marginal** distribution of z_1 is

- multivariate normal with mean μ_1 and covariance matrix Σ_{11}.

The **marginal** distribution of z_2 is

- multivariate normal with mean μ_2 and covariance matrix Σ_{22}.

The distribution of z_1 **conditional** on z_2 is

- multivariate normal with mean

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

and covariance matrix

$$\hat{\Sigma}_{11} = \Sigma_{11} - \Sigma_{12} \Sigma_{22}^{-1} \Sigma_{21} = \Sigma_{11} - \beta \Sigma_{22} \beta'$$

where

$$\beta = \Sigma_{12} \Sigma_{22}^{-1}$$

is an $(N-k) \times k$ matrix of **population regression coefficients** of the $(N-k) \times 1$ random vector $z_1 - \mu_1$ on the $k \times 1$ random vector $z_2 - \mu_2$.

The following class constructs a multivariate normal distribution instance with two methods.

- a method `partition` computes β, taking k as an input
- a method `cond_dist` computes either the distribution of z_1 conditional on z_2 or the distribution of z_2 conditional on z_1

```
class MultivariateNormal:
    """
    Class of multivariate normal distribution.

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

    Arguments
    ---------
    μ, Σ:
        see parameters
    μs: list(ndarray(float, dim=1))
        list of mean vectors μ1 and μ2 in order
    Σs: list(list(ndarray(float, dim=2)))
        2 dimensional list of covariance matrices
        Σ11, Σ12, Σ21, Σ22 in order
    βs: list(ndarray(float, dim=1))
        list of regression coefficients β1 and β2 in order
    """

    def __init__(self, μ, Σ):
        "initialization"
        self.μ = np.array(μ)
        self.Σ = np.atleast_2d(Σ)

    def partition(self, k):
        """
        Given k, partition the random vector z into a size k vector z1
        and a size N-k vector z2. Partition the mean vector μ into
        μ1 and μ2, and the covariance matrix Σ into Σ11, Σ12, Σ21, Σ22
        correspondingly. Compute the regression coefficients β1 and β2
        using the partitioned arrays.
        """
        μ = self.μ
        Σ = self.Σ

        self.μs = [μ[:k], μ[k:]]
        self.Σs = [[Σ[:k, :k], Σ[:k, k:]],
                   [Σ[k:, :k], Σ[k:, k:]]]

        self.βs = [self.Σs[0][1] @ np.linalg.inv(self.Σs[1][1]),
                   self.Σs[1][0] @ np.linalg.inv(self.Σs[0][0])]

    def cond_dist(self, ind, z):
        """
        Compute the conditional distribution of z1 given z2, or reversely.
        Argument ind determines whether we compute the conditional
        distribution of z1 (ind=0) or z2 (ind=1).

        Returns
        ---------
        μ_hat: ndarray(float, ndim=1)
            The conditional mean of z1 or z2.
```

(continues on next page)

(continued from previous page)

```
        Σ_hat: ndarray(float, ndim=2)
            The conditional covariance matrix of z1 or z2.
        """
        β = self.βs[ind]
        μs = self.μs
        Σs = self.Σs

        μ_hat = μs[ind] + β @ (z - μs[1-ind])
        Σ_hat = Σs[ind][ind] - β @ Σs[1-ind][1-ind] @ β.T

        return μ_hat, Σ_hat
```

Let's put this code to work on a suite of examples.

We begin with a simple bivariate example; after that we'll turn to a trivariate example.

We'll compute population moments of some conditional distributions using our `MultivariateNormal` class.

For fun we'll also compute sample analogs of the associated population regressions by generating simulations and then computing linear least squares regressions.

We'll compare those linear least squares regressions for the simulated data to their population counterparts.

13.3 Bivariate Example

We start with a bivariate normal distribution pinned down by

$$\mu = \left[\begin{array}{c} .5 \\ 1.0 \end{array}\right], \quad \Sigma = \left[\begin{array}{cc} 1 & .5 \\ .5 & 1 \end{array}\right]$$

```
μ = np.array([.5, 1.])
Σ = np.array([[1., .5], [.5 ,1.]])

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

```
k = 1 # choose partition

# partition and compute regression coefficients
multi_normal.partition(k)
multi_normal.βs[0],multi_normal.βs[1]
```

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

Let's illustrate the fact that you *can regress anything on anything else*.

We have computed everything we need to compute two regression lines, one of z_2 on z_1, the other of z_1 on z_2.

We'll represent these regressions as

$$z_1 = a_1 + b_1 z_2 + \epsilon_1$$

and

$$z_2 = a_2 + b_2 z_1 + \epsilon_2$$

where we have the population least squares orthogonality conditions

$$E\epsilon_1 z_2 = 0$$

and

$$E\epsilon_2 z_1 = 0$$

Let's compute a_1, a_2, b_1, b_2.

```
beta = multi_normal.βs

a1 = μ[0] - beta[0]*μ[1]
b1 = beta[0]

a2 = μ[1] - beta[1]*μ[0]
b2 = beta[1]
```

Let's print out the intercepts and slopes.

For the regression of z_1 on z_2 we have

```
print ("a1 = ", a1)
print ("b1 = ", b1)
```

```
a1 =  [[0.]]
b1 =  [[0.5]]
```

For the regression of z_2 on z_1 we have

```
print ("a2 = ", a2)
print ("b2 = ", b2)
```

```
a2 =  [[0.75]]
b2 =  [[0.5]]
```

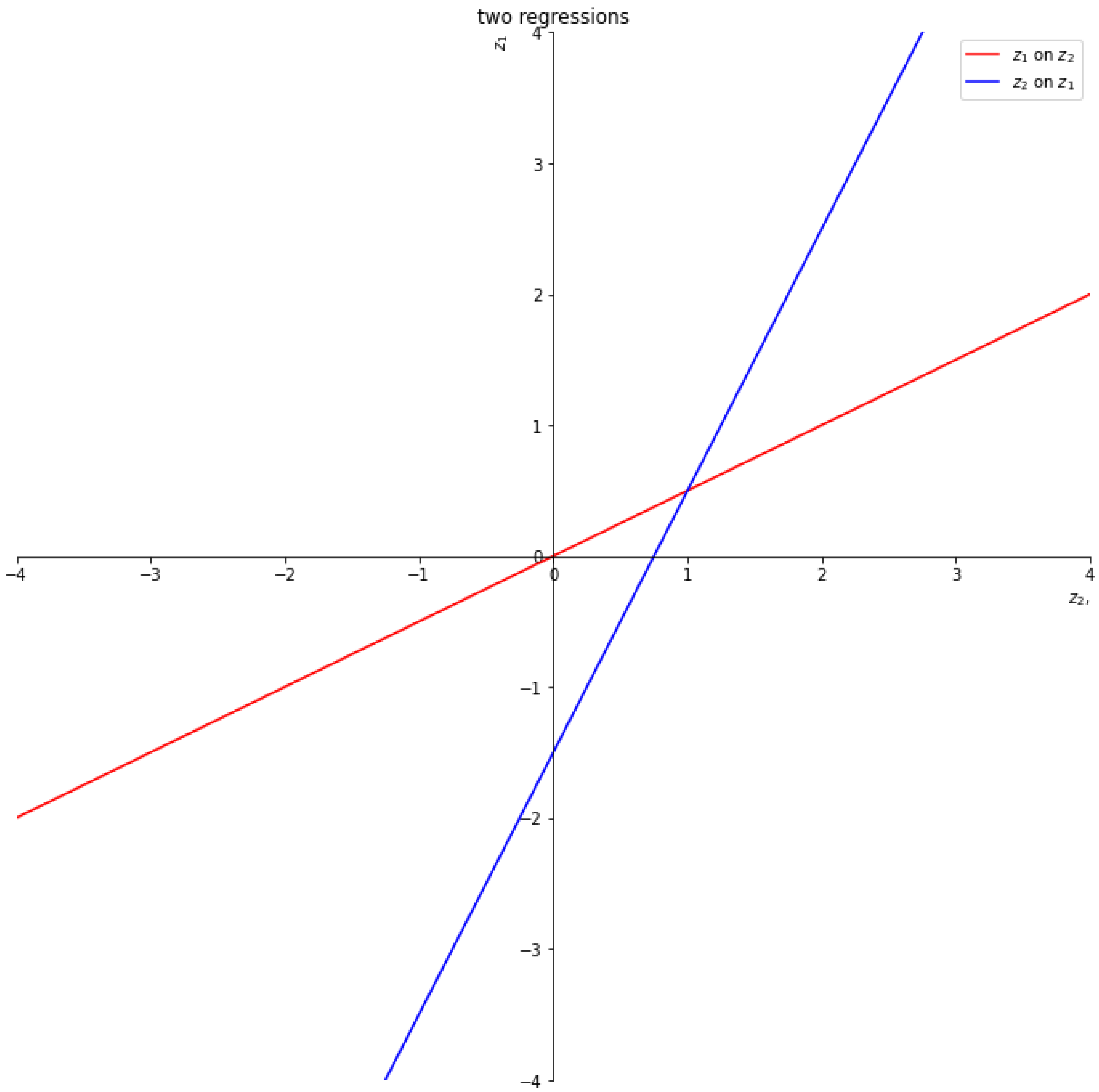
Now let's plot the two regression lines and stare at them.

```
z2 = np.linspace(-4,4,100)


a1 = np.squeeze(a1)
b1 = np.squeeze(b1)

a2 = np.squeeze(a2)
b2 = np.squeeze(b2)

z1  = b1*z2 + a1


z1h = z2/b2 - a2/b2


fig = plt.figure(figsize=(12,12))
ax = fig.add_subplot(1, 1, 1)
```

(continues on next page)

(continued from previous page)

```
ax.set(xlim=(-4, 4), ylim=(-4, 4))
ax.spines['left'].set_position('center')
ax.spines['bottom'].set_position('zero')
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')
plt.ylabel('$z_1$', loc = 'top')
plt.xlabel('$z_2$,', loc = 'right')
plt.title('two regressions')
plt.plot(z2,z1, 'r', label = "$z_1$ on $z_2$")
plt.plot(z2,z1h, 'b', label = "$z_2$ on $z_1$")
plt.legend()
plt.show()
```



The red line is the expectation of z_1 conditional on z_2.

The intercept and slope of the red line are

```
print("a1 = ", a1)
print("b1 = ", b1)
```

```
a1 =  0.0
b1 =  0.5
```

The blue line is the expectation of z_2 conditional on z_1.

The intercept and slope of the blue line are

```
print("-a2/b2 = ", - a2/b2)
print("1/b2 = ", 1/b2)
```

```
-a2/b2 =  -1.5
1/b2 =  2.0
```

We can use these regression lines or our code to compute conditional expectations.

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

After that we'll reverse what are on the left and right sides of the regression.

```
# compute the cond. dist. of z1
ind = 1
z1 = np.array([5.]) # given z1

μ2_hat, Σ2_hat = multi_normal.cond_dist(ind, z1)
print('μ2_hat, Σ2_hat = ', μ2_hat, Σ2_hat)
```

```
μ2_hat, Σ2_hat =  [3.25] [[0.75]]
```

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

```
# compute the cond. dist. of z1
ind = 0
z2 = np.array([5.]) # given z2

μ1_hat, Σ1_hat = multi_normal.cond_dist(ind, z2)
print('μ1_hat, Σ1_hat = ', μ1_hat, Σ1_hat)
```

```
μ1_hat, Σ1_hat =  [2.5] [[0.75]]
```

Let's compare the preceding population mean and variance with outcomes from drawing a large sample and then regressing $z_1 - \mu_1$ on $z_2 - \mu_2$.

We know that

$$E z_1 | z_2 = (\mu_1 - \beta\mu_2) + \beta z_2$$

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 $\hat{\Sigma}_1$.

```
n = 1_000_000 # sample size

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

# OLS regression
μ1, μ2 = multi_normal.μs
results = sm.OLS(z1_data - μ1, z2_data - μ2).fit()
```

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

```
multi_normal.βs[0], results.params
```

```
(array([[0.5]]), array([0.49951561]))
```

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

```
Σ1_hat, results.resid @ results.resid.T / (n - 1)
```

```
(array([[0.75]]), 0.7499568468007555)
```

Lastly, let's compute the estimate of $\widehat{Ez_1|z_2}$ and compare it with $\hat{\mu}_1$

```
μ1_hat, results.predict(z2 - μ2) + μ1
```

```
(array([2.5]), array([2.49806245]))
```

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

A Law of Large Numbers explains why sample analogues approximate population objects.

13.4 Trivariate Example

Let's apply our code to a trivariate example.

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

```
μ = np.random.random(3)
C = np.random.random((3, 3))
Σ = C @ C.T # positive semi-definite

multi_normal = MultivariateNormal(μ, Σ)
```

```
μ, Σ
```

```
(array([0.81665356, 0.64520806, 0.89203644]),
 array([[2.3328485 , 0.69063069, 1.74677053],
        [0.69063069, 0.2965043 , 0.59653754],
        [1.74677053, 0.59653754, 1.37811334]]))
```

```
k = 1
multi_normal.partition(k)
```

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

```
ind = 0
z2 = np.array([2., 5.])

μ1_hat, Σ1_hat = multi_normal.cond_dist(ind, z2)
```

```
n = 1_000_000
data = np.random.multivariate_normal(μ, Σ, size=n)
z1_data = data[:, :k]
z2_data = data[:, k:]
```

```
μ1, μ2 = multi_normal.μs
results = sm.OLS(z1_data - μ1, z2_data - μ2).fit()
```

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

```
multi_normal.βs[0], results.params
```

```
(array([[-1.71053213,  2.00793873]]), array([-1.70978271,  2.0076083 ]))
```

```
Σ1_hat, results.resid @ results.resid.T / (n - 1)
```

```
(array([[0.00678627]]), 0.006790326919375728)
```

```
μ1_hat, results.predict(z2 - μ2) + μ1
```

```
(array([6.74777753]), array([6.74743547]))
```

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

13.5 One Dimensional Intelligence (IQ)

Let's move closer to a real-life example, namely, inferring a one-dimensional measure of intelligence called IQ from a list of test scores.

The ith test score y_i equals the sum of an unknown scalar IQ θ and a random variable w_i.

$$y_i = \theta + \sigma_y w_i, \quad i = 1, \dots, n$$

The distribution of IQ's for a cross-section of people is a normal random variable described by

$$\theta = \mu_\theta + \sigma_\theta w_{n+1}.$$

We assume that the noises $\{w_i\}_{i=1}^N$ in the test scores are IID and not correlated with IQ.

We also assume that $\{w_i\}_{i=1}^{n+1}$ are i.i.d. standard normal:

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

The following system describes the $(n+1) \times 1$ random vector X that interests us:

$$X = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \\ \theta \end{bmatrix} = \begin{bmatrix} \mu_\theta \\ \mu_\theta \\ \vdots \\ \mu_\theta \\ \mu_\theta \end{bmatrix} + \begin{bmatrix} \sigma_y & 0 & \cdots & 0 & \sigma_\theta \\ 0 & \sigma_y & \cdots & 0 & \sigma_\theta \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & \sigma_y & \sigma_\theta \\ 0 & 0 & \cdots & 0 & \sigma_\theta \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \\ w_{n+1} \end{bmatrix},$$

or equivalently,

$$X = \mu_\theta 1_{n+1} + Dw$$

where $X = \begin{bmatrix} y \\ \theta \end{bmatrix}$, 1_{n+1} is a vector of 1s of size $n+1$, and D is an $n+1$ by $n+1$ matrix.

Let's define a Python function that constructs the mean μ and covariance matrix Σ of the random vector X that we know is governed by a multivariate normal distribution.

As arguments, the function takes the number of tests n, the mean μ_θ and the standard deviation σ_θ of the IQ distribution, and the standard deviation of the randomness in test scores σ_y.

```python
def construct_moments_IQ(n, μθ, σθ, σy):

    μ_IQ = np.full(n+1, μθ)

    D_IQ = np.zeros((n+1, n+1))
    D_IQ[range(n), range(n)] = σy
    D_IQ[:, n] = σθ

    Σ_IQ = D_IQ @ D_IQ.T

    return μ_IQ, Σ_IQ, D_IQ
```

Now let's consider a specific instance of this model.

Assume we have recorded 50 test scores and we know that $\mu_\theta = 100$, $\sigma_\theta = 10$, and $\sigma_y = 10$.

We can compute the mean vector and covariance matrix of X easily with our `construct_moments_IQ` function as follows.

```python
n = 50
μθ, σθ, σy = 100., 10., 10.

μ_IQ, Σ_IQ, D_IQ = construct_moments_IQ(n, μθ, σθ, σy)
μ_IQ, Σ_IQ, D_IQ
```

```
(array([100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
        100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
        100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
```

(continues on next page)

(continued from previous page)

```
        100., 100., 100., 100., 100., 100., 100., 100., 100., 100., 100.,
        100., 100., 100., 100., 100., 100., 100.]),
 array([[200., 100., 100., ..., 100., 100., 100.],
        [100., 200., 100., ..., 100., 100., 100.],
        [100., 100., 200., ..., 100., 100., 100.],
        ...,
        [100., 100., 100., ..., 200., 100., 100.],
        [100., 100., 100., ..., 100., 200., 100.],
        [100., 100., 100., ..., 100., 100., 100.]]),
 array([[10.,  0.,  0., ...,  0.,  0., 10.],
        [ 0., 10.,  0., ...,  0.,  0., 10.],
        [ 0.,  0., 10., ...,  0.,  0., 10.],
        ...,
        [ 0.,  0.,  0., ..., 10.,  0., 10.],
        [ 0.,  0.,  0., ...,  0., 10., 10.],
        [ 0.,  0.,  0., ...,  0.,  0., 10.]]))
```

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

We want to regress IQ, the random variable θ (*what we don't know*), on the vector y of test scores (*what we do know*).

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

```
multi_normal_IQ = MultivariateNormal(μ_IQ, Σ_IQ)

k = n
multi_normal_IQ.partition(k)
```

Using the generator `multivariate_normal`, we can make one draw of the random vector from our distribution and then compute the distribution of θ conditional on our test scores.

Let's do that and then print out some pertinent quantities.

```
x = np.random.multivariate_normal(μ_IQ, Σ_IQ)
y = x[:-1] # test scores
θ = x[-1]  # IQ
```

```
# the true value
θ
```

```
104.54899674277466
```

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

In the following code, `ind` sets the variables on the right side of the regression.

Given the way we have defined the vector X, we want to set `ind=1` in order to make θ the left side variable in the population regression.

```
ind = 1
multi_normal_IQ.cond_dist(ind, y)
```

```
(array([104.49298531]), array([[1.96078431]]))
```

The first number is the conditional mean $\hat{\mu}_\theta$ and the second is the conditional variance $\hat{\Sigma}_\theta$.

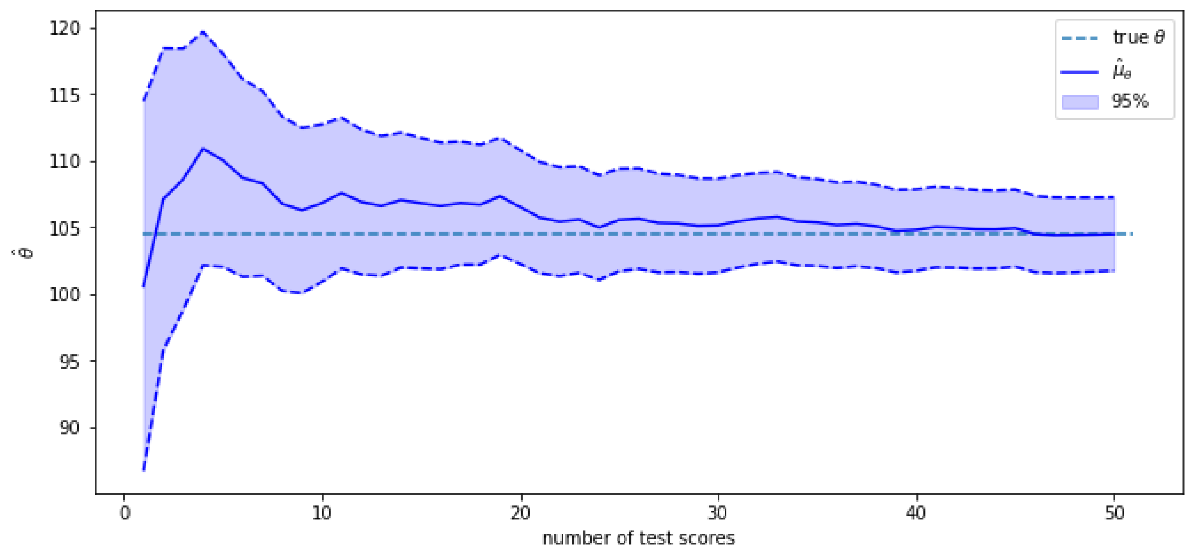
How do additional test scores affect our inferences?

To shed light on this, we compute a sequence of conditional distributions of θ by varying the number of test scores in the conditioning set from 1 to n.

We'll make a pretty graph showing how our judgment of the person's IQ change as more test results come in.

```
# array for containing moments
μθ_hat_arr = np.empty(n)
Σθ_hat_arr = np.empty(n)

# loop over number of test scores
for i in range(1, n+1):
    # construction of multivariate normal distribution instance
    μ_IQ_i, Σ_IQ_i, D_IQ_i = construct_moments_IQ(i, μθ, σθ, σy)
    multi_normal_IQ_i = MultivariateNormal(μ_IQ_i, Σ_IQ_i)

    # partition and compute conditional distribution
    multi_normal_IQ_i.partition(i)
    scores_i = y[:i]
    μθ_hat_i, Σθ_hat_i = multi_normal_IQ_i.cond_dist(1, scores_i)

    # store the results
    μθ_hat_arr[i-1] = μθ_hat_i[0]
    Σθ_hat_arr[i-1] = Σθ_hat_i[0, 0]

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

```
μθ_hat_lower = μθ_hat_arr - 1.96 * σθ_hat_arr
μθ_hat_higher = μθ_hat_arr + 1.96 * σθ_hat_arr

plt.hlines(θ, 1, n+1, ls='--', label='true $θ$')
plt.plot(range(1, n+1), μθ_hat_arr, color='b', label='$\hat{μ}_{θ}$')
plt.plot(range(1, n+1), μθ_hat_lower, color='b', ls='--')
plt.plot(range(1, n+1), μθ_hat_higher, color='b', ls='--')
plt.fill_between(range(1, n+1), μθ_hat_lower, μθ_hat_higher,
                 color='b', alpha=0.2, label='95%')

plt.xlabel('number of test scores')
plt.ylabel('$\hat{θ}$')
plt.legend()

plt.show()
```



The solid blue line in the plot above shows $\hat{\mu}_\theta$ as a function of the number of test scores that we have recorded and conditioned on.

The blue area shows the span that comes from adding or 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 were to drive the number of tests $n \rightarrow +\infty$, the conditional standard deviation $\hat{\sigma}_\theta$ would converge to 0 at rate $\frac{1}{n^{.5}}$.

13.6 Information as Surprise

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

We can represent the random vector X defined above as

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

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

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

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

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

Let c_i be the ith element in the last row of C.

Then we can write

$$\theta = \mu_{\theta} + c_1\epsilon_1 + c_2\epsilon_2 + \dots + c_n\epsilon_n + c_{n+1}\epsilon_{n+1} \tag{13.1}$$

The mutual orthogonality of the ϵ_i's provides us with an informative way to interpret them in light of equation (13.1).

Thus, relative to what is known from tests $i = 1, \dots, n-1$, $c_i\epsilon_i$ is the amount of **new information** about θ brought by the test number i.

Here **new information** means **surprise** or what could not be predicted from earlier information.

Formula (13.1) also provides us with an enlightening way to express conditional means and conditional variances that we computed earlier.

In particular,

$$E\left[\theta \mid y_1, \dots, y_k\right] = \mu_{\theta} + c_1\epsilon_1 + \dots + c_k\epsilon_k$$

and

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

```
C = np.linalg.cholesky(Σ_IQ)
G = np.linalg.inv(C)

ε = G @ (x - μθ)
```

```
cε = C[n, :] * ε

# compute the sequence of μθ and Σθ conditional on y1, y2, ..., yk
μθ_hat_arr_C = np.array([np.sum(cε[:k+1]) for k in range(n)]) + μθ
Σθ_hat_arr_C = np.array([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.

```
# conditional mean
np.max(np.abs(μθ_hat_arr - μθ_hat_arr_C)) < 1e-10
```

```
True
```

```
# conditional variance
np.max(np.abs(Σθ_hat_arr - Σθ_hat_arr_C)) < 1e-10
```

```
True
```

13.7 Cholesky Factor Magic

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

The Cholesky factorization computes these things **recursively**.

Indeed, in formula (13.1),

- the random variable $c_i \epsilon_i$ is information about θ that is not contained by the information in $\epsilon_1, \epsilon_2, \ldots, \epsilon_{i-1}$
- the coefficient c_i is the simple population regression coefficient of $\theta - \mu_\theta$ on ϵ_i

13.8 Math and Verbal Intelligence

We can alter the preceding example to be more realistic.

There is ample evidence that IQ is not a scalar.

Some people are good in math skills but poor in language skills.

Other people are good in language skills but poor in math skills.

So now we shall assume that there are two dimensions of IQ, θ and η.

These determine average performances in math and language tests, respectively.

We observe math scores $\{y_i\}_{i=1}^{n}$ and language scores $\{y_i\}_{i=n+1}^{2n}$.

When $n = 2$, we assume that outcomes are draws from a multivariate normal distribution with representation

$$X = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ \theta \\ \eta \end{bmatrix} = \begin{bmatrix} \mu_\theta \\ \mu_\theta \\ \mu_\eta \\ \mu_\eta \\ \mu_\theta \\ \mu_\eta \end{bmatrix} + \begin{bmatrix} \sigma_y & 0 & 0 & 0 & \sigma_\theta & 0 \\ 0 & \sigma_y & 0 & 0 & \sigma_\theta & 0 \\ 0 & 0 & \sigma_y & 0 & 0 & \sigma_\eta \\ 0 & 0 & 0 & \sigma_y & 0 & \sigma_\eta \\ 0 & 0 & 0 & 0 & \sigma_\theta & 0 \\ 0 & 0 & 0 & 0 & 0 & \sigma_\eta \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \\ w_5 \\ w_6 \end{bmatrix}$$

where $w \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_6 \end{bmatrix}$ is a standard normal random vector.

We construct a Python function `construct_moments_IQ2d` to construct the mean vector and covariance matrix of the joint normal distribution.

```python
def construct_moments_IQ2d(n, μθ, σθ, μη, ση, σy):

    μ_IQ2d = np.empty(2*(n+1))
    μ_IQ2d[:n] = μθ
    μ_IQ2d[2*n] = μθ
    μ_IQ2d[n:2*n] = μη
    μ_IQ2d[2*n+1] = μη
```

(continues on next page)

(continued from previous page)

```
    D_IQ2d = np.zeros((2*(n+1), 2*(n+1)))
    D_IQ2d[range(2*n), range(2*n)] = σy
    D_IQ2d[:n, 2*n] = σθ
    D_IQ2d[2*n, 2*n] = σθ
    D_IQ2d[n:2*n, 2*n+1] = ση
    D_IQ2d[2*n+1, 2*n+1] = ση

    Σ_IQ2d = D_IQ2d @ D_IQ2d.T

    return μ_IQ2d, Σ_IQ2d, D_IQ2d
```

Let's put the function to work.

```
n = 2
# mean and variance of θ, η, and y
μθ, σθ, μη, ση, σy = 100., 10., 100., 10, 10

μ_IQ2d, Σ_IQ2d, D_IQ2d = construct_moments_IQ2d(n, μθ, σθ, μη, ση, σy)
μ_IQ2d, Σ_IQ2d, D_IQ2d
```

```
(array([100., 100., 100., 100., 100., 100.]),
 array([[200., 100.,   0.,   0., 100.,   0.],
        [100., 200.,   0.,   0., 100.,   0.],
        [  0.,   0., 200., 100.,   0., 100.],
        [  0.,   0., 100., 200.,   0., 100.],
        [100., 100.,   0.,   0., 100.,   0.],
        [  0.,   0., 100., 100.,   0., 100.]]),
 array([[10.,  0.,  0.,  0., 10.,  0.],
        [ 0., 10.,  0.,  0., 10.,  0.],
        [ 0.,  0., 10.,  0.,  0., 10.],
        [ 0.,  0.,  0., 10.,  0., 10.],
        [ 0.,  0.,  0.,  0., 10.,  0.],
        [ 0.,  0.,  0.,  0.,  0., 10.]]))
```

```
# take one draw
x = np.random.multivariate_normal(μ_IQ2d, Σ_IQ2d)
y1 = x[:n]
y2 = x[n:2*n]
θ = x[2*n]
η = x[2*n+1]

# the true values
θ, η
```

```
(104.87169696314103, 107.9697815380512)
```

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

```
multi_normal_IQ2d = MultivariateNormal(μ_IQ2d, Σ_IQ2d)

k = 2*n # the length of data vector
```

(continues on next page)

(continued from previous page)

```
multi_normal_IQ2d.partition(k)

multi_normal_IQ2d.cond_dist(1, [*y1, *y2])
```

```
(array([101.17829519, 105.80501858]),
 array([[33.33333333,  0.        ],
        [ 0.        , 33.33333333]]))
```

Now let's compute distributions of θ and μ separately conditional on various subsets of test scores.

It will be fun to compare outcomes with the help of an auxiliary function `cond_dist_IQ2d` that we now construct.

```
def cond_dist_IQ2d(μ, Σ, data):

    n = len(μ)

    multi_normal = MultivariateNormal(μ, Σ)
    multi_normal.partition(n-1)
    μ_hat, Σ_hat = multi_normal.cond_dist(1, data)

    return μ_hat, Σ_hat
```

Let's see how things work for an example.

```
for indices, IQ, conditions in [([*range(2*n), 2*n], 'θ', 'y1, y2, y3, y4'),
                                 ([*range(n), 2*n], 'θ', 'y1, y2'),
                                 ([*range(n, 2*n), 2*n], 'θ', 'y3, y4'),
                                 ([*range(2*n), 2*n+1], 'η', 'y1, y2, y3, y4'),
                                 ([*range(n), 2*n+1], 'η', 'y1, y2'),
                                 ([*range(n, 2*n), 2*n+1], 'η', 'y3, y4')]:

    μ_hat, Σ_hat = cond_dist_IQ2d(μ_IQ2d[indices], Σ_IQ2d[indices][:, indices],␣
↪x[indices[:-1]])
    print(f'The mean and variance of {IQ} conditional on {conditions: <15} are ' +
          f'{μ_hat[0]:1.2f} and {Σ_hat[0, 0]:1.2f} respectively')
```

```
The mean and variance of θ conditional on y1, y2, y3, y4  are 101.18 and 33.33␣
↪respectively
The mean and variance of θ conditional on y1, y2          are 101.18 and 33.33␣
↪respectively
The mean and variance of θ conditional on y3, y4          are 100.00 and 100.00␣
↪respectively
The mean and variance of η conditional on y1, y2, y3, y4  are 105.81 and 33.33␣
↪respectively
The mean and variance of η conditional on y1, y2          are 100.00 and 100.00␣
↪respectively
The mean and variance of η conditional on y3, y4          are 105.81 and 33.33␣
↪respectively
```

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

13.9 Univariate Time Series Analysis

We can use the multivariate normal distribution and a little matrix algebra to present foundations of univariate linear time series analysis.

Let x_t, y_t, v_t, w_{t+1} each be scalars for $t \geq 0$.

Consider the following model:

$$\begin{aligned} x_0 &\sim N\left(0, \sigma_0^2\right) \\ x_{t+1} &= a x_t + b w_{t+1}, \quad w_{t+1} \sim N\left(0, 1\right), t \geq 0 \\ y_t &= c x_t + d v_t, \quad v_t \sim N\left(0, 1\right), t \geq 0 \end{aligned}$$

We can compute the moments of x_t

1. $E x_{t+1}^2 = a^2 E x_t^2 + b^2, t \geq 0$, where $E x_0^2 = \sigma_0^2$
2. $E x_{t+j} x_t = a^j E x_t^2, \forall t \, \forall j$

Given some T, we can formulate the sequence $\{x_t\}_{t=0}^T$ as a random vector

$$X = \left[\begin{array}{c} x_0 \\ x_1 \\ \vdots \\ x_T \end{array}\right]$$

and the covariance matrix Σ_x can be constructed using the moments we have computed above.

Similarly, we can define

$$Y = \left[\begin{array}{c} y_0 \\ y_1 \\ \vdots \\ y_T \end{array}\right], \quad v = \left[\begin{array}{c} v_0 \\ v_1 \\ \vdots \\ v_T \end{array}\right]$$

and therefore

$$Y = CX + DV$$

where C and D are both diagonal matrices with constant c and d as diagonal respectively.

Consequently, the covariance matrix of Y is

$$\Sigma_y = E Y Y^{\prime} = C \Sigma_x C^{\prime} + D D^{\prime}$$

By stacking X and Y, we can write

$$Z = \left[\begin{array}{c} X \\ Y \end{array}\right]$$

and

$$\Sigma_z = E Z Z^{\prime} = \left[\begin{array}{cc} \Sigma_x & \Sigma_x C^{\prime} \\ C \Sigma_x & \Sigma_y \end{array}\right]$$

Thus, the stacked sequences $\{x_t\}_{t=0}^T$ and $\{y_t\}_{t=0}^T$ jointly follow the multivariate normal distribution $N\left(0, \Sigma_z\right)$.

```
# as an example, consider the case where T = 3
T = 3
```

```
# variance of the initial distribution x_0
σ0 = 1.

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

```
# construct the covariance matrix of X
Σx = np.empty((T+1, T+1))

Σx[0, 0] = σ0 ** 2
for i in range(T):
    Σx[i, i+1:] = Σx[i, i] * a ** np.arange(1, T+1-i)
    Σx[i+1:, i] = Σx[i, i+1:]

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

```
Σx
```

```
array([[1.      , 0.9     , 0.81    , 0.729   ],
       [0.9     , 1.81    , 1.629   , 1.4661  ],
       [0.81    , 1.629   , 2.4661  , 2.21949 ],
       [0.729   , 1.4661  , 2.21949 , 2.997541]])
```

```
# construct the covariance matrix of Y
C = np.eye(T+1) * c
D = np.eye(T+1) * d

Σy = C @ Σx @ C.T + D @ D.T
```

```
# construct the covariance matrix of Z
Σz = np.empty((2*(T+1), 2*(T+1)))

Σz[:T+1, :T+1] = Σx
Σz[:T+1, T+1:] = Σx @ C.T
Σz[T+1:, :T+1] = C @ Σx
Σz[T+1:, T+1:] = Σy
```

```
Σz
```

```
array([[1.      , 0.9     , 0.81    , 0.729   , 1.      , 0.9     ,
        0.81    , 0.729   ],
       [0.9     , 1.81    , 1.629   , 1.4661  , 0.9     , 1.81    ,
        1.629   , 1.4661  ],
       [0.81    , 1.629   , 2.4661  , 2.21949 , 0.81    , 1.629   ,
        2.4661  , 2.21949 ],
       [0.729   , 1.4661  , 2.21949 , 2.997541, 0.729   , 1.4661  ,
        2.21949 , 2.997541],
       [1.      , 0.9     , 0.81    , 0.729   , 1.0025  , 0.9     ,
        0.81    , 0.729   ],
```

(continues on next page)

(continued from previous page)

```
        [0.9     , 1.81    , 1.629   , 1.4661  , 0.9     , 1.8125  ,
         1.629   , 1.4661  ],
        [0.81    , 1.629   , 2.4661  , 2.21949 , 0.81    , 1.629   ,
         2.4686  , 2.21949 ],
        [0.729   , 1.4661  , 2.21949 , 2.997541, 0.729   , 1.4661  ,
         2.21949 , 3.000041]])
```

```
# construct the mean vector of Z
μz = np.zeros(2*(T+1))
```

The following Python code lets us sample random vectors X and Y.

This is going to be very useful for doing the conditioning to be used in the fun exercises below.

```
z = np.random.multivariate_normal(μz, Σz)

x = z[:T+1]
y = z[T+1:]
```

13.9.1 Smoothing Example

This is an instance of a classic `smoothing` calculation whose purpose is to compute $EX \mid Y$.

An interpretation of this example is

- X is a random sequence of hidden Markov state variables x_t
- Y is a sequence of observed signals y_t bearing information about the hidden state

```
# construct a MultivariateNormal instance
multi_normal_ex1 = MultivariateNormal(μz, Σz)
x = z[:T+1]
y = z[T+1:]
```

```
# partition Z into X and Y
multi_normal_ex1.partition(T+1)
```

```
# compute the conditional mean and covariance matrix of X given Y=y

print("X = ", x)
print("Y = ", y)
print(" E [ X | Y] = ", )

multi_normal_ex1.cond_dist(0, y)
```

```
X =  [1.04425612 1.12782548 0.54973228 0.49249607]
Y =  [1.1415452  1.12814704 0.65694875 0.44324541]
 E [ X | Y] =
```

```
(array([1.1389275 , 1.12708895, 0.65750761, 0.44361577]),
 array([[2.48875094e-03, 5.57449314e-06, 1.24861729e-08, 2.80235835e-11],
```

(continues on next page)

(continued from previous page)

```
         [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]]))
```

13.9.2 Filtering Exercise

Compute $E[x_t \mid y_{t-1}, y_{t-2}, \dots, y_0]$.

To do so, we need to first construct the mean vector and the covariance matrix of the subvector $[x_t, y_0, \dots, y_{t-2}, y_{t-1}]$.

For example, let's say that we want the conditional distribution of x_3.

```
t = 3
```

```
# mean of the subvector
sub_μz = np.zeros(t+1)

# covariance matrix of the subvector
sub_Σz = np.empty((t+1, t+1))

sub_Σz[0, 0] = Σz[t, t] # x_t
sub_Σz[0, 1:] = Σz[t, T+1:T+t+1]
sub_Σz[1:, 0] = Σz[T+1:T+t+1, t]
sub_Σz[1:, 1:] = Σz[T+1:T+t+1, T+1:T+t+1]
```

```
sub_Σz
```

```
array([[2.997541, 0.729   , 1.4661  , 2.21949 ],
       [0.729   , 1.0025  , 0.9     , 0.81    ],
       [1.4661  , 0.9     , 1.8125  , 1.629   ],
       [2.21949 , 0.81    , 1.629   , 2.4686  ]])
```

```
multi_normal_ex2 = MultivariateNormal(sub_μz, sub_Σz)
multi_normal_ex2.partition(1)
```

```
sub_y = y[:t]

multi_normal_ex2.cond_dist(0, sub_y)
```

```
(array([0.59205609]), array([[1.00201996]]))
```

13.9.3 Prediction Exercise

Compute $E\left[y_t \mid y_{t-j}, \dots, y_0\right]$.

As what we did in exercise 2, we will construct the mean vector and covariance matrix of the subvector $\left[y_t, y_0, \dots, y_{t-j-1}, y_{t-j}\right]$.

For example, we take a case in which $t = 3$ and $j = 2$.

```
t = 3
j = 2
```

```
sub_μz = np.zeros(t-j+2)
sub_Σz = np.empty((t-j+2, t-j+2))

sub_Σz[0, 0] = Σz[T+t+1, T+t+1]
sub_Σz[0, 1:] = Σz[T+t+1, T+1:T+t-j+2]
sub_Σz[1:, 0] = Σz[T+1:T+t-j+2, T+t+1]
sub_Σz[1:, 1:] = Σz[T+1:T+t-j+2, T+1:T+t-j+2]
```

```
sub_Σz
```

```
array([[3.000041, 0.729   , 1.4661  ],
       [0.729   , 1.0025  , 0.9     ],
       [1.4661  , 0.9     , 1.8125  ]])
```

```
multi_normal_ex3 = MultivariateNormal(sub_μz, sub_Σz)
multi_normal_ex3.partition(1)
```

```
sub_y = y[:t-j+1]

multi_normal_ex3.cond_dist(0, sub_y)
```

```
(array([0.91359083]), array([[1.81413617]]))
```

13.9.4 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

$$y_t = h_{t,t}\epsilon_t + h_{t,t-1}\epsilon_{t-1} + \dots + h_{t,0}\epsilon_0.$$

```
H = np.linalg.cholesky(Σy)

H
```

```
array([[1.00124922, 0.        , 0.        , 0.        ],
       [0.8988771 , 1.00225743, 0.        , 0.        ],
       [0.80898939, 0.89978675, 1.00225743, 0.        ],
       [0.72809046, 0.80980808, 0.89978676, 1.00225743]])
```

```
ε = np.linalg.inv(H) @ y

ε
```

```
array([ 1.14012093,  0.10308573, -0.35734549, -0.1484755 ])
```

```
y
```

```
array([1.1415452 , 1.12814704, 0.65694875, 0.44324541])
```

This example is an instance of what is known as a **Wold representation** in time series analysis.

13.10 Stochastic Difference Equation

Consider the stochastic second-order linear difference equation

$$y_t = \alpha_0 + \alpha_1 y_{y-1} + \alpha_2 y_{t-2} + u_t$$

where $u_t \sim N\left(0, \sigma_u^2\right)$ and

$$\left[\begin{array}{c} y_{-1} \\ y_0 \end{array}\right] \sim N\left(\mu_{\tilde{y}}, \Sigma_{\tilde{y}}\right)$$

It can be written as a stacked system

$$\underbrace{\left[\begin{array}{cccccccc} 1 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ -\alpha_1 & 1 & 0 & 0 & \cdots & 0 & 0 & 0 \\ -\alpha_2 & -\alpha_1 & 1 & 0 & \cdots & 0 & 0 & 0 \\ 0 & -\alpha_2 & -\alpha_1 & 1 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & -\alpha_2 & -\alpha_1 & 1 \end{array}\right]}_{\equiv A} \left[\begin{array}{c} y_1 \\ y_2 \\ y_3 \\ y_4 \\ \vdots \\ y_T \end{array}\right] = \underbrace{\left[\begin{array}{c} \alpha_0 + \alpha_1 y_0 + \alpha_2 y_{-1} \\ \alpha_0 + \alpha_2 y_0 \\ \alpha_0 \\ \alpha_0 \\ \vdots \\ \alpha_0 \end{array}\right]}_{\equiv b}$$

We can compute y by solving the system

$$y = A^{-1}\left(b + u\right)$$

We have

$$\begin{aligned} \mu_y &= A^{-1}\mu_b \\ \Sigma_y &= A^{-1}E\left[\left(b-\mu_b+u\right)\left(b-\mu_b+u\right)'\right]\left(A^{-1}\right)' \\ &= A^{-1}\left(\Sigma_b+\Sigma_u\right)\left(A^{-1}\right)' \end{aligned}$$

where

$$\mu_b = \left[\begin{array}{c} \alpha_0 + \alpha_1 \mu_{y_0} + \alpha_2 \mu_{y_{-1}} \\ \alpha_0 + \alpha_2 \mu_{y_0} \\ \alpha_0 \\ \vdots \\ \alpha_0 \end{array}\right]$$

$$\Sigma_b = \left[\begin{array}{cc} C\Sigma_{\tilde{y}}C' & 0_{N-2\times N-2} \\ 0_{N-2\times 2} & 0_{N-2\times N-2} \end{array} \right], \quad C = \left[\begin{array}{cc} \alpha_2 & \alpha_1 \\ 0 & \alpha_2 \end{array} \right]$$

$$\Sigma_u = \left[\begin{array}{cccc} \sigma_u^2 & 0 & \cdots & 0 \\ 0 & \sigma_u^2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & \sigma_u^2 \end{array} \right]$$

```python
# set parameters
T = 80
T = 160
# coefficients of the second order difference equation
□0 = 10
□1 = 1.53
□2 = -.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]])
```

```python
# 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] = -□1

    if i-2 >= 0:
        A[i, i-2] = -□2

A_inv = np.linalg.inv(A)
```

```python
# compute the mean vectors of b and y
μb = np.full(T, □0)
μb[0] += □1 * μy_tilde[1] + □2 * μy_tilde[0]
μb[1] += □2 * μy_tilde[1]

μy = A_inv @ μb
```

```python
# compute the covariance matrices of b and y
Σu = np.eye(T) * σu ** 2

Σb = np.zeros((T, T))

C = np.array([[□2, □1], [0, □2]])
Σb[:2, :2] = C @ Σy_tilde @ C.T

Σy = A_inv @ (Σb + Σu) @ A_inv.T
```

13.11 Application to Stock Price Model

Let

$$p_t = \sum_{j=0}^{T-t} \beta^j y_{t+j}$$

Form

$$\underbrace{\begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ \vdots \\ p_T \end{bmatrix}}_{\equiv p} = \underbrace{\begin{bmatrix} 1 & \beta & \beta^2 & \cdots & \beta^{T-1} \\ 0 & 1 & \beta & \cdots & \beta^{T-2} \\ 0 & 0 & 1 & \cdots & \beta^{T-3} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix}}_{\equiv B} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_T \end{bmatrix}$$

we have

$$\mu_p = B\mu_y$$
$$\Sigma_p = B\Sigma_y B'$$

```
β = .96
```

```
# construct B
B = np.zeros((T, T))

for i in range(T):
    B[i, i:] = β ** np.arange(0, T-i)
```

Denote

$$z = \begin{bmatrix} y \\ p \end{bmatrix} = \underbrace{\begin{bmatrix} I \\ B \end{bmatrix}}_{\equiv D} y$$

Thus, $\{y_t\}_{t=1}^T$ and $\{p_t\}_{t=1}^T$ jointly follow the multivariate normal distribution $N(\mu_z, \Sigma_z)$, where

$$\mu_z = D\mu_y$$

$$\Sigma_z = D\Sigma_y D'$$

```
D = np.vstack([np.eye(T), B])
```

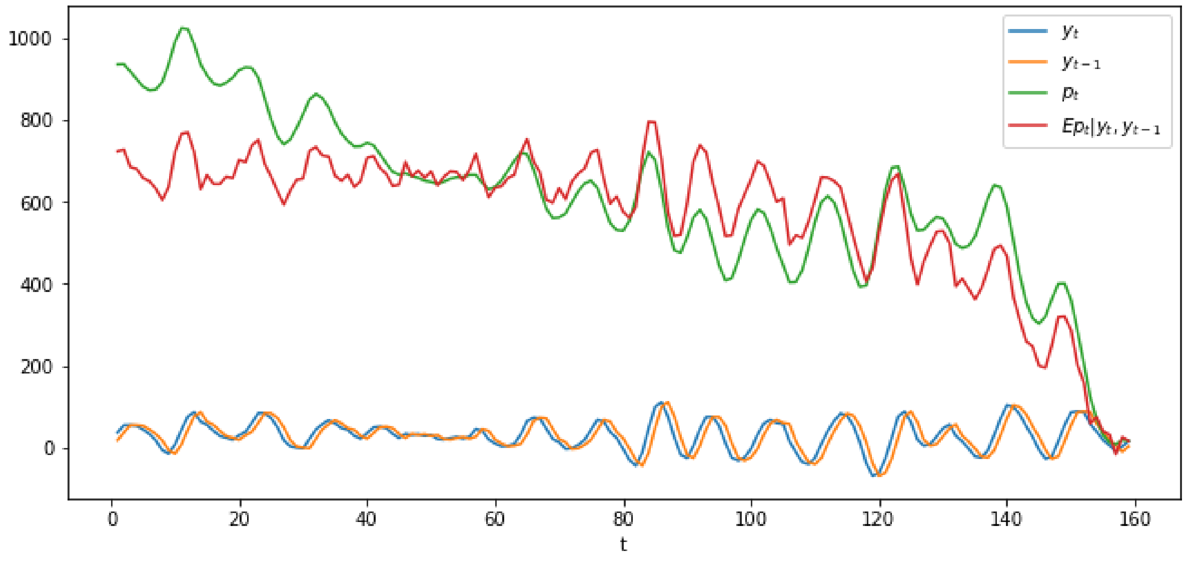
```
μz = D @ μy
Σz = D @ Σy @ D.T
```

We can simulate paths of y_t and p_t and compute the conditional mean $E[p_t \mid y_{t-1}, y_t]$ using the `MultivariateNormal` class.

```
z = np.random.multivariate_normal(μz, Σz)
y, p = z[:T], z[T:]
```

```
cond_Ep = np.empty(T-1)

sub_μ = np.empty(3)
sub_Σ = np.empty((3, 3))
for t in range(2, T+1):
    sub_μ[:] = μz[[t-2, t-1, T-1+t]]
    sub_Σ[:, :] = Σz[[t-2, t-1, T-1+t], :][:, [t-2, t-1, T-1+t]]

    multi_normal = MultivariateNormal(sub_μ, sub_Σ)
    multi_normal.partition(2)

    cond_Ep[t-2] = multi_normal.cond_dist(1, y[t-2:t])[0][0]
```

```
plt.plot(range(1, T), y[1:], label='$y_{t}$')
plt.plot(range(1, T), y[:-1], label='$y_{t-1}$')
plt.plot(range(1, T), p[1:], label='$p_{t}$')
plt.plot(range(1, T), cond_Ep, label='$Ep_{t}|y_{t}, y_{t-1}$')

plt.xlabel('t')
plt.legend(loc=1)
plt.show()
```



In the above graph, the green line is what the price of the stock would be if people had perfect foresight about the path of dividends while the green line is the conditional expectation $Ep_t|y_t, y_{t-1}$, which is what the price would be if people did not have perfect foresight but were optimally predicting future dividends on the basis of the information y_t, y_{t-1} at time t.

13.12 Filtering Foundations

Assume that x_0 is an $n \times 1$ random vector and that y_0 is a $p \times 1$ random vector determined by the *observation equation*

$$y_0 = Gx_0 + v_0, \quad x_0 \sim \mathcal{N}(\hat{x}_0, \Sigma_0), \quad v_0 \sim \mathcal{N}(0, R)$$

where v_0 is orthogonal to x_0, G is a $p \times n$ matrix, and R is a $p \times p$ positive definite matrix.

We consider the problem of someone who

- *observes* y_0
- does not observe x_0,
- knows $\hat{x}_0, \Sigma_0, G, R$ and therefore the joint probability distribution of the vector $\begin{bmatrix} x_0 \\ y_0 \end{bmatrix}$
- wants to infer x_0 from y_0 in light of what he knows about that joint probability distribution.

Therefore, the person wants to construct the probability distribution of x_0 conditional on the random vector y_0.

The joint distribution of $\begin{bmatrix} x_0 \\ y_0 \end{bmatrix}$ is multivariate normal $\mathcal{N}(\mu, \Sigma)$ with

$$\mu = \begin{bmatrix} \hat{x}_0 \\ G\hat{x}_0 \end{bmatrix}, \quad \Sigma = \begin{bmatrix} \Sigma_0 & \Sigma_0 G' \\ G\Sigma_0 & G\Sigma_0 G' + R \end{bmatrix}$$

By applying an appropriate instance of the above formulas for the mean vector $\hat{\mu}_1$ and covariance matrix $\hat{\Sigma}_{11}$ of z_1 conditional on z_2, we find that the probability distribution of x_0 conditional on y_0 is $\mathcal{N}(\tilde{x}_0, \tilde{\Sigma}_0)$ where

$$\begin{aligned} \beta_0 &= \Sigma_0 G'(G\Sigma_0 G' + R)^{-1} \\ \tilde{x}_0 &= \hat{x}_0 + \beta_0(y_0 - G\hat{x}_0) \\ \tilde{\Sigma}_0 &= \Sigma_0 - \Sigma_0 G'(G\Sigma_0 G' + R)^{-1} G\Sigma_0 \end{aligned}$$

13.12.1 Step toward dynamics

Now suppose that we are in a time series setting and that we have the one-step state transition equation

$$x_1 = Ax_0 + Cw_1, \quad w_1 \sim \mathcal{N}(0, I)$$

where A is an $n \times n$ matrix and C is an $n \times m$ matrix.

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

$$\begin{aligned} \hat{x}_1 &= A\tilde{x}_0 \\ \Sigma_1 &= A\tilde{\Sigma}_0 A' + CC' \end{aligned}$$

13.12.2 Dynamic version

Suppose now that for $t \geq 0$, $\{x_{t+1}, y_t\}_{t=0}^{\infty}$ are governed by the equations

$$\begin{aligned} x_{t+1} &= Ax_t + Cw_{t+1} \\ y_t &= Gx_t + v_t \end{aligned}$$

where as before $x_0 \sim \mathcal{N}(\hat{x}_0, \Sigma_0)$, w_{t+1} is the $t+1$th component of an i.i.d. stochastic process distributed as $w_{t+1} \sim \mathcal{N}(0, I)$, and v_t is the tth component of an i.i.d. process distributed as $v_t \sim \mathcal{N}(0, R)$ and the $\{w_{t+1}\}_{t=0}^{\infty}$ and $\{v_t\}_{t=0}^{\infty}$ processes are orthogonal at all pairs of dates.

The logic and formulas that we applied above imply that the probability distribution of x_t conditional on $y_0, y_1, \ldots, y_{t-1} = y^{t-1}$ is

$$x_t | y^{t-1} \sim \mathcal{N}(A\tilde{x}_t, A\tilde{\Sigma}_t A' + CC')$$

where $\{\tilde{x}_t, \tilde{\Sigma}_t\}_{t=1}^{\infty}$ can be computed by iterating on the following equations starting from $t = 1$ and initial conditions for $\tilde{x}_0, \tilde{\Sigma}_0$ computed as we have above:

$$\begin{aligned} \Sigma_t &= A\tilde{\Sigma}_{t-1}A' + CC' \\ \hat{x}_t &= A\tilde{x}_{t-1} \\ \beta_t &= \Sigma_t G'(G\Sigma_t G' + R)^{-1} \\ \tilde{x}_t &= \hat{x}_t + \beta_t(y_t - G\hat{x}_t) \\ \tilde{\Sigma}_t &= \Sigma_t - \Sigma_t G'(G\Sigma_t G' + R)^{-1} G\Sigma_t \end{aligned}$$

If we shift the first equation forward one period and then substitute the expression for $\tilde{\Sigma}_t$ on the right side of the fifth equation into it we obtain

$$\Sigma_{t+1} = CC' + A\Sigma_t A' - A\Sigma_t G'(G\Sigma_t G' + R)^{-1} G\Sigma_t A'.$$

This is a matrix Riccati difference equation that is closely related to another matrix Riccati difference equation that appears in a quantecon lecture on the basics of linear quadratic control theory.

That equation has the form

$$P_{t-1} = R + A'P_t A - A'P_t B(B'P_t B + Q)^{-1} B'P_t A.$$

Stare at the two preceding equations for a moment or two, the first being a matrix difference equation for a conditional covariance matrix, the second being a matrix difference equation in the matrix appearing in a quadratic form for an intertemporal cost of value function.

Although the two equations are not identical, they display striking family resemblences.

- the first equation tells dynamics that work **forward** in time
- the second equation tells dynamics that work **backward** in time
- while many of the terms are similar, one equation seems to apply matrix transformations to some matrices that play similar roles in the other equation

The family resemblences of these two equations reflects a transcendent **duality** between control theory and filtering theory.

13.12.3 An example

We can use the Python class *MultivariateNormal* to construct examples.

Here is an example for a single period problem at time 0

```
G = np.array([[1., 3.]])
R = np.array([[1.]])

x0_hat = np.array([0., 1.])
Σ0 = np.array([[1., .5], [.3, 2.]])

μ = np.hstack([x0_hat, G @ x0_hat])
Σ = np.block([[Σ0, Σ0 @ G.T], [G @ Σ0, G @ Σ0 @ G.T + R]])
```

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

```
multi_normal.partition(2)
```

```
# the observation of y
y0 = 2.3

# conditional distribution of x0
μ1_hat, Σ11 = multi_normal.cond_dist(0, y0)
μ1_hat, Σ11
```

```
(array([-0.078125,  0.803125]),
 array([[ 0.72098214, -0.203125  ],
        [-0.403125  ,  0.228125  ]]))
```

```
A = np.array([[0.5, 0.2], [-0.1, 0.3]])
C = np.array([[2.], [1.]])

# conditional distribution of x1
x1_cond = A @ μ1_hat
Σ1_cond = C @ C.T + A @ Σ11 @ A.T
x1_cond, Σ1_cond
```

```
(array([0.1215625, 0.24875  ]),
 array([[4.12874554, 1.95523214],
        [1.92123214, 1.04592857]]))
```

13.12.4 Code for Iterating

Here is code for solving a dynamic filtering problem by iterating on our equations, followed by an example.

```python
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
```

```python
iterate(x0_hat, Σ0, A, C, G, R, [2.3, 1.2, 3.2])
```

```
(array([[0.        , 1.        ],
        [0.1215625 , 0.24875   ],
        [0.18680212, 0.06904689],
        [0.75576875, 0.05558463]]),
 array([[[1.        , 0.5       ],
         [0.3       , 2.        ]],

        [[4.12874554, 1.95523214],
         [1.92123214, 1.04592857]],

        [[4.08198663, 1.99218488],
         [1.98640488, 1.00886423]],

        [[4.06457628, 2.00041999],
         [1.99943739, 1.00275526]]]))
```

The iterative algorithm just described is a version of the celebrated **Kalman filter**.

We describe the Kalman filter and some applications of it in *A First Look at the Kalman Filter*

13.13 Classic Factor Analysis Model

The factor analysis model widely used in psychology and other fields can be represented as

$$Y = \Lambda f + U$$

where

1. Y is $n \times 1$ random vector, $EUU' = D$ is a diagonal matrix,
2. Λ is $n \times k$ coefficient matrix,
3. f is $k \times 1$ random vector, $Eff' = I$,
4. U is $n \times 1$ random vector, and $U \perp f$ (i.e., $EUf' = 0$)
5. It is presumed that k is small relative to n; often k is only 1 or 2, as in our IQ examples.

This implies that

$$\begin{aligned} \Sigma_y = EYY' &= \Lambda\Lambda' + D \\ EYf' &= \Lambda \\ EfY' &= \Lambda' \end{aligned}$$

Thus, the covariance matrix Σ_Y is the sum of a diagonal matrix D and a positive semi-definite matrix $\Lambda\Lambda'$ of rank k.

This means that all covariances among the n components of the Y vector are intermediated by their common dependencies on the $k <$ factors.

Form

$$Z = \left(\begin{array}{c} f \\ Y \end{array}\right)$$

the covariance matrix of the expanded random vector Z can be computed as

$$\Sigma_z = EZZ' = \left(\begin{array}{cc} I & \Lambda' \\ \Lambda & \Lambda\Lambda' + D \end{array}\right)$$

In the following, we first construct the mean vector and the covariance matrix for the case where $N = 10$ and $k = 2$.

```
N = 10
k = 2
```

We set the coefficient matrix Λ and the covariance matrix of U to be

$$\Lambda = \left(\begin{array}{cc} 1 & 0 \\ \vdots & \vdots \\ 1 & 0 \\ 0 & 1 \\ \vdots & \vdots \\ 0 & 1 \end{array}\right), \quad D = \left(\begin{array}{cccc} \sigma_u^2 & 0 & \cdots & 0 \\ 0 & \sigma_u^2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & \sigma_u^2 \end{array}\right)$$

where the first half of the first column of Λ is filled with 1s and 0s for the rest half, and symmetrically for the second column.

D is a diagonal matrix with parameter σ_u^2 on the diagonal.

```
Λ = np.zeros((N, k))
Λ[:N//2, 0] = 1
Λ[N//2:, 1] = 1

σu = .5
D = np.eye(N) * σu ** 2
```

```
# compute Σy
Σy = Λ @ Λ.T + D
```

We can now construct the mean vector and the covariance matrix for Z.

```
μz = np.zeros(k+N)

Σz = np.empty((k+N, k+N))

Σz[:k, :k] = np.eye(k)
Σz[:k, k:] = Λ.T
Σz[k:, :k] = Λ
Σz[k:, k:] = Σy
```

```
z = np.random.multivariate_normal(μz, Σz)

f = z[:k]
y = z[k:]
```

```
multi_normal_factor = MultivariateNormal(μz, Σz)
multi_normal_factor.partition(k)
```

Let's compute the conditional distribution of the hidden factor f on the observations Y, namely, $f \mid Y = y$.

```
multi_normal_factor.cond_dist(0, y)
```

```
(array([0.37829706, 0.31441423]),
 array([[0.04761905, 0.        ],
        [0.        , 0.04761905]]))
```

We can verify that the conditional mean $E[f \mid Y = y] = BY$ where $B = \Lambda' \Sigma_y^{-1}$.

```
B = Λ.T @ np.linalg.inv(Σy)

B @ y
```

```
array([0.37829706, 0.31441423])
```

Similarly, we can compute the conditional distribution $Y \mid f$.

```
multi_normal_factor.cond_dist(1, f)
```

```
(array([0.34553632, 0.34553632, 0.34553632, 0.34553632, 0.34553632,
        0.37994724, 0.37994724, 0.37994724, 0.37994724, 0.37994724]),
```

(continues on next page)

(continued from previous page)

```
array([[0.25, 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 0.25, 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 0.  , 0.25, 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 0.  , 0.  , 0.25, 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 0.  , 0.  , 0.  , 0.25, 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 0.  , 0.  , 0.  , 0.  , 0.25, 0.  , 0.  , 0.  , 0.  ],
       [0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.25, 0.  , 0.  , 0.  ],
       [0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.25, 0.  , 0.  ],
       [0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.25, 0.  ],
       [0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.25]]))
```

It can be verified that the mean is $\Lambda I^{-1} f = \Lambda f$.

```
Λ @ f
```

```
array([0.34553632, 0.34553632, 0.34553632, 0.34553632, 0.34553632,
       0.37994724, 0.37994724, 0.37994724, 0.37994724, 0.37994724])
```

13.14 PCA and Factor Analysis

To learn about Principal Components Analysis (PCA), please see this lecture *Singular Value Decompositions*.

For fun, let's apply a PCA decomposition to a covariance matrix Σ_y that in fact is governed by our factor-analytic model.

Technically, this means that the PCA model is misspecified. (Can you explain why?)

Nevertheless, this exercise will let us study how well the first two principal components from a PCA can approximate the conditional expectations $Ef_i|Y$ for our two factors f_i, $i = 1, 2$ for the factor analytic model that we have assumed truly governs the data on Y we have generated.

So we compute the PCA decomposition

$$\Sigma_y = P\tilde{\Lambda}P'$$

where $\tilde{\Lambda}$ is a diagonal matrix.

We have

$$Y = P\epsilon$$

and

$$\epsilon = P'Y$$

Note that we will arrange the eigenvectors in P in the *descending* order of eigenvalues.

```
⍰_tilde, P = np.linalg.eigh(Σy)

# arrange the eigenvectors by eigenvalues
ind = sorted(range(N), key=lambda x: ⍰_tilde[x], reverse=True)

P = P[:, ind]
⍰_tilde = ⍰_tilde[ind]
```

(continues on next page)

(continued from previous page)

```
Λ_tilde = np.diag(�_tilde)

print('�_tilde =', �_tilde)
```

```
�_tilde = [5.25 5.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25]
```

```
# verify the orthogonality of eigenvectors
np.abs(P @ P.T - np.eye(N)).max()
```

```
4.440892098500626e-16
```

```
# verify the eigenvalue decomposition is correct
P @ Λ_tilde @ P.T
```

```
array([[1.25, 1.  , 1.  , 1.  , 1.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [1.  , 1.25, 1.  , 1.  , 1.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [1.  , 1.  , 1.25, 1.  , 1.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [1.  , 1.  , 1.  , 1.25, 1.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [1.  , 1.  , 1.  , 1.  , 1.25, 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 0.  , 0.  , 0.  , 0.  , 1.25, 1.  , 1.  , 1.  , 1.  ],
       [0.  , 0.  , 0.  , 0.  , 0.  , 1.  , 1.25, 1.  , 1.  , 1.  ],
       [0.  , 0.  , 0.  , 0.  , 0.  , 1.  , 1.  , 1.25, 1.  , 1.  ],
       [0.  , 0.  , 0.  , 0.  , 0.  , 1.  , 1.  , 1.  , 1.25, 1.  ],
       [0.  , 0.  , 0.  , 0.  , 0.  , 1.  , 1.  , 1.  , 1.  , 1.25]])
```

```
ε = P.T @ y

print("ε = ", ε)
```

```
ε =  [ 0.88819283  0.73820417 -0.18694674  0.33595973 -1.08939006 -0.05565398
  0.57290059  0.38344127 -0.37376177 -0.14303457]
```

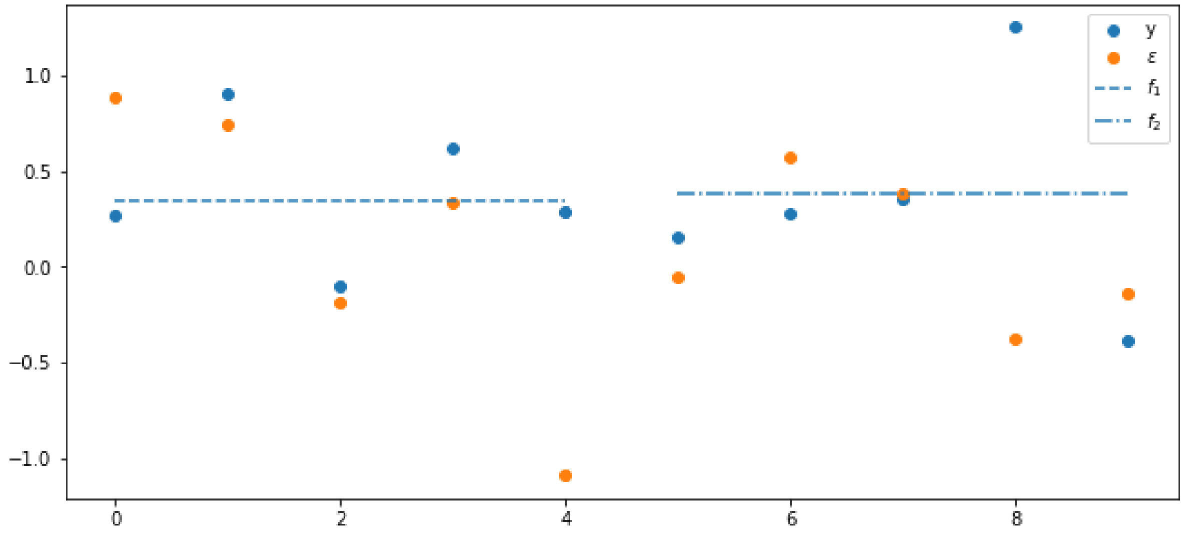
```
# print the values of the two factors

print('f = ', f)
```

```
f =  [0.34553632 0.37994724]
```

Below we'll plot several things

- the N values of y
- the N values of the principal components ϵ
- the value of the first factor f_1 plotted only for the first $N/2$ observations of y for which it receives a non-zero loading in Λ
- the value of the second factor f_2 plotted only for the final $N/2$ observations for which it receives a non-zero loading in Λ

```
plt.scatter(range(N), y, label='y')
plt.scatter(range(N), ε, label='$\epsilon$')
plt.hlines(f[0], 0, N//2-1, ls='--', label='$f_{1}$')
plt.hlines(f[1], N//2, N-1, ls='-.', label='$f_{2}$')
plt.legend()

plt.show()
```



Consequently, the first two ϵ_j correspond to the largest two eigenvalues.

Let's look at them, after which we'll look at $Ef|y = By$

```
ε[:2]
```

```
array([0.88819283, 0.73820417])
```

```
# compare with Ef|y
B @ y
```

```
array([0.37829706, 0.31441423])
```

The fraction of variance in y_t explained by the first two principal components can be computed as below.

```
_tilde[:2].sum() / _tilde.sum()
```

```
0.84
```

Compute

$$\hat{Y} = P_j \epsilon_j + P_k \epsilon_k$$

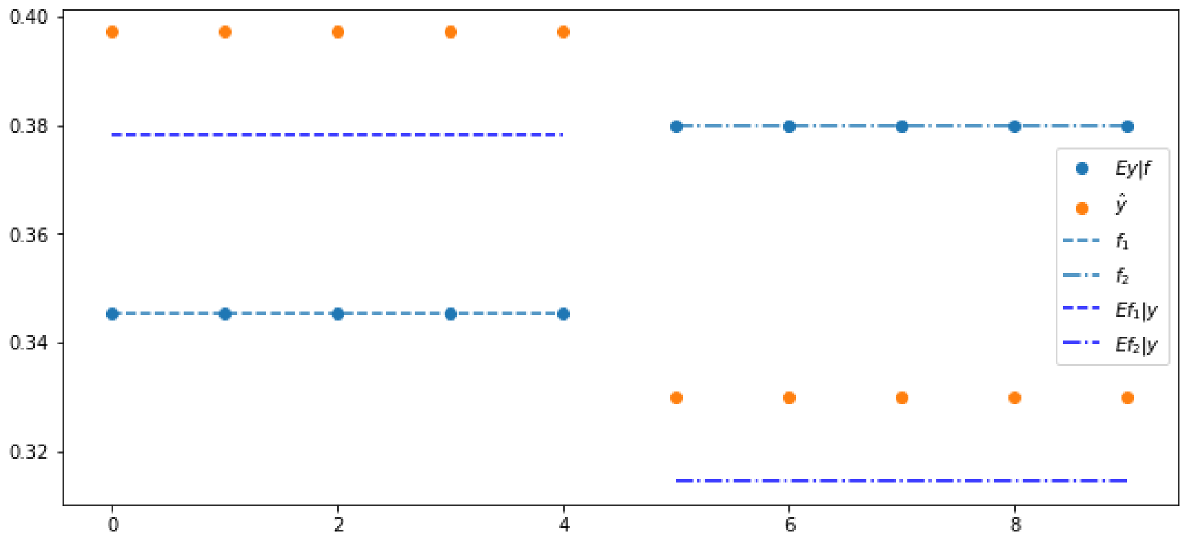
where P_j and P_k correspond to the largest two eigenvalues.

```
y_hat = P[:, :2] @ ε[:2]
```

In this example, it turns out that the projection $\hat{Y}$ of Y on the first two principal components does a good job of approximating $Ef \mid y$.

We confirm this in the following plot of f, $Ey \mid f$, $Ef \mid y$, and $\hat{y}$ on the coordinate axis versus y on the ordinate axis.

```
plt.scatter(range(N), Λ @ f, label='$Ey|f$')
plt.scatter(range(N), y_hat, label='$\hat{y}$')
plt.hlines(f[0], 0, N//2-1, ls='--', label='$f_{1}$')
plt.hlines(f[1], N//2, N-1, ls='-.', label='$f_{2}$')

Efy = B @ y
plt.hlines(Efy[0], 0, N//2-1, ls='--', color='b', label='$Ef_{1}|y$')
plt.hlines(Efy[1], N//2, N-1, ls='-.', color='b', label='$Ef_{2}|y$')
plt.legend()

plt.show()
```



The covariance matrix of $\hat{Y}$ can be computed by first constructing the covariance matrix of ϵ and then use the upper left block for ϵ_1 and ϵ_2.

```
Σεjk = (P.T @ Σy @ P)[:2, :2]

Pjk = P[:, :2]

Σy_hat = Pjk @ Σεjk @ Pjk.T
print('Σy_hat = \n', Σy_hat)
```

```
Σy_hat =
 [[1.05 1.05 1.05 1.05 1.05 0.   0.   0.   0.   0.  ]
 [1.05 1.05 1.05 1.05 1.05 0.   0.   0.   0.   0.  ]
 [1.05 1.05 1.05 1.05 1.05 0.   0.   0.   0.   0.  ]
 [1.05 1.05 1.05 1.05 1.05 0.   0.   0.   0.   0.  ]
 [1.05 1.05 1.05 1.05 1.05 0.   0.   0.   0.   0.  ]
 [0.   0.   0.   0.   0.   1.05 1.05 1.05 1.05 1.05]
```

(continues on next page)

(continued from previous page)

```
 [0.   0.   0.   0.   0.   1.05 1.05 1.05 1.05 1.05]
 [0.   0.   0.   0.   0.   1.05 1.05 1.05 1.05 1.05]
 [0.   0.   0.   0.   0.   1.05 1.05 1.05 1.05 1.05]
 [0.   0.   0.   0.   0.   1.05 1.05 1.05 1.05 1.05]]
```

CHAPTER

FOURTEEN

HEAVY-TAILED DISTRIBUTIONS

Contents

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

```
!pip install quantecon
!pip install --upgrade yfinance
```

14.1 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., [Vil96], [BB18]),
- the firm size distribution ([Axt01], [Gab16]}),
- the distribution of returns on holding assets over short time horizons ([Man63], [Rac03]), and

- the distribution of city sizes ([RRGM11], [Gab16]).

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., [AR02], [GSS03], [BEGS18] or [AKM+18].

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

Let's start with some imports:

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
import quantecon as qe
```

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:

```
from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters()
```

14.2 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.

14.2.1 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.

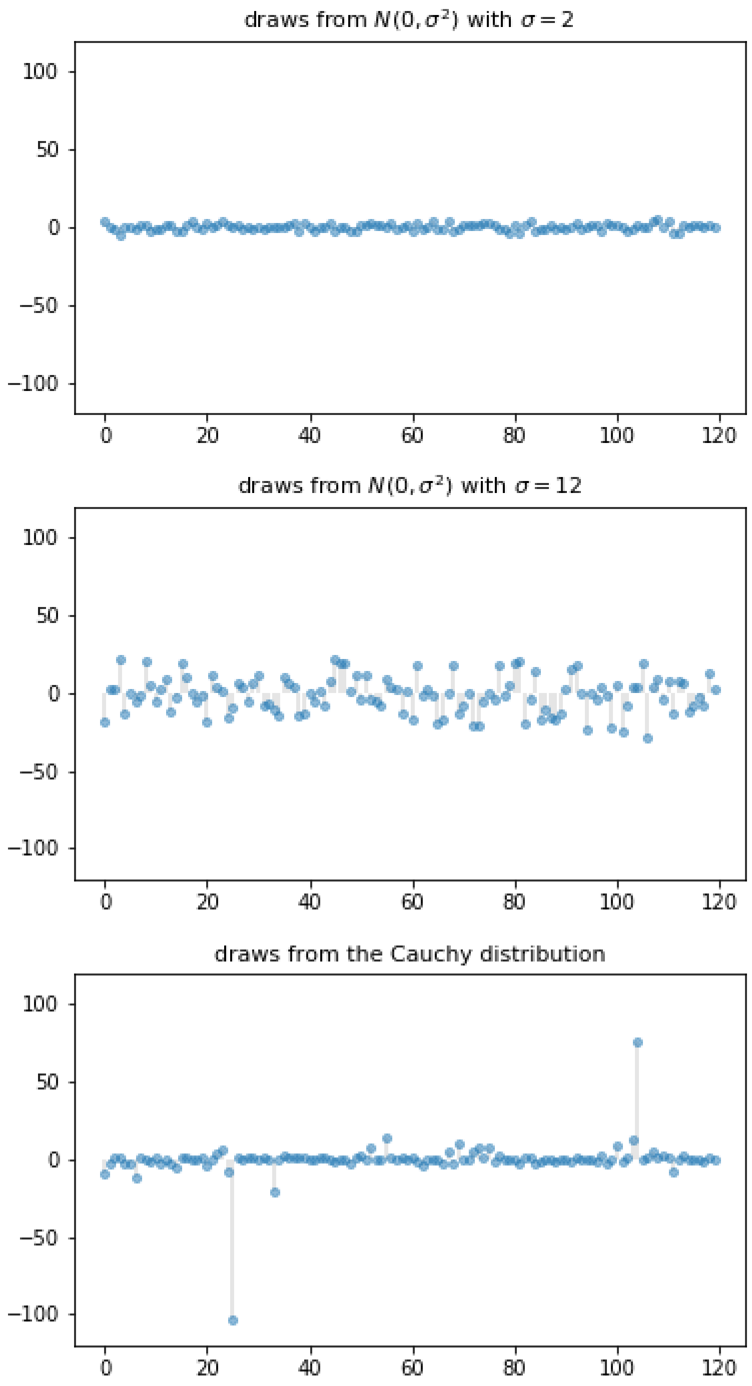
The bottom subfigure shows 120 independent draws from the Cauchy distribution, which is heavy-tailed.

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.


draws from $N(0, \sigma^2)$ with $\sigma = 2$
draws from $N(0, \sigma^2)$ with $\sigma = 12$
draws from the Cauchy distribution
100
50
0
−50
−100
0
20
40
60
80
100
120

14.2.2 Heavy Tails in Asset Returns

Next let's look at some financial data.

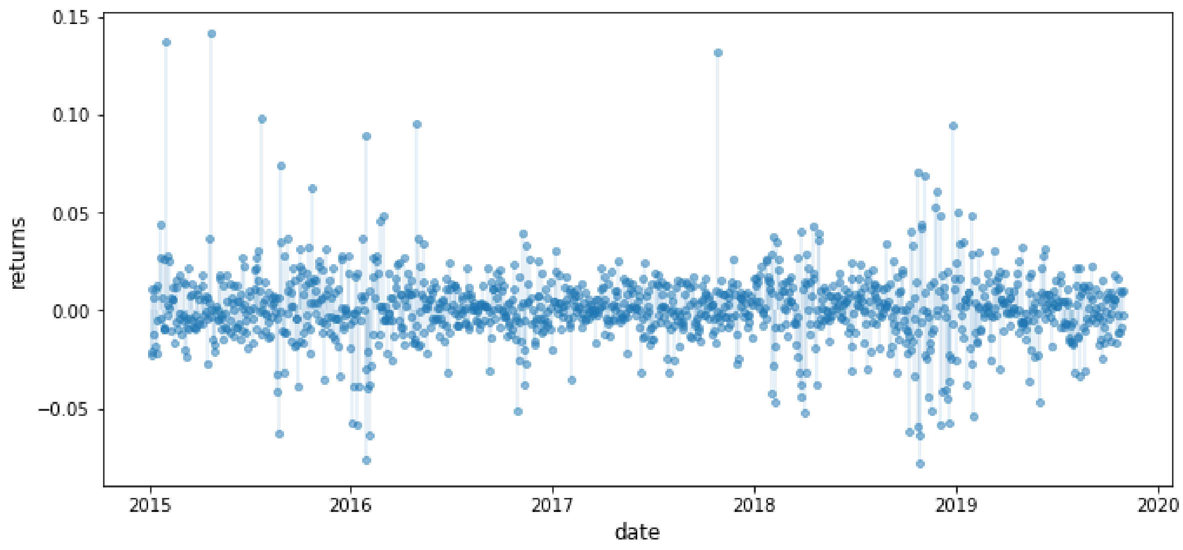
Our aim is to plot the daily change in the price of Amazon (AMZN) stock for the period from 1st January 2015 to 1st November 2019.

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.

```python
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.

14.3 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 $X_1, \ldots, X_n$ 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

$$\mathbb{P}\left\{\bar{X}_n \to \mu \text{ as } n \to \infty\right\} = 1 \tag{14.1}$$

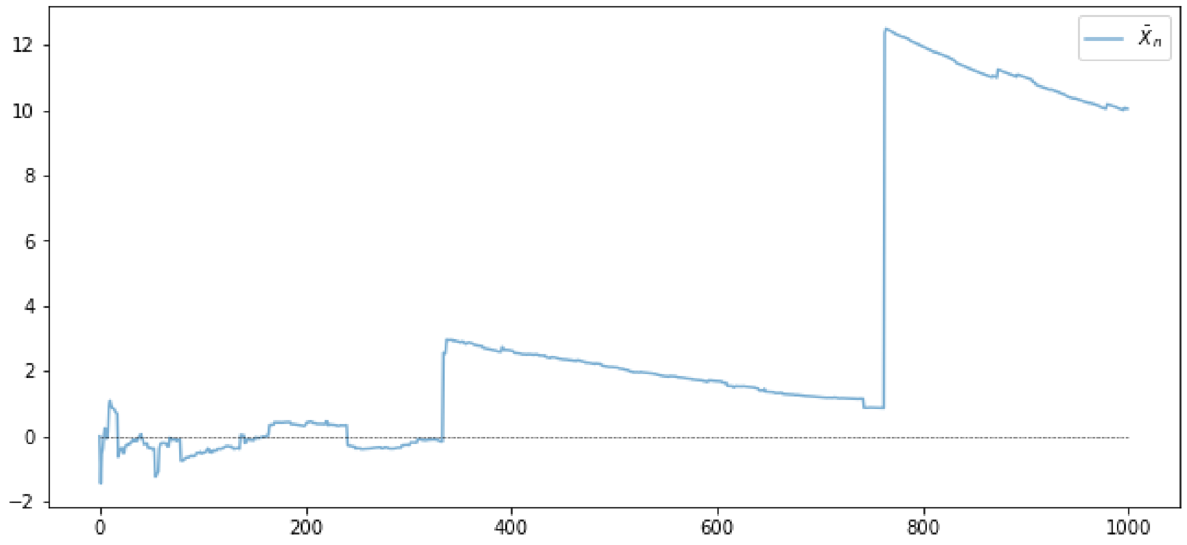
where $\mu := \mathbb{E} X_i = \int x F(x)$ is the common mean of the sample.

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.

```python
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

$$\phi(t) = \mathbb{E}e^{itX} = \int e^{itx} f(x)dx = e^{-|t|} \tag{14.2}$$

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 (14.2), 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.

14.4 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.

14.4.1 Light and Heavy Tails

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

$$\int_0^\infty \exp(tx) F(dx) = \infty \quad \text{for all } t > 0. \tag{14.3}$$

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.

14.4.2 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

$$\lim_{x \to \infty} x^\alpha \, \mathbb{P}\{X > x\} = c. \tag{14.4}$$

Evidently (14.4) implies the existence of positive constants b and $\bar{x}$ such that $\mathbb{P}\{X > x\} \geq b x^{-\alpha}$ 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 (14.4) is said to have a **power law tail**.

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

$$F(x) = \begin{cases} 1 - (\bar{x}/x)^\alpha & \text{if } x \geq \bar{x} \\ 0 & \text{if } x < \bar{x} \end{cases} \tag{14.5}$$

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

It is easy to see that if $X \sim F$, then $\mathbb{P}\{X > x\}$ satisfies (14.4).

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

14.4.3 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 [NOM04].

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.

14.5 Exercises

Exercise 14.5.1

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

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

Exercise 14.5.2

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

Exercise 14.5.3

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.

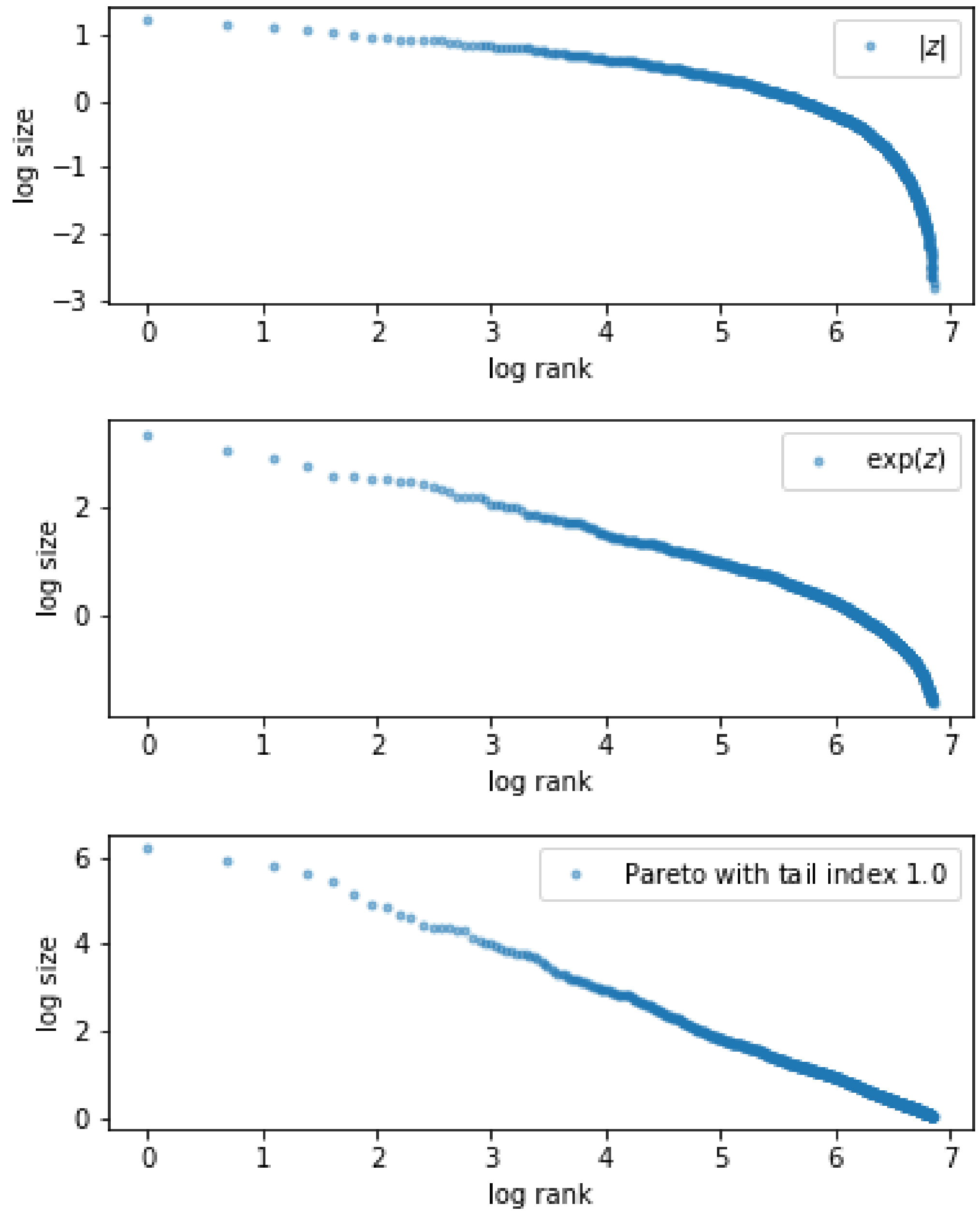
Exercise 14.5.4

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.

Exercise 14.5.5



|z|
exp(z)
Pareto with tail index 1.0
log size
log rank

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., [FDGA+04], [KLS18] or [ST19a]).

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 (14.5) with $\bar{x} = 1$ and $\alpha = 1.05$.

(The value the tail index α is plausible given the data [Gab16].)

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.)

14.6 Solutions

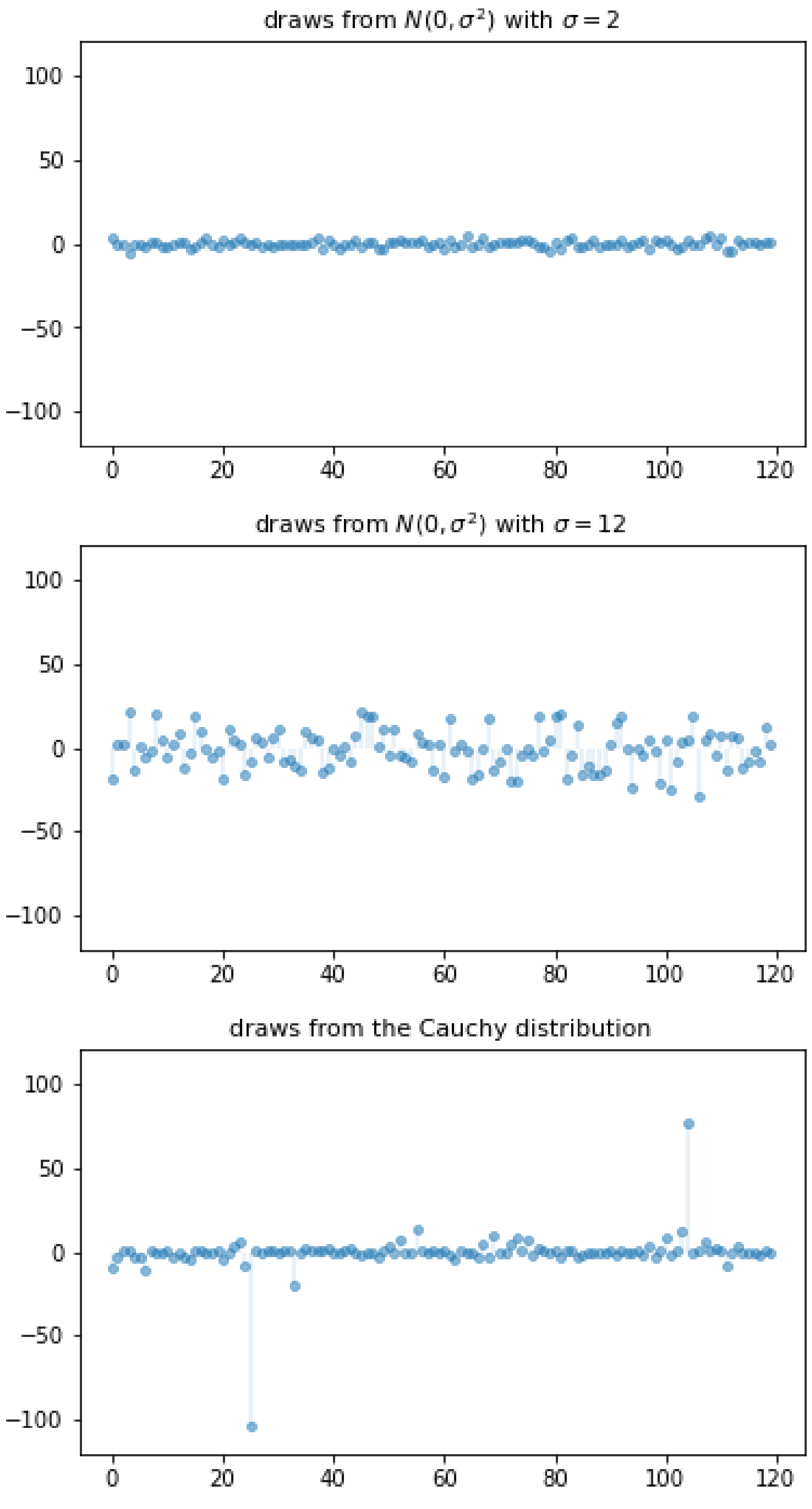
Solution to Exercise 14.5.1

```python
n = 120
np.random.seed(11)

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

for ax in axes:
```

(continues on next page)

(continued from previous page)

```
    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(f"draws from the Cauchy distribution", fontsize=11)

plt.subplots_adjust(hspace=0.25)

plt.show()
```


draws from $N(0, \sigma^2)$ with $\sigma = 2$
draws from $N(0, \sigma^2)$ with $\sigma = 12$
draws from the Cauchy distribution
100
50
0
−50
−100
0
20
40
60
80
100
120

Solution to Exercise 14.5.2

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

Fix $r \geq \alpha$.

As discussed after (14.4), we can take positive constants b and $\bar{x}$ such that

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

But then

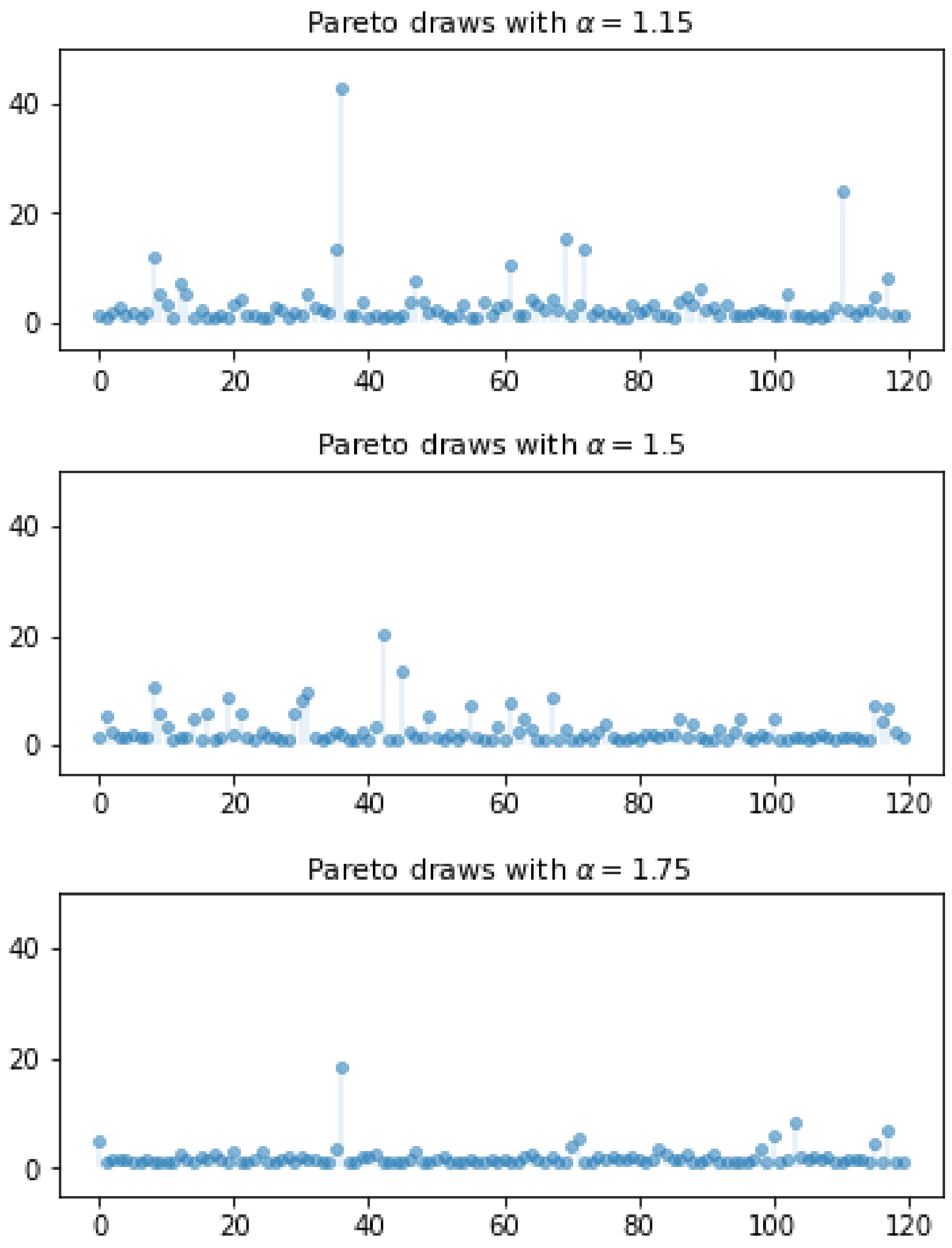
$$\mathbb{E} X^r = r \int_0^\infty x^{r-1} \mathbb{P}\{X > x\} x \geq r \int_0^{\bar{x}} x^{r-1} \mathbb{P}\{X > x\} x + r \int_{\bar{x}}^\infty x^{r-1} b x^{-\alpha} x.$$

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 $\mathbb{E} X^r = \infty$.

Solution to Exercise 14.5.3

```python
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_ylim((-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()
```



Solution to Exercise 14.5.4

First let's generate the data for the plots:

```
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))
```

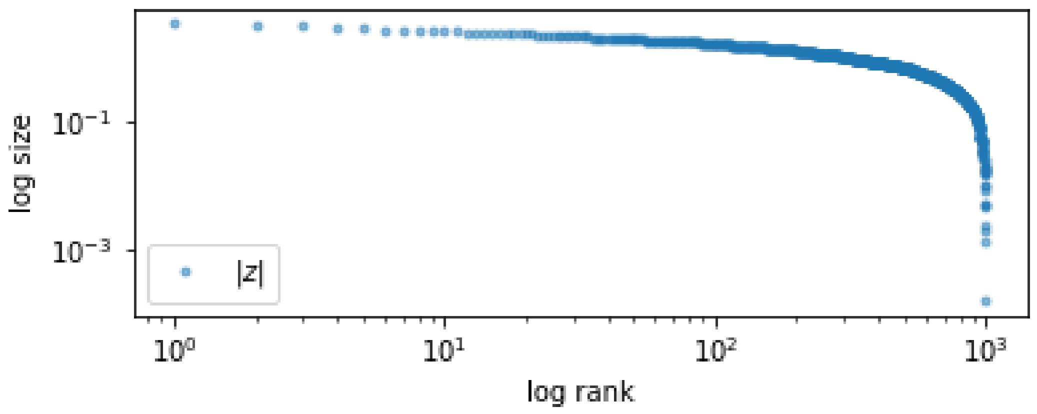
(continues on next page)

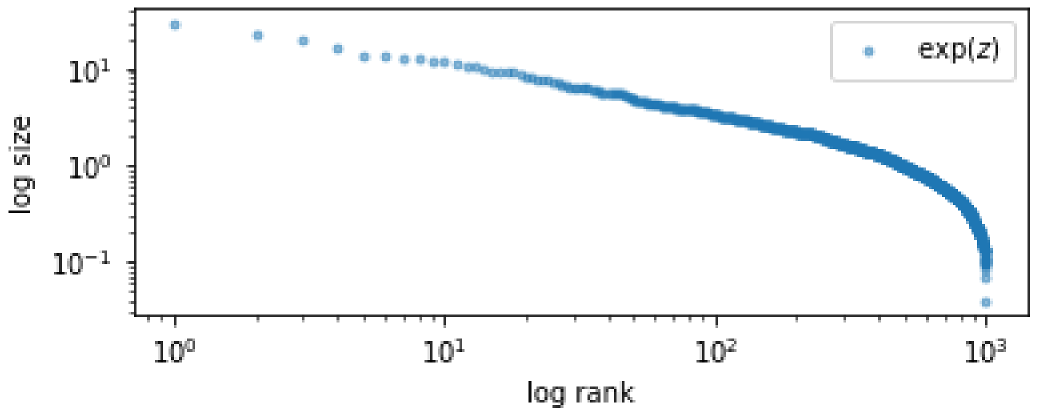
(continued from previous page)

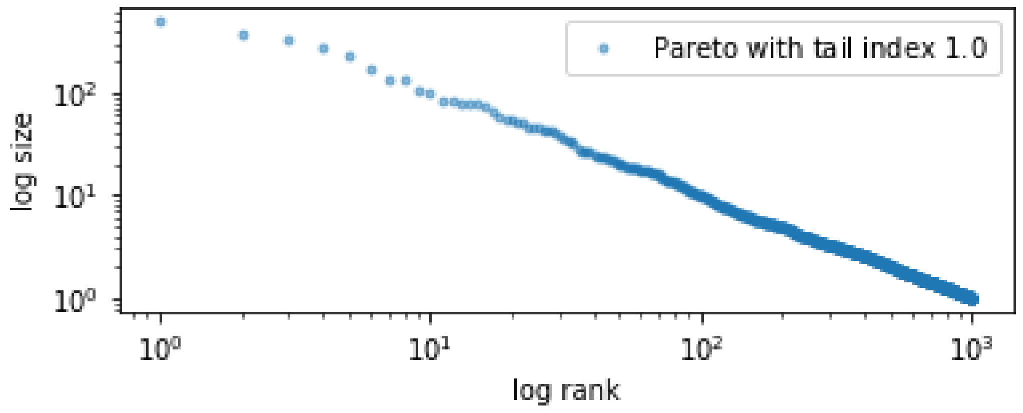
```python
data_list = [data_1, data_2, data_3]
```

Now we plot the data:

```python
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()
```







Solution to Exercise 14.5.5

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 (14.5) 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.

```
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:

```
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`.

```
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
```

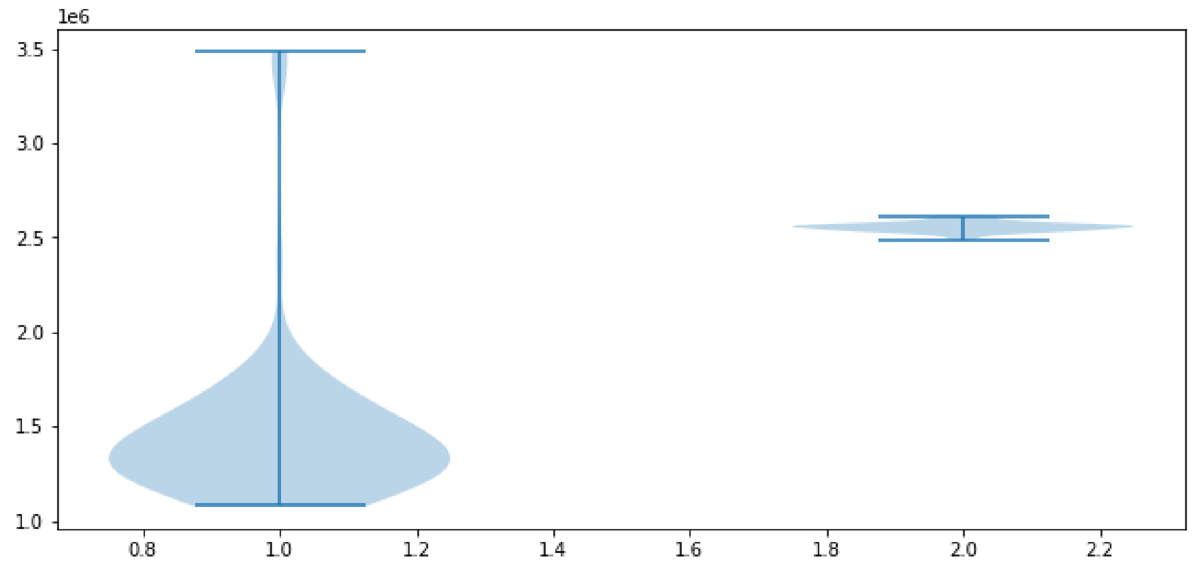
Now let's generate the violin plot.

```
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()
```

(continues on next page)

(continued from previous page)

```
data = tax_rev_pareto, tax_rev_lognorm

ax.violinplot(data)

plt.show()
```



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

```
tax_rev_pareto.mean(), tax_rev_pareto.std()
```

```
(1.4587290546623734e+06, 406089.3613661567)
```

```
tax_rev_lognorm.mean(), tax_rev_lognorm.std()
```

```
(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

FIFTEEN

FAULT TREE UNCERTAINTIES

15.1 Overview

This lecture puts elementary tools to work to approximate probability distributions of the annual failure rates of a system consisting of a number of critical parts.

We'll use log normal distributions to approximate probability distributions of critical component parts.

To approximate the probability distribution of the **sum** of n log normal probability distributions that describes the failure rate of the entire system, we'll compute the convolution of those n log normal probability distributions.

We'll use the following concepts and tools:

- log normal distributions
- the convolution theorem that describes the probability distribution of the sum independent random variables
- fault tree analysis for approximating a failure rate of a multi-component system
- a hierarchical probability model for describing uncertain probabilities
- Fourier transforms and inverse Fourier tranforms as efficient ways of computing convolutions of sequences

For more about Fourier transforms see this quantecon lecture Circulant Matrices as well as these lecture Covariance Stationary Processes and Estimation of Spectra.

El-Shanawany, Ardron, and Walker [ESAW18] and Greenfield and Sargent [GS93] used some of the methods described here to approximate probabilities of failures of safety systems in nuclear facilities.

These methods respond to some of the recommendations made by Apostolakis [Apo90] for constructing procedures for quantifying uncertainty about the reliability of a safety system.

We'll start by bringing in some Python machinery.

```
!pip install tabulate
```

```
Requirement already satisfied: tabulate in /usr/share/miniconda3/envs/quantecon/
↪lib/python3.9/site-packages (0.8.9)
```

```
import numpy as np
from numpy import fft
import matplotlib.pyplot as plt
import scipy as sc
from scipy.signal import fftconvolve
from tabulate import tabulate
```

(continues on next page)

(continued from previous page)

```
import time
%matplotlib inline
```

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

15.2 Log normal distribution

If a random variable x follows a normal distribution with mean μ and variance σ^2, then the natural logarithm of x, say $y = \log(x)$, follows a **log normal distribution** with parameters μ, σ^2.

Notice that we said **parameters** and not **mean and variance** μ, σ^2.

- μ and σ^2 are the mean and variance of $x = \exp(y)$
- they are **not** the mean and variance of y
- instead, the mean of y is $e^{\mu+\frac{1}{2}\sigma^2}$ and the variance of y is $(e^{\sigma^2} - 1)e^{2\mu+\sigma^2}$

A log normal random variable y is nonnegative.

The density for a log normal random variate y is

$$f(y) = \frac{1}{y\sigma\sqrt{2\pi}} \exp\left(\frac{-(\log y - \mu)^2}{2\sigma^2}\right)$$

for $y \geq 0$.

Important features of a log normal random variable are

$$\begin{aligned}
\text{mean:} \quad & e^{\mu+\frac{1}{2}\sigma^2} \\
\text{variance:} \quad & (e^{\sigma^2} - 1)e^{2\mu+\sigma^2} \\
\text{median:} \quad & e^{\mu} \\
\text{mode:} \quad & e^{\mu-\sigma^2} \\
\text{.95 quantile:} \quad & e^{\mu+1.645\sigma} \\
\text{.95-.05 quantile ratio:} \quad & e^{1.645\sigma}
\end{aligned}$$

Recall the following *stability* property of two independent normally distributed random variables:

If x_1 is normal with mean μ_1 and variance σ_1^2 and x_2 is independent of x_1 and normal with mean μ_2 and variance σ_2^2, then $x_1 + x_2$ is normally distributed with mean $\mu_1 + \mu_2$ and variance $\sigma_1^2 + \sigma_2^2$.

Independent log normal distributions have a different *stability* property.

The **product** of independent log normal random variables is also log normal.

In particular, if y_1 is log normal with parameters (μ_1, σ_1^2) and y_2 is log normal with parameters (μ_2, σ_2^2), then the product $y_1 y_2$ is log normal with parameters $(\mu_1 + \mu_2, \sigma_1^2 + \sigma_2^2)$.

Note: While the product of two log normal distributions is log normal, the **sum** of two log normal distributions is **not** log normal.

This observation sets the stage for challenge that confronts us in this lecture, namely, to approximate probability distributions of **sums** of independent log normal random variables.

To compute the probability distribution of the sum of two log normal distributions, we can use the following convolution property of a probability distribution that is a sum of independent random variables.

15.3 The Convolution Property

Let x be a random variable with probability density $f(x)$, where $x \in \mathbf{R}$.

Let y be a random variable with probability density $g(y)$, where $y \in \mathbf{R}$.

Let x and y be independent random variables and let $z = x + y \in \mathbf{R}$.

Then the probability distribution of z is

$$h(z) = (f * g)(z) \equiv \int_{-\infty}^{\infty} f(z) g(z - \tau) d\tau$$

where $(f * g)$ denotes the **convolution** of the two functions f and g.

If the random variables are both nonnegative, then the above formula specializes to

$$h(z) = (f * g)(z) \equiv \int_{0}^{\infty} f(z) g(z - \tau) d\tau$$

Below, we'll use a discretized version of the preceding formula.

In particular, we'll replace both f and g with discretized counterparts, normalized to sum to 1 so that they are probability distributions.

- by **discretized** we mean an equally spaced sampled version

Then we'll use the following version of the above formula

$$h_n = (f * g)_n = \sum_{m=0}^{\infty} f_m g_{n-m}, n \geq 0$$

to compute a discretized version of the probability distribution of the sum of two random variables, one with probability mass function f, the other with probability mass function g.

Before applying the convolution property to sums of log normal distributions, let's practice on some simple discrete distributions.

To take one example, let's consider the following two probability distributions

$$f_j = \text{Prob}(X = j), j = 0, 1$$

and

$$g_j = \text{Prob}(Y = j), j = 0, 1, 2, 3$$

and

$$h_j = \text{Prob}(Z \equiv X + Y = j), j = 0, 1, 2, 3, 4$$

The convolution property tells us that

$$h = f * g = g * f$$

Let's compute an example using the `numpy.convolve` and `scipy.signal.fftconvolve`.

```
f = [.75, .25]
g = [0., .6,  0., .4]
h = np.convolve(f,g)
hf = fftconvolve(f,g)

print("f = ", f,  ", np.sum(f) = ", np.sum(f))
print("g = ", g, ", np.sum(g) = ", np.sum(g))
print("h = ", h, ", np.sum(h) = ", np.sum(h))
print("hf = ", hf, ",np.sum(hf) = ", np.sum(hf))
```

```
f =  [0.75, 0.25] , np.sum(f) =  1.0
g =  [0.0, 0.6, 0.0, 0.4] , np.sum(g) =  1.0
h =  [0.   0.45 0.15 0.3  0.1 ] , np.sum(h) =  1.0
hf =  [0.   0.45 0.15 0.3  0.1 ] ,np.sum(hf) =  1.0000000000000002
```

A little later we'll explain some advantages that come from using `scipy.signal.ftconvolve` rather than `numpy.convolve`.numpy program convolve.

They provide the same answers but `scipy.signal.ftconvolve` is much faster.

That's why we rely on it later in this lecture.

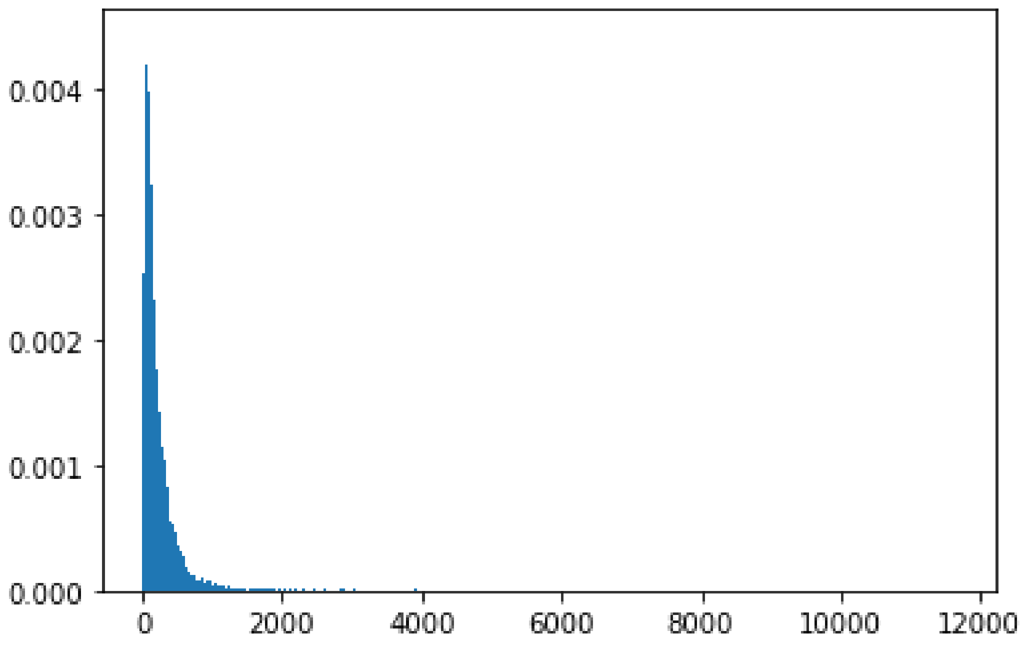
15.4 Approximating Distributions

We'll construct an example to verify that discretized distributions can do a good job of approximating samples drawn from underlying continuous distributions.

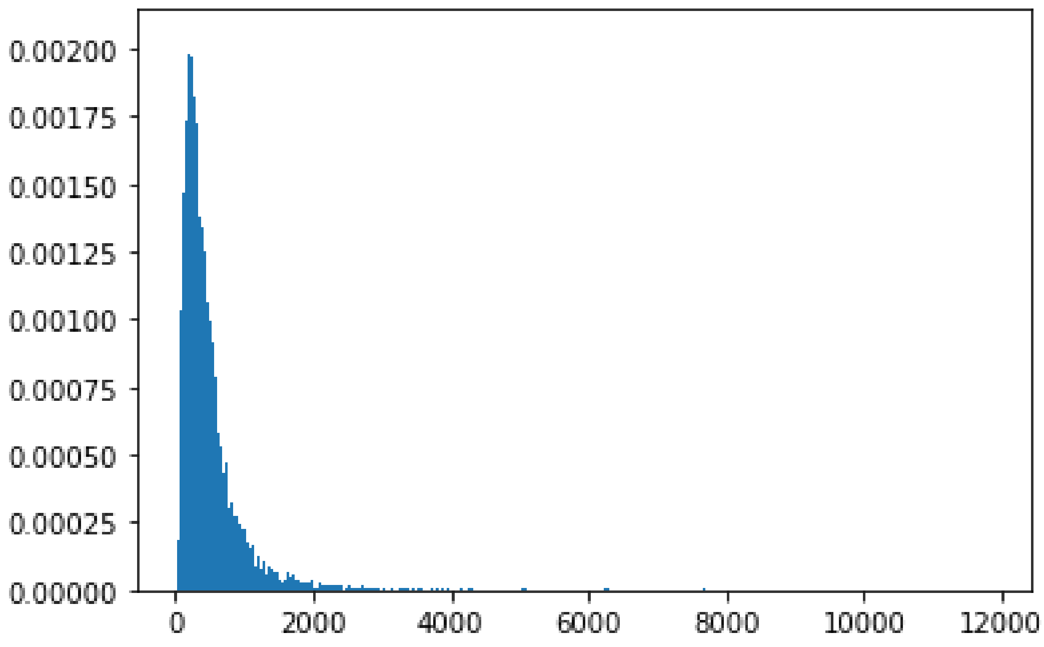
We'll start by generating samples of size 25000 of three independent log normal random variates as well as pairwise and triple-wise sums.

Then we'll plot histograms and compare them with convolutions of appropriate discretized log normal distributions.

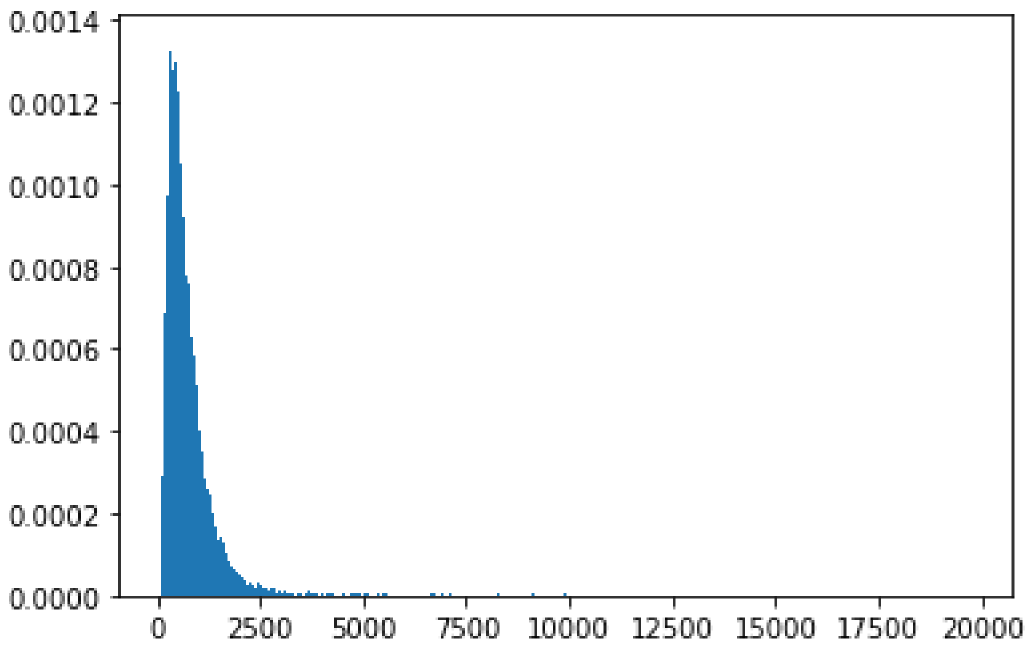
```
## create sums of two and three log normal random variates ssum2 = s1 + s2 and ssum3␣
↪= s1 + s2 + s3


mu1, sigma1 = 5., 1. # mean and standard deviation
s1 = np.random.lognormal(mu1, sigma1, 25000)

mu2, sigma2 = 5., 1. # mean and standard deviation
s2 = np.random.lognormal(mu2, sigma2, 25000)

mu3, sigma3 = 5., 1. # mean and standard deviation
s3 = np.random.lognormal(mu3, sigma3, 25000)

ssum2 = s1 + s2

ssum3 = s1 + s2 + s3

count, bins, ignored = plt.hist(s1, 1000, density=True, align='mid')
```



```python
count, bins, ignored = plt.hist(ssum2, 1000, density=True, align='mid')
```



```python
count, bins, ignored = plt.hist(ssum3, 1000, density=True, align='mid')
```



```
samp_mean2 = np.mean(s2)
pop_mean2 = np.exp(mu2+ (sigma2**2)/2)

pop_mean2, samp_mean2, mu2, sigma2
```

```
(2.4469193226422038e+02, 245.64259335142356, 5.0, 1.0)
```

Here are helper functions that create a discretized version of a log normal probability density function.

```
def p_log_normal(x,μ,σ):
    p = 1 / (σ*x*np.sqrt(2*np.pi)) * np.exp(-1/2*((np.log(x) - μ)/σ)**2)
    return p

def pdf_seq(μ,σ,I,m):
    x = np.arange(1e-7,I,m)
    p_array = p_log_normal(x,μ,σ)
    p_array_norm = p_array/np.sum(p_array)
    return p_array,p_array_norm,x
```

Now we shall set a grid length I and a grid increment size $m = 1$ for our discretizations.

Note: We set I equal to a power of two because we want to be free to use a Fast Fourier Transform to compute a convolution of two sequences (discrete distributions).

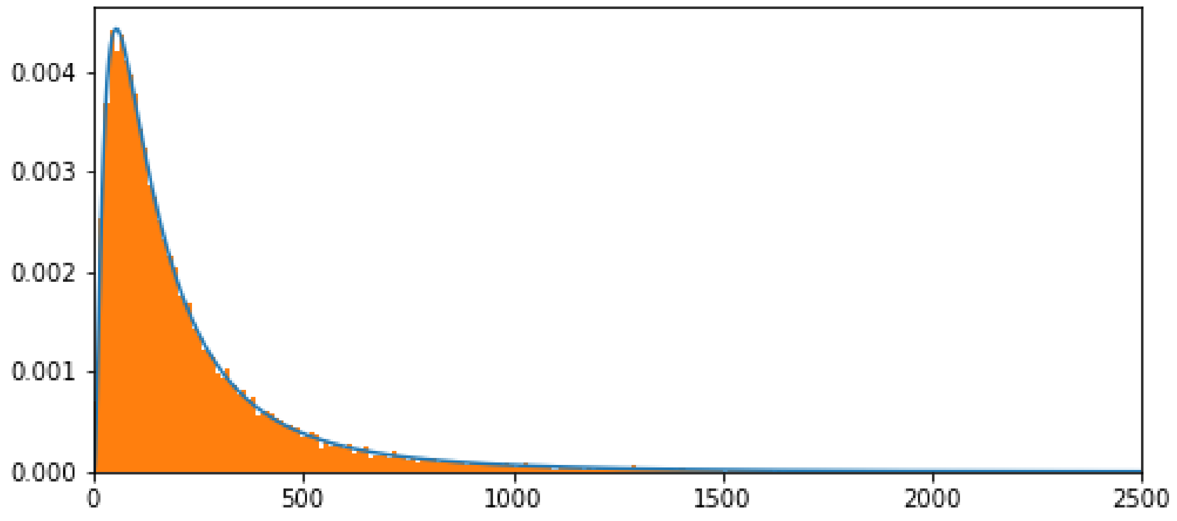
We recommend experimenting with different values of the power p of 2.

Setting it to 15 rather than 12, for example, improves how well the discretized probability mass function approximates the original continuous probability density function being studied.

```
p=15
I = 2**p # Truncation value
m = .1 # increment size
```

```
## Cell to check -- note what happens when don't normalize!
## things match up without adjustment. Compare with above

p1,p1_norm,x = pdf_seq(mu1,sigma1,I,m)
## compute number of points to evaluate the probability mass function
NT = x.size

plt.figure(figsize = (8,8))
plt.subplot(2,1,1)
plt.plot(x[:np.int(NT)],p1[:np.int(NT)],label = '')
plt.xlim(0,2500)
count, bins, ignored = plt.hist(s1, 1000, density=True, align='mid')

plt.show()
```

```
/tmp/ipykernel_15783/2130497458.py:10: DeprecationWarning: `np.int` is a␣
↪deprecated alias for the builtin `int`. To silence this warning, use `int` by␣
↪itself. Doing this will not modify any behavior and is safe. When replacing `np.
↪int`, you may wish to use e.g. `np.int64` or `np.int32` to specify the precision.
↪ If you wish to review your current use, check the release note link for␣
↪additional information.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/
↪release/1.20.0-notes.html#deprecations
  plt.plot(x[:np.int(NT)],p1[:np.int(NT)],label = '')
```



```
# Compute mean from discretized pdf and compare with the theoretical value

mean= np.sum(np.multiply(x[:NT],p1_norm[:NT]))
meantheory = np.exp(mu1+.5*sigma1**2)
mean, meantheory
```

```
(2.446905989830291e+02, 244.69193226422038)
```

15.5 Convolving Probability Mass Functions

Now let's use the convolution theorem to compute the probability distribution of a sum of the two log normal random variables we have parameterized above.

We'll also compute the probability of a sum of three log normal distributions constructed above.

Before we do these things, we shall explain our choice of Python algorithm to compute a convolution of two sequences.

Because the sequences that we convolve are long, we use the `scipy.signal.fftconvolve` function rather than the numpy.convove function.

These two functions give virtually equivalent answers but for long sequences `scipy.signal.fftconvolve` is much faster.

The program `scipy.signal.fftconvolve` uses fast Fourier transforms and their inverses to calculate convolutions.

Let's define the Fourier transform and the inverse Fourier transform.

The **Fourier transform** of a sequence $\{x_t\}_{t=0}^{T-1}$ is a sequence of complex numbers $\{x(\omega_j)\}_{j=0}^{T-1}$ given by

$$x(\omega_j) = \sum_{t=0}^{T-1} x_t \exp(-i\omega_j t) \tag{15.1}$$

where $\omega_j = \frac{2\pi j}{T}$ for $j = 0, 1, \dots, T-1$.

The **inverse Fourier transform** of the sequence $\{x(\omega_j)\}_{j=0}^{T-1}$ is

$$x_t = T^{-1} \sum_{j=0}^{T-1} x(\omega_j) \exp(i\omega_j t) \tag{15.2}$$

The sequences $\{x_t\}_{t=0}^{T-1}$ and $\{x(\omega_j)\}_{j=0}^{T-1}$ contain the same information.

The pair of equations (15.1) and (15.2) tell how to recover one series from its Fourier partner.

The program `scipy.signal.fftconvolve` deploys the theorem that a convolution of two sequences $\{f_k\}, \{g_k\}$ can be computed in the following way:

- Compute Fourier transforms $F(\omega), G(\omega)$ of the $\{f_k\}$ and $\{g_k\}$ sequences, respectively
- Form the product $H(\omega) = F(\omega)G(\omega)$
- The convolution of $f * g$ is the inverse Fourier transform of $H(\omega)$

The **fast Fourier transform** and the associated **inverse fast Fourier transform** execute these calculations very quickly.

This is the algorithm that `scipy.signal.fftconvolve` uses.

Let's do a warmup calculation that compares the times taken by `numpy.convove` and `scipy.signal.fftconvolve`.

```
p1,p1_norm,x = pdf_seq(mu1,sigma1,I,m)
p2,p2_norm,x = pdf_seq(mu2,sigma2,I,m)
p3,p3_norm,x = pdf_seq(mu3,sigma3,I,m)

tic = time.perf_counter()

c1 = np.convolve(p1_norm,p2_norm)
c2 = np.convolve(c1,p3_norm)
```

(continues on next page)

(continued from previous page)

```


toc = time.perf_counter()

tdiff1 = toc - tic

tic = time.perf_counter()

c1f = fftconvolve(p1_norm,p2_norm)
c2f = fftconvolve(c1f,p3_norm)
toc = time.perf_counter()

toc = time.perf_counter()

tdiff2 = toc - tic

print("time with np.convolve = ", tdiff1,  "; time with fftconvolve = ",  tdiff2)
```

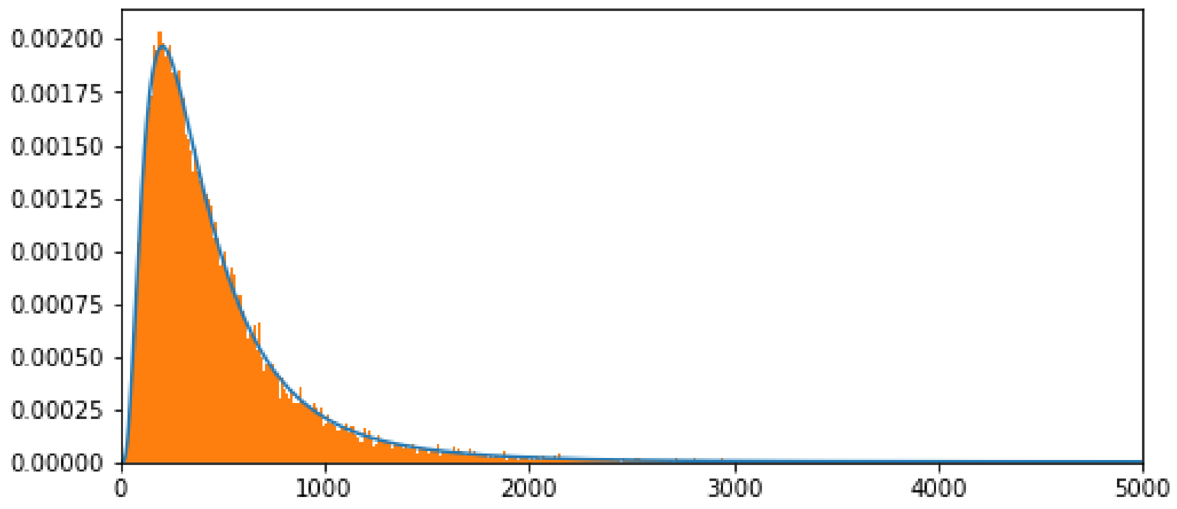
```
time with np.convolve =  78.51970149099998 ; time with fftconvolve =  0.
 ↪11671551499966881
```

The fast Fourier transform is two orders of magnitude faster than `numpy.convolve`

Now let's plot our computed probability mass function approximation for the sum of two log normal random variables against the histogram of the sample that we formed above.

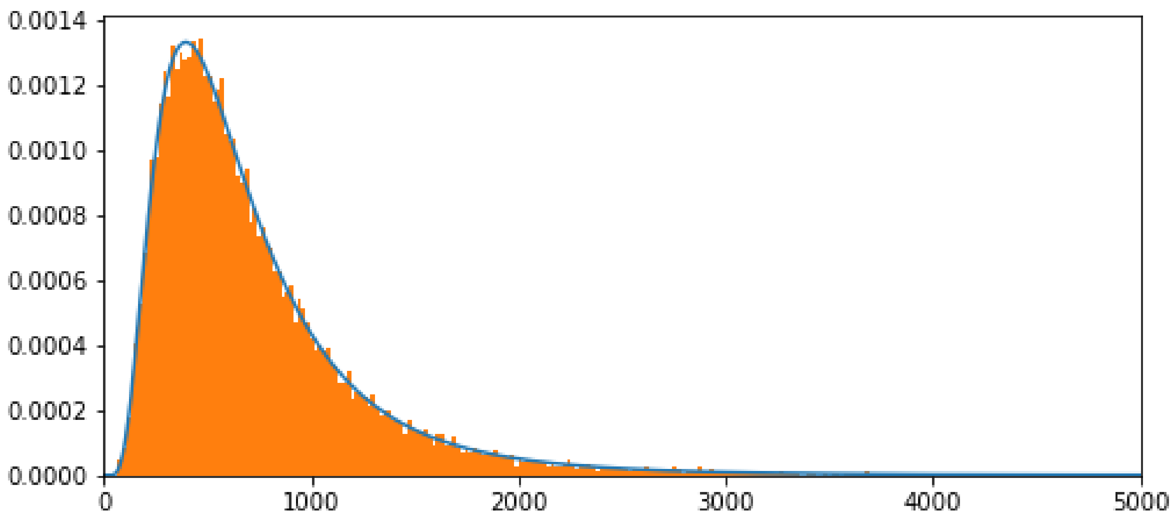
```
NT= np.size(x)

plt.figure(figsize = (8,8))
plt.subplot(2,1,1)
plt.plot(x[:np.int(NT)],c1f[:np.int(NT)]/m,label = '')
plt.xlim(0,5000)

count, bins, ignored = plt.hist(ssum2, 1000, density=True, align='mid')
# plt.plot(P2P3[:10000],label = 'FFT method',linestyle = '--')

plt.show()
```

```
/tmp/ipykernel_15783/2364144025.py:5: DeprecationWarning: `np.int` is a deprecated␣
 ↪alias for the builtin `int`. To silence this warning, use `int` by itself. Doing␣
 ↪this will not modify any behavior and is safe. When replacing `np.int`, you may␣
 ↪wish to use e.g. `np.int64` or `np.int32` to specify the precision. If you wish␣
 ↪to review your current use, check the release note link for additional␣
 ↪information.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/
 ↪release/1.20.0-notes.html#deprecations
  plt.plot(x[:np.int(NT)],c1f[:np.int(NT)]/m,label = '')
```



```python
NT= np.size(x)
plt.figure(figsize = (8,8))
plt.subplot(2,1,1)
plt.plot(x[:np.int(NT)],c2f[:np.int(NT)]/m,label = '')
plt.xlim(0,5000)

count, bins, ignored = plt.hist(ssum3, 1000, density=True, align='mid')
# plt.plot(P2P3[:10000],label = 'FFT method',linestyle = '--')

plt.show()
```

```
/tmp/ipykernel_15783/3883904051.py:4: DeprecationWarning: `np.int` is a deprecated␣
↪alias for the builtin `int`. To silence this warning, use `int` by itself. Doing␣
↪this will not modify any behavior and is safe. When replacing `np.int`, you may␣
↪wish to use e.g. `np.int64` or `np.int32` to specify the precision. If you wish␣
↪to review your current use, check the release note link for additional␣
↪information.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/
↪release/1.20.0-notes.html#deprecations
  plt.plot(x[:np.int(NT)],c2f[:np.int(NT)]/m,label = '')
```



```
## Let's compute the mean of the discretized pdf
mean= np.sum(np.multiply(x[:NT],c1f[:NT]))
# meantheory = np.exp(mu1+.5*sigma1**2)
mean, 2*meantheory
```

```
(489.38109740938546, 489.38386452844077)
```

```
## Let's compute the mean of the discretized pdf
mean= np.sum(np.multiply(x[:NT],c2f[:NT]))
# meantheory = np.exp(mu1+.5*sigma1**2)
mean, 3*meantheory
```

```
(734.0714863312252, 734.0757967926611)
```

15.6 Failure Tree Analysis

We shall soon apply the convolution theorem to compute the probability of a **top event** in a failure tree analysis.

Before applying the convolution theorem, we first describe the model that connects constituent events to the **top** end whose failure rate we seek to quantify.

The model is an example of the widely used **failure tree analysis** described by El-Shanawany, Ardron, and Walker [ESAW18].

To construct the statistical model, we repeatedly use what is called the **rare event approximation**.

We want to compute the probabilty of an event $A \cup B$.

- the union $A \cup B$ is the event that A OR B occurs

A law of probability tells us that A OR B occurs with probability

$$P(A \cup B) = P(A) + P(B) - P(A \cap B)$$

where the intersection $A \cap B$ is the event that A **AND** B both occur and the union $A \cup B$ is the event that A **OR** B occurs.

If A and B are independent, then

$$P(A \cap B) = P(A)P(B)$$

If $P(A)$ and $P(B)$ are both small, then $P(A)P(B)$ is even smaller.

The **rare event approximation** is

$$P(A \cup B) \approx P(A) + P(B)$$

This approximation is widely used in evaluating system failures.

15.7 Application

A system has been designed with the feature a system failure occurs when **any** of n critical components fails.

The failure probability $P(A_i)$ of each event A_i is small.

We assume that failures of the components are statistically independent random variables.

We repeatedly apply a **rare event approximation** to obtain the following formula for the problem of a system failure:

$$P(F) \approx P(A_1) + P(A_2) + \cdots + P(A_n)$$

or

$$P(F) \approx \sum_{i=1}^{n} P(A_i) \tag{15.3}$$

Probabilities for each event are recorded as failure rates per year.

15.8 Failure Rates Unknown

Now we come to the problem that really interests us, following [ESAW18] and Greenfield and Sargent [GS93] in the spirit of Apostolakis [Apo90].

The constituent probabilities or failure rates $P(A_i)$ are not known a priori and have to be estimated.

We address this problem by specifying **probabilities of probabilities** that capture one notion of not knowing the constituent probabilities that are inputs into a failure tree analysis.

Thus, we assume that a system analyst is uncertain about the failure rates $P(A_i), i = 1, \ldots, n$ for components of a system.

The analyst copes with this situation by regarding the systems failure probability $P(F)$ and each of the component probabilities $P(A_i)$ as random variables.

- dispersions of the probability distribution of $P(A_i)$ characterizes the analyst's uncertainty about the failure probability $P(A_i)$
- the dispersion of the implied probability distribution of $P(F)$ characterizes his uncertainty about the probability of a system's failure.

This leads to what is sometimes called a **hierarchical** model in which the analyst has probabilities about the probabilities $P(A_i)$.

The analyst formalizes his uncertainty by assuming that

- the failure probability $P(A_i)$ is itself a log normal random variable with parameters (μ_i, σ_i).

- failure rates $P(A_i)$ and $P(A_j)$ are statistically independent for all pairs with $i \neq j$.

The analyst calibrates the parameters (μ_i, σ_i) for the failure events $i = 1, \ldots, n$ by reading reliability studies in engineering papers that have studied historical failure rates of components that are as similar as possible to the components being used in the system under study.

The analyst assumes that such information about the observed dispersion of annual failure rates, or times to failure, can inform him of what to expect about parts' performances in his system.

The analyst assumes that the random variables $P(A_i)$ are statistically mutually independent.

The analyst wants to approximate a probability mass function and cumulative distribution function of the systems failure probability $P(F)$.

- We say probability mass function because of how we discretize each random variable, as described earlier.

The analyst calculates the probability mass function for the **top event** F, i.e., a **system failure**, by repeatedly applying the convolution theorem to compute the probability distribution of a sum of independent log normal random variables, as described in equation (15.3).

15.9 Waste Hoist Failure Rate

We'll take close to a real world example by assuming that $n = 14$.

The example estimates the annual failure rate of a critical hoist at a nuclear waste facility.

A regulatory agency wants the sytem to be designed in a way that makes the failure rate of the top event small with high probability.

This example is Design Option B-2 (Case I) described in Table 10 on page 27 of [GS93].

The table describes parameters μ_i, σ_i for fourteen log normal random variables that consist of **seven pairs** of random variables that are identically and independently distributed.

- Within a pair, parameters μ_i, σ_i are the same
- As described in table 10 of [GS93] p. 27, parameters of log normal distributions for the seven unique probabilities $P(A_i)$ have been calibrated to be the values in the following Python code:

```
mu1, sigma1 = 4.28, 1.1947
mu2, sigma2 = 3.39, 1.1947
mu3, sigma3 = 2.795, 1.1947
mu4, sigma4 = 2.717, 1.1947
mu5, sigma5 = 2.717, 1.1947
mu6, sigma6 = 1.444, 1.4632
mu7, sigma7 = -.040, 1.4632
```

Note: Because the failure rates are all very small, log normal distributions with the above parameter values actually describe $P(A_i)$ times 10^{-09}.

So the probabilities that we'll put on the x axis of the probability mass function and associated cumulative distribution function should be multiplied by 10^{-09}

To extract a table that summarizes computed quantiles, we'll use a helper function

```
def find_nearest(array, value):
    array = np.asarray(array)
    idx = (np.abs(array - value)).argmin()
    return idx
```

We compute the required thirteen convolutions in the following code.

(Please feel free to try different values of the power parameter p that we use to set the number of points in our grid for constructing the probability mass functions that discretize the continuous log normal distributions.)

We'll plot a counterpart to the cumulative distribution function (CDF) in figure 5 on page 29 of [GS93] and we'll also present a counterpart to their Table 11 on page 28.

```
p=15
I = 2**p # Truncation value
m =  .05 # increment size




p1,p1_norm,x = pdf_seq(mu1,sigma1,I,m)
p2,p2_norm,x = pdf_seq(mu2,sigma2,I,m)
p3,p3_norm,x = pdf_seq(mu3,sigma3,I,m)
p4,p4_norm,x = pdf_seq(mu4,sigma4,I,m)
p5,p5_norm,x = pdf_seq(mu5,sigma5,I,m)
p6,p6_norm,x = pdf_seq(mu6,sigma6,I,m)
p7,p7_norm,x = pdf_seq(mu7,sigma7,I,m)
p8,p8_norm,x = pdf_seq(mu7,sigma7,I,m)
p9,p9_norm,x = pdf_seq(mu7,sigma7,I,m)
p10,p10_norm,x = pdf_seq(mu7,sigma7,I,m)
p11,p11_norm,x = pdf_seq(mu7,sigma7,I,m)
p12,p12_norm,x = pdf_seq(mu7,sigma7,I,m)
p13,p13_norm,x = pdf_seq(mu7,sigma7,I,m)
p14,p14_norm,x = pdf_seq(mu7,sigma7,I,m)

tic = time.perf_counter()

c1 = fftconvolve(p1_norm,p2_norm)
c2 = fftconvolve(c1,p3_norm)
c3 = fftconvolve(c2,p4_norm)
c4 = fftconvolve(c3,p5_norm)
c5 = fftconvolve(c4,p6_norm)
c6 = fftconvolve(c5,p7_norm)
c7 = fftconvolve(c6,p8_norm)
c8 = fftconvolve(c7,p9_norm)
c9 = fftconvolve(c8,p10_norm)
c10 = fftconvolve(c9,p11_norm)
c11 = fftconvolve(c10,p12_norm)
c12 = fftconvolve(c11,p13_norm)
c13 = fftconvolve(c12,p14_norm)

toc = time.perf_counter()

tdiff13 = toc - tic

print("time for 13 convolutions = ", tdiff13)
```

```
time for 13 convolutions =  6.735937851999552
```

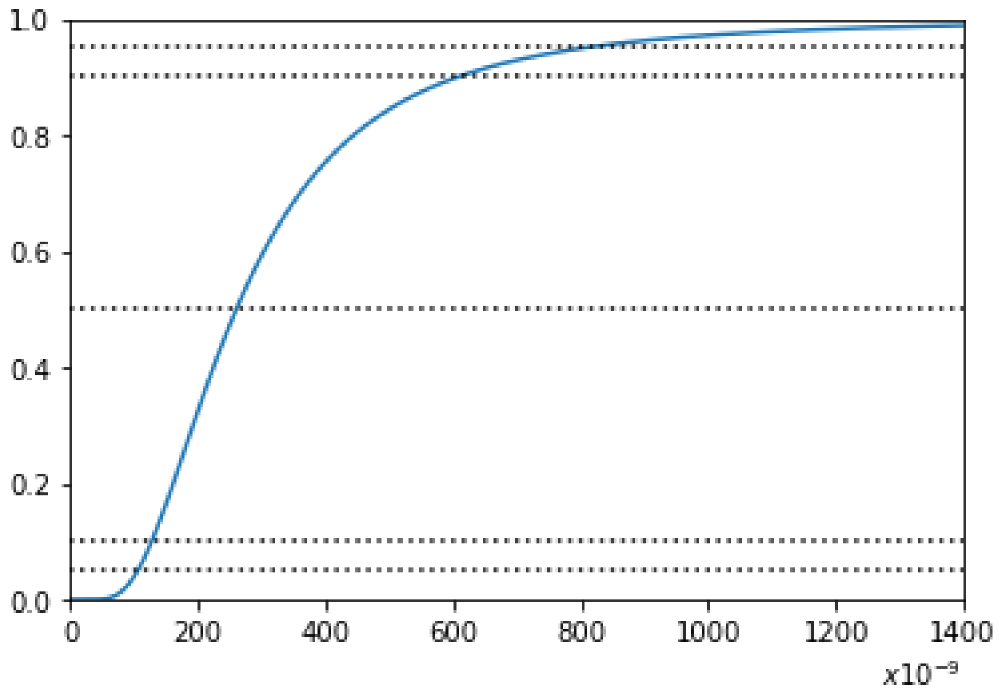
```
d13 = np.cumsum(c13)
Nx=np.int(1400)
plt.figure()
```

(continues on next page)

(continued from previous page)

```
plt.plot(x[0:np.int(Nx/m)],d13[0:np.int(Nx/m)])  # show Yad this -- I multiplied by m␣
↪-- step size
plt.hlines(0.5,min(x),Nx,linestyles='dotted',colors = {'black'})
plt.hlines(0.9,min(x),Nx,linestyles='dotted',colors = {'black'})
plt.hlines(0.95,min(x),Nx,linestyles='dotted',colors = {'black'})
plt.hlines(0.1,min(x),Nx,linestyles='dotted',colors = {'black'})
plt.hlines(0.05,min(x),Nx,linestyles='dotted',colors = {'black'})
plt.ylim(0,1)
plt.xlim(0,Nx)
plt.xlabel("$x10^{-9}$",loc = "right")
plt.show()

x_1 = x[find_nearest(d13,0.01)]
x_5 = x[find_nearest(d13,0.05)]
x_10 = x[find_nearest(d13,0.1)]
x_50 = x[find_nearest(d13,0.50)]
x_66 = x[find_nearest(d13,0.665)]
x_85 = x[find_nearest(d13,0.85)]
x_90 = x[find_nearest(d13,0.90)]
x_95 = x[find_nearest(d13,0.95)]
x_99 = x[find_nearest(d13,0.99)]
x_9978 = x[find_nearest(d13,0.9978)]

print(tabulate([
    ['1%',f"{x_1}"],
    ['5%',f"{x_5}"],
    ['10%',f"{x_10}"],
    ['50%',f"{x_50}"],
    ['66.5%',f"{x_66}"],
    ['85%',f"{x_85}"],
    ['90%',f"{x_90}"],
    ['95%',f"{x_95}"],
    ['99%',f"{x_99}"],
    ['99.78%',f"{x_9978}"]],
    headers = ['Percentile', 'x * 1e-9']))
```

```
/tmp/ipykernel_15783/3082528578.py:2: DeprecationWarning: `np.int` is a deprecated␣
↪alias for the builtin `int`. To silence this warning, use `int` by itself. Doing␣
↪this will not modify any behavior and is safe. When replacing `np.int`, you may␣
↪wish to use e.g. `np.int64` or `np.int32` to specify the precision. If you wish␣
↪to review your current use, check the release note link for additional␣
↪information.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/
↪release/1.20.0-notes.html#deprecations
  Nx=np.int(1400)
/tmp/ipykernel_15783/3082528578.py:4: DeprecationWarning: `np.int` is a deprecated␣
↪alias for the builtin `int`. To silence this warning, use `int` by itself. Doing␣
↪this will not modify any behavior and is safe. When replacing `np.int`, you may␣
↪wish to use e.g. `np.int64` or `np.int32` to specify the precision. If you wish␣
↪to review your current use, check the release note link for additional␣
↪information.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/
↪release/1.20.0-notes.html#deprecations
  plt.plot(x[0:np.int(Nx/m)],d13[0:np.int(Nx/m)])  # show Yad this -- I multiplied␣
↪by m -- step size
```



```
Percentile      x * 1e-9
------------  ----------
1%                 76.15
5%                106.5
10%               128.2
50%               260.55
66.5%             338.55
85%               509.4
90%               608.8
95%               807.6
99%              1470.2
99.78%           2474.85
```

The above table agrees closely with column 2 of Table 11 on p. 28 of of [GS93].

Discrepancies are probably due to slight differences in the number of digits retained in inputting $\mu_i, \sigma_i, i = 1, \ldots, 14$ and in the number of points deployed in the discretizations.

CHAPTER

SIXTEEN

INTRODUCTION TO ARTIFICIAL NEURAL NETWORKS

```
!pip install --upgrade jax jaxlib
!conda install -y -c plotly plotly plotly-orca retrying
```

Note: If you are running this on Google Colab the above cell will present an error. This is because Google Colab doesn't use Anaconda to manage the Python packages. However this lecture will still execute as Google Colab has `plotly` installed.

16.1 Overview

Substantial parts of **machine learning** and **artificial intelligence** are about

- approximating an unknown function with a known function
- estimating the known function from a set of data on the left- and right-hand variables

This lecture describes the structure of a plain vanilla **artificial neural network** (ANN) of a type that is widely used to approximate a function f that maps x in a space X into y in a space Y.

To introduce elementary concepts, we study an example in which x and y are scalars.

We'll describe the following concepts that are brick and mortar for neural networks:

- a neuron
- an activation function
- a network of neurons
- A neural network as a composition of functions
- back-propagation and its relationship to the chain rule of differential calculus

16.2 A Deep (but not Wide) Artificial Neural Network

We describe a “deep” neural network of “width” one.

Deep means that the network composes a large number of functions organized into nodes of a graph.

Width refers to the number of right hand side variables on the right hand side of the function being approximated.

Setting “width” to one means that the network composes just univariate functions.

Let $x \in \mathbb{R}$ be a scalar and $y \in \mathbb{R}$ be another scalar.

We assume that y is a nonlinear function of x:

$$y = f(x)$$

We want to approximate $f(x)$ with another function that we define recursively.

For a network of depth $N \geq 1$, each **layer** $i = 1, \ldots N$ consists of

- an input x_i
- an **affine function** $w_i x_i + bI$, where w_i is a scalar **weight** placed on the input x_i and b_i is a scalar **bias**
- an **activation function** h_i that takes $(w_i x_i + b_i)$ as an argument and produces an output x_{i+1}

An example of an activation function h is the **sigmoid** function

$$h(z) = \frac{1}{1 + e^{-z}}$$

Another popular activation function is the **rectified linear unit** (ReLU) function

$$h(z) = \max(0, z)$$

Yet another activation function is the identity function

$$h(z) = z$$

As activation functions below, we’ll use the sigmoid function for layers 1 to $N - 1$ and the identity function for layer N.

To approximate a function $f(x)$ we construct $\hat{f}(x)$ by proceeding as follows.

Let

$$l_i\left(x\right) = w_i x + b_i.$$

We construct $\hat{f}$ by iterating on compositions of functions $h_i \circ l_i$:

$$f(x) \approx \hat{f}(x) = h_N \circ l_N \circ h_{N-1} \circ l_1 \circ \cdots \circ h_1 \circ l_1(x)$$

If $N > 1$, we call the right side a “deep” neural net.

The larger is the integer N, the “deeper” is the neural net.

Evidently, if we know the parameters $\{w_i, b_i\}_{i=1}^N$, then we can compute $\hat{f}(x)$ for a given $x = \tilde{x}$ by iterating on the recursion

$$x_{i+1} = h_i \circ l_i(x_i), \quad , i = 1, \ldots N \tag{16.1}$$

starting from $x_1 = \tilde{x}$.

The value of x_{N+1} that emerges from this iterative scheme equals $\hat{f}(\tilde{x})$.