Part III

Introduction to Dynamics

CHAPTER

FIFTEEN

DYNAMICS IN ONE DIMENSION

Contents

15.1 Overview

In this lecture we give a quick introduction to discrete time dynamics in one dimension.

In one-dimensional models, the state of the system is described by a single variable.

Although most interesting dynamic models have two or more state variables, the one-dimensional setting is a good place to learn the foundations of dynamics and build intuition.

Let's start with some standard imports:

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

15.2 Some Definitions

This section sets out the objects of interest and the kinds of properties we study.

15.2.1 Difference Equations

A **time homogeneous first order difference equation** is an equation of the form

$$x_{t+1} = g(x_t) \tag{1}$$

where g is a function from some subset S of $\mathbb{R}$ to itself.

Here S is called the **state space** and x is called the **state variable**.

In the definition,

- time homogeneity means that g is the same at each time t
- first order means dependence on only one lag (i.e., earlier states such as x_{t-1} do not enter into (1)).

If $x_0 \in S$ is given, then (1) recursively defines the sequence

$$x_0, \quad x_1 = g(x_0), \quad x_2 = g(x_1) = g(g(x_0)), \quad \text{etc.} \tag{2}$$

This sequence is called the **trajectory** of x_0 under g.

If we define g^n to be n compositions of g with itself, then we can write the trajectory more simply as $x_t = g^t(x_0)$ for $t \geq 0$.

15.2.2 Example: A Linear Model

One simple example is the **linear difference equation**

$$x_{t+1} = ax_t + b, \qquad S = \mathbb{R}$$

where a, b are fixed constants.

In this case, given x_0, the trajectory (2) is

$$x_0, \quad ax_0 + b, \quad a^2x_0 + ab + b, \quad \text{etc.} \tag{3}$$

Continuing in this way, and using our knowledge of *geometric series*, we find that, for any $t \geq 0$,

$$x_t = a^t x_0 + b\frac{1 - a^t}{1 - a} \tag{4}$$

This is about all we need to know about the linear model.

We have an exact expression for x_t for all t and hence a full understanding of the dynamics.

Notice in particular that $|a| < 1$, then, by (4), we have

$$x_t \to \frac{b}{1 - a} \text{ as } t \to \infty \tag{5}$$

regardless of x_0

This is an example of what is called global stability, a topic we return to below.

15.2.3 Example: A Nonlinear Model

In the linear example above, we obtained an exact analytical expression for x_t in terms of arbitrary t and x_0.

This made analysis of dynamics very easy.

When models are nonlinear, however, the situation can be quite different.

For example, recall how we previously studied the law of motion for the Solow growth model, a simplified version of which is

$$k_{t+1} = szk_t^\alpha + (1-\delta)k_t \tag{6}$$

Here k is capital stock and s, z, α, δ are positive parameters with $0 < \alpha, \delta < 1$.

If you try to iterate like we did in (3), you will find that the algebra gets messy quickly.

Analyzing the dynamics of this model requires a different method (see below).

15.2.4 Stability

A **steady state** of the difference equation $x_{t+1} = g(x_t)$ is a point x^* in S such that $x^* = g(x^*)$.

In other words, x^* is a **fixed point** of the function g in S.

For example, for the linear model $x_{t+1} = ax_t + b$, you can use the definition to check that

- $x^* := b/(1-a)$ is a steady state whenever $a \neq 1$.
- if $a = 1$ and $b = 0$, then every $x \in \mathbb{R}$ is a steady state.
- if $a = 1$ and $b \neq 0$, then the linear model has no steady state in $\mathbb{R}$.

A steady state x^* of $x_{t+1} = g(x_t)$ is called **globally stable** if, for all $x_0 \in S$,

$$x_t = g^t(x_0) \to x^* \text{ as } t \to \infty$$

For example, in the linear model $x_{t+1} = ax_t + b$ with $a \neq 1$, the steady state x^*

- is globally stable if $|a| < 1$ and
- fails to be globally stable otherwise.

This follows directly from (4).

A steady state x^* of $x_{t+1} = g(x_t)$ is called **locally stable** if there exists an $\epsilon > 0$ such that

$$|x_0 - x^*| < \epsilon \implies x_t = g^t(x_0) \to x^* \text{ as } t \to \infty$$

Obviously every globally stable steady state is also locally stable.

We will see examples below where the converse is not true.

15.3 Graphical Analysis

As we saw above, analyzing the dynamics for nonlinear models is nontrivial.

There is no single way to tackle all nonlinear models.

However, there is one technique for one-dimensional models that provides a great deal of intuition.

This is a graphical approach based on **45 degree diagrams**.

Let's look at an example: the Solow model with dynamics given in (6).

We begin with some plotting code that you can ignore at first reading.

The function of the code is to produce 45 degree diagrams and time series plots.

```
def subplots(fs):
    "Custom subplots with axes throught the origin"
    fig, ax = plt.subplots(figsize=fs)

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

    return fig, ax


def plot45(g, xmin, xmax, x0, num_arrows=6, var='x'):

    xgrid = np.linspace(xmin, xmax, 200)

    fig, ax = subplots((6.5, 6))
    ax.set_xlim(xmin, xmax)
    ax.set_ylim(xmin, xmax)

    hw = (xmax - xmin) * 0.01
    hl = 2 * hw
    arrow_args = dict(fc="k", ec="k", head_width=hw,
            length_includes_head=True, lw=1,
            alpha=0.6, head_length=hl)

    ax.plot(xgrid, g(xgrid), 'b-', lw=2, alpha=0.6, label='g')
    ax.plot(xgrid, xgrid, 'k-', lw=1, alpha=0.7, label='45')

    x = x0
    xticks = [xmin]
    xtick_labels = [xmin]

    for i in range(num_arrows):
        if i == 0:
            ax.arrow(x, 0.0, 0.0, g(x), **arrow_args) # x, y, dx, dy
        else:
            ax.arrow(x, x, 0.0, g(x) - x, **arrow_args)
            ax.plot((x, x), (0, x), 'k', ls='dotted')

        ax.arrow(x, g(x), g(x) - x, 0, **arrow_args)
```

(continues on next page)

(continued from previous page)

```
        xticks.append(x)
        xtick_labels.append(r'${}_{}$'.format(var, str(i)))

        x = g(x)
        xticks.append(x)
        xtick_labels.append(r'${}_{}$'.format(var, str(i+1)))
        ax.plot((x, x), (0, x), 'k-', ls='dotted')

    xticks.append(xmax)
    xtick_labels.append(xmax)
    ax.set_xticks(xticks)
    ax.set_yticks(xticks)
    ax.set_xticklabels(xtick_labels)
    ax.set_yticklabels(xtick_labels)

    bbox = (0., 1.04, 1., .104)
    legend_args = {'bbox_to_anchor': bbox, 'loc': 'upper right'}

    ax.legend(ncol=2, frameon=False, **legend_args, fontsize=14)
    plt.show()

def ts_plot(g, xmin, xmax, x0, ts_length=6, var='x'):
    fig, ax = subplots((7, 5.5))
    ax.set_ylim(xmin, xmax)
    ax.set_xlabel(r'$t$', fontsize=14)
    ax.set_ylabel(r'${}_t$'.format(var), fontsize=14)
    x = np.empty(ts_length)
    x[0] = x0
    for t in range(ts_length-1):
        x[t+1] = g(x[t])
    ax.plot(range(ts_length),
            x,
            'bo-',
            alpha=0.6,
            lw=2,
            label=r'${}_t$'.format(var))
    ax.legend(loc='best', fontsize=14)
    ax.set_xticks(range(ts_length))
    plt.show()
```

Let's create a 45 degree diagram for the Solow model with a fixed set of parameters

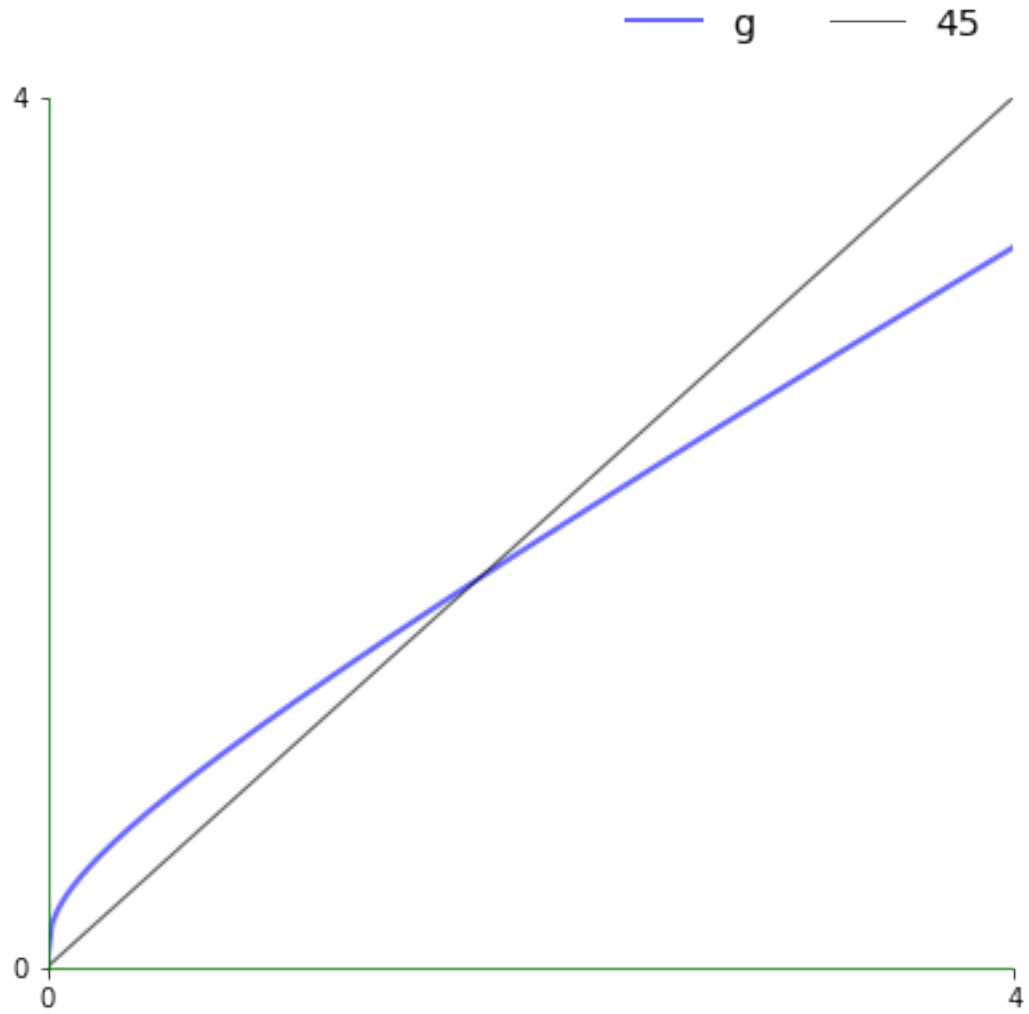
```
A, s, alpha, delta = 2, 0.3, 0.3, 0.4
```

Here's the update function corresponding to the model.

```
def g(k):
    return A * s * k**alpha + (1 - delta) * k
```

Here is the 45 degree plot.

```
xmin, xmax = 0, 4  # Suitable plotting region.

plot45(g, xmin, xmax, 0, num_arrows=0)
```



The plot shows the function g and the 45 degree line.

Think of k_t as a value on the horizontal axis.

To calculate k_{t+1}, we can use the graph of g to see its value on the vertical axis.

Clearly,

- If g lies above the 45 degree line at this point, then we have $k_{t+1} > k_t$.
- If g lies below the 45 degree line at this point, then we have $k_{t+1} < k_t$.
- If g hits the 45 degree line at this point, then we have $k_{t+1} = k_t$, so k_t is a steady state.

For the Solow model, there are two steady states when $S = \mathbb{R}_+ = [0, \infty)$.

- the origin $k = 0$
- the unique positive number such that $k = szk^\alpha + (1 - \delta)k$.

By using some algebra, we can show that in the second case, the steady state is

$$k^* = \left(\frac{sz}{\delta}\right)^{1/(1-\alpha)}$$

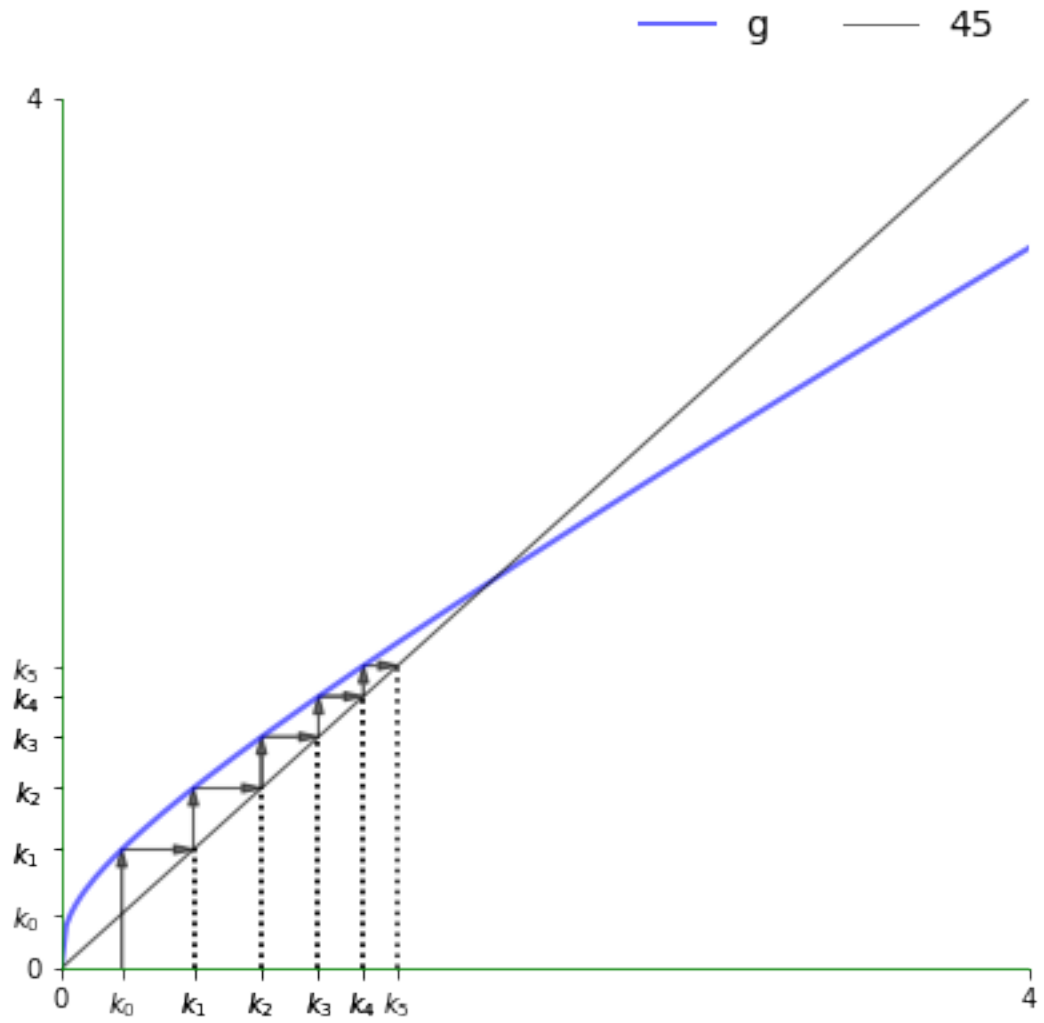
15.3.1 Trajectories

By the preceding discussion, in regions where g lies above the 45 degree line, we know that the trajectory is increasing.

The next figure traces out a trajectory in such a region so we can see this more clearly.

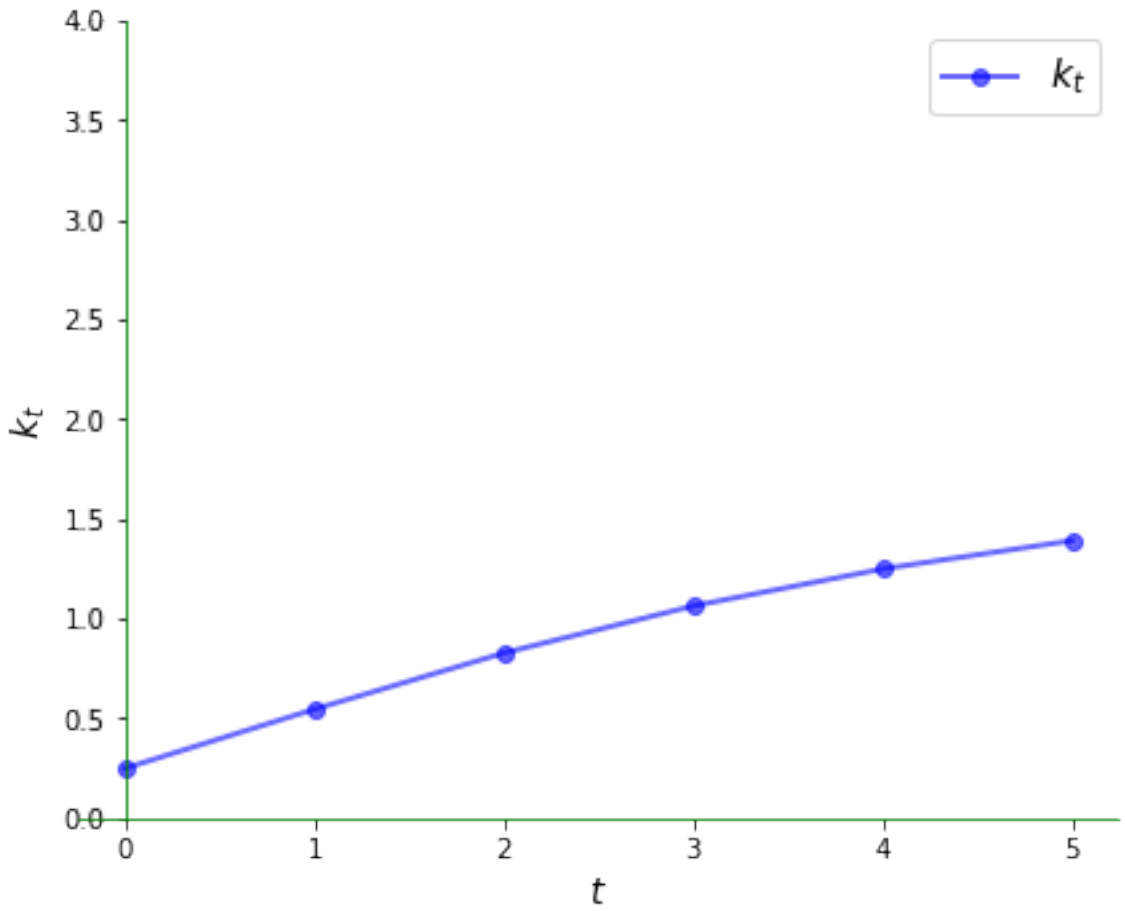
The initial condition is $k_0 = 0.25$.

```
k0 = 0.25

plot45(g, xmin, xmax, k0, num_arrows=5, var='k')
```



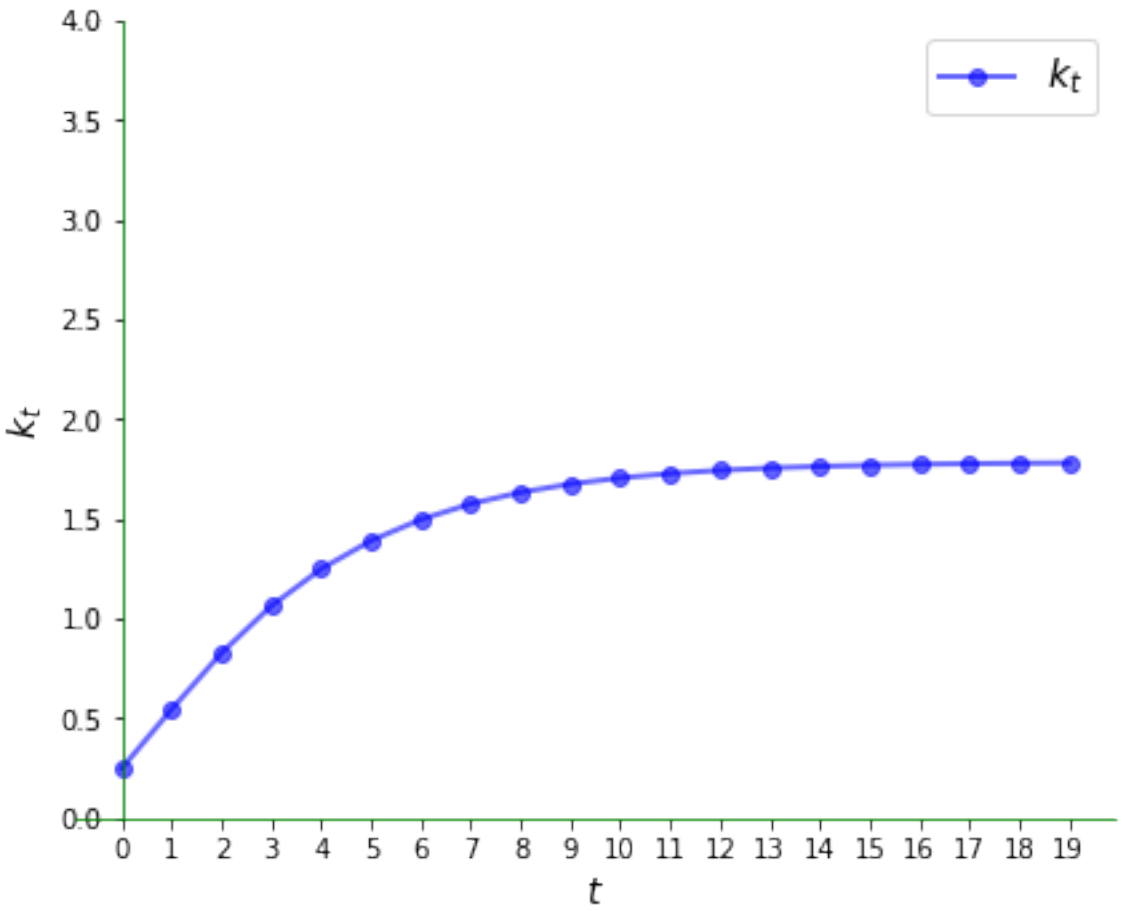
We can plot the time series of capital corresponding to the figure above as follows:

```
ts_plot(g, xmin, xmax, k0, var='k')
```



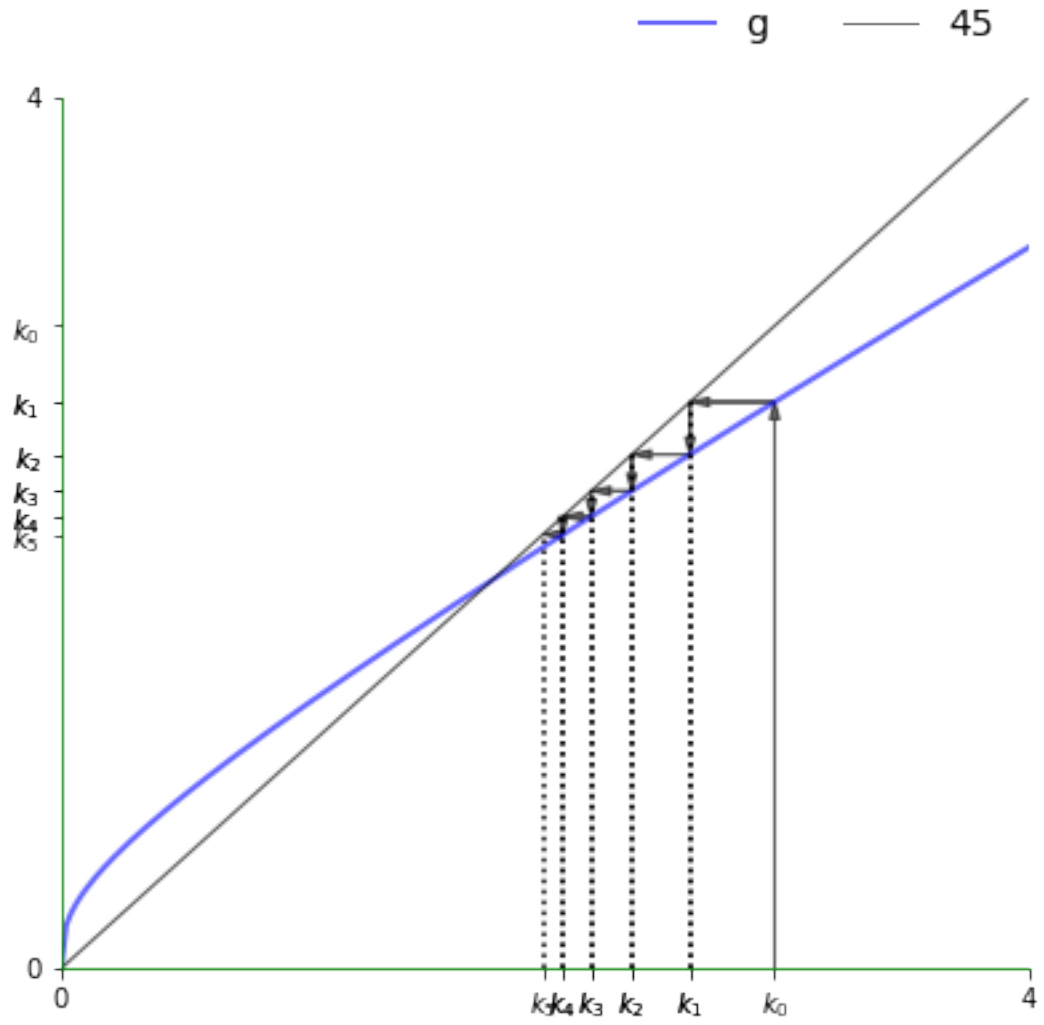
Here's a somewhat longer view:

```
ts_plot(g, xmin, xmax, k0, ts_length=20, var='k')
```



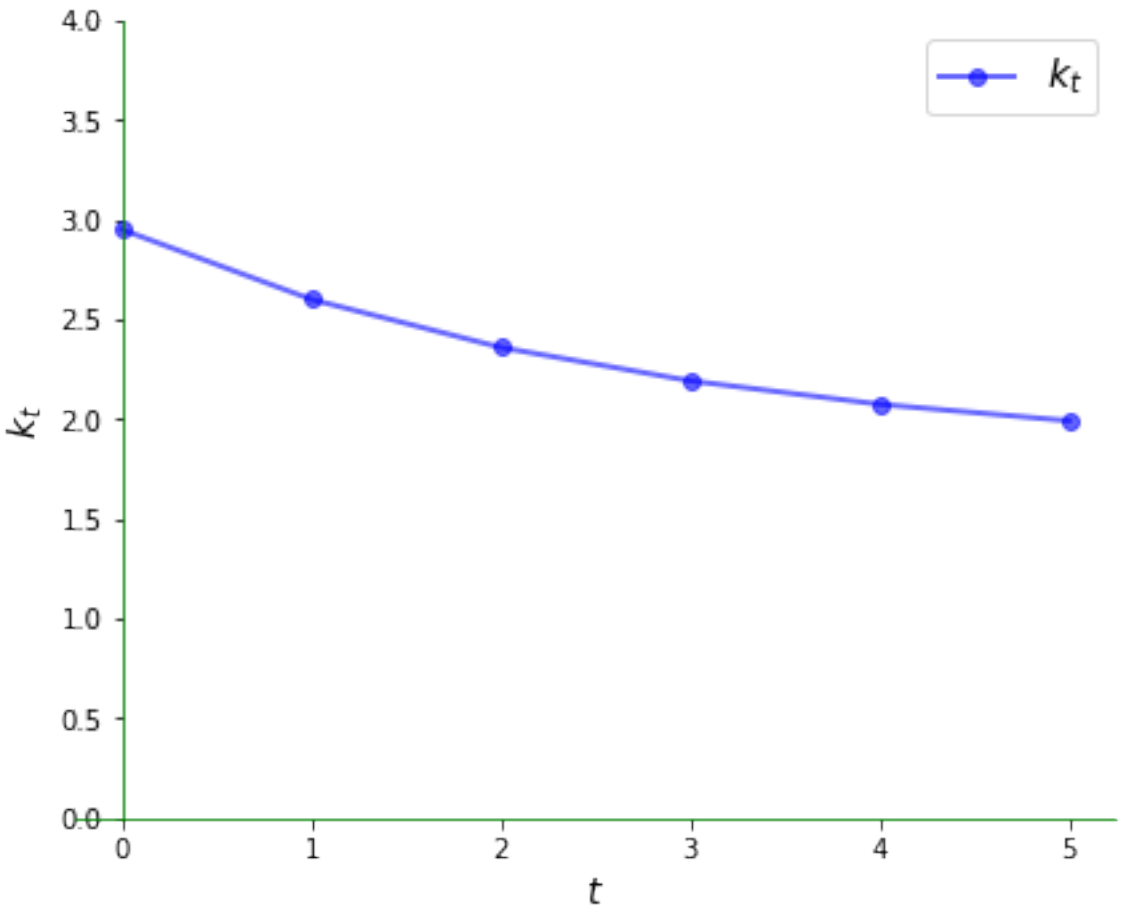
When capital stock is higher than the unique positive steady state, we see that it declines:

```
k0 = 2.95

plot45(g, xmin, xmax, k0, num_arrows=5, var='k')
```



Here is the time series:

```
ts_plot(g, xmin, xmax, k0, var='k')
```



15.3.2 Complex Dynamics

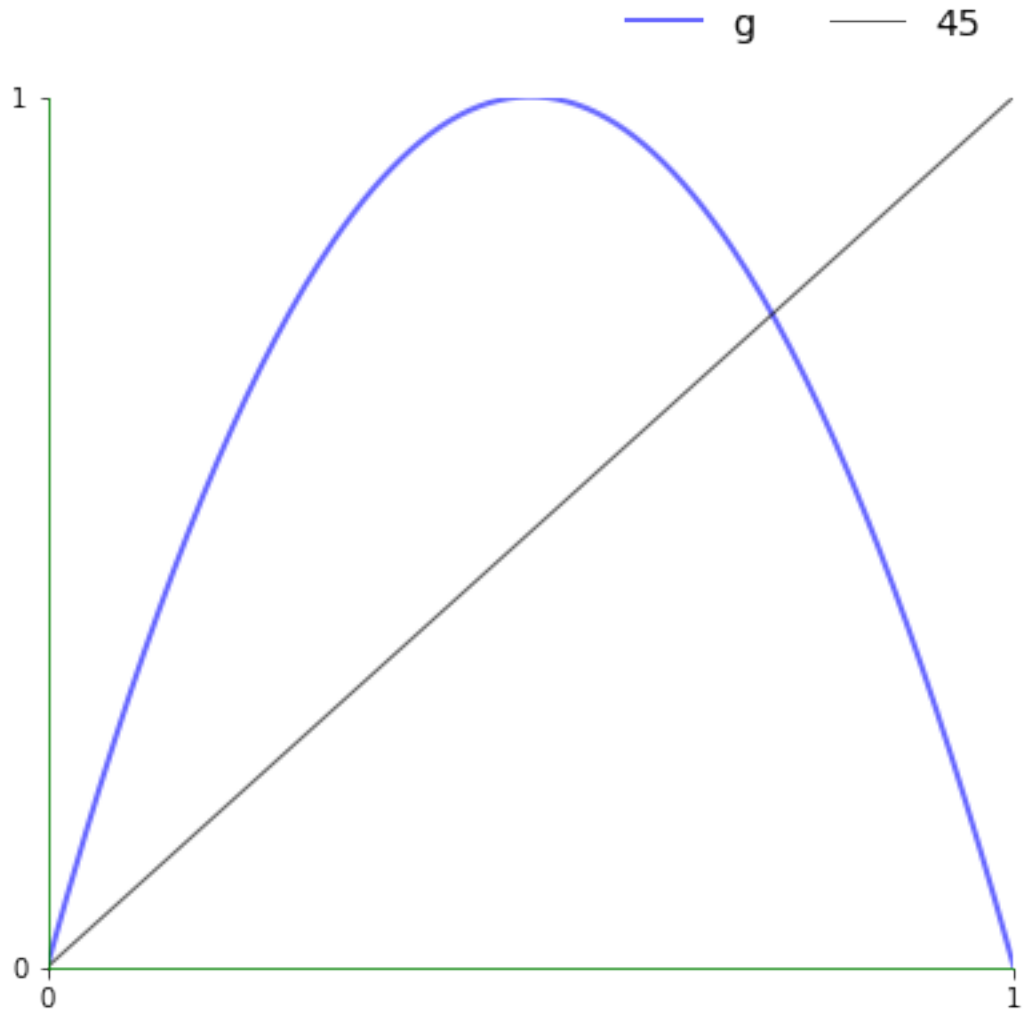
The Solow model is nonlinear but still generates very regular dynamics.

One model that generates irregular dynamics is the **quadratic map**

$$g(x) = 4x(1-x), \qquad x \in [0,1]$$

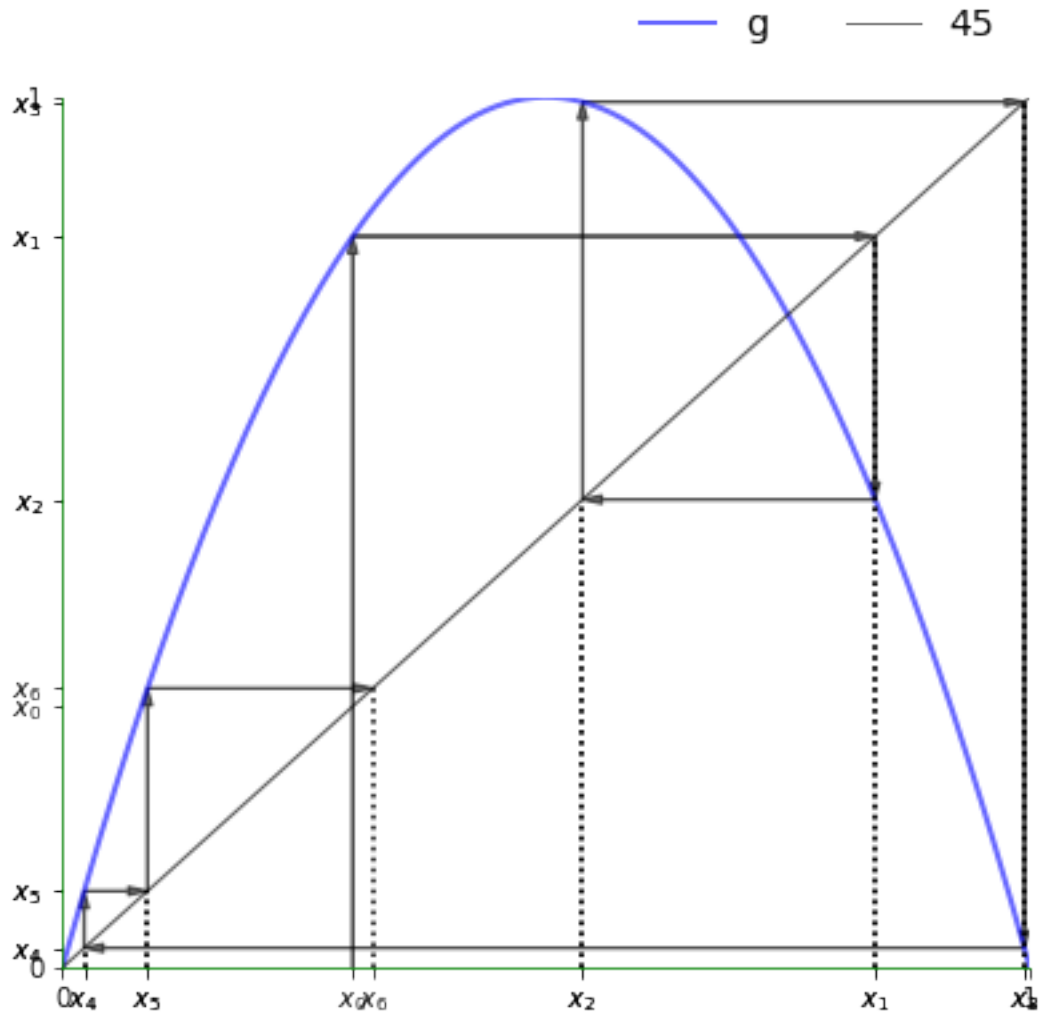
Let's have a look at the 45 degree diagram.

```
xmin, xmax = 0, 1
g = lambda x: 4 * x * (1 - x)

x0 = 0.3
plot45(g, xmin, xmax, x0, num_arrows=0)
```



Now let's look at a typical trajectory.

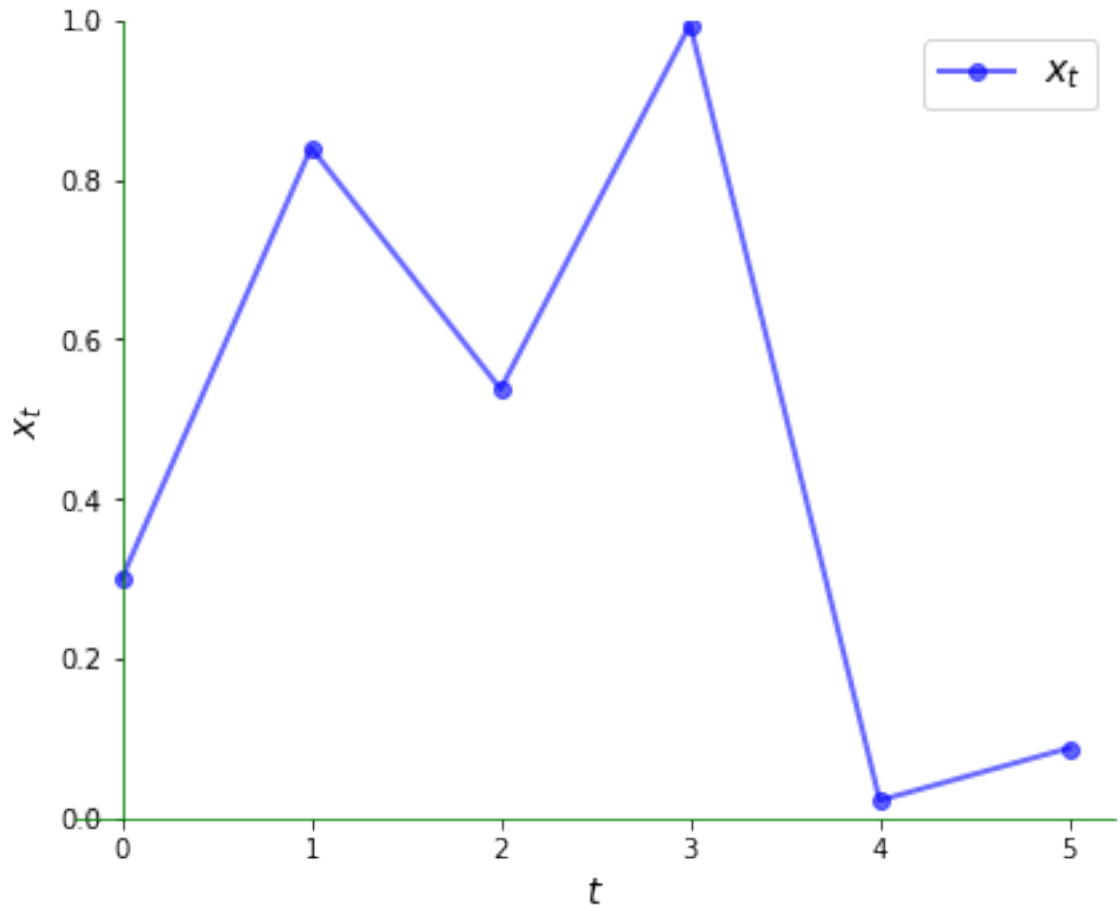
```
plot45(g, xmin, xmax, x0, num_arrows=6)
```



Notice how irregular it is.

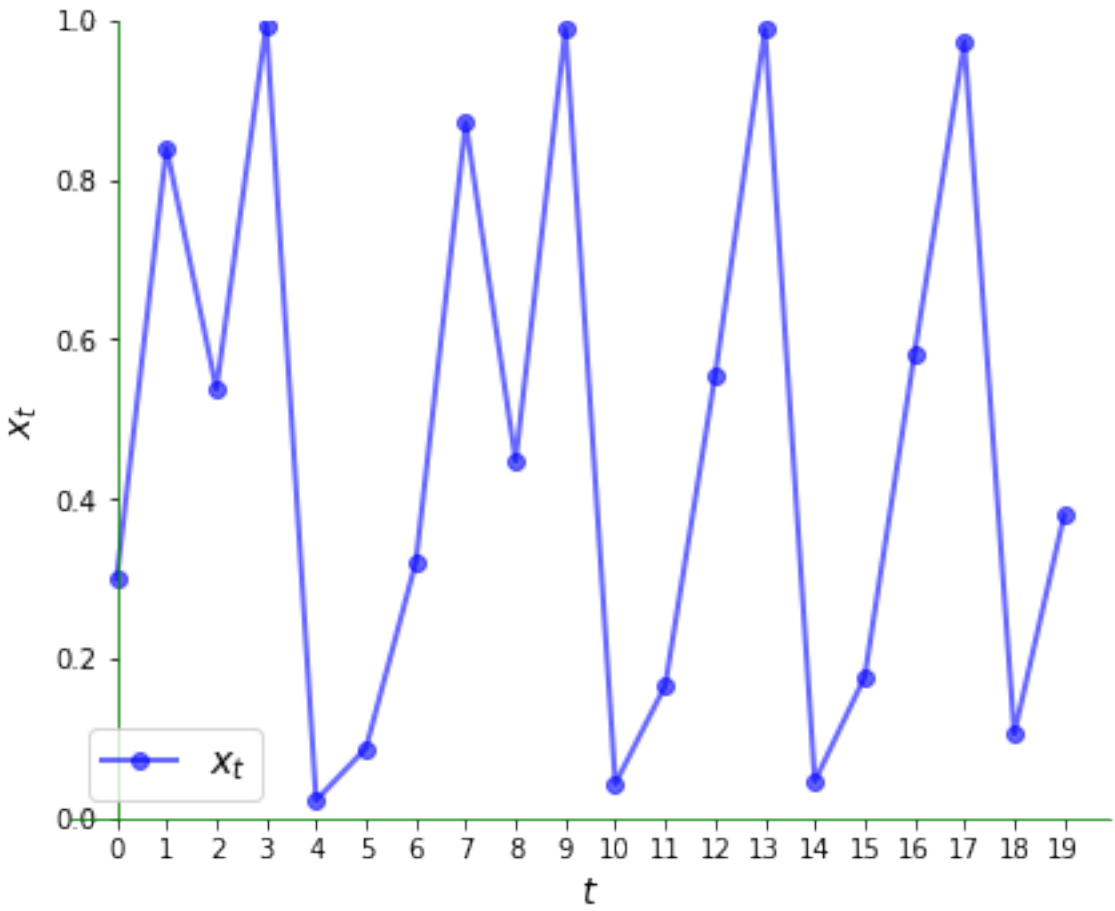
Here is the corresponding time series plot.

```
ts_plot(g, xmin, xmax, x0, ts_length=6)
```



The irregularity is even clearer over a longer time horizon:

```
ts_plot(g, xmin, xmax, x0, ts_length=20)
```



15.4 Exercises

15.4.1 Exercise 1

Consider again the linear model $x_{t+1} = ax_t + b$ with $a \neq 1$.

The unique steady state is $b/(1-a)$.

The steady state is globally stable if $|a| < 1$.

Try to illustrate this graphically by looking at a range of initial conditions.

What differences do you notice in the cases $a \in (-1, 0)$ and $a \in (0, 1)$?

Use $a = 0.5$ and then $a = -0.5$ and study the trajectories

Set $b = 1$ throughout.

15.5 Solutions

15.5.1 Exercise 1

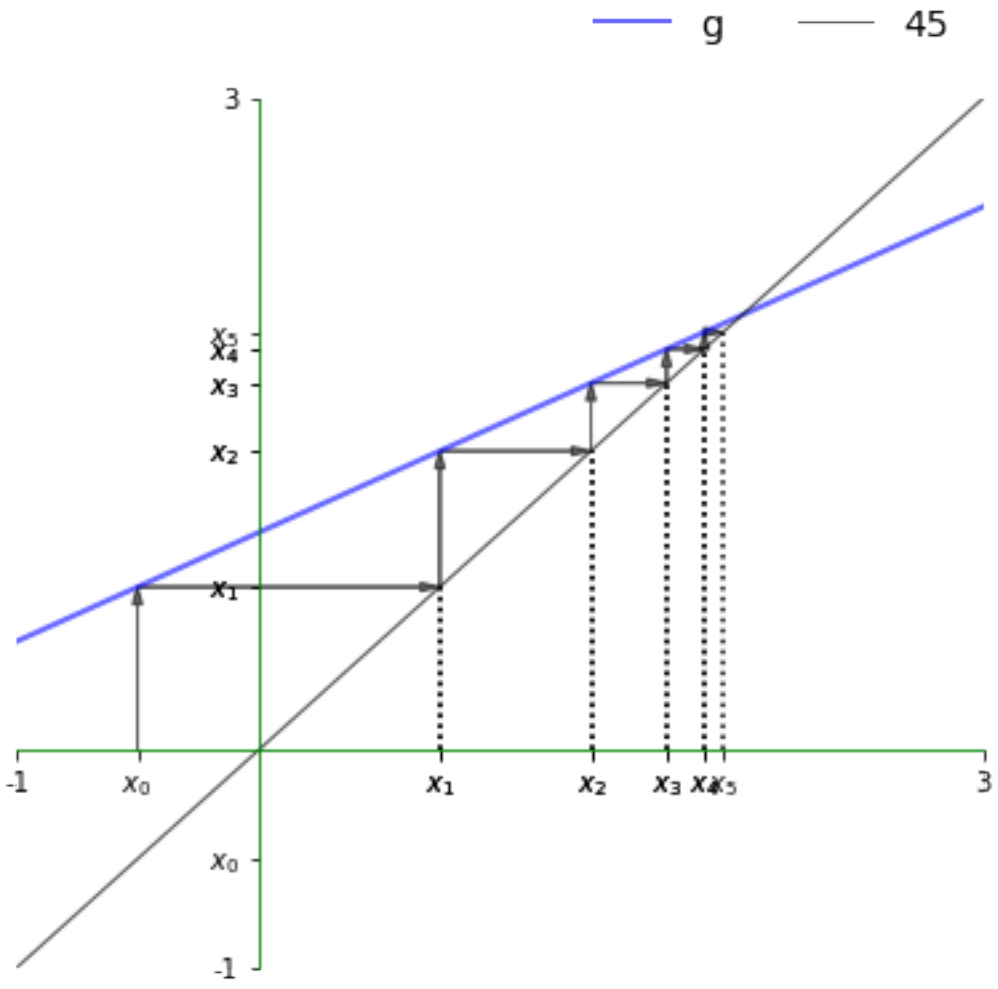
We will start with the case $a = 0.5$.

Let's set up the model and plotting region:

```
a, b = 0.5, 1
xmin, xmax = -1, 3
g = lambda x: a * x + b
```

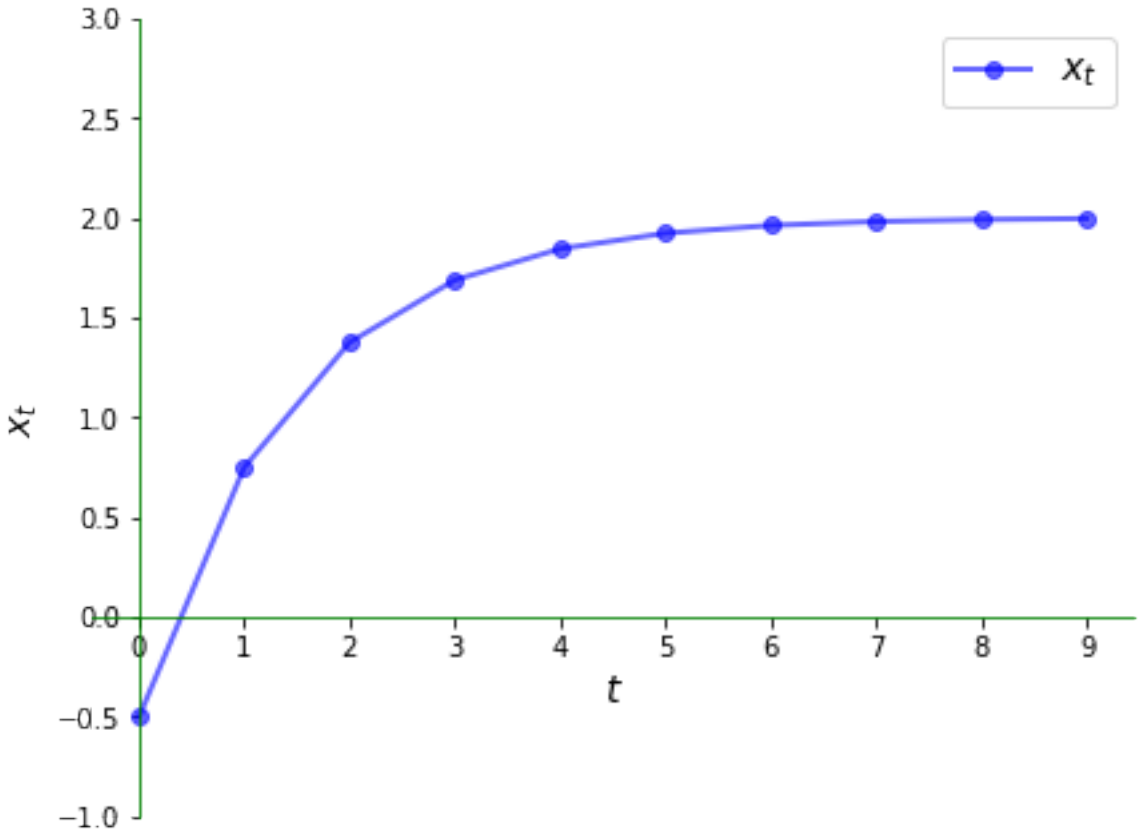
Now let's plot a trajectory:

```
x0 = -0.5
plot45(g, xmin, xmax, x0, num_arrows=5)
```



Here is the corresponding time series, which converges towards the steady state.

```
ts_plot(g, xmin, xmax, x0, ts_length=10)
```



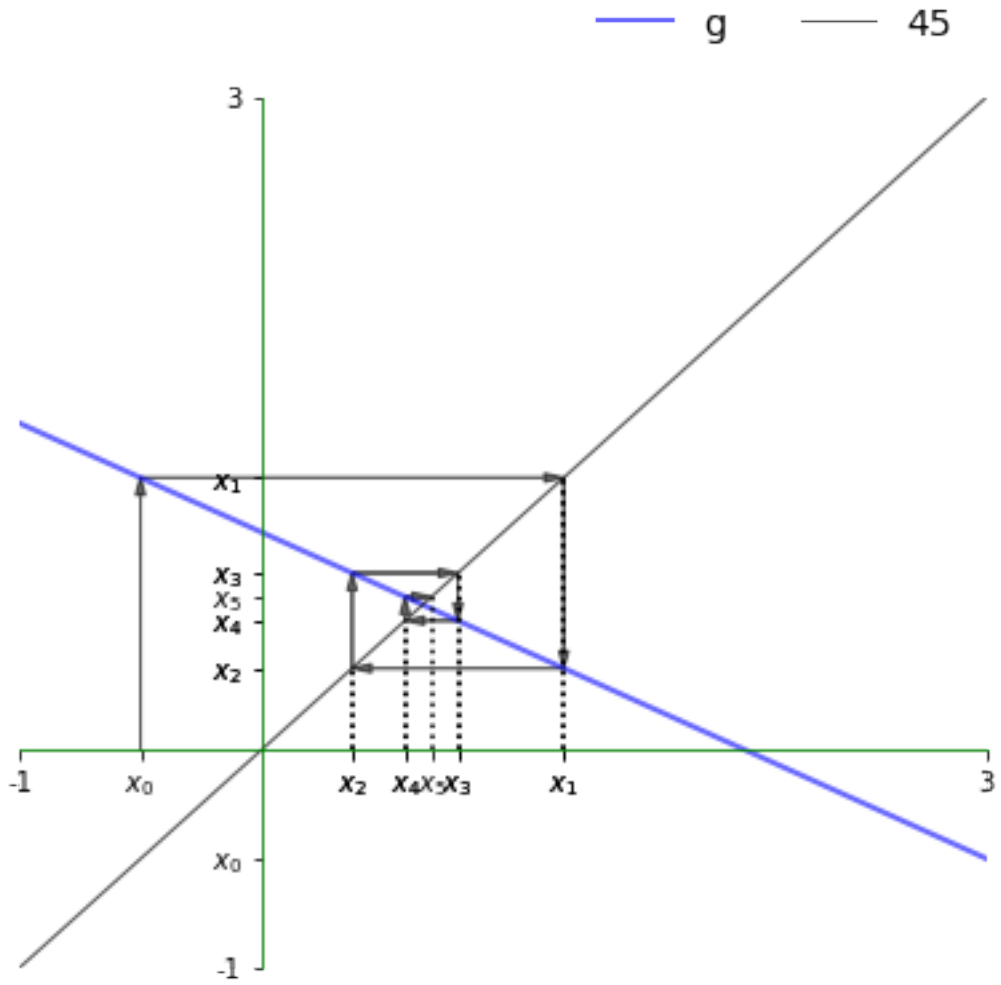
Now let's try $a = -0.5$ and see what differences we observe.

Let's set up the model and plotting region:

```
a, b = -0.5, 1
xmin, xmax = -1, 3
g = lambda x: a * x + b
```

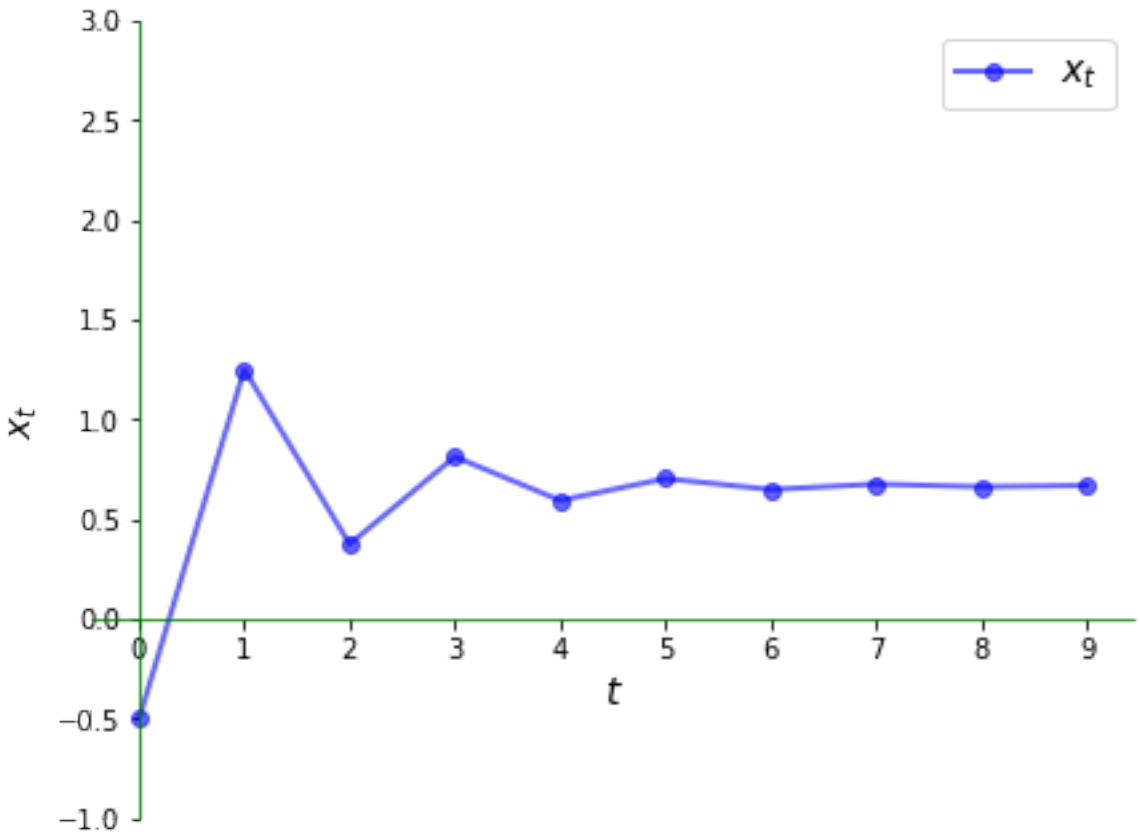
Now let's plot a trajectory:

```
x0 = -0.5
plot45(g, xmin, xmax, x0, num_arrows=5)
```



Here is the corresponding time series, which converges towards the steady state.

```
ts_plot(g, xmin, xmax, x0, ts_length=10)
```



Once again, we have convergence to the steady state but the nature of convergence differs.

In particular, the time series jumps from above the steady state to below it and back again.

In the current context, the series is said to exhibit **damped oscillations**.

CHAPTER

SIXTEEN

AR1 PROCESSES

Contents

16.1 Overview

In this lecture we are going to study a very simple class of stochastic models called AR(1) processes.

These simple models are used again and again in economic research to represent the dynamics of series such as

- labor income
- dividends
- productivity, etc.

AR(1) processes can take negative values but are easily converted into positive processes when necessary by a transformation such as exponentiation.

We are going to study AR(1) processes partly because they are useful and partly because they help us understand important concepts.

Let's start with some imports:

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

16.2 The AR(1) Model

The **AR(1) model** (autoregressive model of order 1) takes the form

$$X_{t+1} = aX_t + b + cW_{t+1} \tag{1}$$

where a, b, c are scalar-valued parameters.

This law of motion generates a time series $\{X_t\}$ as soon as we specify an initial condition X_0.

This is called the **state process** and the state space is $\mathbb{R}$.

To make things even simpler, we will assume that

- the process $\{W_t\}$ is IID and standard normal,
- the initial condition X_0 is drawn from the normal distribution $N(\mu_0, v_0)$ and
- the initial condition X_0 is independent of $\{W_t\}$.

16.2.1 Moving Average Representation

Iterating backwards from time t, we obtain

$$X_t = aX_{t-1} + b + cW_t = a^2 X_{t-2} + ab + acW_{t-1} + b + cW_t = \cdots$$

If we work all the way back to time zero, we get

$$X_t = a^t X_0 + b \sum_{j=0}^{t-1} a^j + c \sum_{j=0}^{t-1} a^j W_{t-j} \tag{2}$$

Equation (2) shows that X_t is a well defined random variable, the value of which depends on

- the parameters,
- the initial condition X_0 and
- the shocks $W_1, \ldots W_t$ from time $t = 1$ to the present.

Throughout, the symbol ψ_t will be used to refer to the density of this random variable X_t.

16.2.2 Distribution Dynamics

One of the nice things about this model is that it's so easy to trace out the sequence of distributions $\{\psi_t\}$ corresponding to the time series $\{X_t\}$.

To see this, we first note that X_t is normally distributed for each t.

This is immediate from (2), since linear combinations of independent normal random variables are normal.

Given that X_t is normally distributed, we will know the full distribution ψ_t if we can pin down its first two moments.

Let μ_t and v_t denote the mean and variance of X_t respectively.

We can pin down these values from (2) or we can use the following recursive expressions:

$$\mu_{t+1} = a\mu_t + b \quad \text{and} \quad v_{t+1} = a^2 v_t + c^2 \tag{3}$$

These expressions are obtained from (1) by taking, respectively, the expectation and variance of both sides of the equality.

In calculating the second expression, we are using the fact that X_t and W_{t+1} are independent.

(This follows from our assumptions and (2).)

Given the dynamics in (2) and initial conditions μ_0, v_0, we obtain μ_t, v_t and hence

$$\psi_t = N(\mu_t, v_t)$$

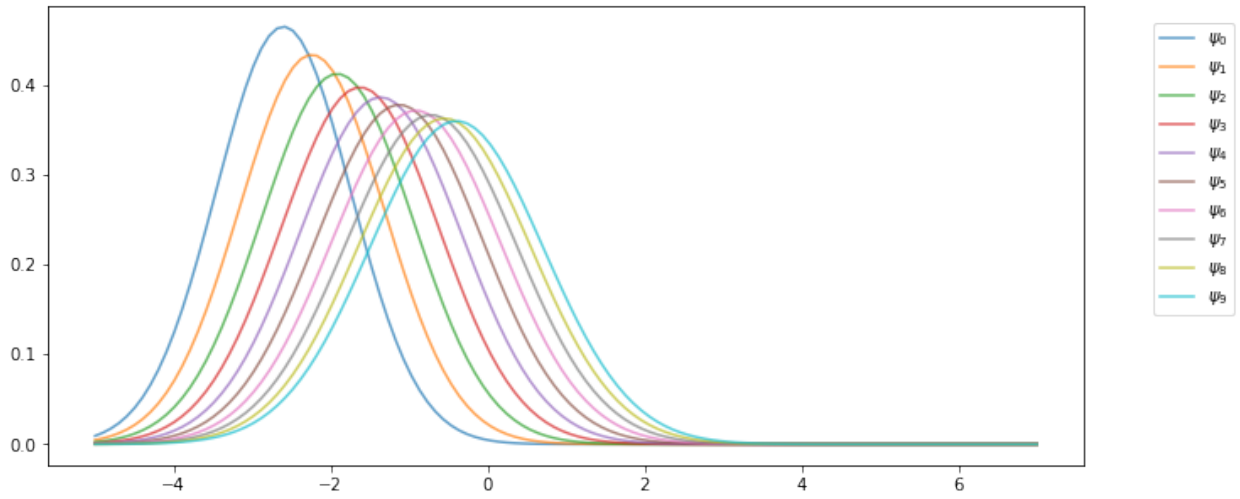
The following code uses these facts to track the sequence of marginal distributions $\{\psi_t\}$.

The parameters are

```
a, b, c = 0.9, 0.1, 0.5

mu, v = -3.0, 0.6  # initial conditions mu_0, v_0
```

Here's the sequence of distributions:

```
from scipy.stats import norm

sim_length = 10
grid = np.linspace(-5, 7, 120)

fig, ax = plt.subplots()

for t in range(sim_length):
    mu = a * mu + b
    v = a**2 * v + c**2
    ax.plot(grid, norm.pdf(grid, loc=mu, scale=np.sqrt(v)),
            label=f"$\psi_{t}$",
            alpha=0.7)

ax.legend(bbox_to_anchor=[1.05,1],loc=2,borderaxespad=1)

plt.show()
```

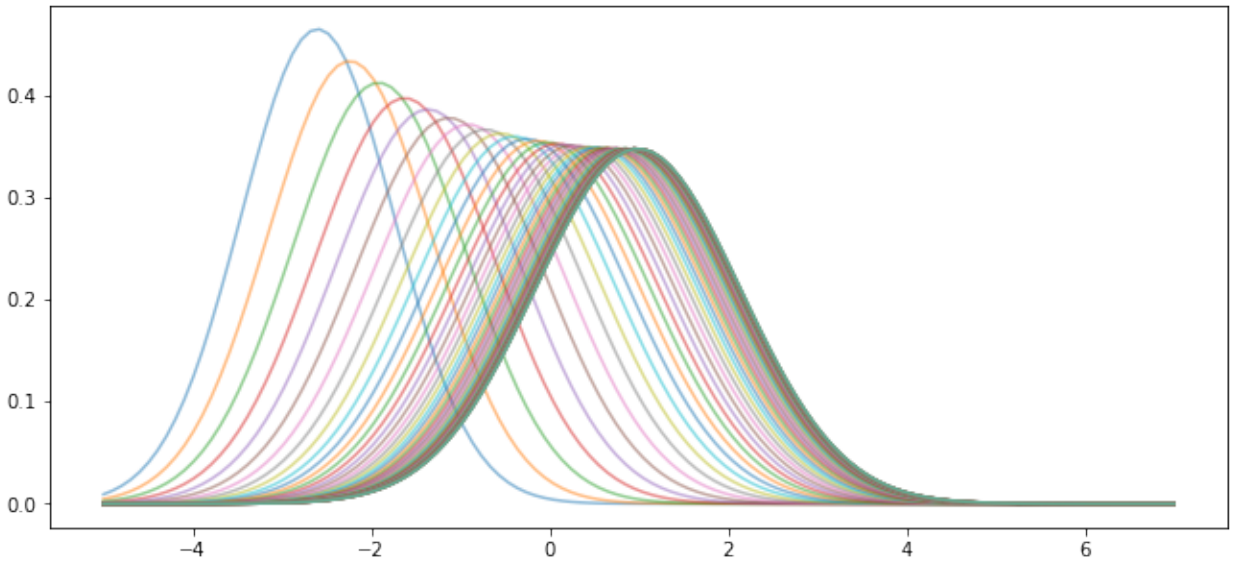


16.3 Stationarity and Asymptotic Stability

Notice that, in the figure above, the sequence $\{\psi_t\}$ seems to be converging to a limiting distribution.

This is even clearer if we project forward further into the future:

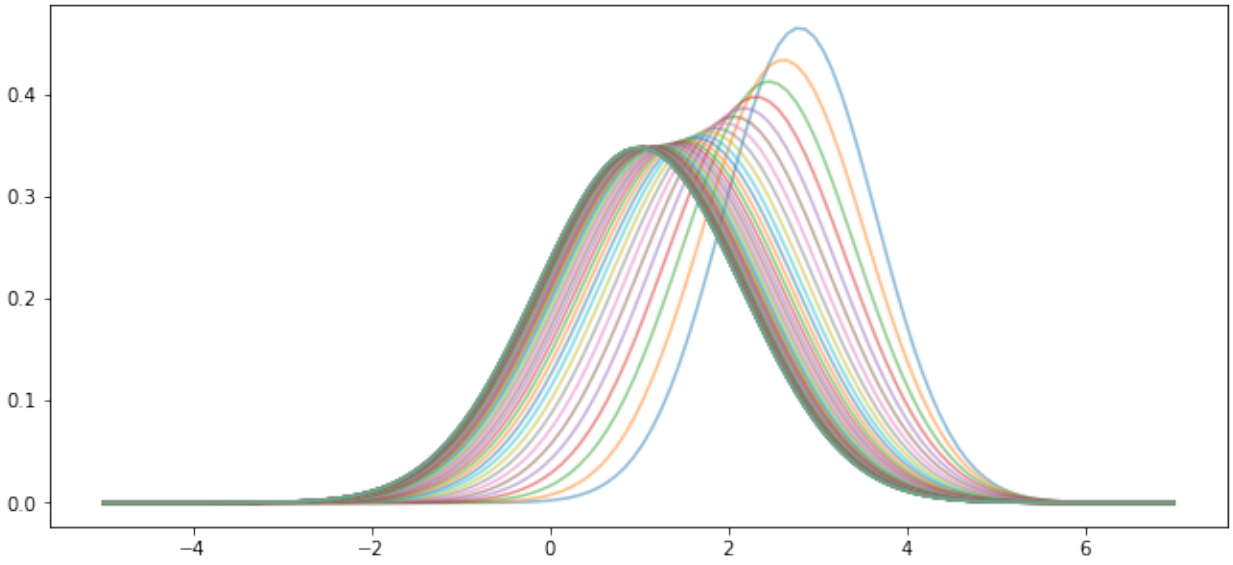
```
def plot_density_seq(ax, mu_0=-3.0, v_0=0.6, sim_length=60):
    mu, v = mu_0, v_0
    for t in range(sim_length):
        mu = a * mu + b
        v = a**2 * v + c**2
        ax.plot(grid,
                norm.pdf(grid, loc=mu, scale=np.sqrt(v)),
                alpha=0.5)

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



Moreover, the limit does not depend on the initial condition.

For example, this alternative density sequence also converges to the same limit.

```
fig, ax = plt.subplots()
plot_density_seq(ax, mu_0=3.0)
plt.show()
```



In fact it's easy to show that such convergence will occur, regardless of the initial condition, whenever $|a| < 1$.

To see this, we just have to look at the dynamics of the first two moments, as given in (3).

When $|a| < 1$, these sequences converge to the respective limits

$$\mu^* := \frac{b}{1-a} \quad \text{and} \quad v^* = \frac{c^2}{1-a^2} \tag{4}$$

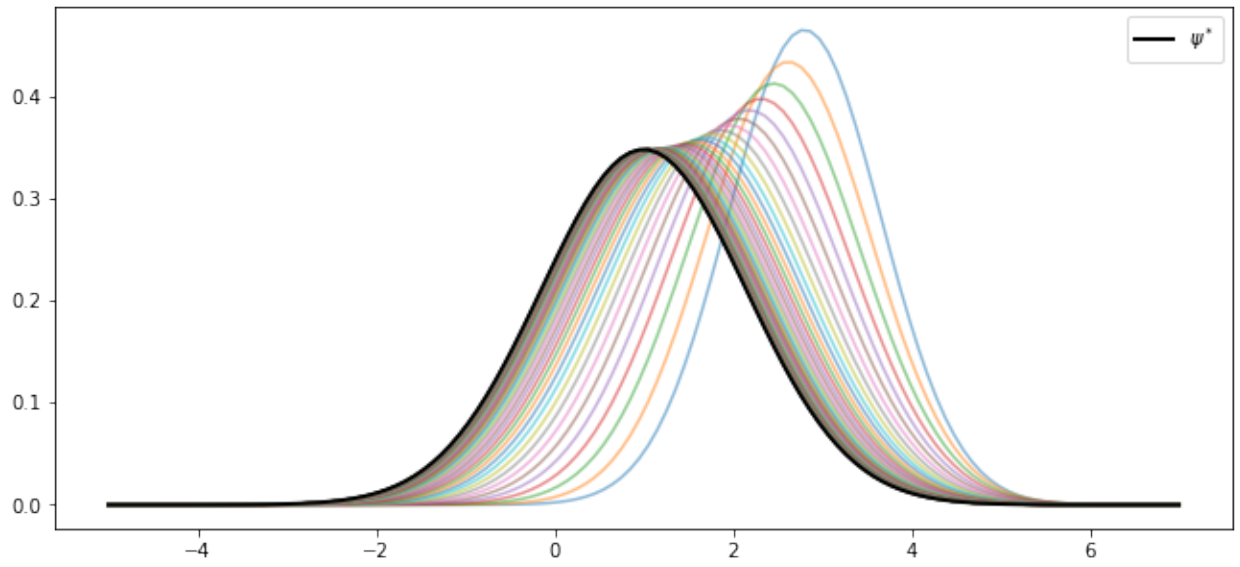
(See our *lecture on one dimensional dynamics* for background on deterministic convergence.)

Hence

$$\psi_t \to \psi^* = N(\mu^*, v^*) \quad \text{as } t \to \infty \tag{5}$$

We can confirm this is valid for the sequence above using the following code.

```
fig, ax = plt.subplots()
plot_density_seq(ax, mu_0=3.0)

mu_star = b / (1 - a)
std_star = np.sqrt(c**2 / (1 - a**2))  # square root of v_star
psi_star = norm.pdf(grid, loc=mu_star, scale=std_star)
ax.plot(grid, psi_star, 'k-', lw=2, label="$\psi^*$")
ax.legend()

plt.show()
```



As claimed, the sequence $\{\psi_t\}$ converges to ψ^*.

16.3.1 Stationary Distributions

A stationary distribution is a distribution that is a fixed point of the update rule for distributions.

In other words, if ψ_t is stationary, then $\psi_{t+j} = \psi_t$ for all j in $\mathbb{N}$.

A different way to put this, specialized to the current setting, is as follows: a density ψ on $\mathbb{R}$ is **stationary** for the AR(1) process if

$$X_t \sim \psi \quad \Longrightarrow \quad aX_t + b + cW_{t+1} \sim \psi$$

The distribution ψ^* in (5) has this property — checking this is an exercise.

(Of course, we are assuming that $|a| < 1$ so that ψ^* is well defined.)

In fact, it can be shown that no other distribution on $\mathbb{R}$ has this property.

Thus, when $|a| < 1$, the AR(1) model has exactly one stationary density and that density is given by ψ^*.

16.4 Ergodicity

The concept of ergodicity is used in different ways by different authors.

One way to understand it in the present setting is that a version of the Law of Large Numbers is valid for $\{X_t\}$, even though it is not IID.

In particular, averages over time series converge to expectations under the stationary distribution.

Indeed, it can be proved that, whenever $|a| < 1$, we have

$$\frac{1}{m}\sum_{t=1}^{m} h(X_t) \to \int h(x)\psi^*(x)dx \quad \text{as } m \to \infty \tag{6}$$

whenever the integral on the right hand side is finite and well defined.

Notes:

- In (6), convergence holds with probability one.
- The textbook by [MT09] is a classic reference on ergodicity.

For example, if we consider the identity function $h(x) = x$, we get

$$\frac{1}{m}\sum_{t=1}^{m} X_t \to \int x\psi^*(x)dx \quad \text{as } m \to \infty$$

In other words, the time series sample mean converges to the mean of the stationary distribution.

As will become clear over the next few lectures, ergodicity is a very important concept for statistics and simulation.

16.5 Exercises

16.5.1 Exercise 1

Let k be a natural number.

The k-th central moment of a random variable is defined as

$$M_k := \mathbb{E}[(X - \mathbb{E}X)^k]$$

When that random variable is $N(\mu, \sigma^2)$, it is known that

$$M_k = \begin{cases} 0 & \text{if } k \text{ is odd} \\ \sigma^k (k-1)!! & \text{if } k \text{ is even} \end{cases}$$

Here $n!!$ is the double factorial.

According to (6), we should have, for any $k \in \mathbb{N}$,

$$\frac{1}{m}\sum_{t=1}^{m}(X_t - \mu^*)^k \approx M_k$$

when m is large.

Confirm this by simulation at a range of k using the default parameters from the lecture.

16.5.2 Exercise 2

Write your own version of a one dimensional kernel density estimator, which estimates a density from a sample.

Write it as a class that takes the data X and bandwidth h when initialized and provides a method f such that

$$f(x) = \frac{1}{hn}\sum_{i=1}^{n} K\left(\frac{x - X_i}{h}\right)$$

For K use the Gaussian kernel (K is the standard normal density).

Write the class so that the bandwidth defaults to Silverman's rule (see the "rule of thumb" discussion on this page). Test the class you have written by going through the steps

1. simulate data $X_1, \ldots, X_n$ from distribution ϕ
2. plot the kernel density estimate over a suitable range

3. plot the density of ϕ on the same figure

for distributions ϕ of the following types

- beta distribution with $\alpha = \beta = 2$
- beta distribution with $\alpha = 2$ and $\beta = 5$
- beta distribution with $\alpha = \beta = 0.5$

Use $n = 500$.

Make a comment on your results. (Do you think this is a good estimator of these distributions?)

16.5.3 Exercise 3

In the lecture we discussed the following fact: for the $AR(1)$ process

$$X_{t+1} = aX_t + b + cW_{t+1}$$

with $\{W_t\}$ iid and standard normal,

$$\psi_t = N(\mu, s^2) \implies \psi_{t+1} = N(a\mu + b, a^2 s^2 + c^2)$$

Confirm this, at least approximately, by simulation. Let

- $a = 0.9$
- $b = 0.0$
- $c = 0.1$
- $\mu = -3$
- $s = 0.2$

First, plot ψ_t and ψ_{t+1} using the true distributions described above.

Second, plot ψ_{t+1} on the same figure (in a different color) as follows:

1. Generate n draws of X_t from the $N(\mu, s^2)$ distribution
2. Update them all using the rule $X_{t+1} = aX_t + b + cW_{t+1}$
3. Use the resulting sample of X_{t+1} values to produce a density estimate via kernel density estimation.

Try this for $n = 2000$ and confirm that the simulation based estimate of ψ_{t+1} does converge to the theoretical distribution.

16.6 Solutions

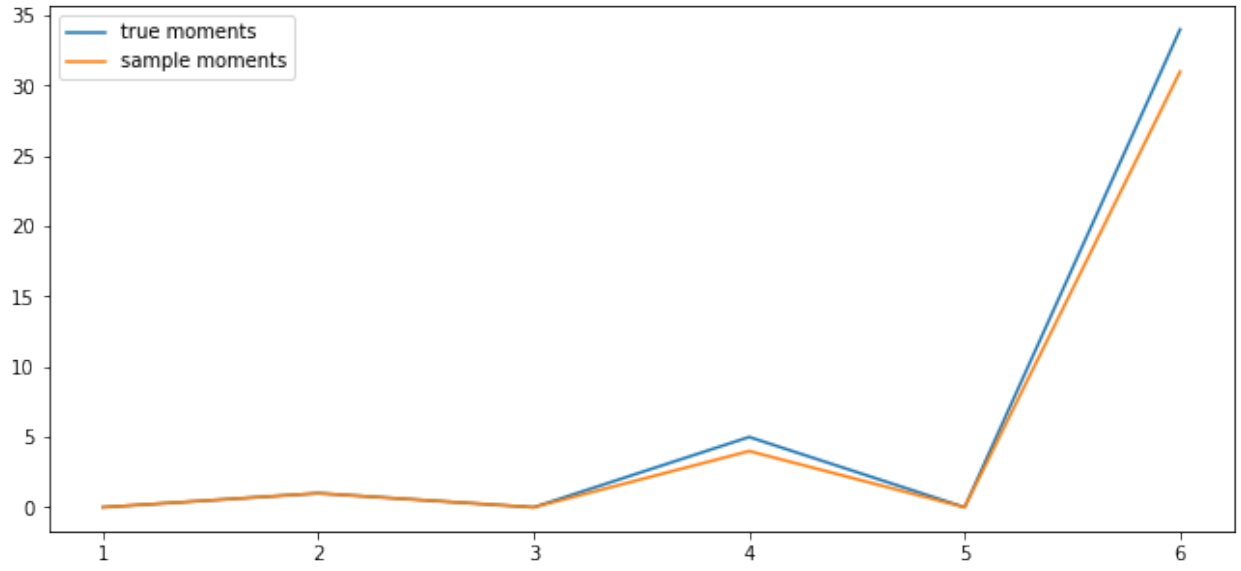
16.6.1 Exercise 1

```
from numba import njit
from scipy.special import factorial2

@njit
def sample_moments_ar1(k, m=100_000, mu_0=0.0, sigma_0=1.0, seed=1234):
    np.random.seed(seed)
    sample_sum = 0.0
```

(continues on next page)

(continued from previous page)

```
    x = mu_0 + sigma_0 * np.random.randn()
    for t in range(m):
        sample_sum += (x - mu_star)**k
        x = a * x + b + c * np.random.randn()
    return sample_sum / m

def true_moments_ar1(k):
    if k % 2 == 0:
        return std_star**k * factorial2(k - 1)
    else:
        return 0

k_vals = np.arange(6) + 1
sample_moments = np.empty_like(k_vals)
true_moments = np.empty_like(k_vals)

for k_idx, k in enumerate(k_vals):
    sample_moments[k_idx] = sample_moments_ar1(k)
    true_moments[k_idx] = true_moments_ar1(k)

fig, ax = plt.subplots()
ax.plot(k_vals, true_moments, label="true moments")
ax.plot(k_vals, sample_moments, label="sample moments")
ax.legend()

plt.show()
```



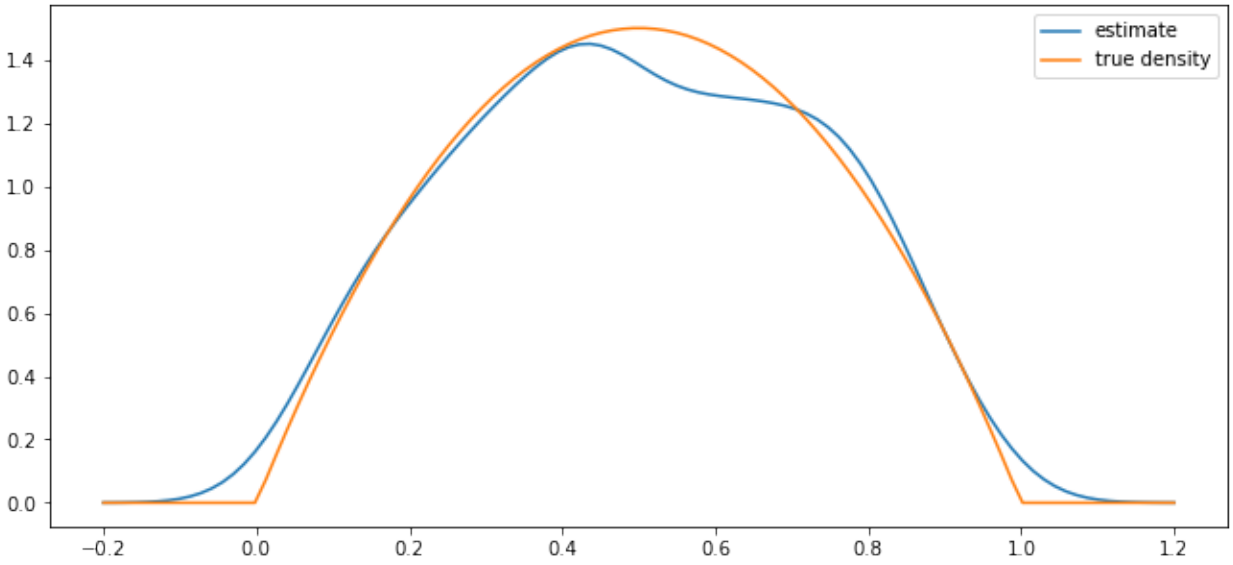
16.6.2 Exercise 2

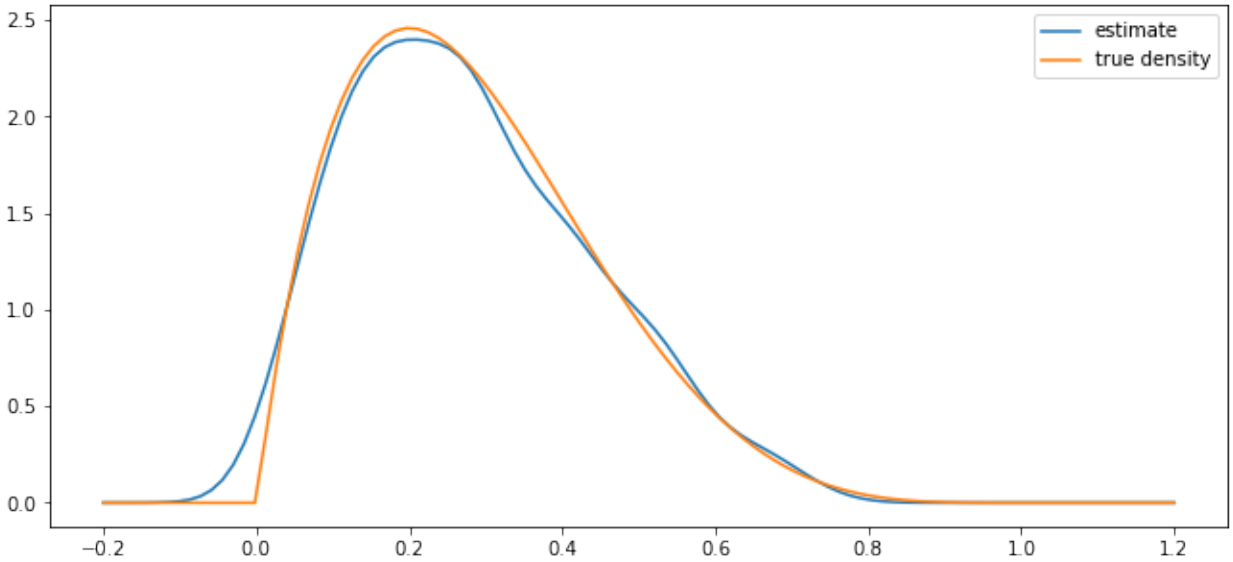
Here is one solution:

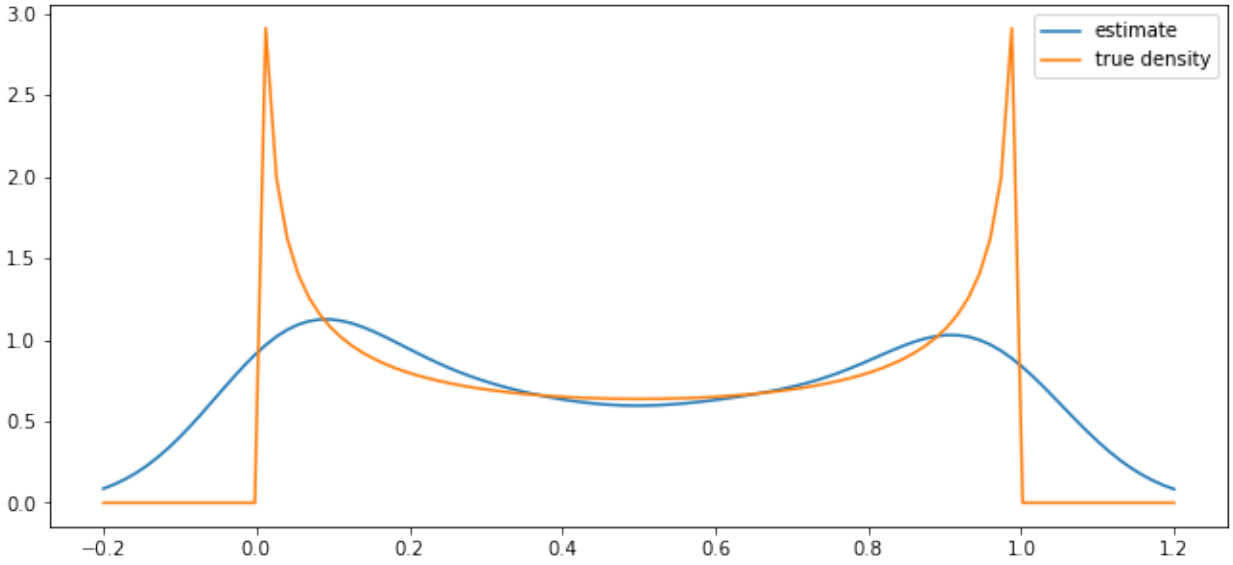
```
K = norm.pdf

class KDE:

    def __init__(self, x_data, h=None):

        if h is None:
            c = x_data.std()
            n = len(x_data)
            h = 1.06 * c * n**(-1/5)
        self.h = h
        self.x_data = x_data

    def f(self, x):
        if np.isscalar(x):
            return K((x - self.x_data) / self.h).mean() * (1/self.h)
        else:
            y = np.empty_like(x)
            for i, x_val in enumerate(x):
                y[i] = K((x_val - self.x_data) / self.h).mean() * (1/self.h)
            return y
```

```
def plot_kde(ϕ, x_min=-0.2, x_max=1.2):
    x_data = ϕ.rvs(n)
    kde = KDE(x_data)

    x_grid = np.linspace(-0.2, 1.2, 100)
    fig, ax = plt.subplots()
    ax.plot(x_grid, kde.f(x_grid), label="estimate")
    ax.plot(x_grid, ϕ.pdf(x_grid), label="true density")
    ax.legend()
    plt.show()
```

```
from scipy.stats import beta

n = 500
parameter_pairs= (2, 2), (2, 5), (0.5, 0.5)
for α, β in parameter_pairs:
    plot_kde(beta(α, β))
```


estimate
true density
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
0.0
0.2
0.4
0.6
0.8
1.0
1.2


estimate
true density
2.5
2.0
1.5
1.0
0.5
0.0
−0.2
0.0
0.2
0.4
0.6
0.8
1.0
1.2



We see that the kernel density estimator is effective when the underlying distribution is smooth but less so otherwise.

16.6.3 Exercise 3

Here is our solution

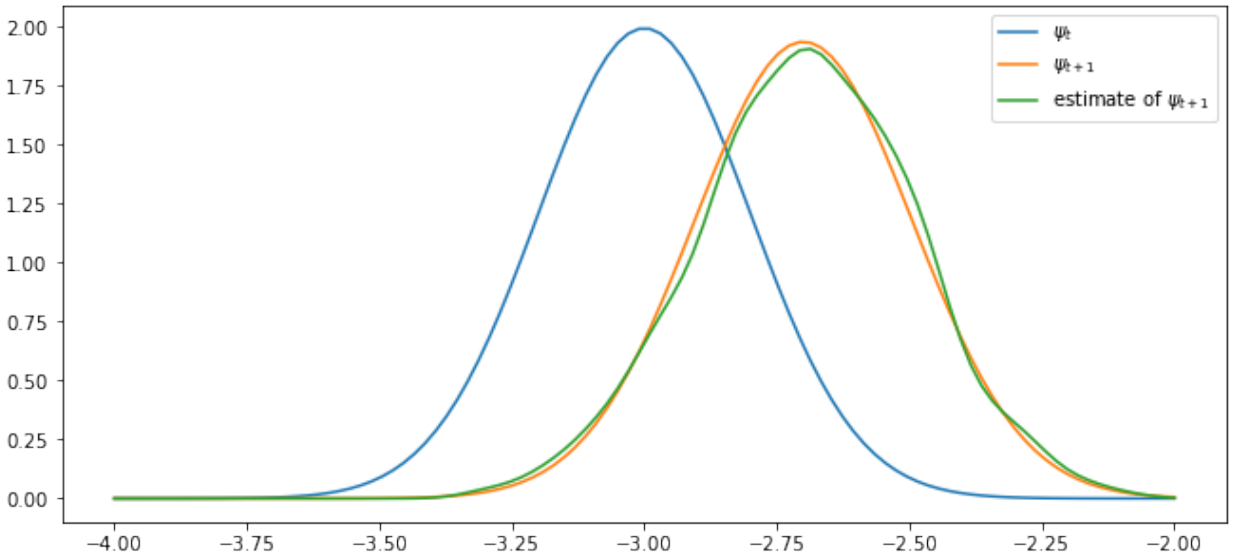
```
a = 0.9
b = 0.0
c = 0.1
μ = -3
s = 0.2
```

```
μ_next = a * μ + b
s_next = np.sqrt(a**2 * s**2 + c**2)
```

```
ψ = lambda x: K((x - μ) / s)
ψ_next = lambda x: K((x - μ_next) / s_next)
```

```
ψ = norm(μ, s)
ψ_next = norm(μ_next, s_next)
```

```
n = 2000
x_draws = ψ.rvs(n)
x_draws_next = a * x_draws + b + c * np.random.randn(n)
kde = KDE(x_draws_next)

x_grid = np.linspace(μ - 1, μ + 1, 100)
fig, ax = plt.subplots()

ax.plot(x_grid, ψ.pdf(x_grid), label="$\psi_t$")
ax.plot(x_grid, ψ_next.pdf(x_grid), label="$\psi_{t+1}$")
ax.plot(x_grid, kde.f(x_grid), label="estimate of $\psi_{t+1}$")

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



The simulated distribution approximately coincides with the theoretical distribution, as predicted.

CHAPTER

SEVENTEEN

FINITE MARKOV CHAINS

Contents

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

```
!conda install -y quantecon
```

17.1 Overview

Markov chains are one of the most useful classes of stochastic processes, being

- simple, flexible and supported by many elegant theoretical results
- valuable for building intuition about random dynamic models
- central to quantitative modeling in their own right

You will find them in many of the workhorse models of economics and finance.

In this lecture, we review some of the theory of Markov chains.

We will also introduce some of the high-quality routines for working with Markov chains available in QuantEcon.py.

Prerequisite knowledge is basic probability and linear algebra.

Let's start with some standard imports:

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

17.2 Definitions

The following concepts are fundamental.

17.2.1 Stochastic Matrices

A **stochastic matrix** (or **Markov matrix**) is an $n \times n$ square matrix P such that

1. each element of P is nonnegative, and
2. each row of P sums to one

Each row of P can be regarded as a probability mass function over n possible outcomes.

It is too not difficult to check[1] that if P is a stochastic matrix, then so is the k-th power P^k for all $k \in \mathbb{N}$.

17.2.2 Markov Chains

There is a close connection between stochastic matrices and Markov chains.

To begin, let S be a finite set with n elements $\{x_1, \ldots, x_n\}$.

The set S is called the **state space** and $x_1, \ldots, x_n$ are the **state values**.

A **Markov chain** $\{X_t\}$ on S is a sequence of random variables on S that have the **Markov property**.

This means that, for any date t and any state $y \in S$,

$$\mathbb{P}\{X_{t+1} = y \,|\, X_t\} = \mathbb{P}\{X_{t+1} = y \,|\, X_t, X_{t-1}, \ldots\} \tag{1}$$

In other words, knowing the current state is enough to know probabilities for future states.

In particular, the dynamics of a Markov chain are fully determined by the set of values

$$P(x, y) := \mathbb{P}\{X_{t+1} = y \,|\, X_t = x\} \qquad (x, y \in S) \tag{2}$$

By construction,

- $P(x, y)$ is the probability of going from x to y in one unit of time (one step)
- $P(x, \cdot)$ is the conditional distribution of X_{t+1} given $X_t = x$

We can view P as a stochastic matrix where

$$P_{ij} = P(x_i, x_j) \qquad 1 \leq i, j \leq n$$

Going the other way, if we take a stochastic matrix P, we can generate a Markov chain $\{X_t\}$ as follows:

[1] Hint: First show that if P and Q are stochastic matrices then so is their product — to check the row sums, try post multiplying by a column vector of ones. Finally, argue that P^n is a stochastic matrix using induction.

- draw X_0 from some specified distribution
- for each $t = 0, 1, ...$, draw X_{t+1} from $P(X_t, \cdot)$

By construction, the resulting process satisfies (2).

17.2.3 Example 1

Consider a worker who, at any given time t, is either unemployed (state 0) or employed (state 1).

Suppose that, over a one month period,

1. An unemployed worker finds a job with probability $\alpha \in (0, 1)$.
2. An employed worker loses her job and becomes unemployed with probability $\beta \in (0, 1)$.

In terms of a Markov model, we have

- $S = \{0, 1\}$
- $P(0, 1) = \alpha$ and $P(1, 0) = \beta$

We can write out the transition probabilities in matrix form as

$$P = \begin{pmatrix} 1 - \alpha & \alpha \\ \beta & 1 - \beta \end{pmatrix} \tag{3}$$

Once we have the values α and β, we can address a range of questions, such as

- What is the average duration of unemployment?
- Over the long-run, what fraction of time does a worker find herself unemployed?
- Conditional on employment, what is the probability of becoming unemployed at least once over the next 12 months?

We'll cover such applications below.

17.2.4 Example 2

Using US unemployment data, Hamilton [Ham05] estimated the stochastic matrix

$$P = \begin{pmatrix} 0.971 & 0.029 & 0 \\ 0.145 & 0.778 & 0.077 \\ 0 & 0.508 & 0.492 \end{pmatrix}$$

where

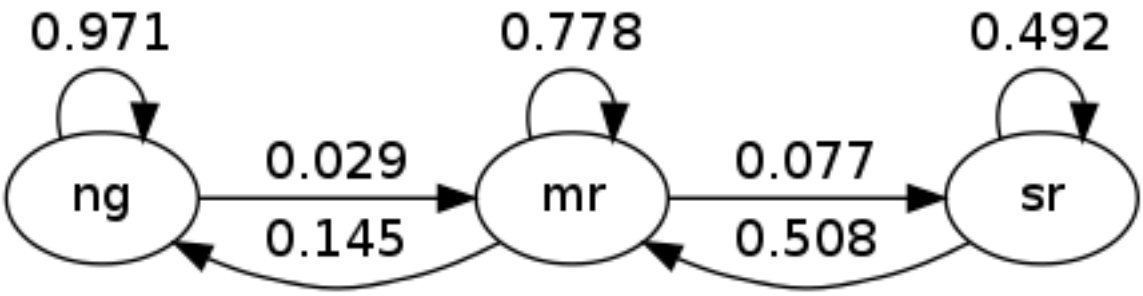
- the frequency is monthly
- the first state represents "normal growth"
- the second state represents "mild recession"
- the third state represents "severe recession"

For example, the matrix tells us that when the state is normal growth, the state will again be normal growth next month with probability 0.97.

In general, large values on the main diagonal indicate persistence in the process $\{X_t\}$.

This Markov process can also be represented as a directed graph, with edges labeled by transition probabilities

Here "ng" is normal growth, "mr" is mild recession, etc.



17.3 Simulation

One natural way to answer questions about Markov chains is to simulate them.

(To approximate the probability of event E, we can simulate many times and count the fraction of times that E occurs).

Nice functionality for simulating Markov chains exists in QuantEcon.py.

- Efficient, bundled with lots of other useful routines for handling Markov chains.

However, it's also a good exercise to roll our own routines — let's do that first and then come back to the methods in QuantEcon.py.

In these exercises, we'll take the state space to be $S = 0, \ldots, n-1$.

17.3.1 Rolling Our Own

To simulate a Markov chain, we need its stochastic matrix P and a probability distribution ψ for the initial state to be drawn from.

The Markov chain is then constructed as discussed above. To repeat:

1. At time $t = 0$, the X_0 is chosen from ψ.
2. At each subsequent time t, the new state X_{t+1} is drawn from $P(X_t, \cdot)$.

To implement this simulation procedure, we need a method for generating draws from a discrete distribution.

For this task, we'll use `random.draw` from QuantEcon, which works as follows:

```
ψ = (0.3, 0.7)           # probabilities over {0, 1}
cdf = np.cumsum(ψ)       # convert into cummulative distribution
qe.random.draw(cdf, 5)   # generate 5 independent draws from ψ
```

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

We'll write our code as a function that takes the following three arguments

- A stochastic matrix `P`
- An initial state `init`
- A positive integer `sample_size` representing the length of the time series the function should return

```
def mc_sample_path(P, ψ_0=None, sample_size=1_000):

    # set up
```

(continues on next page)

(continued from previous page)

```
    P = np.asarray(P)
    X = np.empty(sample_size, dtype=int)

    # Convert each row of P into a cdf
    n = len(P)
    P_dist = [np.cumsum(P[i, :]) for i in range(n)]

    # draw initial state, defaulting to 0
    if ψ_0 is not None:
        X_0 = qe.random.draw(np.cumsum(ψ_0))
    else:
        X_0 = 0

    # simulate
    X[0] = X_0
    for t in range(sample_size - 1):
        X[t+1] = qe.random.draw(P_dist[X[t]])

    return X
```

Let's see how it works using the small matrix

```
P = [[0.4, 0.6],
     [0.2, 0.8]]
```

As we'll see later, for a long series drawn from `P`, the fraction of the sample that takes value 0 will be about 0.25.

Moreover, this is true, regardless of the initial distribution from with X_0 is drawn.

The following code illustrates this

```
X = mc_sample_path(P, ψ_0=[0.1, 0.9], sample_size=100_000)
np.mean(X == 0)
```

```
0.25007
```

You can try changing the initial distribution to confirm that the output is always close to 0.25.

17.3.2 Using QuantEcon's Routines

As discussed above, QuantEcon.py has routines for handling Markov chains, including simulation.

Here's an illustration using the same P as the preceding example

```
from quantecon import MarkovChain

mc = qe.MarkovChain(P)
X = mc.simulate(ts_length=1_000_000)
np.mean(X == 0)
```

```
0.248611
```

The QuantEcon.py routine is JIT compiled and much faster.

```
%time mc_sample_path(P, sample_size=1_000_000) # Our version
```

```
CPU times: user 912 ms, sys: 0 ns, total: 912 ms
Wall time: 911 ms
```

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

```
%time mc.simulate(ts_length=1_000_000) # qe version
```

```
CPU times: user 37.1 ms, sys: 70 µs, total: 37.2 ms
Wall time: 36.9 ms
```

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

Adding State Values and Initial Conditions

If we wish to, we can provide a specification of state values to `MarkovChain`.

These state values can be integers, floats, or even strings.

The following code illustrates

```
mc = qe.MarkovChain(P, state_values=('unemployed', 'employed'))
mc.simulate(ts_length=4, init='employed')
```

```
array(['employed', 'unemployed', 'unemployed', 'employed'], dtype='<U10')
```

```
mc.simulate(ts_length=4, init='unemployed')
```

```
array(['unemployed', 'unemployed', 'employed', 'employed'], dtype='<U10')
```

```
mc.simulate(ts_length=4)  # Start at randomly chosen initial state
```

```
array(['employed', 'employed', 'unemployed', 'employed'], dtype='<U10')
```

If we want to simulate with output as indices rather than state values we can use

```
mc.simulate_indices(ts_length=4)
```

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

17.4 Marginal Distributions

Suppose that

1. $\{X_t\}$ is a Markov chain with stochastic matrix P
2. the distribution of X_t is known to be ψ_t

What then is the distribution of X_{t+1}, or, more generally, of X_{t+m}?

To answer this, we let ψ_t be the distribution of X_t for $t = 0, 1, 2, ...$

Our first aim is to find ψ_{t+1} given ψ_t and P.

To begin, pick any $y \in S$.

Using the law of total probability, we can decompose the probability that $X_{t+1} = y$ as follows:

$$\mathbb{P}\{X_{t+1} = y\} = \sum_{x \in S} \mathbb{P}\{X_{t+1} = y \,|\, X_t = x\} \cdot \mathbb{P}\{X_t = x\}$$

In words, to get the probability of being at y tomorrow, we account for all ways this can happen and sum their probabilities.

Rewriting this statement in terms of marginal and conditional probabilities gives

$$\psi_{t+1}(y) = \sum_{x \in S} P(x, y)\psi_t(x)$$

There are n such equations, one for each $y \in S$.

If we think of ψ_{t+1} and ψ_t as *row vectors* (as is traditional in this literature), these n equations are summarized by the matrix expression

$$\psi_{t+1} = \psi_t P \tag{4}$$

In other words, to move the distribution forward one unit of time, we postmultiply by P.

By repeating this m times we move forward m steps into the future.

Hence, iterating on (4), the expression $\psi_{t+m} = \psi_t P^m$ is also valid — here P^m is the m-th power of P.

As a special case, we see that if ψ_0 is the initial distribution from which X_0 is drawn, then $\psi_0 P^m$ is the distribution of X_m.

This is very important, so let's repeat it

$$X_0 \sim \psi_0 \quad \Longrightarrow \quad X_m \sim \psi_0 P^m \tag{5}$$

and, more generally,

$$X_t \sim \psi_t \quad \Longrightarrow \quad X_{t+m} \sim \psi_t P^m \tag{6}$$

17.4.1 Multiple Step Transition Probabilities

We know that the probability of transitioning from x to y in one step is $P(x, y)$.

It turns out that the probability of transitioning from x to y in m steps is $P^m(x, y)$, the (x, y)-th element of the m-th power of P.

To see why, consider again (6), but now with ψ_t putting all probability on state x

- 1 in the x-th position and zero elsewhere

Inserting this into (6), we see that, conditional on $X_t = x$, the distribution of X_{t+m} is the x-th row of P^m.

In particular

$$\mathbb{P}\{X_{t+m} = y \,|\, X_t = x\} = P^m(x, y) = (x, y)\text{-th element of } P^m$$

17.4.2 Example: Probability of Recession

Recall the stochastic matrix P for recession and growth *considered above*.

Suppose that the current state is unknown — perhaps statistics are available only at the *end* of the current month.

We estimate the probability that the economy is in state x to be $\psi(x)$.

The probability of being in recession (either mild or severe) in 6 months time is given by the inner product

$$\psi P^6 \cdot \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}$$

17.4.3 Example 2: Cross-Sectional Distributions

The marginal distributions we have been studying can be viewed either as probabilities or as cross-sectional frequencies in large samples.

To illustrate, recall our model of employment/unemployment dynamics for a given worker *discussed above*.

Consider a large population of workers, each of whose lifetime experience is described by the specified dynamics, independent of one another.

Let ψ be the current *cross-sectional* distribution over $\{0, 1\}$.

The cross-sectional distribution records the fractions of workers employed and unemployed at a given moment.

- For example, $\psi(0)$ is the unemployment rate.

What will the cross-sectional distribution be in 10 periods hence?

The answer is ψP^{10}, where P is the stochastic matrix in (3).

This is because each worker is updated according to P, so ψP^{10} represents probabilities for a single randomly selected worker.

But when the sample is large, outcomes and probabilities are roughly equal (by the Law of Large Numbers).

So for a very large (tending to infinite) population, ψP^{10} also represents the fraction of workers in each state.

This is exactly the cross-sectional distribution.

17.5 Irreducibility and Aperiodicity

Irreducibility and aperiodicity are central concepts of modern Markov chain theory.

Let's see what they're about.

17.5.1 Irreducibility

Let P be a fixed stochastic matrix.

Two states x and y are said to **communicate** with each other if there exist positive integers j and k such that

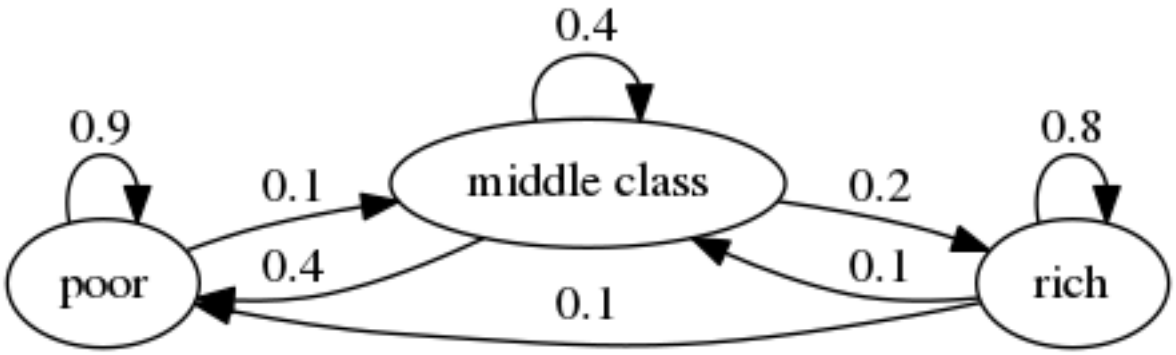
$$P^j(x, y) > 0 \quad \text{and} \quad P^k(y, x) > 0$$

In view of our discussion *above*, this means precisely that

- state x can be reached eventually from state y, and
- state y can be reached eventually from state x

The stochastic matrix P is called **irreducible** if all states communicate; that is, if x and y communicate for all (x, y) in $S \times S$.

For example, consider the following transition probabilities for wealth of a fictitious set of households



We can translate this into a stochastic matrix, putting zeros where there's no edge between nodes

$$P := \begin{pmatrix} 0.9 & 0.1 & 0 \\ 0.4 & 0.4 & 0.2 \\ 0.1 & 0.1 & 0.8 \end{pmatrix}$$

It's clear from the graph that this stochastic matrix is irreducible: we can reach any state from any other state eventually.

We can also test this using QuantEcon.py's MarkovChain class

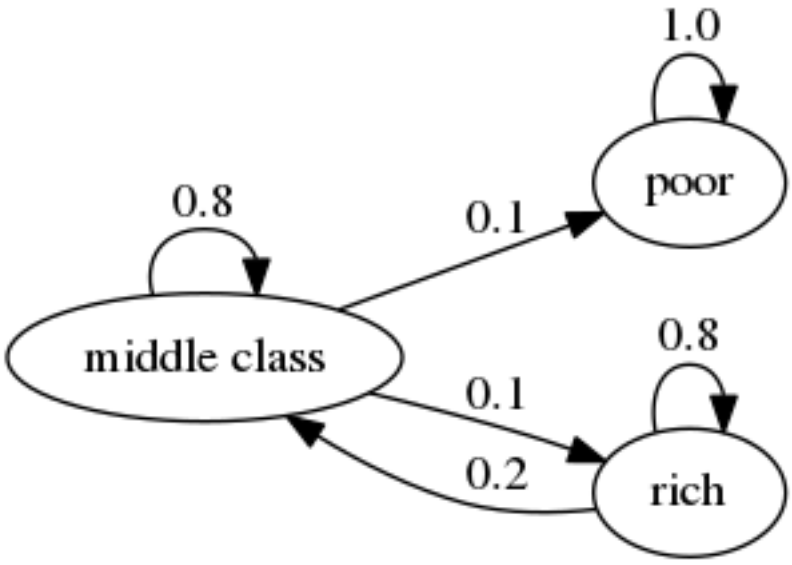
```
P = [[0.9, 0.1, 0.0],
     [0.4, 0.4, 0.2],
     [0.1, 0.1, 0.8]]

mc = qe.MarkovChain(P, ('poor', 'middle', 'rich'))
mc.is_irreducible
```

```
True
```

Here's a more pessimistic scenario, where the poor are poor forever

This stochastic matrix is not irreducible, since, for example, rich is not accessible from poor.

Let's confirm this




```
P = [[1.0, 0.0, 0.0],
     [0.1, 0.8, 0.1],
     [0.0, 0.2, 0.8]]

mc = qe.MarkovChain(P, ('poor', 'middle', 'rich'))
mc.is_irreducible
```

```
False
```

We can also determine the "communication classes"

```
mc.communication_classes
```

```
[array(['poor'], dtype='<U6'), array(['middle', 'rich'], dtype='<U6')]
```

It might be clear to you already that irreducibility is going to be important in terms of long run outcomes.

For example, poverty is a life sentence in the second graph but not the first.

We'll come back to this a bit later.

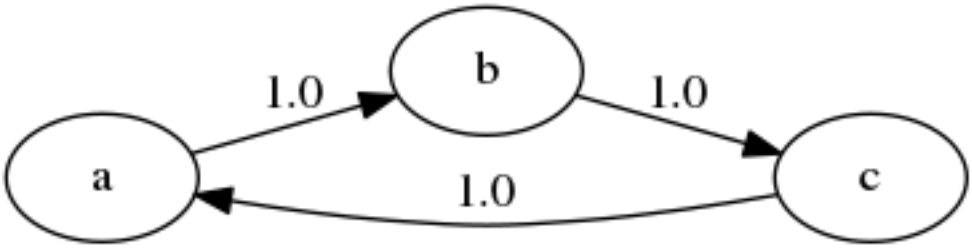
17.5.2 Aperiodicity

Loosely speaking, a Markov chain is called periodic if it cycles in a predictable way, and aperiodic otherwise.

Here's a trivial example with three states

The chain cycles with period 3:

```
P = [[0, 1, 0],
     [0, 0, 1],
     [1, 0, 0]]

mc = qe.MarkovChain(P)
mc.period
```




```
3
```

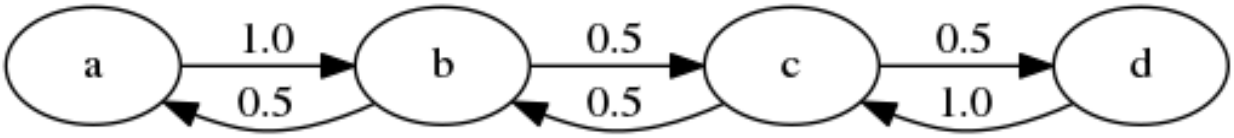
More formally, the **period** of a state x is the greatest common divisor of the set of integers

$$D(x) := \{j \geq 1 : P^j(x, x) > 0\}$$

In the last example, $D(x) = \{3, 6, 9, ...\}$ for every state x, so the period is 3.

A stochastic matrix is called **aperiodic** if the period of every state is 1, and **periodic** otherwise.

For example, the stochastic matrix associated with the transition probabilities below is periodic because, for example, state a has period 2



We can confirm that the stochastic matrix is periodic as follows

```
P = [[0.0, 1.0, 0.0, 0.0],
     [0.5, 0.0, 0.5, 0.0],
     [0.0, 0.5, 0.0, 0.5],
     [0.0, 0.0, 1.0, 0.0]]

mc = qe.MarkovChain(P)
mc.period
```

```
2
```

```
mc.is_aperiodic
```

```
False
```

17.6 Stationary Distributions

As seen in (4), we can shift probabilities forward one unit of time via postmultiplication by P.

Some distributions are invariant under this updating process — for example,

```
P = np.array([[0.4, 0.6],
              [0.2, 0.8]])
ψ = (0.25, 0.75)
ψ @ P
```

```
array([0.25, 0.75])
```

Such distributions are called **stationary**, or **invariant**.

Formally, a distribution ψ^* on S is called **stationary** for P if $\psi^* = \psi^* P$.

(This is the same notion of stationarity that we learned about in the *lecture on AR(1) processes* applied to a different setting.)

From this equality, we immediately get $\psi^* = \psi^* P^t$ for all t.

This tells us an important fact: If the distribution of X_0 is a stationary distribution, then X_t will have this same distribution for all t.

Hence stationary distributions have a natural interpretation as stochastic steady states — we'll discuss this more in just a moment.

Mathematically, a stationary distribution is a fixed point of P when P is thought of as the map $\psi \mapsto \psi P$ from (row) vectors to (row) vectors.

Theorem. Every stochastic matrix P has at least one stationary distribution.

(We are assuming here that the state space S is finite; if not more assumptions are required)

For proof of this result, you can apply Brouwer's fixed point theorem, or see EDTC, theorem 4.3.5.

There may in fact be many stationary distributions corresponding to a given stochastic matrix P.

- For example, if P is the identity matrix, then all distributions are stationary.

Since stationary distributions are long run equilibria, to get uniqueness we require that initial conditions are not infinitely persistent.

Infinite persistence of initial conditions occurs if certain regions of the state space cannot be accessed from other regions, which is the opposite of irreducibility.

This gives some intuition for the following fundamental theorem.

Theorem. If P is both aperiodic and irreducible, then

1. P has exactly one stationary distribution ψ^*.
2. For any initial distribution ψ_0, we have $\|\psi_0 P^t - \psi^*\| \to 0$ as $t \to \infty$.

For a proof, see, for example, theorem 5.2 of [Haggstrom02].

(Note that part 1 of the theorem requires only irreducibility, whereas part 2 requires both irreducibility and aperiodicity)

A stochastic matrix satisfying the conditions of the theorem is sometimes called **uniformly ergodic**.

One easy sufficient condition for aperiodicity and irreducibility is that every element of P is strictly positive.

- Try to convince yourself of this.

17.6.1 Example

Recall our model of employment/unemployment dynamics for a given worker *discussed above*.

Assuming $\alpha \in (0,1)$ and $\beta \in (0,1)$, the uniform ergodicity condition is satisfied.

Let $\psi^* = (p, 1-p)$ be the stationary distribution, so that p corresponds to unemployment (state 0).

Using $\psi^* = \psi^* P$ and a bit of algebra yields

$$p = \frac{\beta}{\alpha + \beta}$$

This is, in some sense, a steady state probability of unemployment — more on interpretation below.

Not surprisingly it tends to zero as $\beta \to 0$, and to one as $\alpha \to 0$.

17.6.2 Calculating Stationary Distributions

As discussed above, a given Markov matrix P can have many stationary distributions.

That is, there can be many row vectors ψ such that $\psi = \psi P$.

In fact if P has two distinct stationary distributions ψ_1, ψ_2 then it has infinitely many, since in this case, as you can verify,

$$\psi_3 := \lambda \psi_1 + (1-\lambda)\psi_2$$

is a stationary distribution for P for any $\lambda \in [0,1]$.

If we restrict attention to the case where only one stationary distribution exists, one option for finding it is to try to solve the linear system $\psi(I_n - P) = 0$ for ψ, where I_n is the $n \times n$ identity.

But the zero vector solves this equation, so we need to proceed carefully.

In essence, we need to impose the restriction that the solution must be a probability distribution.

There are various ways to do this.

One option is to regard this as an eigenvector problem: a vector ψ such that $\psi = \psi P$ is a left eigenvector associated with the unit eigenvalue $\lambda = 1$.

A stable and sophisticated algorithm specialized for stochastic matrices is implemented in QuantEcon.py.

This is the one we recommend you to use:

```
P = [[0.4, 0.6],
     [0.2, 0.8]]

mc = qe.MarkovChain(P)
mc.stationary_distributions  # Show all stationary distributions
```

```
array([[0.25, 0.75]])
```

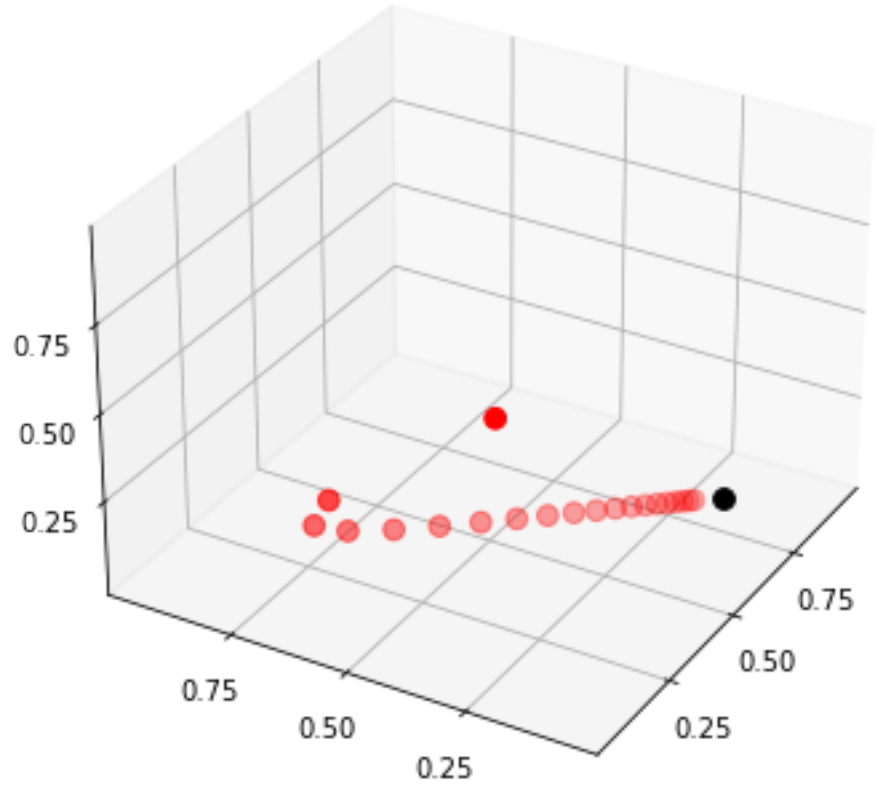
17.6.3 Convergence to Stationarity

Part 2 of the Markov chain convergence theorem *stated above* tells us that the distribution of X_t converges to the stationary distribution regardless of where we start off.

This adds considerable weight to our interpretation of ψ^* as a stochastic steady state.

The convergence in the theorem is illustrated in the next figure

```
P = ((0.971, 0.029, 0.000),
     (0.145, 0.778, 0.077),
     (0.000, 0.508, 0.492))
P = np.array(P)

ψ = (0.0, 0.2, 0.8)         # Initial condition

fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')

ax.set(xlim=(0, 1), ylim=(0, 1), zlim=(0, 1),
       xticks=(0.25, 0.5, 0.75),
       yticks=(0.25, 0.5, 0.75),
       zticks=(0.25, 0.5, 0.75))

x_vals, y_vals, z_vals = [], [], []
for t in range(20):
    x_vals.append(ψ[0])
    y_vals.append(ψ[1])
    z_vals.append(ψ[2])
    ψ = ψ @ P

ax.scatter(x_vals, y_vals, z_vals, c='r', s=60)
ax.view_init(30, 210)

mc = qe.MarkovChain(P)
ψ_star = mc.stationary_distributions[0]
ax.scatter(ψ_star[0], ψ_star[1], ψ_star[2], c='k', s=60)

plt.show()
```



Here

- P is the stochastic matrix for recession and growth *considered above*.
- The highest red dot is an arbitrarily chosen initial probability distribution ψ, represented as a vector in $\mathbb{R}^3$.
- The other red dots are the distributions ψP^t for $t = 1, 2, \ldots$.
- The black dot is ψ^*.

You might like to try experimenting with different initial conditions.

17.7 Ergodicity

Under irreducibility, yet another important result obtains: For all $x \in S$,

$$\frac{1}{m} \sum_{t=1}^{m} \mathbf{1}\{X_t = x\} \to \psi^*(x) \quad \text{as } m \to \infty \tag{7}$$

Here

- $\mathbf{1}\{X_t = x\} = 1$ if $X_t = x$ and zero otherwise
- convergence is with probability one
- the result does not depend on the distribution (or value) of X_0

The result tells us that the fraction of time the chain spends at state x converges to $\psi^*(x)$ as time goes to infinity.

This gives us another way to interpret the stationary distribution — provided that the convergence result in (7) is valid.

The convergence in (7) is a special case of a law of large numbers result for Markov chains — see EDTC, section 4.3.4 for some additional information.

17.7.1 Example

Recall our cross-sectional interpretation of the employment/unemployment model *discussed above*.

Assume that $\alpha \in (0,1)$ and $\beta \in (0,1)$, so that irreducibility and aperiodicity both hold.

We saw that the stationary distribution is $(p, 1-p)$, where

$$p = \frac{\beta}{\alpha + \beta}$$

In the cross-sectional interpretation, this is the fraction of people unemployed.

In view of our latest (ergodicity) result, it is also the fraction of time that a worker can expect to spend unemployed.

Thus, in the long-run, cross-sectional averages for a population and time-series averages for a given person coincide.

This is one interpretation of the notion of ergodicity.

17.8 Computing Expectations

We are interested in computing expectations of the form

$$\mathbb{E}[h(X_t)] \tag{8}$$

and conditional expectations such as

$$\mathbb{E}[h(X_{t+k}) \mid X_t = x] \tag{9}$$

where

- $\{X_t\}$ is a Markov chain generated by $n \times n$ stochastic matrix P
- h is a given function, which, in expressions involving matrix algebra, we'll think of as the column vector

$$h = \begin{pmatrix} h(x_1) \\ \vdots \\ h(x_n) \end{pmatrix}$$

The unconditional expectation (8) is easy: We just sum over the distribution of X_t to get

$$\mathbb{E}[h(X_t)] = \sum_{x \in S} (\psi P^t)(x) h(x)$$

Here ψ is the distribution of X_0.

Since ψ and hence ψP^t are row vectors, we can also write this as

$$\mathbb{E}[h(X_t)] = \psi P^t h$$

For the conditional expectation (9), we need to sum over the conditional distribution of X_{t+k} given $X_t = x$.

We already know that this is $P^k(x, \cdot)$, so

$$\mathbb{E}[h(X_{t+k}) \mid X_t = x] = (P^k h)(x) \tag{10}$$

The vector $P^k h$ stores the conditional expectation $\mathbb{E}[h(X_{t+k}) \mid X_t = x]$ over all x.

17.8.1 Expectations of Geometric Sums

Sometimes we also want to compute expectations of a geometric sum, such as $\sum_t \beta^t h(X_t)$.

In view of the preceding discussion, this is

$$\mathbb{E}\left[\sum_{j=0}^{\infty} \beta^j h(X_{t+j}) \mid X_t = x\right] = [(I - \beta P)^{-1} h](x)$$

where

$$(I - \beta P)^{-1} = I + \beta P + \beta^2 P^2 + \cdots$$

Premultiplication by $(I - \beta P)^{-1}$ amounts to "applying the **resolvent operator**".

17.9 Exercises

17.9.1 Exercise 1

According to the discussion *above*, if a worker's employment dynamics obey the stochastic matrix

$$P = \begin{pmatrix} 1-\alpha & \alpha \\ \beta & 1-\beta \end{pmatrix}$$

with $\alpha \in (0,1)$ and $\beta \in (0,1)$, then, in the long-run, the fraction of time spent unemployed will be

$$p := \frac{\beta}{\alpha + \beta}$$

In other words, if $\{X_t\}$ represents the Markov chain for employment, then $\bar{X}_m \to p$ as $m \to \infty$, where

$$\bar{X}_m := \frac{1}{m} \sum_{t=1}^{m} \mathbf{1}\{X_t = 0\}$$

The exercise is to illustrate this convergence by computing $\bar{X}_m$ for large m and checking that it is close to p.

You will see that this statement is true regardless of the choice of initial condition or the values of α, β, provided both lie in $(0,1)$.

17.9.2 Exercise 2

A topic of interest for economics and many other disciplines is *ranking*.

Let's now consider one of the most practical and important ranking problems — the rank assigned to web pages by search engines.

(Although the problem is motivated from outside of economics, there is in fact a deep connection between search ranking systems and prices in certain competitive equilibria — see [DLP13].)

To understand the issue, consider the set of results returned by a query to a web search engine.

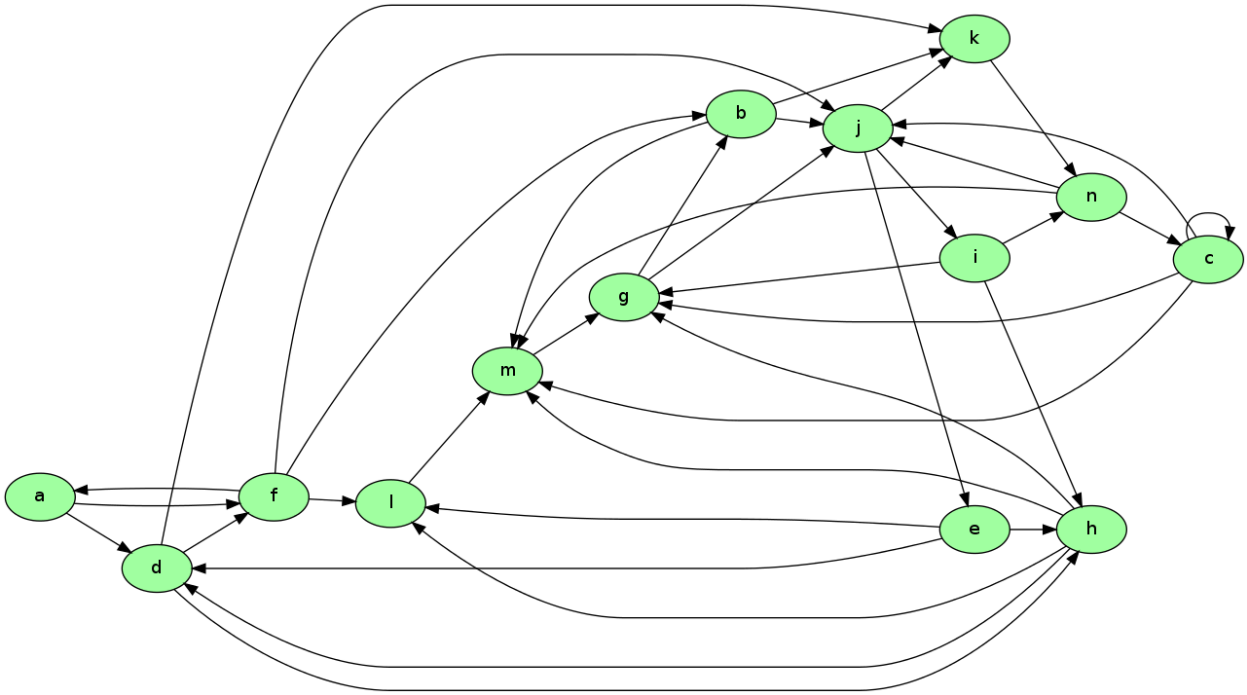
For the user, it is desirable to

1. receive a large set of accurate matches
2. have the matches returned in order, where the order corresponds to some measure of "importance"

Ranking according to a measure of importance is the problem we now consider.

The methodology developed to solve this problem by Google founders Larry Page and Sergey Brin is known as PageRank.

To illustrate the idea, consider the following diagram



Imagine that this is a miniature version of the WWW, with

- each node representing a web page
- each arrow representing the existence of a link from one page to another

Now let's think about which pages are likely to be important, in the sense of being valuable to a search engine user.

One possible criterion for the importance of a page is the number of inbound links — an indication of popularity.

By this measure, `m` and `j` are the most important pages, with 5 inbound links each.

However, what if the pages linking to `m`, say, are not themselves important?

Thinking this way, it seems appropriate to weight the inbound nodes by relative importance.

The PageRank algorithm does precisely this.

A slightly simplified presentation that captures the basic idea is as follows.

Letting j be (the integer index of) a typical page and r_j be its ranking, we set

$$r_j = \sum_{i \in L_j} \frac{r_i}{\ell_i}$$

where

- ℓ_i is the total number of outbound links from i
- L_j is the set of all pages i such that i has a link to j

This is a measure of the number of inbound links, weighted by their own ranking (and normalized by $1/\ell_i$).

There is, however, another interpretation, and it brings us back to Markov chains.

Let P be the matrix given by $P(i,j) = \mathbf{1}\{i \to j\}/\ell_i$ where $\mathbf{1}\{i \to j\} = 1$ if i has a link to j and zero otherwise.

The matrix P is a stochastic matrix provided that each page has at least one link.

With this definition of P we have

$$r_j = \sum_{i \in L_j} \frac{r_i}{\ell_i} = \sum_{\text{all } i} \mathbf{1}\{i \to j\} \frac{r_i}{\ell_i} = \sum_{\text{all } i} P(i,j) r_i$$

Writing r for the row vector of rankings, this becomes $r = rP$.

Hence r is the stationary distribution of the stochastic matrix P.

Let's think of $P(i,j)$ as the probability of "moving" from page i to page j.

The value $P(i,j)$ has the interpretation

- $P(i,j) = 1/k$ if i has k outbound links and j is one of them
- $P(i,j) = 0$ if i has no direct link to j

Thus, motion from page to page is that of a web surfer who moves from one page to another by randomly clicking on one of the links on that page.

Here "random" means that each link is selected with equal probability.

Since r is the stationary distribution of P, assuming that the uniform ergodicity condition is valid, we *can interpret* r_j as the fraction of time that a (very persistent) random surfer spends at page j.

Your exercise is to apply this ranking algorithm to the graph pictured above and return the list of pages ordered by rank.

There is a total of 14 nodes (i.e., web pages), the first named `a` and the last named `n`.

A typical line from the file has the form

```
d -> h;
```

This should be interpreted as meaning that there exists a link from `d` to `h`.

The data for this graph is shown below, and read into a file called `web_graph_data.txt` when the cell is executed.

```
%%file web_graph_data.txt
a -> d;
a -> f;
b -> j;
b -> k;
b -> m;
c -> c;
c -> g;
c -> j;
c -> m;
d -> f;
d -> h;
d -> k;
e -> d;
e -> h;
e -> l;
f -> a;
f -> b;
f -> j;
f -> l;
g -> b;
g -> j;
```

(continues on next page)

(continued from previous page)

```
h -> d;
h -> g;
h -> l;
h -> m;
i -> g;
i -> h;
i -> n;
j -> e;
j -> i;
j -> k;
k -> n;
l -> m;
m -> g;
n -> c;
n -> j;
n -> m;
```

```
Overwriting web_graph_data.txt
```

To parse this file and extract the relevant information, you can use regular expressions.

The following code snippet provides a hint as to how you can go about this

```
import re
re.findall('\w', 'x +++ y ****** z')  # \w matches alphanumerics
```

```
['x', 'y', 'z']
```

```
re.findall('\w', 'a ^^ b &&& $$ c')
```

```
['a', 'b', 'c']
```

When you solve for the ranking, you will find that the highest ranked node is in fact `g`, while the lowest is `a`.

17.9.3 Exercise 3

In numerical work, it is sometimes convenient to replace a continuous model with a discrete one.

In particular, Markov chains are routinely generated as discrete approximations to AR(1) processes of the form

$$y_{t+1} = \rho y_t + u_{t+1}$$

Here u_t is assumed to be IID and $N(0, \sigma_u^2)$.

The variance of the stationary probability distribution of $\{y_t\}$ is

$$\sigma_y^2 := \frac{\sigma_u^2}{1 - \rho^2}$$

Tauchen's method [Tau86] is the most common method for approximating this continuous state process with a finite state Markov chain.

A routine for this already exists in QuantEcon.py but let's write our own version as an exercise.

As a first step, we choose

- n, the number of states for the discrete approximation

- m, an integer that parameterizes the width of the state space

Next, we create a state space $\{x_0, \ldots, x_{n-1}\} \subset \mathbb{R}$ and a stochastic $n \times n$ matrix P such that

- $x_0 = -m\,\sigma_y$
- $x_{n-1} = m\,\sigma_y$
- $x_{i+1} = x_i + s$ where $s = (x_{n-1} - x_0)/(n-1)$

Let F be the cumulative distribution function of the normal distribution $N(0, \sigma_u^2)$.

The values $P(x_i, x_j)$ are computed to approximate the AR(1) process — omitting the derivation, the rules are as follows:

1. If $j = 0$, then set

$$P(x_i, x_j) = P(x_i, x_0) = F(x_0 - \rho x_i + s/2)$$

2. If $j = n - 1$, then set

$$P(x_i, x_j) = P(x_i, x_{n-1}) = 1 - F(x_{n-1} - \rho x_i - s/2)$$

3. Otherwise, set

$$P(x_i, x_j) = F(x_j - \rho x_i + s/2) - F(x_j - \rho x_i - s/2)$$

The exercise is to write a function `approx_markov(rho, sigma_u, m=3, n=7)` that returns $\{x_0, \ldots, x_{n-1}\} \subset \mathbb{R}$ and $n \times n$ matrix P as described above.

- Even better, write a function that returns an instance of QuantEcon.py's MarkovChain class.

17.10 Solutions

17.10.1 Exercise 1

We will address this exercise graphically.

The plots show the time series of $\bar{X}_m - p$ for two initial conditions.

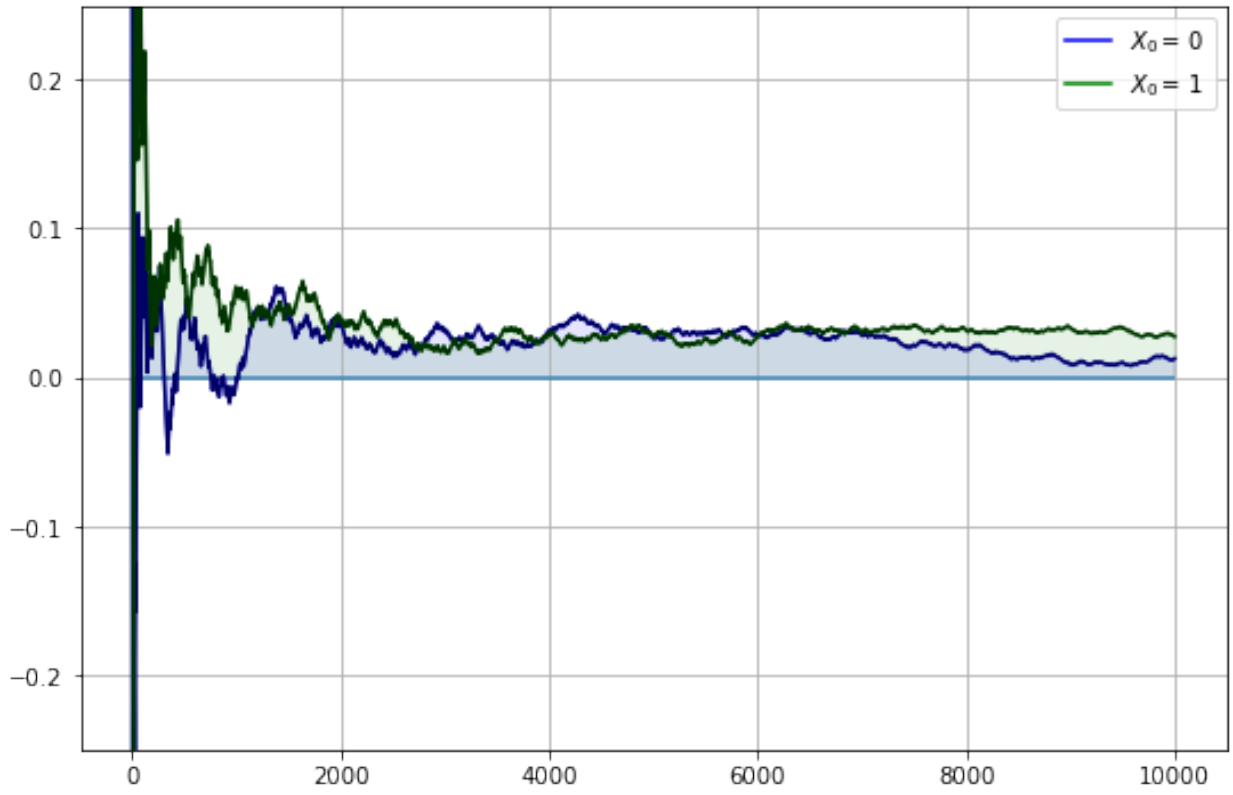
As m gets large, both series converge to zero.

```
α = β = 0.1
N = 10000
p = β / (α + β)

P = ((1 - α,       α),               # Careful: P and p are distinct
     (    β,   1 - β))
mc = MarkovChain(P)

fig, ax = plt.subplots(figsize=(9, 6))
ax.set_ylim(-0.25, 0.25)
ax.grid()
ax.hlines(0, 0, N, lw=2, alpha=0.6)   # Horizonal line at zero

for x0, col in ((0, 'blue'), (1, 'green')):
    # Generate time series for worker that starts at x0
    X = mc.simulate(N, init=x0)
```

(continues on next page)

(continued from previous page)

```
    # Compute fraction of time spent unemployed, for each n
    X_bar = (X == 0).cumsum() / (1 + np.arange(N, dtype=float))
    # Plot
    ax.fill_between(range(N), np.zeros(N), X_bar - p, color=col, alpha=0.1)
    ax.plot(X_bar - p, color=col, label=f'$X_0 = \, {x0} $')
    # Overlay in black--make lines clearer
    ax.plot(X_bar - p, 'k-', alpha=0.6)

ax.legend(loc='upper right')
plt.show()
```



17.10.2 Exercise 2

```
"""
Return list of pages, ordered by rank
"""
import re
from operator import itemgetter

infile = 'web_graph_data.txt'
alphabet = 'abcdefghijklmnopqrstuvwxyz'

n = 14 # Total number of web pages (nodes)

# Create a matrix Q indicating existence of links
#  * Q[i, j] = 1 if there is a link from i to j
```

(continues on next page)

(continued from previous page)

```
#  * Q[i, j] = 0 otherwise
Q = np.zeros((n, n), dtype=int)
f = open(infile, 'r')
edges = f.readlines()
f.close()
for edge in edges:
    from_node, to_node = re.findall('\w', edge)
    i, j = alphabet.index(from_node), alphabet.index(to_node)
    Q[i, j] = 1
# Create the corresponding Markov matrix P
P = np.empty((n, n))
for i in range(n):
    P[i, :] = Q[i, :] / Q[i, :].sum()
mc = MarkovChain(P)
# Compute the stationary distribution r
r = mc.stationary_distributions[0]
ranked_pages = {alphabet[i] : r[i] for i in range(n)}
# Print solution, sorted from highest to lowest rank
print('Rankings\n ***')
for name, rank in sorted(ranked_pages.items(), key=itemgetter(1), reverse=1):
    print(f'{name}: {rank:.4}')
```

```
Rankings
 ***
g: 0.1607
j: 0.1594
m: 0.1195
n: 0.1088
k: 0.09106
b: 0.08326
e: 0.05312
i: 0.05312
c: 0.04834
h: 0.0456
l: 0.03202
d: 0.03056
f: 0.01164
a: 0.002911
```

17.10.3 Exercise 3

A solution from the QuantEcon.py library can be found here.

CHAPTER

EIGHTEEN

INVENTORY DYNAMICS

Contents

18.1 Overview

In this lecture we will study the time path of inventories for firms that follow so-called s-S inventory dynamics.

Such firms

1. wait until inventory falls below some level s and then
2. order sufficient quantities to bring their inventory back up to capacity S.

These kinds of policies are common in practice and also optimal in certain circumstances.

A review of early literature and some macroeconomic implications can be found in [Cap85].

Here our main aim is to learn more about simulation, time series and Markov dynamics.

While our Markov environment and many of the concepts we consider are related to those found in our *lecture on finite Markov chains*, the state space is a continuum in the current application.

Let's start with some imports

```
%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, float64, prange
from numba.experimental import jitclass
```

18.2 Sample Paths

Consider a firm with inventory X_t.

The firm waits until $X_t \leq s$ and then restocks up to S units.

It faces stochastic demand $\{D_t\}$, which we assume is IID.

With notation $a^+ := \max\{a, 0\}$, inventory dynamics can be written as

$$X_{t+1} = \begin{cases} (S - D_{t+1})^+ & \text{if } X_t \leq s \\ (X_t - D_{t+1})^+ & \text{if } X_t > s \end{cases}$$

In what follows, we will assume that each D_t is lognormal, so that

$$D_t = \exp(\mu + \sigma Z_t)$$

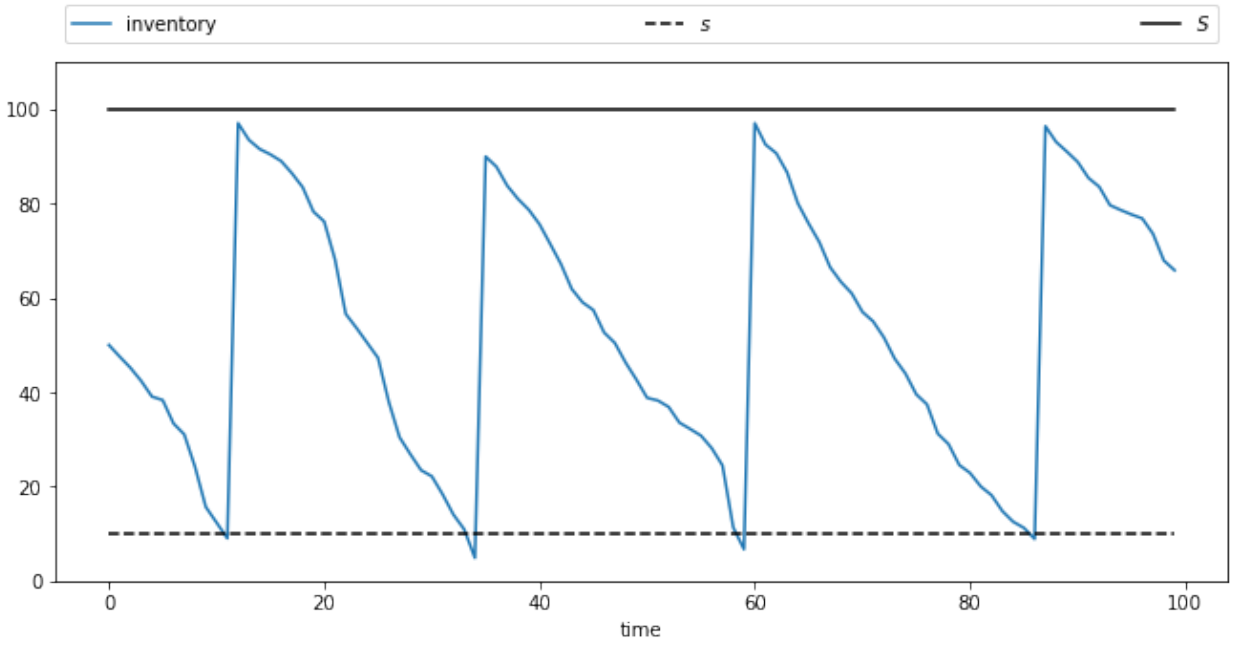
where μ and σ are parameters and $\{Z_t\}$ is IID and standard normal.

Here's a class that stores parameters and generates time paths for inventory.

```
firm_data = [
    ('s', float64),          # restock trigger level
    ('S', float64),          # capacity
    ('mu', float64),         # shock location parameter
    ('sigma', float64)       # shock scale parameter
]


@jitclass(firm_data)
class Firm:

    def __init__(self, s=10, S=100, mu=1.0, sigma=0.5):

        self.s, self.S, self.mu, self.sigma = s, S, mu, sigma

    def update(self, x):
        "Update the state from t to t+1 given current state x."

        Z = np.random.randn()
        D = np.exp(self.mu + self.sigma * Z)
        if x <= self.s:
            return max(self.S - D, 0)
        else:
            return max(x - D, 0)

    def sim_inventory_path(self, x_init, sim_length):

        X = np.empty(sim_length)
        X[0] = x_init

        for t in range(sim_length-1):
            X[t+1] = self.update(X[t])
        return X
```

Let's run a first simulation, of a single path:

```
firm = Firm()

s, S = firm.s, firm.S
sim_length = 100
x_init = 50

X = firm.sim_inventory_path(x_init, sim_length)

fig, ax = plt.subplots()
bbox = (0., 1.02, 1., .102)
legend_args = {'ncol': 3,
               'bbox_to_anchor': bbox,
               'loc': 3,
               'mode': 'expand'}

ax.plot(X, label="inventory")
ax.plot(np.full(sim_length, s), 'k--', label="$s$")
ax.plot(np.full(sim_length, S), 'k-', label="$S$")
ax.set_ylim(0, S+10)
ax.set_xlabel("time")
ax.legend(**legend_args)

plt.show()
```



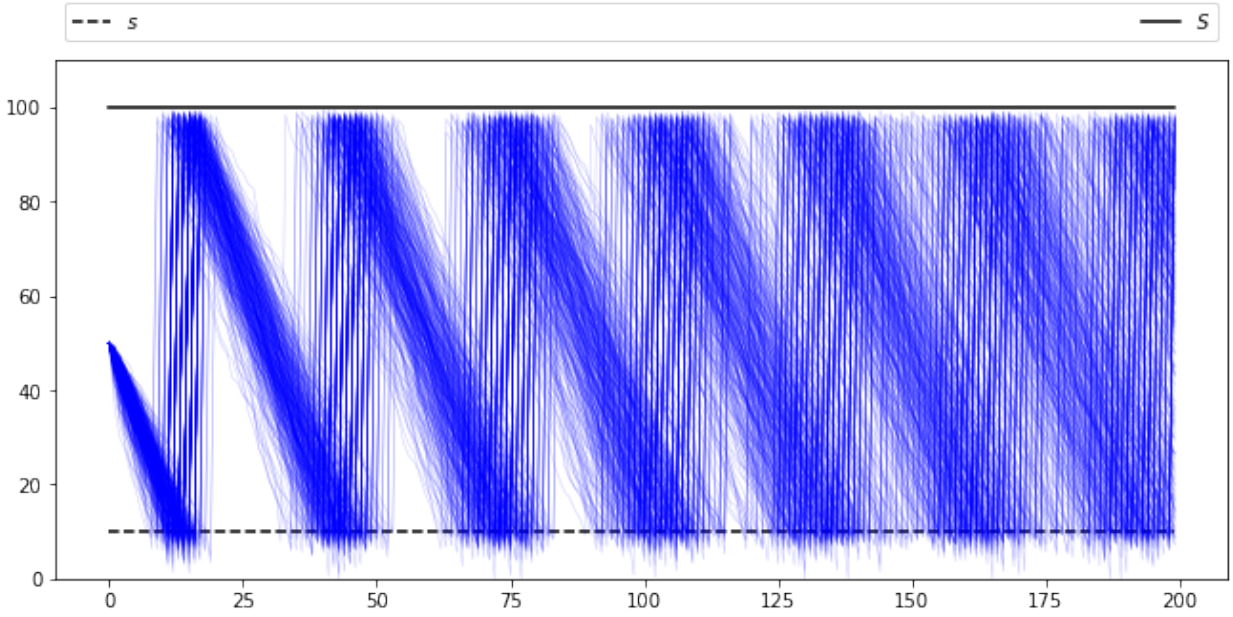
Now let's simulate multiple paths in order to build a more complete picture of the probabilities of different outcomes:

```
sim_length=200
fig, ax = plt.subplots()

ax.plot(np.full(sim_length, s), 'k--', label="$s$")
ax.plot(np.full(sim_length, S), 'k-', label="$S$")
ax.set_ylim(0, S+10)
ax.legend(**legend_args)
```

(continues on next page)

(continued from previous page)

```
for i in range(400):
    X = firm.sim_inventory_path(x_init, sim_length)
    ax.plot(X, 'b', alpha=0.2, lw=0.5)

plt.show()
```



18.3 Marginal Distributions

Now let's look at the marginal distribution ψ_T of X_T for some fixed T.

We will do this by generating many draws of X_T given initial condition X_0.

With these draws of X_T we can build up a picture of its distribution ψ_T.

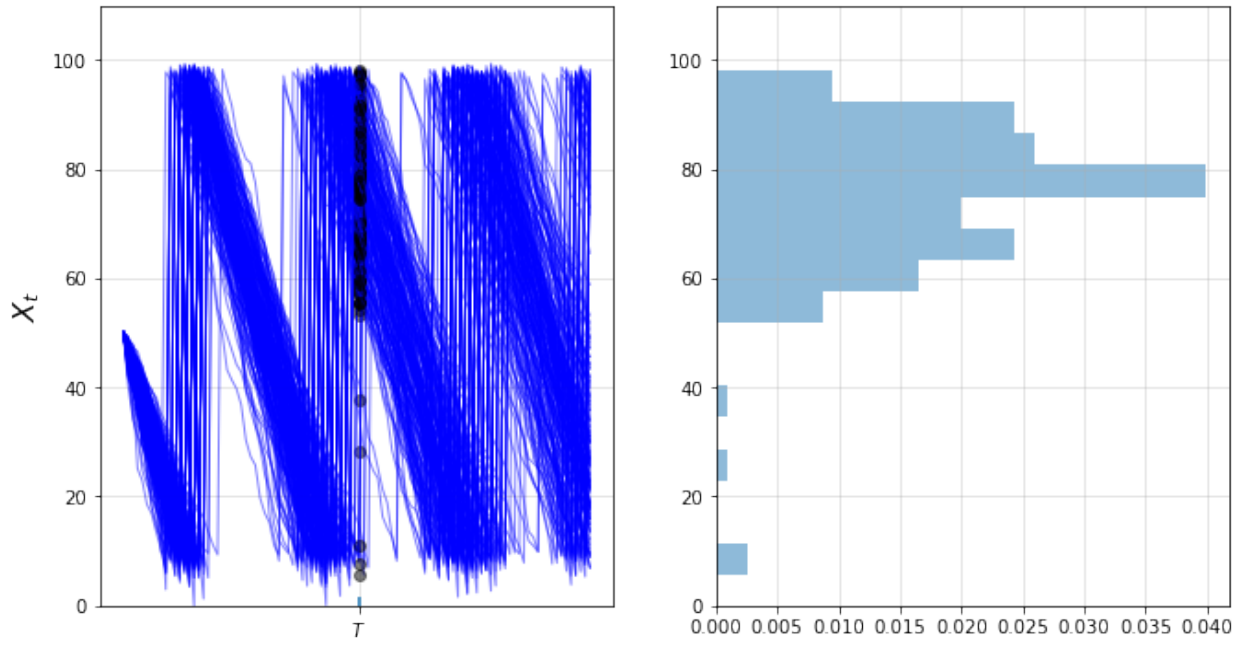
Here's one visualization, with $T = 50$.

```
T = 50
M = 200  # Number of draws

ymin, ymax = 0, S + 10

fig, axes = plt.subplots(1, 2, figsize=(11, 6))

for ax in axes:
    ax.grid(alpha=0.4)

ax = axes[0]

ax.set_ylim(ymin, ymax)
ax.set_ylabel('$X_t$', fontsize=16)
ax.vlines((T,), -1.5, 1.5)

ax.set_xticks((T,))
```

(continues on next page)

(continued from previous page)

```
ax.set_xticklabels((r'$T$',))

sample = np.empty(M)
for m in range(M):
    X = firm.sim_inventory_path(x_init, 2 * T)
    ax.plot(X, 'b-', lw=1, alpha=0.5)
    ax.plot((T,), (X[T+1],), 'ko', alpha=0.5)
    sample[m] = X[T+1]

axes[1].set_ylim(ymin, ymax)

axes[1].hist(sample,
             bins=16,
             density=True,
             orientation='horizontal',
             histtype='bar',
             alpha=0.5)

plt.show()
```



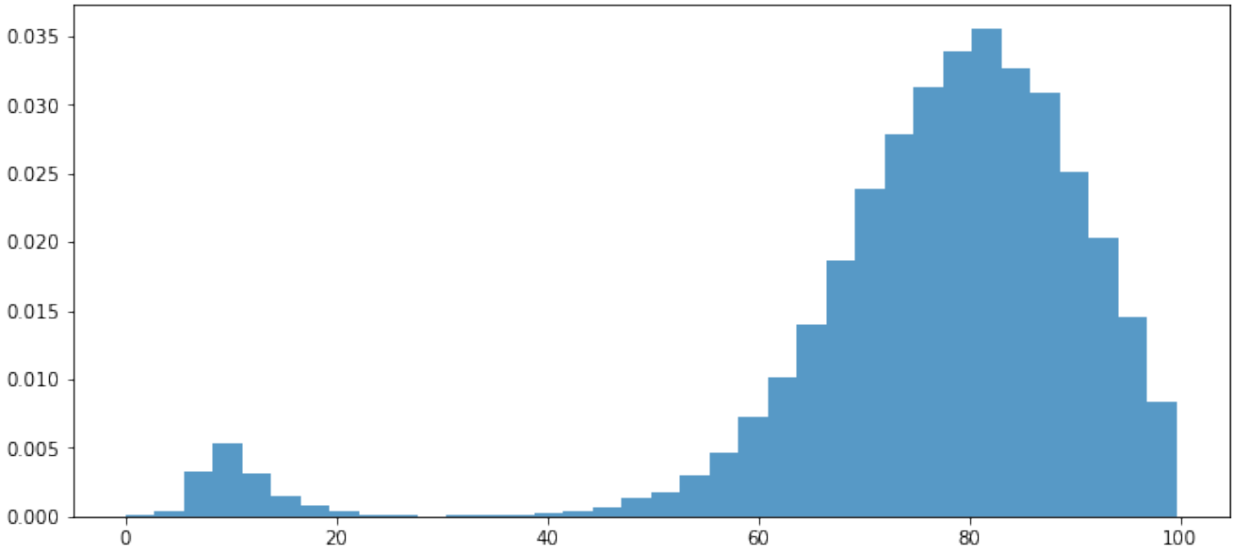
We can build up a clearer picture by drawing more samples

```
T = 50
M = 50_000

fig, ax = plt.subplots()

sample = np.empty(M)
for m in range(M):
    X = firm.sim_inventory_path(x_init, T+1)
    sample[m] = X[T]

ax.hist(sample,
```

(continues on next page)

(continued from previous page)

```
        bins=36,
        density=True,
        histtype='bar',
        alpha=0.75)

plt.show()
```



Note that the distribution is bimodal

- Most firms have restocked twice but a few have restocked only once (see figure with paths above).
- Firms in the second category have lower inventory.

We can also approximate the distribution using a kernel density estimator.

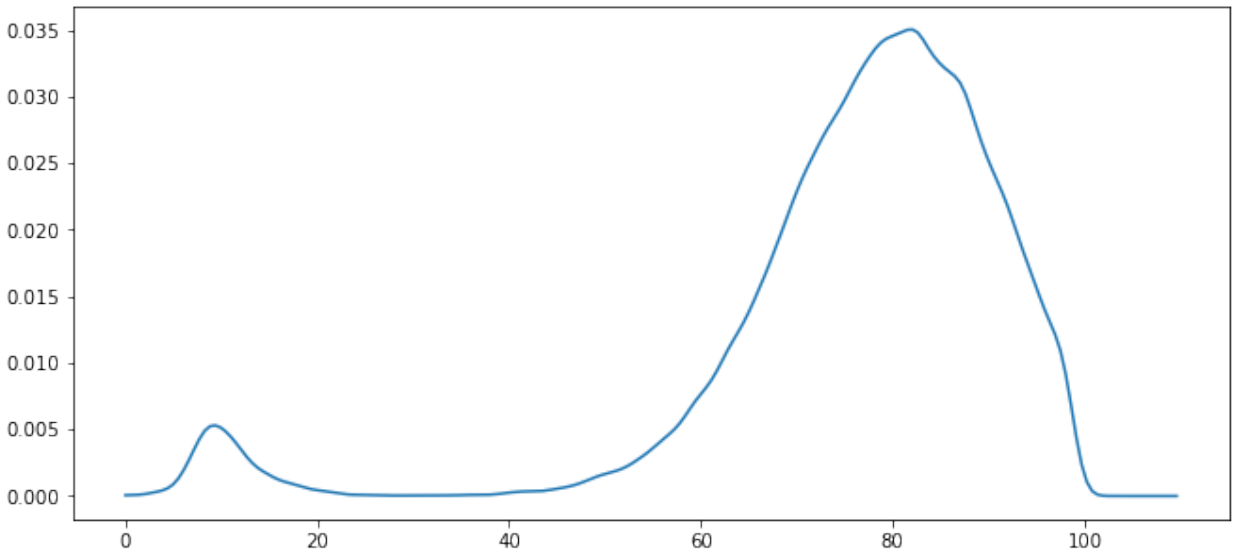
Kernel density estimators can be thought of as smoothed histograms.

They are preferable to histograms when the distribution being estimated is likely to be smooth.

We will use a kernel density estimator from scikit-learn

```
from sklearn.neighbors import KernelDensity

def plot_kde(sample, ax, label=''):

    xmin, xmax = 0.9 * min(sample), 1.1 * max(sample)
    xgrid = np.linspace(xmin, xmax, 200)
    kde = KernelDensity(kernel='gaussian').fit(sample[:, None])
    log_dens = kde.score_samples(xgrid[:, None])

    ax.plot(xgrid, np.exp(log_dens), label=label)
```

```
fig, ax = plt.subplots()
plot_kde(sample, ax)
plt.show()
```



The allocation of probability mass is similar to what was shown by the histogram just above.

18.4 Exercises

18.4.1 Exercise 1

This model is asymptotically stationary, with a unique stationary distribution.

(See the discussion of stationarity in *our lecture on AR(1) processes* for background — the fundamental concepts are the same.)

In particular, the sequence of marginal distributions $\{\psi_t\}$ is converging to a unique limiting distribution that does not depend on initial conditions.

Although we will not prove this here, we can investigate it using simulation.

Your task is to generate and plot the sequence $\{\psi_t\}$ at times $t = 10, 50, 250, 500, 750$ based on the discussion above.

(The kernel density estimator is probably the best way to present each distribution.)

You should see convergence, in the sense that differences between successive distributions are getting smaller.

Try different initial conditions to verify that, in the long run, the distribution is invariant across initial conditions.

18.4.2 Exercise 2

Using simulation, calculate the probability that firms that start with $X_0 = 70$ need to order twice or more in the first 50 periods.

You will need a large sample size to get an accurate reading.

18.5 Solutions

18.5.1 Exercise 1

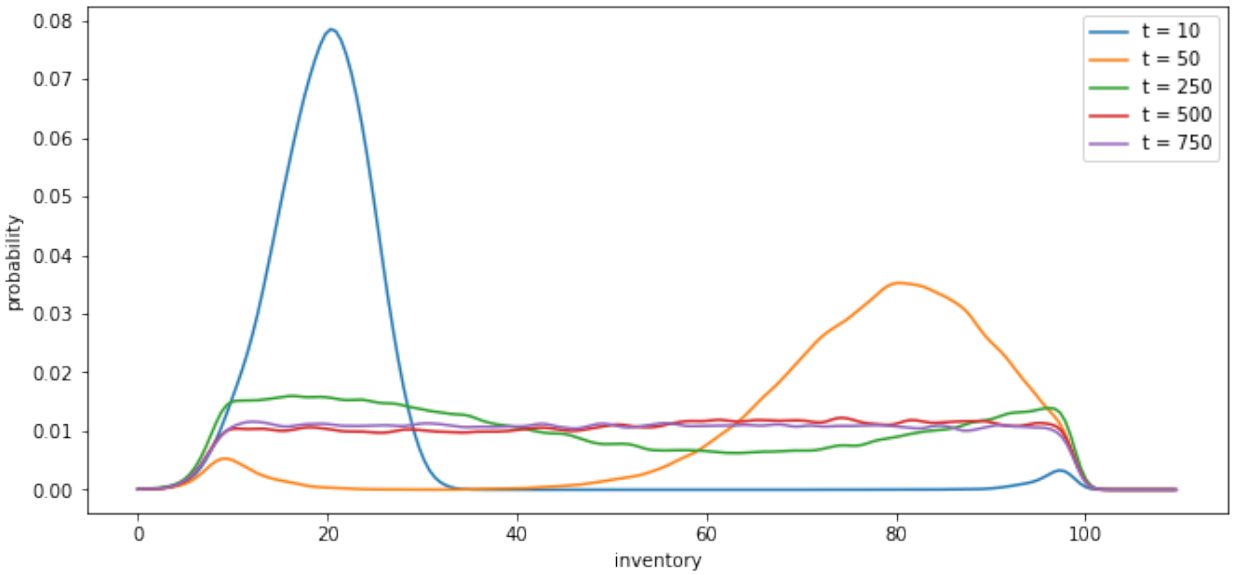
Below is one possible solution:

The computations involve a lot of CPU cycles so we have tried to write the code efficiently.

This meant writing a specialized function rather than using the class above.

```
s, S, mu, sigma = firm.s, firm.S, firm.mu, firm.sigma

@njit(parallel=True)
def shift_firms_forward(current_inventory_levels, num_periods):

    num_firms = len(current_inventory_levels)
    new_inventory_levels = np.empty(num_firms)

    for f in prange(num_firms):
        x = current_inventory_levels[f]
        for t in range(num_periods):
            Z = np.random.randn()
            D = np.exp(mu + sigma * Z)
            if x <= s:
                x = max(S - D, 0)
            else:
                x = max(x - D, 0)
        new_inventory_levels[f] = x

    return new_inventory_levels
```

```
x_init = 50
num_firms = 50_000

sample_dates = 0, 10, 50, 250, 500, 750

first_diffs = np.diff(sample_dates)

fig, ax = plt.subplots()

X = np.full(num_firms, x_init)

current_date = 0
for d in first_diffs:
    X = shift_firms_forward(X, d)
    current_date += d
    plot_kde(X, ax, label=f't = {current_date}')

ax.set_xlabel('inventory')
ax.set_ylabel('probability')
ax.legend()
plt.show()
```



Notice that by $t = 500$ or $t = 750$ the densities are barely changing.

We have reached a reasonable approximation of the stationary density.

You can convince yourself that initial conditions don't matter by testing a few of them.

For example, try rerunning the code above will all firms starting at $X_0 = 20$ or $X_0 = 80$.

18.5.2 Exercise 2

Here is one solution.

Again, the computations are relatively intensive so we have written a a specialized function rather than using the class above.

We will also use parallelization across firms.

```
@njit(parallel=True)
def compute_freq(sim_length=50, x_init=70, num_firms=1_000_000):

    firm_counter = 0  # Records number of firms that restock 2x or more
    for m in prange(num_firms):
        x = x_init
        restock_counter = 0  # Will record number of restocks for firm m

        for t in range(sim_length):
            Z = np.random.randn()
            D = np.exp(mu + sigma * Z)
            if x <= s:
                x = max(S - D, 0)
                restock_counter += 1
            else:
                x = max(x - D, 0)

        if restock_counter > 1:
            firm_counter += 1

    return firm_counter / num_firms
```

Note the time the routine takes to run, as well as the output.

```
%%time

freq = compute_freq()
print(f"Frequency of at least two stock outs = {freq}")
```

```
Frequency of at least two stock outs = 0.44694
CPU times: user 3.23 s, sys: 40.1 ms, total: 3.27 s
Wall time: 1.92 s
```

Try switching the `parallel` flag to `False` in the jitted function above.

Depending on your system, the difference can be substantial.

(On our desktop machine, the speed up is by a factor of 5.)

CHAPTER

NINETEEN

LINEAR STATE SPACE MODELS

Contents

> "We may regard the present state of the universe as the effect of its past and the cause of its future" – Marquis de Laplace

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

```
!conda install -y quantecon
```

19.1 Overview

This lecture introduces the **linear state space** dynamic system.

The linear state space system is a generalization of the scalar AR(1) process *we studied before*.

This model is a workhorse that carries a powerful theory of prediction.

Its many applications include:

- representing dynamics of higher-order linear systems
- predicting the position of a system j steps into the future
- predicting a geometric sum of future values of a variable like
  - non-financial income

  - dividends on a stock
  - the money supply
  - a government deficit or surplus, etc.
- key ingredient of useful models
  - Friedman's permanent income model of consumption smoothing.
  - Barro's model of smoothing total tax collections.
  - Rational expectations version of Cagan's model of hyperinflation.
  - Sargent and Wallace's "unpleasant monetarist arithmetic," etc.

Let's start with some imports:

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

19.2 The Linear State Space Model

The objects in play are:

- An $n \times 1$ vector x_t denoting the **state** at time $t = 0, 1, 2, \ldots$.
- An IID sequence of $m \times 1$ random vectors $w_t \sim N(0, I)$.
- A $k \times 1$ vector y_t of **observations** at time $t = 0, 1, 2, \ldots$.
- An $n \times n$ matrix A called the **transition matrix**.
- An $n \times m$ matrix C called the **volatility matrix**.
- A $k \times n$ matrix G sometimes called the **output matrix**.

Here is the linear state-space system

$$
\begin{aligned}
x_{t+1} &= Ax_t + Cw_{t+1} \\
y_t &= Gx_t \\
x_0 &\sim N(\mu_0, \Sigma_0)
\end{aligned}
$$

19.2.1 Primitives

The primitives of the model are

1. the matrices A, C, G
2. shock distribution, which we have specialized to $N(0, I)$
3. the distribution of the initial condition x_0, which we have set to $N(\mu_0, \Sigma_0)$

Given A, C, G and draws of x_0 and $w_1, w_2, ...$, the model (1) pins down the values of the sequences $\{x_t\}$ and $\{y_t\}$.

Even without these draws, the primitives 1–3 pin down the *probability distributions* of $\{x_t\}$ and $\{y_t\}$.

Later we'll see how to compute these distributions and their moments.

Martingale Difference Shocks

We've made the common assumption that the shocks are independent standardized normal vectors.

But some of what we say will be valid under the assumption that $\{w_{t+1}\}$ is a **martingale difference sequence**.

A martingale difference sequence is a sequence that is zero mean when conditioned on past information.

In the present case, since $\{x_t\}$ is our state sequence, this means that it satisfies

$$\mathbb{E}[w_{t+1}|x_t, x_{t-1}, ...] = 0$$

This is a weaker condition than that $\{w_t\}$ is IID with $w_{t+1} \sim N(0, I)$.

19.2.2 Examples

By appropriate choice of the primitives, a variety of dynamics can be represented in terms of the linear state space model.

The following examples help to highlight this point.

They also illustrate the wise dictum *finding the state is an art*.

Second-order Difference Equation

Let $\{y_t\}$ be a deterministic sequence that satisfies

$$y_{t+1} = \phi_0 + \phi_1 y_t + \phi_2 y_{t-1} \quad \text{s.t.} \quad y_0, y_{-1} \text{ given} \tag{1}$$

To map (1) into our state space system (1), we set

$$x_t = \begin{bmatrix} 1 \\ y_t \\ y_{t-1} \end{bmatrix} \qquad A = \begin{bmatrix} 1 & 0 & 0 \\ \phi_0 & \phi_1 & \phi_2 \\ 0 & 1 & 0 \end{bmatrix} \qquad C = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \qquad G = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}$$

You can confirm that under these definitions, (1) and (1) agree.

The next figure shows the dynamics of this process when $\phi_0 = 1.1, \phi_1 = 0.8, \phi_2 = -0.8, y_0 = y_{-1} = 1$.

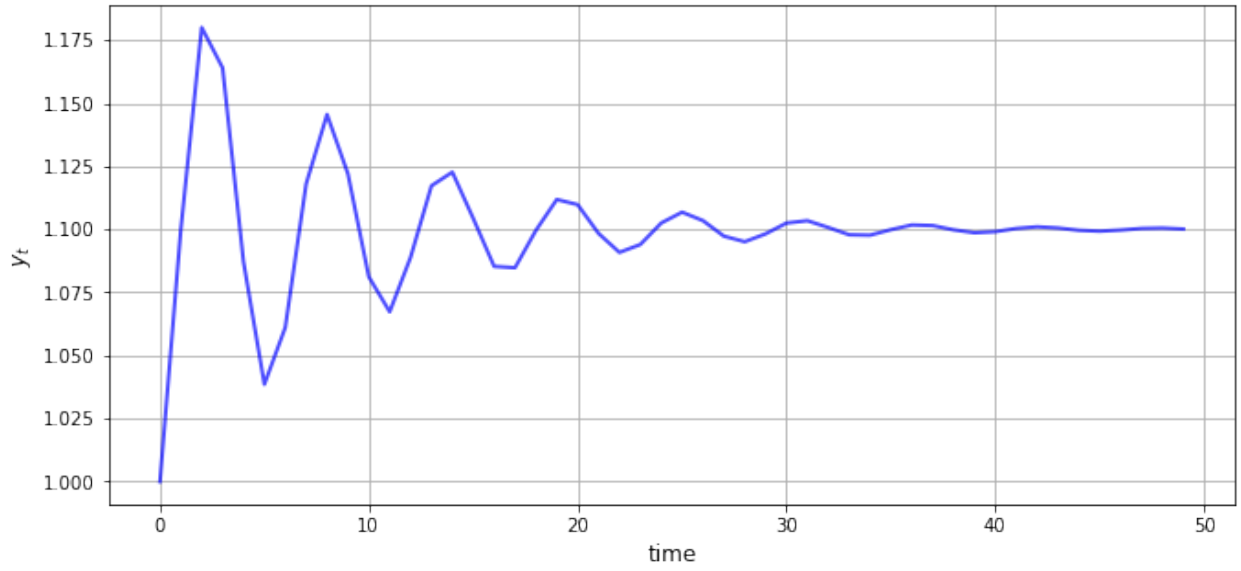
```
def plot_lss(A,
             C,
             G,
             n=3,
             ts_length=50):

    ar = LinearStateSpace(A, C, G, mu_0=np.ones(n))
    x, y = ar.simulate(ts_length)

    fig, ax = plt.subplots()
    y = y.flatten()
    ax.plot(y, 'b-', lw=2, alpha=0.7)
    ax.grid()
```

(continues on next page)

(continued from previous page)

```
    ax.set_xlabel('time', fontsize=12)
    ax.set_ylabel('$y_t$', fontsize=12)
    plt.show()
```

```
ϕ_0, ϕ_1, ϕ_2 = 1.1, 0.8, -0.8

A = [[1,     0,     0  ],
     [ϕ_0,   ϕ_1,   ϕ_2],
     [0,     1,     0  ]]

C = np.zeros((3, 1))
G = [0, 1, 0]

plot_lss(A, C, G)
```



Later you'll be asked to recreate this figure.

Univariate Autoregressive Processes

We can use (1) to represent the model

$$y_{t+1} = \phi_1 y_t + \phi_2 y_{t-1} + \phi_3 y_{t-2} + \phi_4 y_{t-3} + \sigma w_{t+1} \tag{2}$$

where $\{w_t\}$ is IID and standard normal.

To put this in the linear state space format we take $x_t = \begin{bmatrix} y_t & y_{t-1} & y_{t-2} & y_{t-3} \end{bmatrix}'$ and

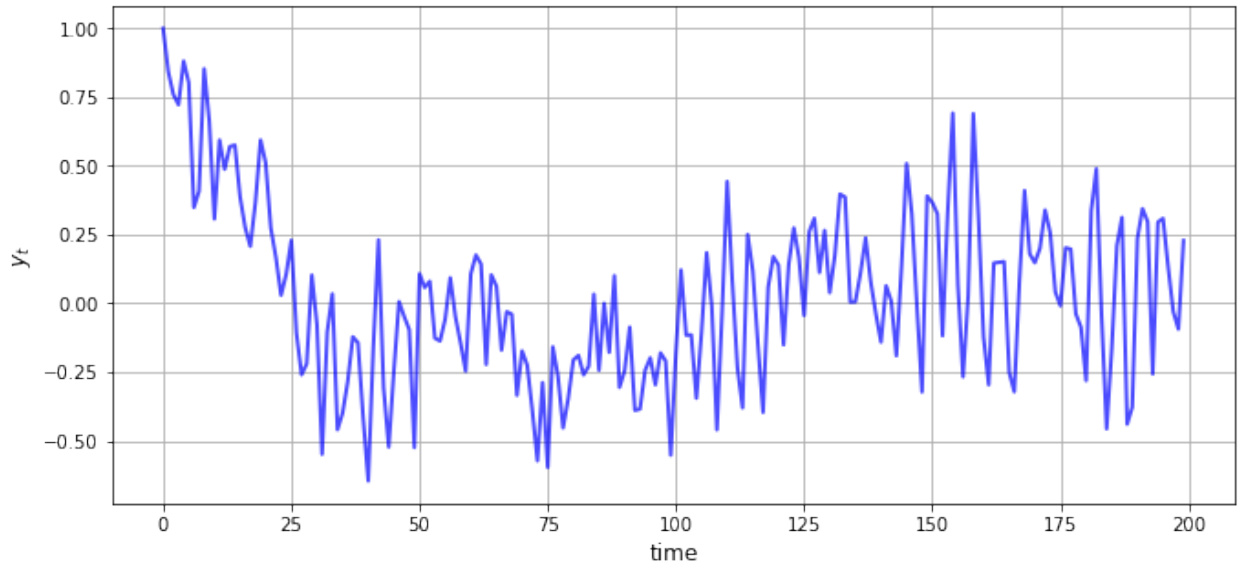
$$A = \begin{bmatrix} \phi_1 & \phi_2 & \phi_3 & \phi_4 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \qquad C = \begin{bmatrix} \sigma \\ 0 \\ 0 \\ 0 \end{bmatrix} \qquad G = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}$$

The matrix A has the form of the *companion matrix* to the vector $\begin{bmatrix} \phi_1 & \phi_2 & \phi_3 & \phi_4 \end{bmatrix}$.

The next figure shows the dynamics of this process when

$$\phi_1 = 0.5, \phi_2 = -0.2, \phi_3 = 0, \phi_4 = 0.5, \sigma = 0.2, y_0 = y_{-1} = y_{-2} = y_{-3} = 1$$

```
ϕ_1, ϕ_2, ϕ_3, ϕ_4 = 0.5, -0.2, 0, 0.5
σ = 0.2

A_1 = [[ϕ_1,   ϕ_2,   ϕ_3,   ϕ_4],
       [1,     0,     0,     0  ],
       [0,     1,     0,     0  ],
       [0,     0,     1,     0  ]]

C_1 = [[σ],
       [0],
       [0],
       [0]]

G_1 = [1, 0, 0, 0]

plot_lss(A_1, C_1, G_1, n=4, ts_length=200)
```



Vector Autoregressions

Now suppose that

- y_t is a $k \times 1$ vector
- ϕ_j is a $k \times k$ matrix and
- w_t is $k \times 1$

Then (2) is termed a *vector autoregression*.

To map this into (1), we set

$$x_t = \begin{bmatrix} y_t \\ y_{t-1} \\ y_{t-2} \\ y_{t-3} \end{bmatrix} \quad A = \begin{bmatrix} \phi_1 & \phi_2 & \phi_3 & \phi_4 \\ I & 0 & 0 & 0 \\ 0 & I & 0 & 0 \\ 0 & 0 & I & 0 \end{bmatrix} \quad C = \begin{bmatrix} \sigma \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad G = \begin{bmatrix} I & 0 & 0 & 0 \end{bmatrix}$$

where I is the $k \times k$ identity matrix and σ is a $k \times k$ matrix.

Seasonals

We can use (1) to represent

1. the *deterministic seasonal* $y_t = y_{t-4}$
2. the *indeterministic seasonal* $y_t = \phi_4 y_{t-4} + w_t$

In fact, both are special cases of (2).

With the deterministic seasonal, the transition matrix becomes

$$A = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

It is easy to check that $A^4 = I$, which implies that x_t is strictly periodic with period 4:[1]

$$x_{t+4} = x_t$$

Such an x_t process can be used to model deterministic seasonals in quarterly time series.

The *indeterministic* seasonal produces recurrent, but aperiodic, seasonal fluctuations.

Time Trends

The model $y_t = at + b$ is known as a *linear time trend*.

We can represent this model in the linear state space form by taking

$$A = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \qquad C = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \qquad G = \begin{bmatrix} a & b \end{bmatrix} \tag{3}$$

and starting at initial condition $x_0 = \begin{bmatrix} 0 & 1 \end{bmatrix}'$.

In fact, it's possible to use the state-space system to represent polynomial trends of any order.

For instance, we can represent the model $y_t = at^2 + bt + c$ in the linear state space form by taking

$$A = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \qquad C = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \qquad G = \begin{bmatrix} 2a & a+b & c \end{bmatrix}$$

and starting at initial condition $x_0 = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}'$.

It follows that

$$A^t = \begin{bmatrix} 1 & t & t(t-1)/2 \\ 0 & 1 & t \\ 0 & 0 & 1 \end{bmatrix}$$

Then $x_t' = \begin{bmatrix} t(t-1)/2 & t & 1 \end{bmatrix}$. You can now confirm that $y_t = Gx_t$ has the correct form.

[1] The eigenvalues of A are $(1, -1, i, -i)$.

19.2.3 Moving Average Representations

A nonrecursive expression for x_t as a function of $x_0, w_1, w_2, \ldots, w_t$ can be found by using (1) repeatedly to obtain

$$\begin{aligned} x_t &= Ax_{t-1} + Cw_t \\ &= A^2 x_{t-2} + ACw_{t-1} + Cw_t \\ &\quad \vdots \\ &= \sum_{j=0}^{t-1} A^j C w_{t-j} + A^t x_0 \end{aligned}$$

Representation (4) is a *moving average* representation.

It expresses $\{x_t\}$ as a linear function of

1. current and past values of the process $\{w_t\}$ and
2. the initial condition x_0

As an example of a moving average representation, let the model be

$$A = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \qquad C = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

You will be able to show that $A^t = \begin{bmatrix} 1 & t \\ 0 & 1 \end{bmatrix}$ and $A^j C = \begin{bmatrix} 1 & 0 \end{bmatrix}'$.

Substituting into the moving average representation (4), we obtain

$$x_{1t} = \sum_{j=0}^{t-1} w_{t-j} + \begin{bmatrix} 1 & t \end{bmatrix} x_0$$

where x_{1t} is the first entry of x_t.

The first term on the right is a cumulated sum of martingale differences and is therefore a martingale.

The second term is a translated linear function of time.

For this reason, x_{1t} is called a *martingale with drift*.

19.3 Distributions and Moments

19.3.1 Unconditional Moments

Using (1), it's easy to obtain expressions for the (unconditional) means of x_t and y_t.

We'll explain what *unconditional* and *conditional* mean soon.

Letting $\mu_t := \mathbb{E}[x_t]$ and using linearity of expectations, we find that

$$\mu_{t+1} = A\mu_t \quad \text{with} \quad \mu_0 \text{ given} \tag{4}$$

Here μ_0 is a primitive given in (1).

The variance-covariance matrix of x_t is $\Sigma_t := \mathbb{E}[(x_t - \mu_t)(x_t - \mu_t)']$.

Using $x_{t+1} - \mu_{t+1} = A(x_t - \mu_t) + Cw_{t+1}$, we can determine this matrix recursively via

$$\Sigma_{t+1} = A\Sigma_t A' + CC' \quad \text{with} \quad \Sigma_0 \text{ given} \tag{5}$$

As with μ_0, the matrix Σ_0 is a primitive given in (1).

As a matter of terminology, we will sometimes call

- μ_t the *unconditional mean* of x_t
- Σ_t the *unconditional variance-covariance matrix* of x_t

This is to distinguish μ_t and Σ_t from related objects that use conditioning information, to be defined below.

However, you should be aware that these "unconditional" moments do depend on the initial distribution $N(\mu_0, \Sigma_0)$.

Moments of the Observations

Using linearity of expectations again we have

$$\mathbb{E}[y_t] = \mathbb{E}[Gx_t] = G\mu_t \tag{6}$$

The variance-covariance matrix of y_t is easily shown to be

$$\text{Var}[y_t] = \text{Var}[Gx_t] = G\Sigma_t G' \tag{7}$$

19.3.2 Distributions

In general, knowing the mean and variance-covariance matrix of a random vector is not quite as good as knowing the full distribution.

However, there are some situations where these moments alone tell us all we need to know.

These are situations in which the mean vector and covariance matrix are **sufficient statistics** for the population distribution.

(Sufficient statistics form a list of objects that characterize a population distribution)

One such situation is when the vector in question is Gaussian (i.e., normally distributed).

This is the case here, given

1. our Gaussian assumptions on the primitives
2. the fact that normality is preserved under linear operations

In fact, it's well-known that

$$u \sim N(\bar{u}, S) \quad \text{and} \quad v = a + Bu \implies v \sim N(a + B\bar{u}, BSB') \tag{8}$$

In particular, given our Gaussian assumptions on the primitives and the linearity of (1) we can see immediately that both x_t and y_t are Gaussian for all $t \geq 0$[2].

Since x_t is Gaussian, to find the distribution, all we need to do is find its mean and variance-covariance matrix.

But in fact we've already done this, in (4) and (5).

Letting μ_t and Σ_t be as defined by these equations, we have

$$x_t \sim N(\mu_t, \Sigma_t) \tag{9}$$

By similar reasoning combined with (6) and (7),

$$y_t \sim N(G\mu_t, G\Sigma_t G') \tag{10}$$

[2] The correct way to argue this is by induction. Suppose that x_t is Gaussian. Then (1) and (8) imply that x_{t+1} is Gaussian. Since x_0 is assumed to be Gaussian, it follows that every x_t is Gaussian. Evidently, this implies that each y_t is Gaussian.

19.3.3 Ensemble Interpretations

How should we interpret the distributions defined by (9)–(10)?

Intuitively, the probabilities in a distribution correspond to relative frequencies in a large population drawn from that distribution.

Let's apply this idea to our setting, focusing on the distribution of y_T for fixed T.

We can generate independent draws of y_T by repeatedly simulating the evolution of the system up to time T, using an independent set of shocks each time.

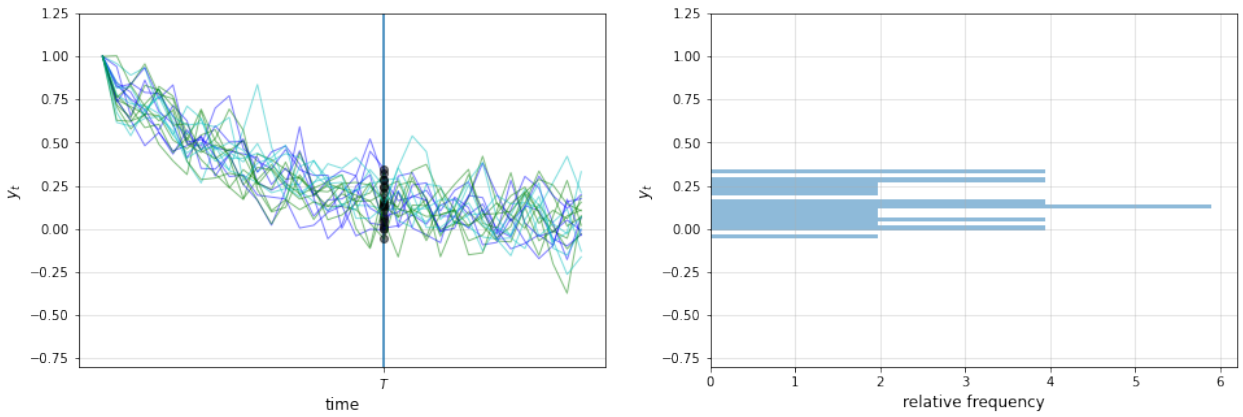
The next figure shows 20 simulations, producing 20 time series for $\{y_t\}$, and hence 20 draws of y_T.

The system in question is the univariate autoregressive model (2).

The values of y_T are represented by black dots in the left-hand figure

```
def cross_section_plot(A,
                       C,
                       G,
                       T=20,                 # Set the time
                       ymin=-0.8,
                       ymax=1.25,
                       sample_size = 20,     # 20 observations/simulations
                       n=4):                 # The number of dimensions for the initial x0

    ar = LinearStateSpace(A, C, G, mu_0=np.ones(n))

    fig, axes = plt.subplots(1, 2, figsize=(16, 5))

    for ax in axes:
        ax.grid(alpha=0.4)
        ax.set_ylim(ymin, ymax)

    ax = axes[0]
    ax.set_ylim(ymin, ymax)
    ax.set_ylabel('$y_t$', fontsize=12)
    ax.set_xlabel('time', fontsize=12)
    ax.vlines((T,), -1.5, 1.5)

    ax.set_xticks((T,))
    ax.set_xticklabels(('$T$',))

    sample = []
    for i in range(sample_size):
        rcolor = random.choice(('c', 'g', 'b', 'k'))
        x, y = ar.simulate(ts_length=T+15)
        y = y.flatten()
        ax.plot(y, color=rcolor, lw=1, alpha=0.5)
        ax.plot((T,), (y[T],), 'ko', alpha=0.5)
        sample.append(y[T])

    y = y.flatten()
    axes[1].set_ylim(ymin, ymax)
    axes[1].set_ylabel('$y_t$', fontsize=12)
    axes[1].set_xlabel('relative frequency', fontsize=12)
    axes[1].hist(sample, bins=16, density=True, orientation='horizontal', alpha=0.5)
    plt.show()
```

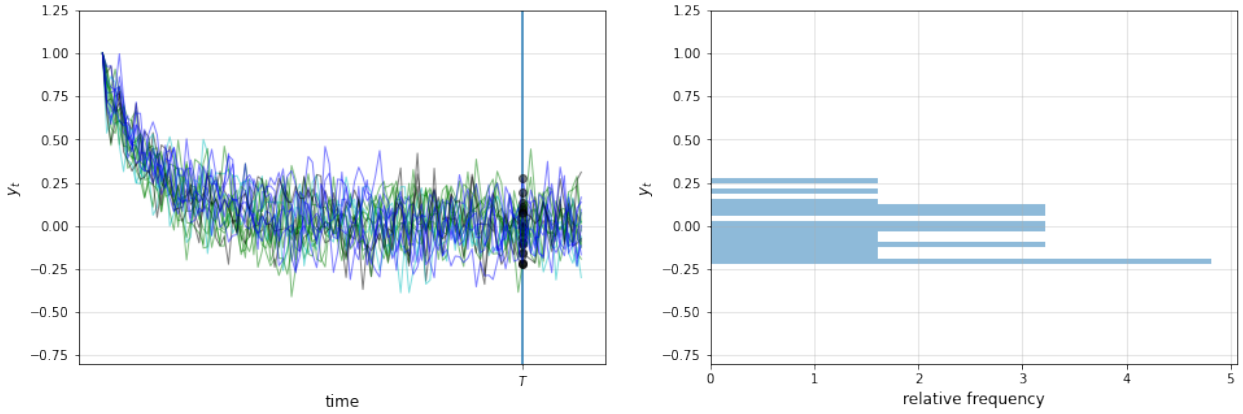
```
ϕ_1, ϕ_2, ϕ_3, ϕ_4 = 0.5, -0.2, 0, 0.5
σ = 0.1

A_2 = [[ϕ_1, ϕ_2, ϕ_3, ϕ_4],
       [1,     0,     0,     0],
       [0,     1,     0,     0],
       [0,     0,     1,     0]]

C_2 = [[σ], [0], [0], [0]]

G_2 = [1, 0, 0, 0]

cross_section_plot(A_2, C_2, G_2)
```



In the right-hand figure, these values are converted into a rotated histogram that shows relative frequencies from our sample of 20 y_T's.

Here is another figure, this time with 100 observations

```
t = 100
cross_section_plot(A_2, C_2, G_2, T=t)
```



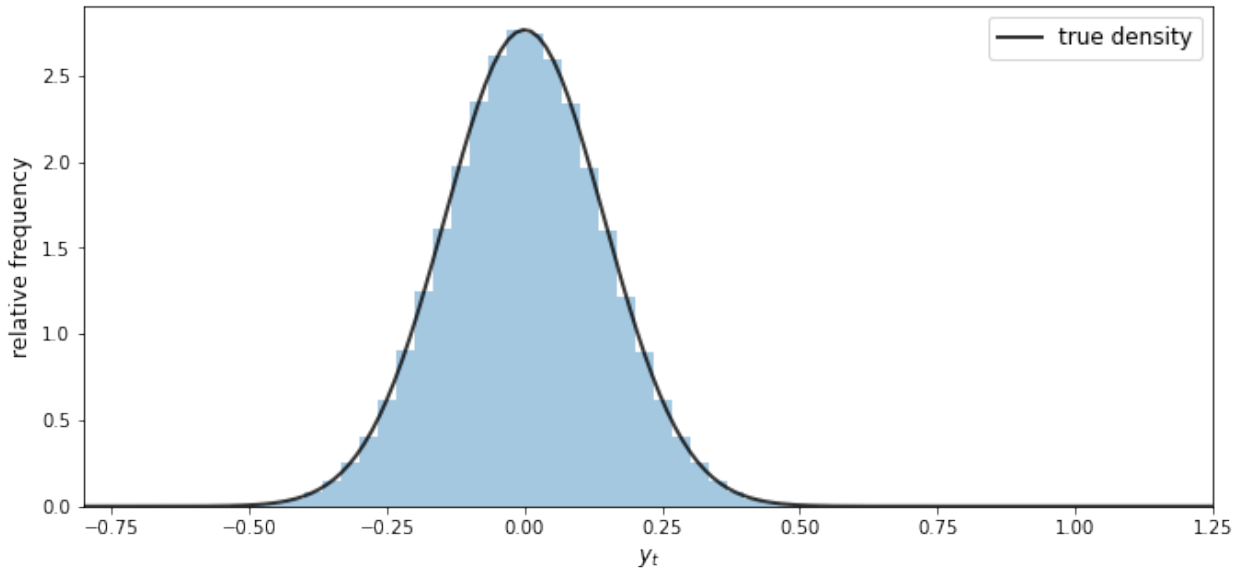
Let's now try with 500,000 observations, showing only the histogram (without rotation)

```
T = 100
ymin=-0.8
ymax=1.25
```

(continues on next page)

(continued from previous page)

```
sample_size = 500_000

ar = LinearStateSpace(A_2, C_2, G_2, mu_0=np.ones(4))
fig, ax = plt.subplots()
x, y = ar.simulate(sample_size)
mu_x, mu_y, Sigma_x, Sigma_y, Sigma_yx = ar.stationary_distributions()
f_y = norm(loc=float(mu_y), scale=float(np.sqrt(Sigma_y)))
y = y.flatten()
ygrid = np.linspace(ymin, ymax, 150)

ax.hist(y, bins=50, density=True, alpha=0.4)
ax.plot(ygrid, f_y.pdf(ygrid), 'k-', lw=2, alpha=0.8, label=r'true density')
ax.set_xlim(ymin, ymax)
ax.set_xlabel('$y_t$', fontsize=12)
ax.set_ylabel('relative frequency', fontsize=12)
ax.legend(fontsize=12)
plt.show()
```



The black line is the population density of y_T calculated from (10).

The histogram and population distribution are close, as expected.

By looking at the figures and experimenting with parameters, you will gain a feel for how the population distribution depends on the model primitives *listed above*, as intermediated by the distribution's sufficient statistics.

Ensemble Means

In the preceding figure, we approximated the population distribution of y_T by

1. generating I sample paths (i.e., time series) where I is a large number
2. recording each observation y_T^i
3. histogramming this sample

Just as the histogram approximates the population distribution, the *ensemble* or *cross-sectional average*

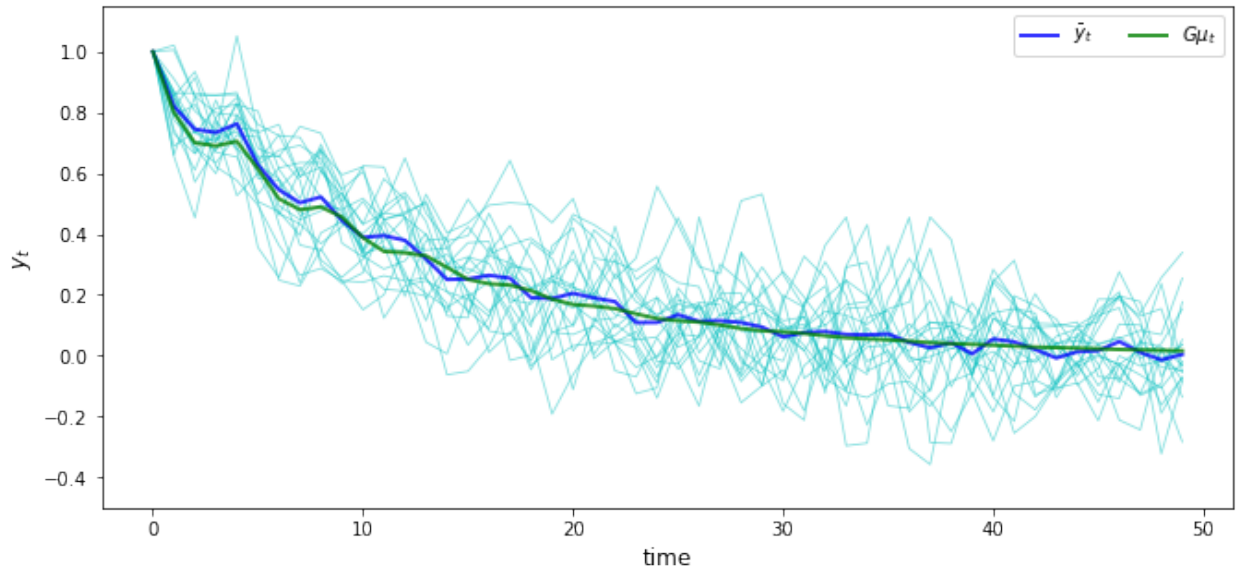
$$\bar{y}_T := \frac{1}{I} \sum_{i=1}^{I} y_T^i$$

approximates the expectation $\mathbb{E}[y_T] = G\mu_T$ (as implied by the law of large numbers).

Here's a simulation comparing the ensemble averages and population means at time points $t = 0, \ldots, 50$.

The parameters are the same as for the preceding figures, and the sample size is relatively small ($I = 20$).

```
I = 20
T = 50
ymin = -0.5
ymax = 1.15

ar = LinearStateSpace(A_2, C_2, G_2, mu_0=np.ones(4))

fig, ax = plt.subplots()

ensemble_mean = np.zeros(T)
for i in range(I):
    x, y = ar.simulate(ts_length=T)
    y = y.flatten()
    ax.plot(y, 'c-', lw=0.8, alpha=0.5)
    ensemble_mean = ensemble_mean + y

ensemble_mean = ensemble_mean / I
ax.plot(ensemble_mean, color='b', lw=2, alpha=0.8, label='$\\bar y_t$')
m = ar.moment_sequence()

population_means = []
for t in range(T):
    μ_x, μ_y, Σ_x, Σ_y = next(m)
    population_means.append(float(μ_y))

ax.plot(population_means, color='g', lw=2, alpha=0.8, label='$G\mu_t$')
ax.set_ylim(ymin, ymax)
ax.set_xlabel('time', fontsize=12)
ax.set_ylabel('$y_t$', fontsize=12)
ax.legend(ncol=2)
plt.show()
```



The ensemble mean for x_t is

$$\bar{x}_T := \frac{1}{I}\sum_{i=1}^{I} x_T^i \to \mu_T \qquad (I \to \infty)$$

The limit μ_T is a "long-run average".

(By *long-run average* we mean the average for an infinite ($I = \infty$) number of sample x_T's)

Another application of the law of large numbers assures us that

$$\frac{1}{I}\sum_{i=1}^{I}(x_T^i - \bar{x}_T)(x_T^i - \bar{x}_T)' \to \Sigma_T \qquad (I \to \infty)$$

19.3.4 Joint Distributions

In the preceding discussion, we looked at the distributions of x_t and y_t in isolation.

This gives us useful information but doesn't allow us to answer questions like

- what's the probability that $x_t \geq 0$ for all t?
- what's the probability that the process $\{y_t\}$ exceeds some value a before falling below b?
- etc., etc.

Such questions concern the *joint distributions* of these sequences.

To compute the joint distribution of $x_0, x_1, \ldots, x_T$, recall that joint and conditional densities are linked by the rule

$$p(x, y) = p(y \,|\, x)p(x) \qquad \text{(joint} = \text{conditional} \times \text{marginal)}$$

From this rule we get $p(x_0, x_1) = p(x_1 \,|\, x_0)p(x_0)$.

The Markov property $p(x_t \,|\, x_{t-1}, \ldots, x_0) = p(x_t \,|\, x_{t-1})$ and repeated applications of the preceding rule lead us to

$$p(x_0, x_1, \ldots, x_T) = p(x_0)\prod_{t=0}^{T-1} p(x_{t+1} \,|\, x_t)$$

The marginal $p(x_0)$ is just the primitive $N(\mu_0, \Sigma_0)$.

In view of (1), the conditional densities are

$$p(x_{t+1} \mid x_t) = N(Ax_t, CC')$$

Autocovariance Functions

An important object related to the joint distribution is the *autocovariance function*

$$\Sigma_{t+j,t} := \mathbb{E}[(x_{t+j} - \mu_{t+j})(x_t - \mu_t)'] \tag{11}$$

Elementary calculations show that

$$\Sigma_{t+j,t} = A^j \Sigma_t \tag{12}$$

Notice that $\Sigma_{t+j,t}$ in general depends on both j, the gap between the two dates, and t, the earlier date.

19.4 Stationarity and Ergodicity

Stationarity and ergodicity are two properties that, when they hold, greatly aid analysis of linear state space models.

Let's start with the intuition.

19.4.1 Visualizing Stability

Let's look at some more time series from the same model that we analyzed above.

This picture shows cross-sectional distributions for y at times T, T', T''

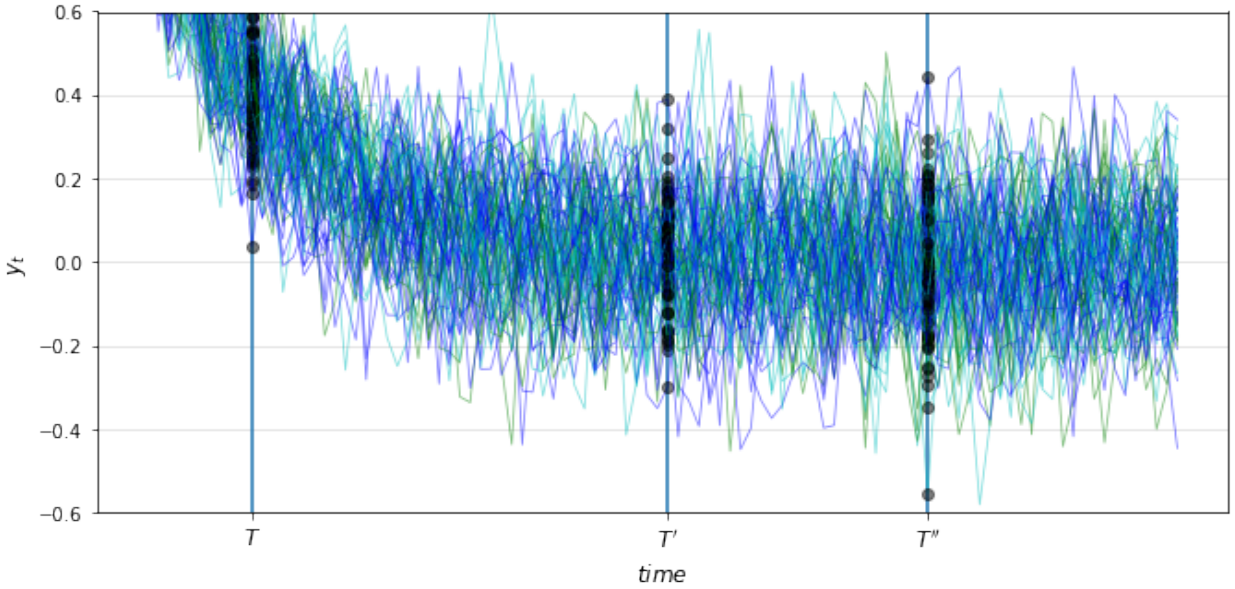
```
def cross_plot(A,
               C,
               G,
               steady_state='False',
               T0 = 10,
               T1 = 50,
               T2 = 75,
               T4 = 100):

    ar = LinearStateSpace(A, C, G, mu_0=np.ones(4))

    if steady_state == 'True':
        μ_x, μ_y, Σ_x, Σ_y, Σ_yx = ar.stationary_distributions()
        ar_state = LinearStateSpace(A, C, G, mu_0=μ_x, Sigma_0=Σ_x)

    ymin, ymax = -0.6, 0.6
    fig, ax = plt.subplots()
    ax.grid(alpha=0.4)
    ax.set_ylim(ymin, ymax)
    ax.set_ylabel('$y_t$', fontsize=12)
    ax.set_xlabel('$time$', fontsize=12)

    ax.vlines((T0, T1, T2), -1.5, 1.5)
    ax.set_xticks((T0, T1, T2))
```

(continues on next page)

(continued from previous page)

```
    ax.set_xticklabels(("$T$", "$T'$", "$T''$"), fontsize=12)
    for i in range(80):
        rcolor = random.choice(('c', 'g', 'b'))

        if steady_state == 'True':
            x, y = ar_state.simulate(ts_length=T4)
        else:
            x, y = ar.simulate(ts_length=T4)

        y = y.flatten()
        ax.plot(y, color=rcolor, lw=0.8, alpha=0.5)
        ax.plot((T0, T1, T2), (y[T0], y[T1], y[T2],), 'ko', alpha=0.5)
    plt.show()
```

```
cross_plot(A_2, C_2, G_2)
```



Note how the time series "settle down" in the sense that the distributions at T' and T'' are relatively similar to each other — but unlike the distribution at T.

Apparently, the distributions of y_t converge to a fixed long-run distribution as $t \to \infty$.

When such a distribution exists it is called a *stationary distribution*.

19.4.2 Stationary Distributions

In our setting, a distribution ψ_∞ is said to be *stationary* for x_t if

$$x_t \sim \psi_\infty \quad \text{and} \quad x_{t+1} = Ax_t + Cw_{t+1} \quad \implies \quad x_{t+1} \sim \psi_\infty$$

Since

1. in the present case, all distributions are Gaussian
2. a Gaussian distribution is pinned down by its mean and variance-covariance matrix

we can restate the definition as follows: ψ_∞ is stationary for x_t if

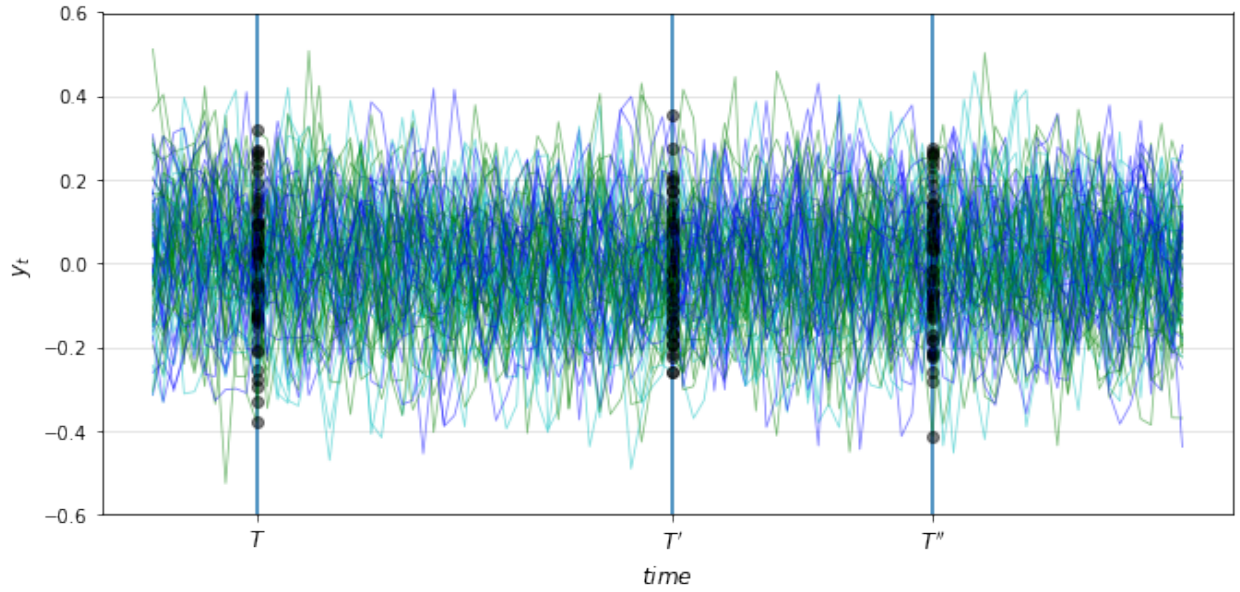
$$\psi_\infty = N(\mu_\infty, \Sigma_\infty)$$

where μ_∞ and Σ_∞ are fixed points of (4) and (5) respectively.

19.4.3 Covariance Stationary Processes

Let's see what happens to the preceding figure if we start x_0 at the stationary distribution.

```
cross_plot(A_2, C_2, G_2, steady_state='True')
```



Now the differences in the observed distributions at T, T' and T'' come entirely from random fluctuations due to the finite sample size.

By

- our choosing $x_0 \sim N(\mu_\infty, \Sigma_\infty)$
- the definitions of μ_∞ and Σ_∞ as fixed points of (4) and (5) respectively

we've ensured that

$$\mu_t = \mu_\infty \quad \text{and} \quad \Sigma_t = \Sigma_\infty \quad \text{for all } t$$

Moreover, in view of (12), the autocovariance function takes the form $\Sigma_{t+j,t} = A^j \Sigma_\infty$, which depends on j but not on t.

This motivates the following definition.

A process $\{x_t\}$ is said to be *covariance stationary* if

- both μ_t and Σ_t are constant in t
- $\Sigma_{t+j,t}$ depends on the time gap j but not on time t

In our setting, $\{x_t\}$ will be covariance stationary if μ_0, Σ_0, A, C assume values that imply that none of $\mu_t, \Sigma_t, \Sigma_{t+j,t}$ depends on t.

19.4.4 Conditions for Stationarity

The Globally Stable Case

The difference equation $\mu_{t+1} = A\mu_t$ is known to have *unique* fixed point $\mu_\infty = 0$ if all eigenvalues of A have moduli strictly less than unity.

That is, if `(np.absolute(np.linalg.eigvals(A)) < 1).all() == True`.

The difference equation (5) also has a unique fixed point in this case, and, moreover

$$\mu_t \to \mu_\infty = 0 \quad \text{and} \quad \Sigma_t \to \Sigma_\infty \quad \text{as} \quad t \to \infty$$

regardless of the initial conditions μ_0 and Σ_0.

This is the *globally stable case* — see these notes for more a theoretical treatment.

However, global stability is more than we need for stationary solutions, and often more than we want.

To illustrate, consider *our second order difference equation example*.

Here the state is $x_t = \begin{bmatrix} 1 & y_t & y_{t-1} \end{bmatrix}'$.

Because of the constant first component in the state vector, we will never have $\mu_t \to 0$.

How can we find stationary solutions that respect a constant state component?

Processes with a Constant State Component

To investigate such a process, suppose that A and C take the form

$$A = \begin{bmatrix} A_1 & a \\ 0 & 1 \end{bmatrix} \qquad C = \begin{bmatrix} C_1 \\ 0 \end{bmatrix}$$

where

- A_1 is an $(n-1) \times (n-1)$ matrix
- a is an $(n-1) \times 1$ column vector

Let $x_t = \begin{bmatrix} x_{1t}' & 1 \end{bmatrix}'$ where x_{1t} is $(n-1) \times 1$.

It follows that

$$x_{1,t+1} = A_1 x_{1t} + a + C_1 w_{t+1}$$

Let $\mu_{1t} = \mathbb{E}[x_{1t}]$ and take expectations on both sides of this expression to get

$$\mu_{1,t+1} = A_1 \mu_{1,t} + a \tag{13}$$

Assume now that the moduli of the eigenvalues of A_1 are all strictly less than one.

Then (13) has a unique stationary solution, namely,

$$\mu_{1\infty} = (I - A_1)^{-1} a$$

The stationary value of μ_t itself is then $\mu_\infty := \begin{bmatrix} \mu_{1\infty}' & 1 \end{bmatrix}'$.

The stationary values of Σ_t and $\Sigma_{t+j,t}$ satisfy

$$\begin{aligned} \Sigma_\infty &= A \Sigma_\infty A' + CC' \\ \Sigma_{t+j,t} &= A^j \Sigma_\infty \end{aligned}$$

Notice that here $\Sigma_{t+j,t}$ depends on the time gap j but not on calendar time t.

In conclusion, if

- $x_0 \sim N(\mu_\infty, \Sigma_\infty)$ and
- the moduli of the eigenvalues of A_1 are all strictly less than unity

then the $\{x_t\}$ process is covariance stationary, with constant state component.

Note: If the eigenvalues of A_1 are less than unity in modulus, then (a) starting from any initial value, the mean and variance-covariance matrix both converge to their stationary values; and (b) iterations on (5) converge to the fixed point of the *discrete Lyapunov equation* in the first line of (14).

19.4.5 Ergodicity

Let's suppose that we're working with a covariance stationary process.

In this case, we know that the ensemble mean will converge to μ_∞ as the sample size I approaches infinity.

Averages over Time

Ensemble averages across simulations are interesting theoretically, but in real life, we usually observe only a *single* realization $\{x_t, y_t\}_{t=0}^T$.

So now let's take a single realization and form the time-series averages

$$\bar{x} := \frac{1}{T}\sum_{t=1}^{T} x_t \quad \text{and} \quad \bar{y} := \frac{1}{T}\sum_{t=1}^{T} y_t$$

Do these time series averages converge to something interpretable in terms of our basic state-space representation?

The answer depends on something called *ergodicity*.

Ergodicity is the property that time series and ensemble averages coincide.

More formally, ergodicity implies that time series sample averages converge to their expectation under the stationary distribution.

In particular,

- $\frac{1}{T}\sum_{t=1}^{T} x_t \to \mu_\infty$
- $\frac{1}{T}\sum_{t=1}^{T} (x_t - \bar{x}_T)(x_t - \bar{x}_T)' \to \Sigma_\infty$
- $\frac{1}{T}\sum_{t=1}^{T} (x_{t+j} - \bar{x}_T)(x_t - \bar{x}_T)' \to A^j \Sigma_\infty$

In our linear Gaussian setting, any covariance stationary process is also ergodic.

19.5 Noisy Observations

In some settings, the observation equation $y_t = Gx_t$ is modified to include an error term.

Often this error term represents the idea that the true state can only be observed imperfectly.

To include an error term in the observation we introduce

- An IID sequence of $\ell \times 1$ random vectors $v_t \sim N(0, I)$.
- A $k \times \ell$ matrix H.

and extend the linear state-space system to

$$\begin{aligned} x_{t+1} &= Ax_t + Cw_{t+1} \\ y_t &= Gx_t + Hv_t \\ x_0 &\sim N(\mu_0, \Sigma_0) \end{aligned}$$

The sequence $\{v_t\}$ is assumed to be independent of $\{w_t\}$.

The process $\{x_t\}$ is not modified by noise in the observation equation and its moments, distributions and stability properties remain the same.

The unconditional moments of y_t from (6) and (7) now become

$$\mathbb{E}[y_t] = \mathbb{E}[Gx_t + Hv_t] = G\mu_t \tag{14}$$

The variance-covariance matrix of y_t is easily shown to be

$$\text{Var}[y_t] = \text{Var}[Gx_t + Hv_t] = G\Sigma_t G' + HH' \tag{15}$$

The distribution of y_t is therefore

$$y_t \sim N(G\mu_t, G\Sigma_t G' + HH')$$

19.6 Prediction

The theory of prediction for linear state space systems is elegant and simple.

19.6.1 Forecasting Formulas – Conditional Means

The natural way to predict variables is to use conditional distributions.

For example, the optimal forecast of x_{t+1} given information known at time t is

$$\mathbb{E}_t[x_{t+1}] := \mathbb{E}[x_{t+1} \mid x_t, x_{t-1}, \ldots, x_0] = Ax_t$$

The right-hand side follows from $x_{t+1} = Ax_t + Cw_{t+1}$ and the fact that w_{t+1} is zero mean and independent of $x_t, x_{t-1}, \ldots, x_0$.

That $\mathbb{E}_t[x_{t+1}] = \mathbb{E}[x_{t+1} \mid x_t]$ is an implication of $\{x_t\}$ having the *Markov property*.

The one-step-ahead forecast error is

$$x_{t+1} - \mathbb{E}_t[x_{t+1}] = Cw_{t+1}$$

The covariance matrix of the forecast error is

$$\mathbb{E}[(x_{t+1} - \mathbb{E}_t[x_{t+1}])(x_{t+1} - \mathbb{E}_t[x_{t+1}])'] = CC'$$

More generally, we'd like to compute the j-step ahead forecasts $\mathbb{E}_t[x_{t+j}]$ and $\mathbb{E}_t[y_{t+j}]$.

With a bit of algebra, we obtain

$$x_{t+j} = A^j x_t + A^{j-1} C w_{t+1} + A^{j-2} C w_{t+2} + \cdots + A^0 C w_{t+j}$$

In view of the IID property, current and past state values provide no information about future values of the shock.

Hence $\mathbb{E}_t[w_{t+k}] = \mathbb{E}[w_{t+k}] = 0$.

It now follows from linearity of expectations that the j-step ahead forecast of x is

$$\mathbb{E}_t[x_{t+j}] = A^j x_t$$

The j-step ahead forecast of y is therefore

$$\mathbb{E}_t[y_{t+j}] = \mathbb{E}_t[G x_{t+j} + H v_{t+j}] = G A^j x_t$$

19.6.2 Covariance of Prediction Errors

It is useful to obtain the covariance matrix of the vector of j-step-ahead prediction errors

$$x_{t+j} - \mathbb{E}_t[x_{t+j}] = \sum_{s=0}^{j-1} A^s C w_{t-s+j} \tag{16}$$

Evidently,

$$V_j := \mathbb{E}_t[(x_{t+j} - \mathbb{E}_t[x_{t+j}])(x_{t+j} - \mathbb{E}_t[x_{t+j}])'] = \sum_{k=0}^{j-1} A^k C C' A^{k'} \tag{17}$$

V_j defined in (17) can be calculated recursively via $V_1 = CC'$ and

$$V_j = CC' + A V_{j-1} A', \quad j \geq 2 \tag{18}$$

V_j is the *conditional covariance matrix* of the errors in forecasting x_{t+j}, conditioned on time t information x_t.

Under particular conditions, V_j converges to

$$V_\infty = CC' + A V_\infty A' \tag{19}$$

Equation (19) is an example of a *discrete Lyapunov* equation in the covariance matrix V_∞.

A sufficient condition for V_j to converge is that the eigenvalues of A be strictly less than one in modulus.

Weaker sufficient conditions for convergence associate eigenvalues equaling or exceeding one in modulus with elements of C that equal 0.

19.7 Code

Our preceding simulations and calculations are based on code in the file lss.py from the QuantEcon.py package.

The code implements a class for handling linear state space models (simulations, calculating moments, etc.).

One Python construct you might not be familiar with is the use of a generator function in the method `moment_sequence()`.

Go back and read the relevant documentation if you've forgotten how generator functions work.

Examples of usage are given in the solutions to the exercises.

19.8 Exercises

19.8.1 Exercise 1

In several contexts, we want to compute forecasts of geometric sums of future random variables governed by the linear state-space system (1).

We want the following objects

- Forecast of a geometric sum of future x's, or $\mathbb{E}_t\left[\sum_{j=0}^{\infty}\beta^j x_{t+j}\right]$.
- Forecast of a geometric sum of future y's, or $\mathbb{E}_t\left[\sum_{j=0}^{\infty}\beta^j y_{t+j}\right]$.

These objects are important components of some famous and interesting dynamic models.

For example,

- if $\{y_t\}$ is a stream of dividends, then $\mathbb{E}\left[\sum_{j=0}^{\infty}\beta^j y_{t+j}|x_t\right]$ is a model of a stock price
- if $\{y_t\}$ is the money supply, then $\mathbb{E}\left[\sum_{j=0}^{\infty}\beta^j y_{t+j}|x_t\right]$ is a model of the price level

Show that:

$$\mathbb{E}_t\left[\sum_{j=0}^{\infty}\beta^j x_{t+j}\right] = [I-\beta A]^{-1}x_t$$

and

$$\mathbb{E}_t\left[\sum_{j=0}^{\infty}\beta^j y_{t+j}\right] = G[I-\beta A]^{-1}x_t$$

what must the modulus for every eigenvalue of A be less than?

19.9 Solutions

19.9.1 Exercise 1

Suppose that every eigenvalue of A has modulus strictly less than $\frac{1}{\beta}$.

It *then follows* that $I+\beta A+\beta^2 A^2+\cdots=[I-\beta A]^{-1}$.

This leads to our formulas:

- Forecast of a geometric sum of future x's

$$\mathbb{E}_t\left[\sum_{j=0}^{\infty}\beta^j x_{t+j}\right] = [I+\beta A+\beta^2 A^2+\cdots]x_t = [I-\beta A]^{-1}x_t$$

- Forecast of a geometric sum of future y's

$$\mathbb{E}_t \left[\sum_{j=0}^{\infty} \beta^j y_{t+j} \right] = G[I + \beta A + \beta^2 A^2 + \cdots] x_t = G[I - \beta A]^{-1} x_t$$

CHAPTER

TWENTY

APPLICATION: THE SAMUELSON MULTIPLIER-ACCELERATOR

Contents

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

```
!conda install -y quantecon
```

20.1 Overview

This lecture creates non-stochastic and stochastic versions of Paul Samuelson's celebrated multiplier accelerator model [Sam39].

In doing so, we extend the example of the Solow model class in our second OOP lecture.

Our objectives are to

- provide a more detailed example of OOP and classes
- review a famous model
- review linear difference equations, both deterministic and stochastic

Let's start with some standard imports:

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

We'll also use the following for various tasks described below:

```
from quantecon import LinearStateSpace
import cmath
import math
import sympy
from sympy import Symbol, init_printing
from cmath import sqrt
```

20.1.1 Samuelson's Model

Samuelson used a *second-order linear difference equation* to represent a model of national output based on three components:

- a *national output identity* asserting that national output or national income is the sum of consumption plus investment plus government purchases.
- a Keynesian *consumption function* asserting that consumption at time t is equal to a constant times national output at time $t-1$.
- an investment *accelerator* asserting that investment at time t equals a constant called the *accelerator coefficient* times the difference in output between period $t-1$ and $t-2$.

Consumption plus investment plus government purchases constitute *aggregate demand,* which automatically calls forth an equal amount of *aggregate supply*.

(To read about linear difference equations see here or chapter IX of [Sar87].)

Samuelson used the model to analyze how particular values of the marginal propensity to consume and the accelerator coefficient might give rise to transient *business cycles* in national output.

Possible dynamic properties include

- smooth convergence to a constant level of output
- damped business cycles that eventually converge to a constant level of output
- persistent business cycles that neither dampen nor explode

Later we present an extension that adds a random shock to the right side of the national income identity representing random fluctuations in aggregate demand.

This modification makes national output become governed by a second-order *stochastic linear difference equation* that, with appropriate parameter values, gives rise to recurrent irregular business cycles.

(To read about stochastic linear difference equations see chapter XI of [Sar87].)

20.2 Details

Let's assume that

- $\{G_t\}$ is a sequence of levels of government expenditures – we'll start by setting $G_t = G$ for all t.
- $\{C_t\}$ is a sequence of levels of aggregate consumption expenditures, a key endogenous variable in the model.
- $\{I_t\}$ is a sequence of rates of investment, another key endogenous variable.
- $\{Y_t\}$ is a sequence of levels of national income, yet another endogenous variable.
- a is the marginal propensity to consume in the Keynesian consumption function $C_t = aY_{t-1} + \gamma$.
- b is the "accelerator coefficient" in the "investment accelerator" $I_t = b(Y_{t-1} - Y_{t-2})$.
- $\{\epsilon_t\}$ is an IID sequence standard normal random variables.
- $\sigma \geq 0$ is a "volatility" parameter — setting $\sigma = 0$ recovers the non-stochastic case that we'll start with.

The model combines the consumption function

$$C_t = aY_{t-1} + \gamma \tag{1}$$

with the investment accelerator

$$I_t = b(Y_{t-1} - Y_{t-2}) \tag{2}$$

and the national income identity

$$Y_t = C_t + I_t + G_t \tag{3}$$

- The parameter a is peoples' *marginal propensity to consume* out of income - equation (1) asserts that people consume a fraction of $a \in (0, 1)$ of each additional dollar of income.
- The parameter $b > 0$ is the investment accelerator coefficient - equation (2) asserts that people invest in physical capital when income is increasing and disinvest when it is decreasing.

Equations (1), (2), and (3) imply the following second-order linear difference equation for national income:

$$Y_t = (a + b)Y_{t-1} - bY_{t-2} + (\gamma + G_t)$$

or

$$Y_t = \rho_1 Y_{t-1} + \rho_2 Y_{t-2} + (\gamma + G_t) \tag{4}$$

where $\rho_1 = (a + b)$ and $\rho_2 = -b$.

To complete the model, we require two **initial conditions**.

If the model is to generate time series for $t = 0, \ldots, T$, we require initial values

$$Y_{-1} = \bar{Y}_{-1}, \quad Y_{-2} = \bar{Y}_{-2}$$

We'll ordinarily set the parameters (a, b) so that starting from an arbitrary pair of initial conditions $(\bar{Y}_{-1}, \bar{Y}_{-2})$, national income Y_t converges to a constant value as t becomes large.

We are interested in studying

- the transient fluctuations in Y_t as it converges to its **steady state** level
- the **rate** at which it converges to a steady state level

The deterministic version of the model described so far — meaning that no random shocks hit aggregate demand — has only transient fluctuations.

We can convert the model to one that has persistent irregular fluctuations by adding a random shock to aggregate demand.

20.2.1 Stochastic Version of the Model

We create a **random** or **stochastic** version of the model by adding a random process of **shocks** or **disturbances** $\{\sigma\epsilon_t\}$ to the right side of equation (4), leading to the **second-order scalar linear stochastic difference equation**:

$$Y_t = G_t + a(1-b)Y_{t-1} - abY_{t-2} + \sigma\epsilon_t \tag{5}$$

20.2.2 Mathematical Analysis of the Model

To get started, let's set $G_t \equiv 0$, $\sigma = 0$, and $\gamma = 0$.

Then we can write equation (5) as

$$Y_t = \rho_1 Y_{t-1} + \rho_2 Y_{t-2}$$

or

$$Y_{t+2} - \rho_1 Y_{t+1} - \rho_2 Y_t = 0 \tag{6}$$

To discover the properties of the solution of (6), it is useful first to form the **characteristic polynomial** for (6):

$$z^2 - \rho_1 z - \rho_2 \tag{7}$$

where z is possibly a complex number.

We want to find the two **zeros** (a.k.a. **roots**) – namely λ_1, λ_2 – of the characteristic polynomial.

These are two special values of z, say $z = \lambda_1$ and $z = \lambda_2$, such that if we set z equal to one of these values in expression (7), the characteristic polynomial (7) equals zero:

$$z^2 - \rho_1 z - \rho_2 = (z - \lambda_1)(z - \lambda_2) = 0 \tag{8}$$

Equation (8) is said to **factor** the characteristic polynomial.

When the roots are complex, they will occur as a complex conjugate pair.

When the roots are complex, it is convenient to represent them in the polar form

$$\lambda_1 = re^{i\omega}, \ \lambda_2 = re^{-i\omega}$$

where r is the *amplitude* of the complex number and ω is its *angle* or *phase*.

These can also be represented as

$$\lambda_1 = r(cos(\omega) + i \sin(\omega))$$

$$\lambda_2 = r(cos(\omega) - i \sin(\omega))$$

(To read about the polar form, see here)

Given **initial conditions** Y_{-1}, Y_{-2}, we want to generate a **solution** of the difference equation (6).

It can be represented as

$$Y_t = \lambda_1^t c_1 + \lambda_2^t c_2$$

where c_1 and c_2 are constants that depend on the two initial conditions and on ρ_1, ρ_2.

When the roots are complex, it is useful to pursue the following calculations.

Notice that

$$\begin{aligned} Y_t &= c_1(re^{i\omega})^t + c_2(re^{-i\omega})^t \\ &= c_1 r^t e^{i\omega t} + c_2 r^t e^{-i\omega t} \\ &= c_1 r^t[\cos(\omega t) + i\sin(\omega t)] + c_2 r^t[\cos(\omega t) - i\sin(\omega t)] \\ &= (c_1 + c_2) r^t \cos(\omega t) + i(c_1 - c_2) r^t \sin(\omega t) \end{aligned}$$

The only way that Y_t can be a real number for each t is if $c_1 + c_2$ is a real number and $c_1 - c_2$ is an imaginary number.

This happens only when c_1 and c_2 are complex conjugates, in which case they can be written in the polar forms

$$c_1 = ve^{i\theta}, \;\; c_2 = ve^{-i\theta}$$

So we can write

$$\begin{aligned} Y_t &= ve^{i\theta} r^t e^{i\omega t} + ve^{-i\theta} r^t e^{-i\omega t} \\ &= vr^t[e^{i(\omega t+\theta)} + e^{-i(\omega t+\theta)}] \\ &= 2vr^t \cos(\omega t + \theta) \end{aligned}$$

where v and θ are constants that must be chosen to satisfy initial conditions for Y_{-1}, Y_{-2}.

This formula shows that when the roots are complex, Y_t displays oscillations with **period** $\breve{p} = \frac{2\pi}{\omega}$ and **damping factor** r.

We say that $\breve{p}$ is the **period** because in that amount of time the cosine wave $\cos(\omega t + \theta)$ goes through exactly one complete cycles.

(Draw a cosine function to convince yourself of this please)

Remark: Following [Sam39], we want to choose the parameters a, b of the model so that the absolute values (of the possibly complex) roots λ_1, λ_2 of the characteristic polynomial are both strictly less than one:

$$|\lambda_j| < 1 \qquad \text{for } j = 1, 2$$

Remark: When both roots λ_1, λ_2 of the characteristic polynomial have absolute values strictly less than one, the absolute value of the larger one governs the rate of convergence to the steady state of the non stochastic version of the model.

20.2.3 Things This Lecture Does

We write a function to generate simulations of a $\{Y_t\}$ sequence as a function of time.

The function requires that we put in initial conditions for Y_{-1}, Y_{-2}.

The function checks that a, b are set so that λ_1, λ_2 are less than unity in absolute value (also called "modulus").

The function also tells us whether the roots are complex, and, if they are complex, returns both their real and complex parts.

If the roots are both real, the function returns their values.

We use our function written to simulate paths that are stochastic (when $\sigma > 0$).

We have written the function in a way that allows us to input $\{G_t\}$ paths of a few simple forms, e.g.,

- one time jumps in G at some time
- a permanent jump in G that occurs at some time

We proceed to use the Samuelson multiplier-accelerator model as a laboratory to make a simple OOP example.

The "state" that determines next period's Y_{t+1} is now not just the current value Y_t but also the once lagged value Y_{t-1}.

This involves a little more bookkeeping than is required in the Solow model class definition.

We use the Samuelson multiplier-accelerator model as a vehicle for teaching how we can gradually add more features to the class.

We want to have a method in the class that automatically generates a simulation, either non-stochastic ($\sigma = 0$) or stochastic ($\sigma > 0$).

We also show how to map the Samuelson model into a simple instance of the `LinearStateSpace` class described here.

We can use a `LinearStateSpace` instance to do various things that we did above with our homemade function and class.

Among other things, we show by example that the eigenvalues of the matrix A that we use to form the instance of the `LinearStateSpace` class for the Samuelson model equal the roots of the characteristic polynomial (7) for the Samuelson multiplier accelerator model.

Here is the formula for the matrix A in the linear state space system in the case that government expenditures are a constant G:

$$A = \begin{bmatrix} 1 & 0 & 0 \\ \gamma + G & \rho_1 & \rho_2 \\ 0 & 1 & 0 \end{bmatrix}$$

20.3 Implementation

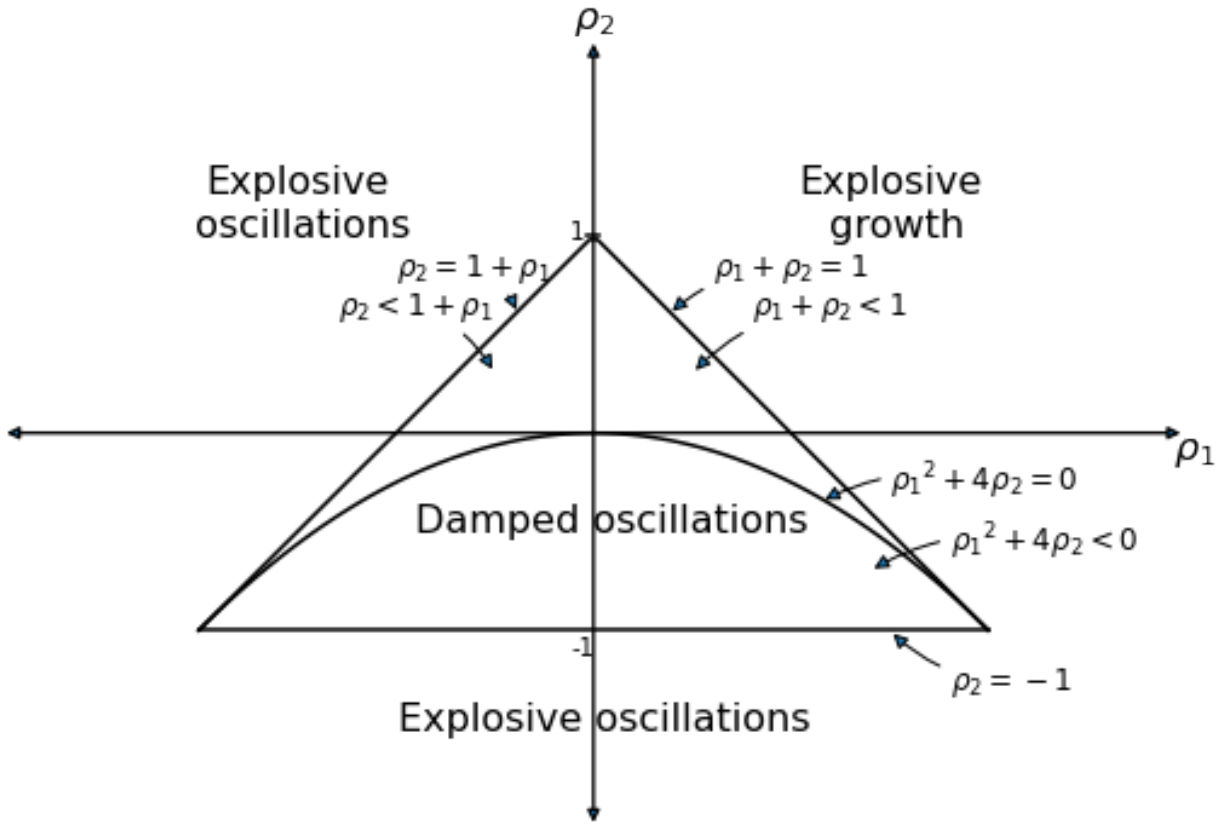
We'll start by drawing an informative graph from page 189 of [Sar87]

```
def param_plot():

    """This function creates the graph on page 189 of
    Sargent Macroeconomic Theory, second edition, 1987.
    """

    fig, ax = plt.subplots(figsize=(10, 6))
    ax.set_aspect('equal')

    # Set axis
    xmin, ymin = -3, -2
    xmax, ymax = -xmin, -ymin
    plt.axis([xmin, xmax, ymin, ymax])

    # Set axis labels
    ax.set(xticks=[], yticks=[])
    ax.set_xlabel(r'$\rho_2$', fontsize=16)
    ax.xaxis.set_label_position('top')
    ax.set_ylabel(r'$\rho_1$', rotation=0, fontsize=16)
    ax.yaxis.set_label_position('right')

    # Draw (t1, t2) points
    ρ1 = np.linspace(-2, 2, 100)
    ax.plot(ρ1, -abs(ρ1) + 1, c='black')
    ax.plot(ρ1, np.full_like(ρ1, -1), c='black')
    ax.plot(ρ1, -(ρ1**2 / 4), c='black')

    # Turn normal axes off
    for spine in ['left', 'bottom', 'top', 'right']:
        ax.spines[spine].set_visible(False)
```

(continues on next page)

(continued from previous page)

```
    # Add arrows to represent axes
    axes_arrows = {'arrowstyle': '<|-|>', 'lw': 1.3}
    ax.annotate('', xy=(xmin, 0), xytext=(xmax, 0), arrowprops=axes_arrows)
    ax.annotate('', xy=(0, ymin), xytext=(0, ymax), arrowprops=axes_arrows)

    # Annotate the plot with equations
    plot_arrowsl = {'arrowstyle': '-|>', 'connectionstyle': "arc3, rad=-0.2"}
    plot_arrowsr = {'arrowstyle': '-|>', 'connectionstyle': "arc3, rad=0.2"}
    ax.annotate(r'$\rho_1 + \rho_2 < 1$', xy=(0.5, 0.3), xytext=(0.8, 0.6),
                arrowprops=plot_arrowsr, fontsize='12')
    ax.annotate(r'$\rho_1 + \rho_2 = 1$', xy=(0.38, 0.6), xytext=(0.6, 0.8),
                arrowprops=plot_arrowsr, fontsize='12')
    ax.annotate(r'$\rho_2 < 1 + \rho_1$', xy=(-0.5, 0.3), xytext=(-1.3, 0.6),
                arrowprops=plot_arrowsl, fontsize='12')
    ax.annotate(r'$\rho_2 = 1 + \rho_1$', xy=(-0.38, 0.6), xytext=(-1, 0.8),
                arrowprops=plot_arrowsl, fontsize='12')
    ax.annotate(r'$\rho_2 = -1$', xy=(1.5, -1), xytext=(1.8, -1.3),
                arrowprops=plot_arrowsl, fontsize='12')
    ax.annotate(r'${\rho_1}^2 + 4\rho_2 = 0$', xy=(1.15, -0.35),
                xytext=(1.5, -0.3), arrowprops=plot_arrowsr, fontsize='12')
    ax.annotate(r'${\rho_1}^2 + 4\rho_2 < 0$', xy=(1.4, -0.7),
                xytext=(1.8, -0.6), arrowprops=plot_arrowsr, fontsize='12')

    # Label categories of solutions
    ax.text(1.5, 1, 'Explosive\n growth', ha='center', fontsize=16)
    ax.text(-1.5, 1, 'Explosive\n oscillations', ha='center', fontsize=16)
    ax.text(0.05, -1.5, 'Explosive oscillations', ha='center', fontsize=16)
    ax.text(0.09, -0.5, 'Damped oscillations', ha='center', fontsize=16)

    # Add small marker to y-axis
    ax.axhline(y=1.005, xmin=0.495, xmax=0.505, c='black')
    ax.text(-0.12, -1.12, '-1', fontsize=10)
    ax.text(-0.12, 0.98, '1', fontsize=10)

    return fig

param_plot()
plt.show()
```



The graph portrays regions in which the (λ_1, λ_2) root pairs implied by the $(\rho_1 = (a + b), \rho_2 = -b)$ difference equation parameter pairs in the Samuelson model are such that:

- (λ_1, λ_2) are complex with modulus less than 1 - in this case, the $\{Y_t\}$ sequence displays damped oscillations.
- (λ_1, λ_2) are both real, but one is strictly greater than 1 - this leads to explosive growth.
- (λ_1, λ_2) are both real, but one is strictly less than -1 - this leads to explosive oscillations.
- (λ_1, λ_2) are both real and both are less than 1 in absolute value - in this case, there is smooth convergence to the steady state without damped cycles.

Later we'll present the graph with a red mark showing the particular point implied by the setting of (a, b).

20.3.1 Function to Describe Implications of Characteristic Polynomial

```
def categorize_solution(ρ1, ρ2):

    """This function takes values of ρ1 and ρ2 and uses them
    to classify the type of solution
    """

    discriminant = ρ1 ** 2 + 4 * ρ2
    if ρ2 > 1 + ρ1 or ρ2 < -1:
        print('Explosive oscillations')
    elif ρ1 + ρ2 > 1:
        print('Explosive growth')
    elif discriminant < 0:
```

(continues on next page)

(continued from previous page)

```
        print('Roots are complex with modulus less than one; \
therefore damped oscillations')
    else:
        print('Roots are real and absolute values are less than one; \
therefore get smooth convergence to a steady state')
```

```
### Test the categorize_solution function

categorize_solution(1.3, -.4)
```

```
Roots are real and absolute values are less than one; therefore get smooth␣
↪convergence to a steady state
```

20.3.2 Function for Plotting Paths

A useful function for our work below is

```
def plot_y(function=None):

    """Function plots path of Y_t"""

    plt.subplots(figsize=(10, 6))
    plt.plot(function)
    plt.xlabel('Time $t$')
    plt.ylabel('$Y_t$', rotation=0)
    plt.grid()
    plt.show()
```

20.3.3 Manual or "by hand" Root Calculations

The following function calculates roots of the characteristic polynomial using high school algebra.

(We'll calculate the roots in other ways later)

The function also plots a Y_t starting from initial conditions that we set

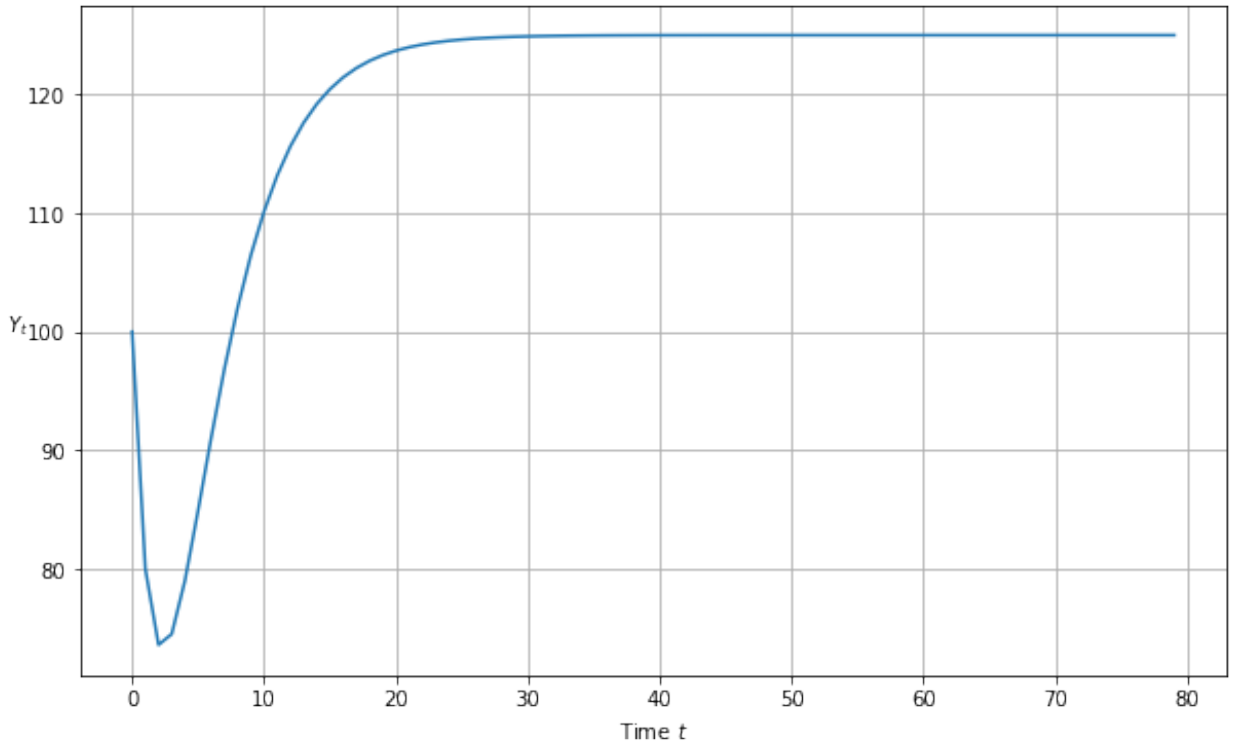
```
# This is a 'manual' method

def y_nonstochastic(y_0=100, y_1=80, α=.92, β=.5, γ=10, n=80):

    """Takes values of parameters and computes the roots of characteristic
    polynomial. It tells whether they are real or complex and whether they
    are less than unity in absolute value.It also computes a simulation of
    length n starting from the two given initial conditions for national
    income
    """

    roots = []

    ρ1 = α + β
    ρ2 = -β

    print(f'ρ_1 is {ρ1}')
```

(continues on next page)

(continued from previous page)

```
    print(f'ρ_2 is {ρ2}')

    discriminant = ρ1 ** 2 + 4 * ρ2

    if discriminant == 0:
        roots.append(-ρ1 / 2)
        print('Single real root: ')
        print(''.join(str(roots)))
    elif discriminant > 0:
        roots.append((-ρ1 + sqrt(discriminant).real) / 2)
        roots.append((-ρ1 - sqrt(discriminant).real) / 2)
        print('Two real roots: ')
        print(''.join(str(roots)))
    else:
        roots.append((-ρ1 + sqrt(discriminant)) / 2)
        roots.append((-ρ1 - sqrt(discriminant)) / 2)
        print('Two complex roots: ')
        print(''.join(str(roots)))

    if all(abs(root) < 1 for root in roots):
        print('Absolute values of roots are less than one')
    else:
        print('Absolute values of roots are not less than one')

    def transition(x, t): return ρ1 * x[t - 1] + ρ2 * x[t - 2] + y

    y_t = [y_0, y_1]

    for t in range(2, n):
        y_t.append(transition(y_t, t))

    return y_t

plot_y(y_nonstochastic())
```

```
ρ_1 is 1.42
ρ_2 is -0.5
Two real roots:
[-0.6459687576256715, -0.7740312423743284]
Absolute values of roots are less than one
```



20.3.4 Reverse-Engineering Parameters to Generate Damped Cycles

The next cell writes code that takes as inputs the modulus r and phase ϕ of a conjugate pair of complex numbers in polar form

$$\lambda_1 = r\exp(i\phi), \quad \lambda_2 = r\exp(-i\phi)$$

- The code assumes that these two complex numbers are the roots of the characteristic polynomial
- It then reverse-engineers (a, b) and (ρ_1, ρ_2), pairs that would generate those roots

```
### code to reverse-engineer a cycle
### y_t = r^t (c_1 cos(ϕ t) + c2 sin(ϕ t))
###

def f(r, ϕ):
    """
    Takes modulus r and angle ϕ of complex number r exp(j ϕ)
    and creates ρ1 and ρ2 of characteristic polynomial for which
    r exp(j ϕ) and r exp(- j ϕ) are complex roots.

    Returns the multiplier coefficient a and the accelerator coefficient b
    that verifies those roots.
    """
    g1 = cmath.rect(r, ϕ)  # Generate two complex roots
    g2 = cmath.rect(r, -ϕ)
    ρ1 = g1 + g2           # Implied ρ1, ρ2
    ρ2 = -g1 * g2
    b = -ρ2                # Reverse-engineer a and b that validate these
    a = ρ1 - b
```

(continues on next page)

(continued from previous page)

```
    return ρ1, ρ2, a, b

## Now let's use the function in an example
## Here are the example parameters

r = .95
period = 10                 # Length of cycle in units of time
ϕ = 2 * math.pi/period

## Apply the function

ρ1, ρ2, a, b = f(r, ϕ)

print(f"a, b = {a}, {b}")
print(f"ρ1, ρ2 = {ρ1}, {ρ2}")
```

```
a, b = (0.6346322893124001+0j), (0.9024999999999999-0j)
ρ1, ρ2 = (1.5371322893124+0j), (-0.9024999999999999+0j)
```

```
## Print the real components of ρ1 and ρ2

ρ1 = ρ1.real
ρ2 = ρ2.real

ρ1, ρ2
```

```
(1.5371322893124, -0.9024999999999999)
```

20.3.5 Root Finding Using Numpy

Here we'll use numpy to compute the roots of the characteristic polynomial

```
r1, r2 = np.roots([1, -ρ1, -ρ2])

p1 = cmath.polar(r1)
p2 = cmath.polar(r2)

print(f"r, ϕ = {r}, {ϕ}")
print(f"p1, p2 = {p1}, {p2}")
# print(f"g1, g2 = {g1}, {g2}")

print(f"a, b = {a}, {b}")
print(f"ρ1, ρ2 = {ρ1}, {ρ2}")
```

```
r, ϕ = 0.95, 0.6283185307179586
p1, p2 = (0.95, 0.6283185307179586), (0.95, -0.6283185307179586)
a, b = (0.6346322893124001+0j), (0.9024999999999999-0j)
ρ1, ρ2 = 1.5371322893124, -0.9024999999999999
```

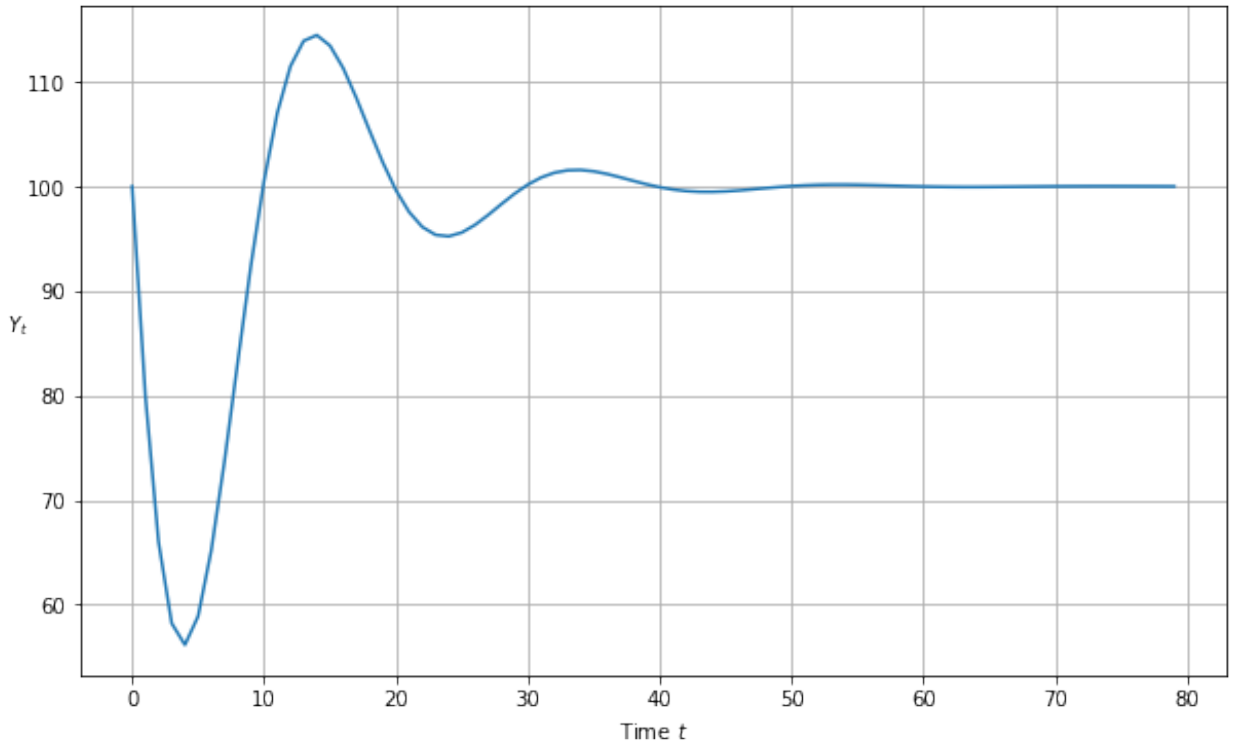
```
##=== This method uses numpy to calculate roots ===#


def y_nonstochastic(y_0=100, y_1=80, α=.9, β=.8, γ=10, n=80):
```

(continues on next page)

(continued from previous page)

```
    """ Rather than computing the roots of the characteristic
    polynomial by hand as we did earlier, this function
    enlists numpy to do the work for us
    """

    # Useful constants
    ρ1 = α + β
    ρ2 = -β

    categorize_solution(ρ1, ρ2)

    # Find roots of polynomial
    roots = np.roots([1, -ρ1, -ρ2])
    print(f'Roots are {roots}')

    # Check if real or complex
    if all(isinstance(root, complex) for root in roots):
        print('Roots are complex')
    else:
        print('Roots are real')

    # Check if roots are less than one
    if all(abs(root) < 1 for root in roots):
        print('Roots are less than one')
    else:
        print('Roots are not less than one')

    # Define transition equation
    def transition(x, t): return ρ1 * x[t - 1] + ρ2 * x[t - 2] + y

    # Set initial conditions
    y_t = [y_0, y_1]

    # Generate y_t series
    for t in range(2, n):
        y_t.append(transition(y_t, t))

    return y_t

plot_y(y_nonstochastic())
```

```
Roots are complex with modulus less than one; therefore damped oscillations
Roots are [0.85+0.27838822j 0.85-0.27838822j]
Roots are complex
Roots are less than one
```



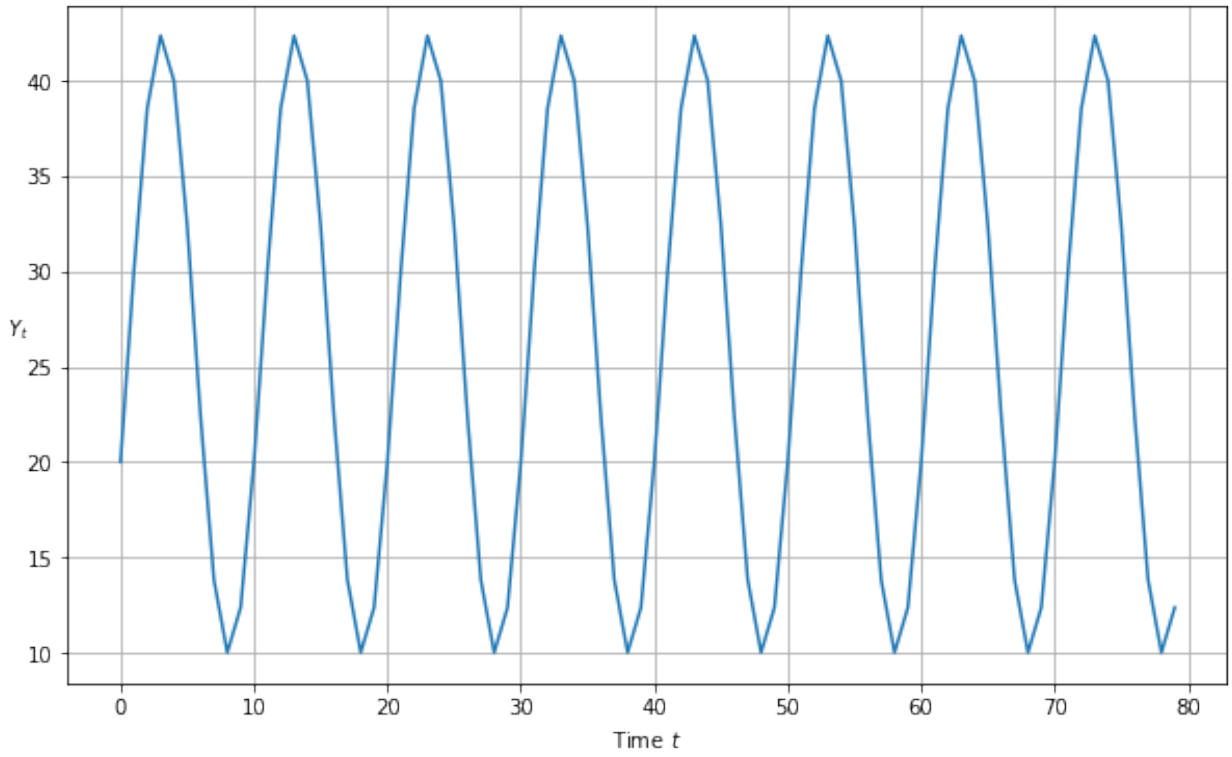
20.3.6 Reverse-Engineered Complex Roots: Example

The next cell studies the implications of reverse-engineered complex roots.

We'll generate an **undamped** cycle of period 10

```
r = 1    # Generates undamped, nonexplosive cycles

period = 10    # Length of cycle in units of time
ϕ = 2 * math.pi/period

## Apply the reverse-engineering function f

ρ1, ρ2, a, b = f(r, ϕ)

# Drop the imaginary part so that it is a valid input into y_nonstochastic
a = a.real
b = b.real

print(f"a, b = {a}, {b}")

ytemp = y_nonstochastic(α=a, β=b, y_0=20, y_1=30)
plot_y(ytemp)
```

```
a, b = 0.6180339887498949, 1.0
Roots are complex with modulus less than one; therefore damped oscillations
Roots are [0.80901699+0.58778525j 0.80901699-0.58778525j]
Roots are complex
Roots are less than one
```



20.3.7 Digression: Using Sympy to Find Roots

We can also use sympy to compute analytic formulas for the roots

```
init_printing()

r1 = Symbol("ρ_1")
r2 = Symbol("ρ_2")
z = Symbol("z")

sympy.solve(z**2 - r1*z - r2, z)
```

$$\left[\rho_1/2 - \sqrt{\rho_1^2 + 4\rho_2}/2,\ \rho_1/2 + \sqrt{\rho_1^2 + 4\rho_2}/2\right]$$

```
a = Symbol("α")
b = Symbol("β")
r1 = a + b
r2 = -b

sympy.solve(z**2 - r1*z - r2, z)
```

$$\left[\alpha/2 + \beta/2 - \sqrt{\alpha^2 + 2\alpha\beta + \beta^2 - 4\beta}/2,\ \alpha/2 + \beta/2 + \sqrt{\alpha^2 + 2\alpha\beta + \beta^2 - 4\beta}/2\right]$$

20.4 Stochastic Shocks

Now we'll construct some code to simulate the stochastic version of the model that emerges when we add a random shock process to aggregate demand

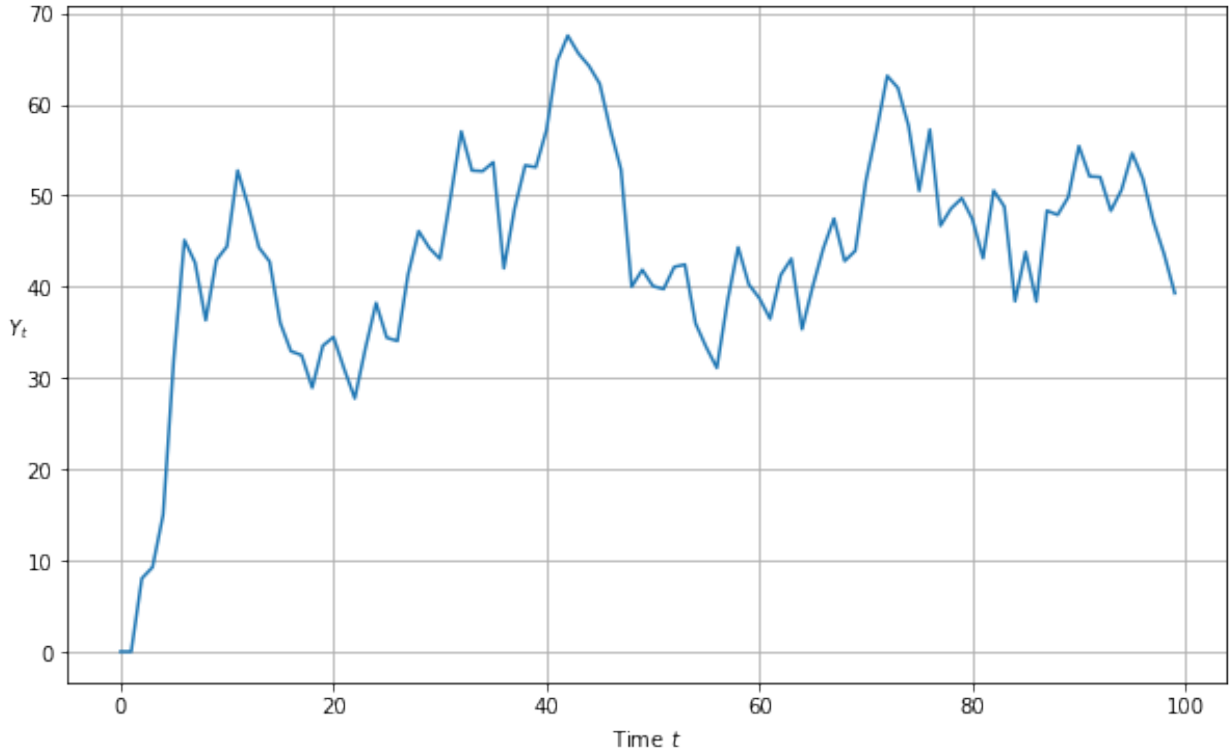
```
def y_stochastic(y_0=0, y_1=0, α=0.8, β=0.2, γ=10, n=100, σ=5):

    """This function takes parameters of a stochastic version of
    the model and proceeds to analyze the roots of the characteristic
    polynomial and also generate a simulation.
    """

    # Useful constants
    ρ1 = α + β
    ρ2 = -β

    # Categorize solution
    categorize_solution(ρ1, ρ2)

    # Find roots of polynomial
    roots = np.roots([1, -ρ1, -ρ2])
    print(roots)

    # Check if real or complex
    if all(isinstance(root, complex) for root in roots):
        print('Roots are complex')
    else:
        print('Roots are real')

    # Check if roots are less than one
    if all(abs(root) < 1 for root in roots):
        print('Roots are less than one')
    else:
        print('Roots are not less than one')

    # Generate shocks
    ϵ = np.random.normal(0, 1, n)

    # Define transition equation
    def transition(x, t): return ρ1 * \
        x[t - 1] + ρ2 * x[t - 2] + γ + σ * ϵ[t]

    # Set initial conditions
    y_t = [y_0, y_1]

    # Generate y_t series
    for t in range(2, n):
        y_t.append(transition(y_t, t))

    return y_t

plot_y(y_stochastic())
```

```
Roots are real and absolute values are less than one; therefore get smooth␣
↪convergence to a steady state
[0.7236068 0.2763932]
Roots are real
```

(continues on next page)

(continued from previous page)

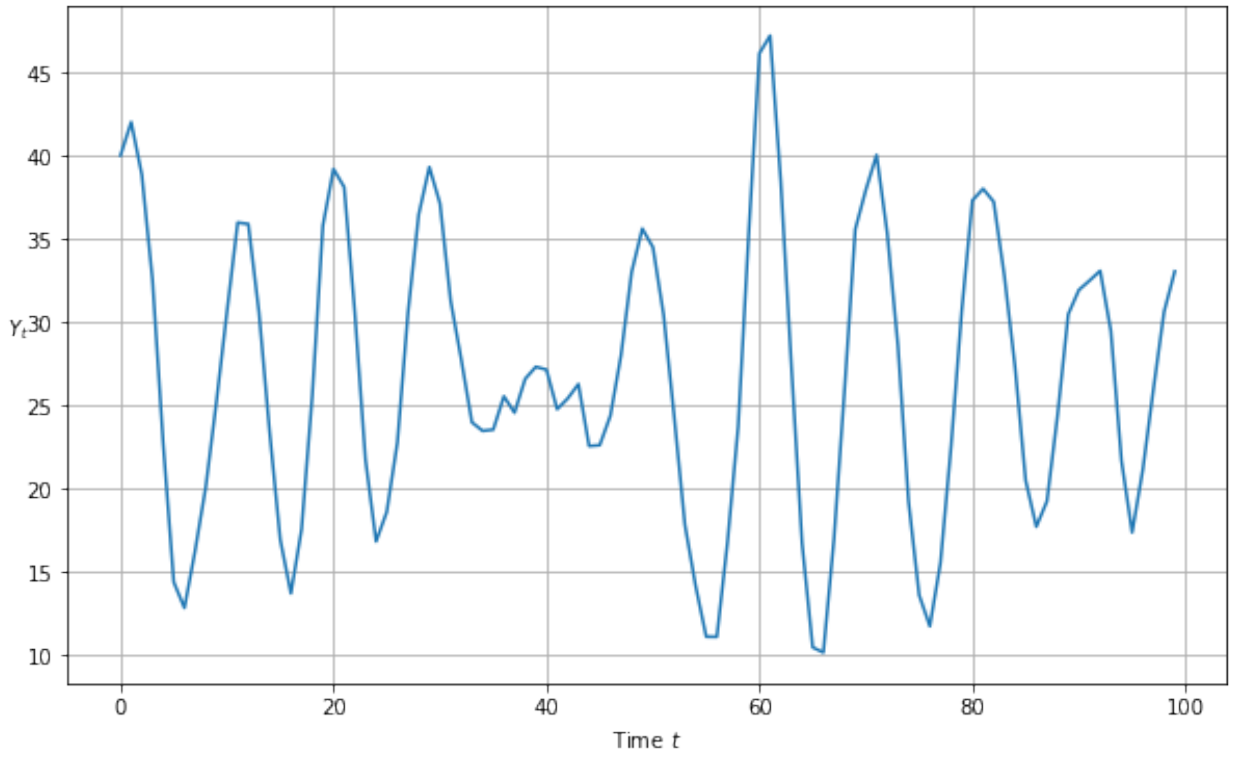
```
Roots are less than one
```



Let's do a simulation in which there are shocks and the characteristic polynomial has complex roots

```
r = .97

period = 10    #  Length of cycle in units of time
ϕ = 2 * math.pi/period

### Apply the  reverse-engineering function f

ρ1, ρ2, a, b = f(r, ϕ)

# Drop the imaginary part so that it is a valid input into y_nonstochastic
a = a.real
b = b.real

print(f"a, b = {a}, {b}")
plot_y(y_stochastic(y_0=40, y_1 = 42, α=a, β=b, σ=2, n=100))
```

```
a, b = 0.6285929690873979, 0.9409000000000001
Roots are complex with modulus less than one; therefore damped oscillations
[0.78474648+0.57015169j 0.78474648-0.57015169j]
Roots are complex
Roots are less than one
```



20.5 Government Spending

This function computes a response to either a permanent or one-off increase in government expenditures

```
def y_stochastic_g(y_0=20,
                   y_1=20,
                   α=0.8,
                   β=0.2,
                   γ=10,
                   n=100,
                   σ=2,
                   g=0,
                   g_t=0,
                   duration='permanent'):

    """This program computes a response to a permanent increase
    in government expenditures that occurs at time 20
    """

    # Useful constants
    ρ1 = α + β
    ρ2 = -β

    # Categorize solution
    categorize_solution(ρ1, ρ2)

    # Find roots of polynomial
    roots = np.roots([1, -ρ1, -ρ2])
```

(continues on next page)

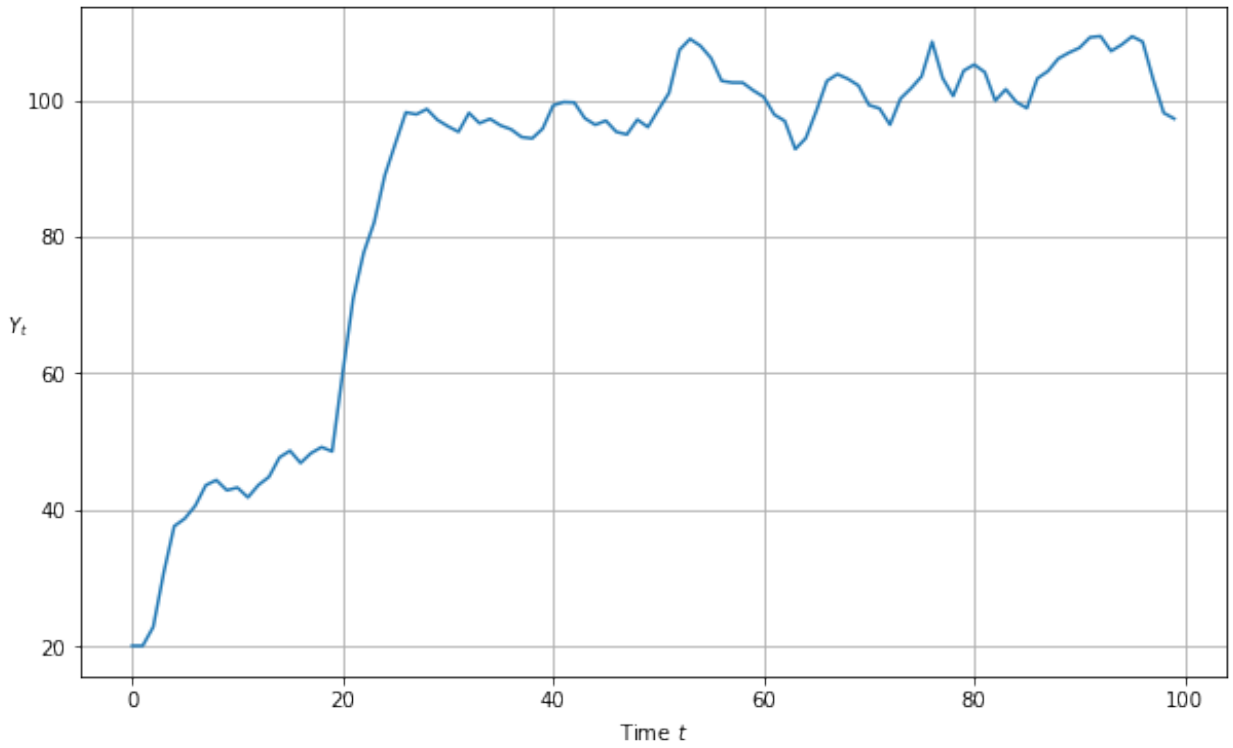
(continued from previous page)

```
    print(roots)

    # Check if real or complex
    if all(isinstance(root, complex) for root in roots):
        print('Roots are complex')
    else:
        print('Roots are real')

    # Check if roots are less than one
    if all(abs(root) < 1 for root in roots):
        print('Roots are less than one')
    else:
        print('Roots are not less than one')

    # Generate shocks
    ϵ = np.random.normal(0, 1, n)

    def transition(x, t, g):

        # Non-stochastic - separated to avoid generating random series
        # when not needed
        if σ == 0:
            return ρ1 * x[t - 1] + ρ2 * x[t - 2] + γ + g

        # Stochastic
        else:
            ϵ = np.random.normal(0, 1, n)
            return ρ1 * x[t - 1] + ρ2 * x[t - 2] + γ + g + σ * ϵ[t]

    # Create list and set initial conditions
    y_t = [y_0, y_1]

    # Generate y_t series
    for t in range(2, n):

        # No government spending
        if g == 0:
            y_t.append(transition(y_t, t))

        # Government spending (no shock)
        elif g != 0 and duration == None:
            y_t.append(transition(y_t, t))

        # Permanent government spending shock
        elif duration == 'permanent':
            if t < g_t:
                y_t.append(transition(y_t, t, g=0))
            else:
                y_t.append(transition(y_t, t, g=g))

        # One-off government spending shock
        elif duration == 'one-off':
            if t == g_t:
                y_t.append(transition(y_t, t, g=g))
            else:
                y_t.append(transition(y_t, t, g=0))
    return y_t
```

A permanent government spending shock can be simulated as follows

```
plot_y(y_stochastic_g(g=10, g_t=20, duration='permanent'))
```

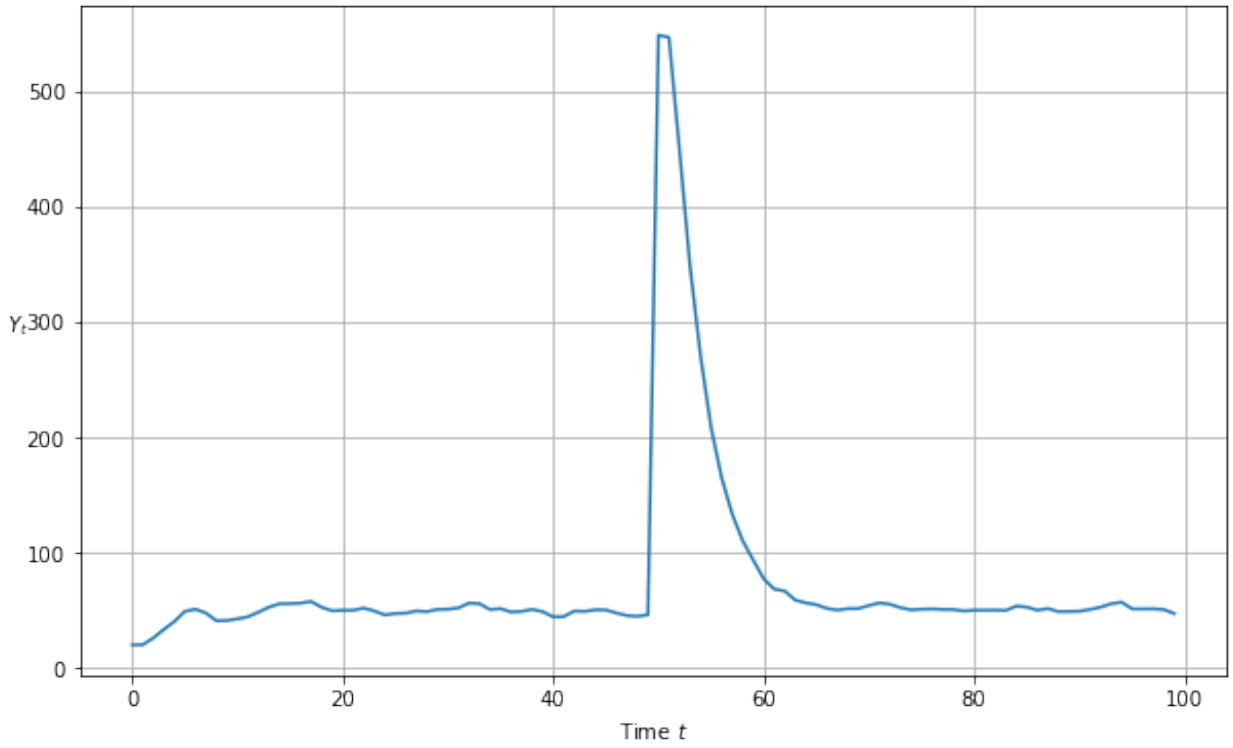
```
Roots are real and absolute values are less than one; therefore get smooth␣
 ↪convergence to a steady state
[0.7236068 0.2763932]
Roots are real
Roots are less than one
```



We can also see the response to a one time jump in government expenditures

```
plot_y(y_stochastic_g(g=500, g_t=50, duration='one-off'))
```

```
Roots are real and absolute values are less than one; therefore get smooth␣
 ↪convergence to a steady state
[0.7236068 0.2763932]
Roots are real
Roots are less than one
```



20.6 Wrapping Everything Into a Class

Up to now, we have written functions to do the work.

Now we'll roll up our sleeves and write a Python class called `Samuelson` for the Samuelson model

```
class Samuelson():

    """This class represents the Samuelson model, otherwise known as the
    multiple-accelerator model. The model combines the Keynesian multiplier
    with the accelerator theory of investment.

    The path of output is governed by a linear second-order difference equation

    .. math::

        Y_t =  + \alpha (1 + \beta) Y_{t-1} - \alpha \beta Y_{t-2}

    Parameters
    ----------
    y_0 : scalar
        Initial condition for Y_0
    y_1 : scalar
        Initial condition for Y_1
    α : scalar
        Marginal propensity to consume
    β : scalar
        Accelerator coefficient
    n : int
```

(continues on next page)

(continued from previous page)

```
        Number of iterations
    σ : scalar
        Volatility parameter. It must be greater than or equal to 0. Set
        equal to 0 for a non-stochastic model.
    g : scalar
        Government spending shock
    g_t : int
        Time at which government spending shock occurs. Must be specified
        when duration != None.
    duration : {None, 'permanent', 'one-off'}
        Specifies type of government spending shock. If none, government
        spending equal to g for all t.

    """

    def __init__(self,
                 y_0=100,
                 y_1=50,
                 α=1.3,
                 β=0.2,
                 γ=10,
                 n=100,
                 σ=0,
                 g=0,
                 g_t=0,
                 duration=None):

        self.y_0, self.y_1, self.α, self.β = y_0, y_1, α, β
        self.n, self.g, self.g_t, self.duration = n, g, g_t, duration
        self.γ, self.σ = γ, σ
        self.ρ1 = α + β
        self.ρ2 = -β
        self.roots = np.roots([1, -self.ρ1, -self.ρ2])

    def root_type(self):
        if all(isinstance(root, complex) for root in self.roots):
            return 'Complex conjugate'
        elif len(self.roots) > 1:
            return 'Double real'
        else:
            return 'Single real'

    def root_less_than_one(self):
        if all(abs(root) < 1 for root in self.roots):
            return True

    def solution_type(self):
        ρ1, ρ2 = self.ρ1, self.ρ2
        discriminant = ρ1 ** 2 + 4 * ρ2
        if ρ2 >= 1 + ρ1 or ρ2 <= -1:
            return 'Explosive oscillations'
        elif ρ1 + ρ2 >= 1:
            return 'Explosive growth'
        elif discriminant < 0:
            return 'Damped oscillations'
        else:
            return 'Steady state'
```

(continues on next page)

(continued from previous page)

```
    def _transition(self, x, t, g):

        # Non-stochastic - separated to avoid generating random series
        # when not needed
        if self.σ == 0:
            return self.ρ1 * x[t - 1] + self.ρ2 * x[t - 2] + self.y + g

        # Stochastic
        else:
            ϵ = np.random.normal(0, 1, self.n)
            return self.ρ1 * x[t - 1] + self.ρ2 * x[t - 2] + self.y + g \
                + self.σ * ϵ[t]

    def generate_series(self):

        # Create list and set initial conditions
        y_t = [self.y_0, self.y_1]

        # Generate y_t series
        for t in range(2, self.n):

            # No government spending
            if self.g == 0:
                y_t.append(self._transition(y_t, t))

            # Government spending (no shock)
            elif self.g != 0 and self.duration == None:
                y_t.append(self._transition(y_t, t))

            # Permanent government spending shock
            elif self.duration == 'permanent':
                if t < self.g_t:
                    y_t.append(self._transition(y_t, t, g=0))
                else:
                    y_t.append(self._transition(y_t, t, g=self.g))

            # One-off government spending shock
            elif self.duration == 'one-off':
                if t == self.g_t:
                    y_t.append(self._transition(y_t, t, g=self.g))
                else:
                    y_t.append(self._transition(y_t, t, g=0))
        return y_t

    def summary(self):
        print('Summary\n' + '-' * 50)
        print(f'Root type: {self.root_type()}')
        print(f'Solution type: {self.solution_type()}')
        print(f'Roots: {str(self.roots)}')

        if self.root_less_than_one() == True:
            print('Absolute value of roots is less than one')
        else:
            print('Absolute value of roots is not less than one')

        if self.σ > 0:
```

(continues on next page)

(continued from previous page)

```
            print('Stochastic series with σ = ' + str(self.σ))
        else:
            print('Non-stochastic series')

        if self.g != 0:
            print('Government spending equal to ' + str(self.g))

        if self.duration != None:
            print(self.duration.capitalize() +
                  ' government spending shock at t = ' + str(self.g_t))

    def plot(self):
        fig, ax = plt.subplots(figsize=(10, 6))
        ax.plot(self.generate_series())
        ax.set(xlabel='Iteration', xlim=(0, self.n))
        ax.set_ylabel('$Y_t$', rotation=0)
        ax.grid()

        # Add parameter values to plot
        paramstr = f'$\\alpha={self.α:.2f}$ \n $\\beta={self.β:.2f}$ \n \
        $\\gamma={self.γ:.2f}$ \n $\\sigma={self.σ:.2f}$ \n \
        $\\rho_1={self.ρ1:.2f}$ \n $\\rho_2={self.ρ2:.2f}$'
        props = dict(fc='white', pad=10, alpha=0.5)
        ax.text(0.87, 0.05, paramstr, transform=ax.transAxes,
                fontsize=12, bbox=props, va='bottom')

        return fig

    def param_plot(self):

        # Uses the param_plot() function defined earlier (it is then able
        # to be used standalone or as part of the model)

        fig = param_plot()
        ax = fig.gca()

        # Add λ values to legend
        for i, root in enumerate(self.roots):
            if isinstance(root, complex):
                # Need to fill operator for positive as string is split apart
                operator = ['+', '']
                label = rf'$\lambda_{i+1} = {sam.roots[i].real:.2f} \
                    {operator[i]} {sam.roots[i].imag:.2f}i$'
            else:
                label = rf'$\lambda_{i+1} = {sam.roots[i].real:.2f}$'
            ax.scatter(0, 0, 0, label=label) # dummy to add to legend

        # Add ρ pair to plot
        ax.scatter(self.ρ1, self.ρ2, 100, 'red', '+',
            label=r'$(\ \rho_1, \ \rho_2 \ )$', zorder=5)

        plt.legend(fontsize=12, loc=3)

        return fig
```

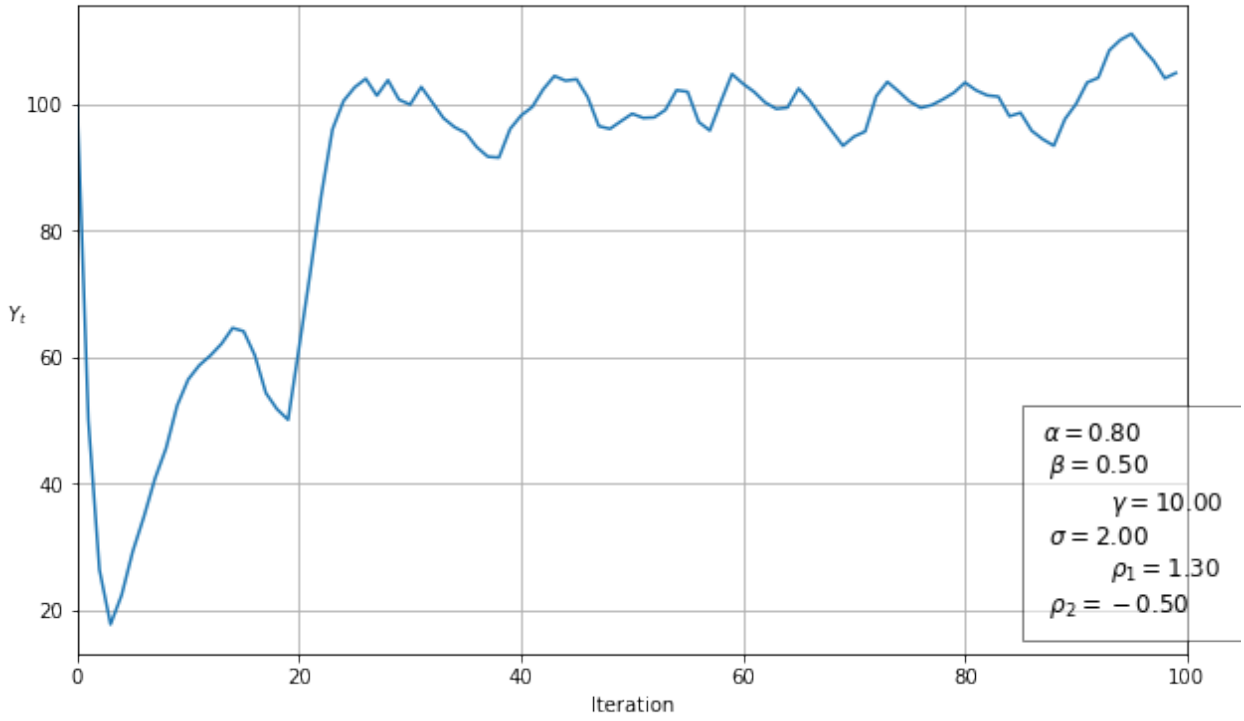
20.6.1 Illustration of Samuelson Class

Now we'll put our Samuelson class to work on an example

```
sam = Samuelson(α=0.8, β=0.5, σ=2, g=10, g_t=20, duration='permanent')
sam.summary()
```

```
Summary
---------------------------------------------------
Root type: Complex conjugate
Solution type: Damped oscillations
Roots: [0.65+0.27838822j 0.65-0.27838822j]
Absolute value of roots is less than one
Stochastic series with σ = 2
Government spending equal to 10
Permanent government spending shock at t = 20
```

```
sam.plot()
plt.show()
```

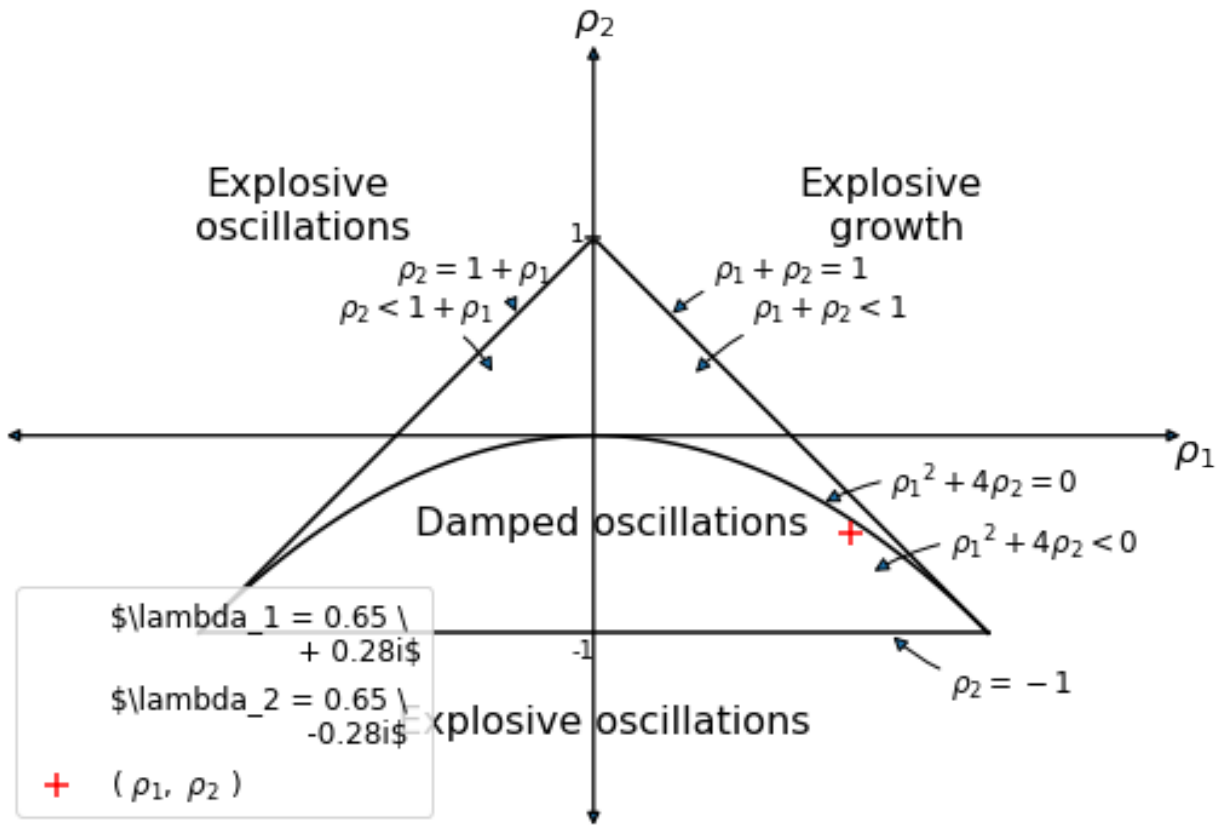


20.6.2 Using the Graph

We'll use our graph to show where the roots lie and how their location is consistent with the behavior of the path just graphed.

The red + sign shows the location of the roots

```
sam.param_plot()
plt.show()
```



20.7 Using the LinearStateSpace Class

It turns out that we can use the QuantEcon.py LinearStateSpace class to do much of the work that we have done from scratch above.

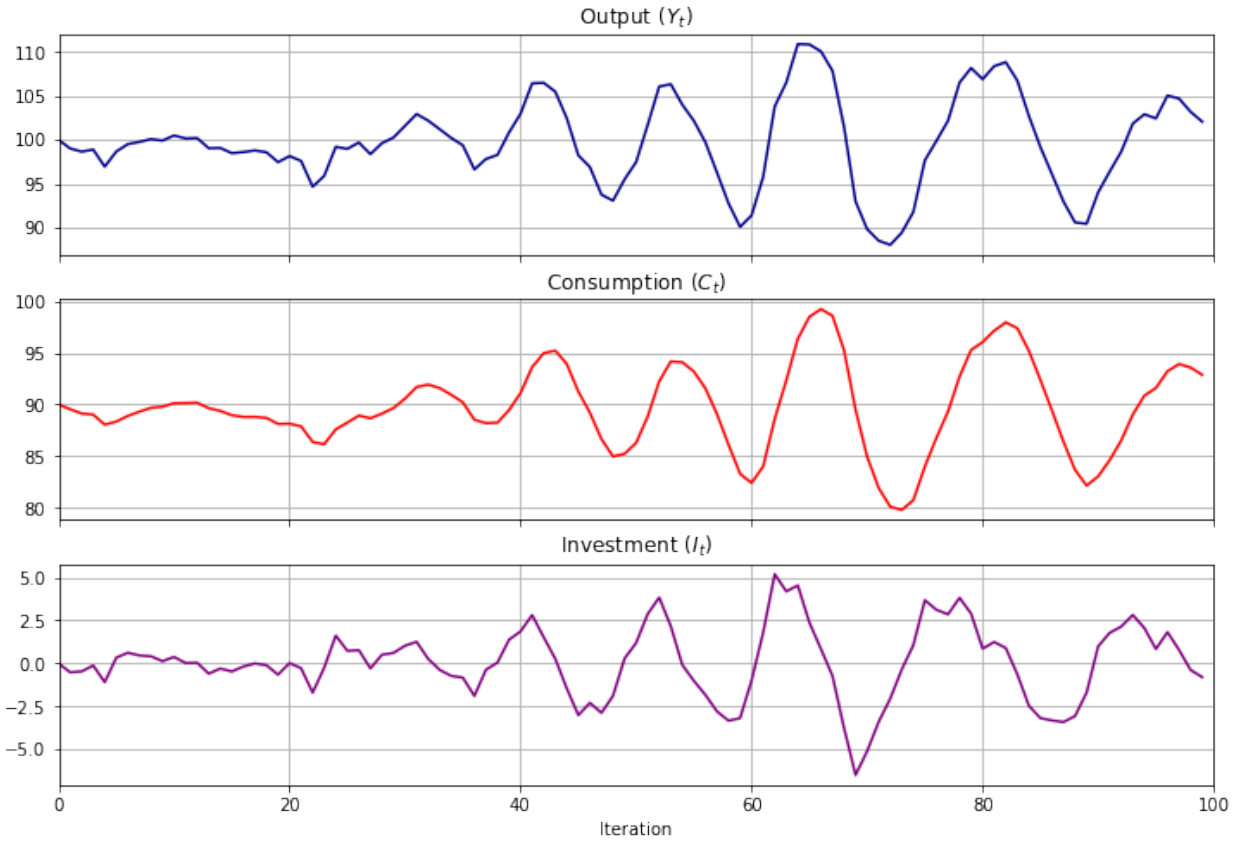
Here is how we map the Samuelson model into an instance of a `LinearStateSpace` class

```
"""This script maps the Samuelson model in the the
``LinearStateSpace`` class
"""
α = 0.8
β = 0.9
ρ1 = α + β
ρ2 = -β
γ = 10
σ = 1
```

(continues on next page)

(continued from previous page)

```
g = 10
n = 100

A = [[1,         0,       0],
     [γ + g,    ρ1,      ρ2],
     [0,         1,       0]]

G = [[γ + g, ρ1,    ρ2],          # this is Y_{t+1}
     [γ,       α,     0],          # this is C_{t+1}
     [0,       β,    -β]]          # this is I_{t+1}

μ_0 = [1, 100, 100]
C = np.zeros((3,1))
C[1] = σ # stochastic

sam_t = LinearStateSpace(A, C, G, mu_0=μ_0)

x, y = sam_t.simulate(ts_length=n)

fig, axes = plt.subplots(3, 1, sharex=True, figsize=(12, 8))
titles = ['Output ($Y_t$)', 'Consumption ($C_t$)', 'Investment ($I_t$)']
colors = ['darkblue', 'red', 'purple']
for ax, series, title, color in zip(axes, y, titles, colors):
    ax.plot(series, color=color)
    ax.set(title=title, xlim=(0, n))
    ax.grid()

axes[-1].set_xlabel('Iteration')

plt.show()
```



20.7.1 Other Methods in the `LinearStateSpace` Class

Let's plot **impulse response functions** for the instance of the Samuelson model using a method in the `LinearStateSpace` class

```
imres = sam_t.impulse_response()
imres = np.asarray(imres)
y1 = imres[:, :, 0]
y2 = imres[:, :, 1]
y1.shape
```

```
(2, 6, 1)
```

Now let's compute the zeros of the characteristic polynomial by simply calculating the eigenvalues of A

```
A = np.asarray(A)
w, v = np.linalg.eig(A)
print(w)
```

```
[0.85+0.42130749j 0.85-0.42130749j 1.  +0.j        ]
```

20.7.2 Inheriting Methods from `LinearStateSpace`

We could also create a subclass of `LinearStateSpace` (inheriting all its methods and attributes) to add more functions to use

```
class SamuelsonLSS(LinearStateSpace):

    """
    This subclass creates a Samuelson multiplier-accelerator model
    as a linear state space system.
    """
    def __init__(self,
                 y_0=100,
                 y_1=100,
                 α=0.8,
                 β=0.9,
                 γ=10,
                 σ=1,
                 g=10):

        self.α, self.β = α, β
        self.y_0, self.y_1, self.g = y_0, y_1, g
        self.γ, self.σ = γ, σ

        # Define intial conditions
        self.μ_0 = [1, y_0, y_1]

        self.ρ1 = α + β
        self.ρ2 = -β

        # Define transition matrix
        self.A = [[1,                 0,          0],
                  [γ + g,       self.ρ1,    self.ρ2],
                  [0,                 1,          0]]

        # Define output matrix
        self.G = [[γ + g, self.ρ1, self.ρ2],          # this is Y_{t+1}
                  [γ,           α,       0],          # this is C_{t+1}
                  [0,           β,      -β]]          # this is I_{t+1}

        self.C = np.zeros((3, 1))
        self.C[1] = σ  # stochastic

        # Initialize LSS with parameters from Samuelson model
        LinearStateSpace.__init__(self, self.A, self.C, self.G, mu_0=self.μ_0)

    def plot_simulation(self, ts_length=100, stationary=True):

        # Temporarily store original parameters
        temp_μ = self.μ_0
        temp_Σ = self.Sigma_0

        # Set distribution parameters equal to their stationary
        # values for simulation
        if stationary == True:
            try:
                self.μ_x, self.μ_y, self.σ_x, self.σ_y, self.σ_yx = \
                    self.stationary_distributions()
```

(continues on next page)

(continued from previous page)

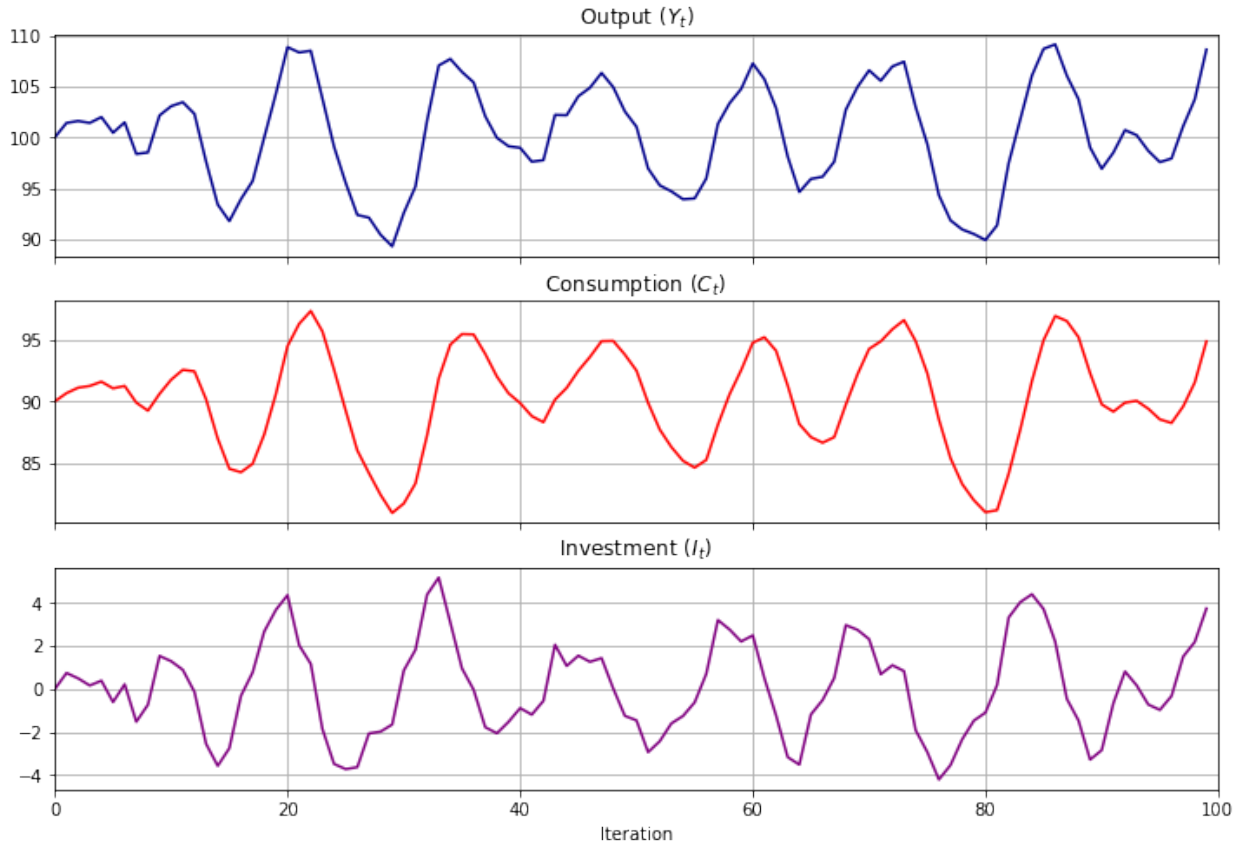
```
                self.μ_0 = self.μ_y
                self.Σ_0 = self.σ_y
            # Exception where no convergence achieved when
            #calculating stationary distributions
            except ValueError:
                print('Stationary distribution does not exist')

        x, y = self.simulate(ts_length)

        fig, axes = plt.subplots(3, 1, sharex=True, figsize=(12, 8))
        titles = ['Output ($Y_t$)', 'Consumption ($C_t$)', 'Investment ($I_t$)']
        colors = ['darkblue', 'red', 'purple']
        for ax, series, title, color in zip(axes, y, titles, colors):
            ax.plot(series, color=color)
            ax.set(title=title, xlim=(0, n))
            ax.grid()

        axes[-1].set_xlabel('Iteration')

        # Reset distribution parameters to their initial values
        self.μ_0 = temp_μ
        self.Sigma_0 = temp_Σ

        return fig

    def plot_irf(self, j=5):

        x, y = self.impulse_response(j)

        # Reshape into 3 x j matrix for plotting purposes
        yimf = np.array(y).flatten().reshape(j+1, 3).T

        fig, axes = plt.subplots(3, 1, sharex=True, figsize=(12, 8))
        labels = ['$Y_t$', '$C_t$', '$I_t$']
        colors = ['darkblue', 'red', 'purple']
        for ax, series, label, color in zip(axes, yimf, labels, colors):
            ax.plot(series, color=color)
            ax.set(xlim=(0, j))
            ax.set_ylabel(label, rotation=0, fontsize=14, labelpad=10)
            ax.grid()

        axes[0].set_title('Impulse Response Functions')
        axes[-1].set_xlabel('Iteration')

        return fig

    def multipliers(self, j=5):
        x, y = self.impulse_response(j)
        return np.sum(np.array(y).flatten().reshape(j+1, 3), axis=0)
```

20.7.3 Illustrations

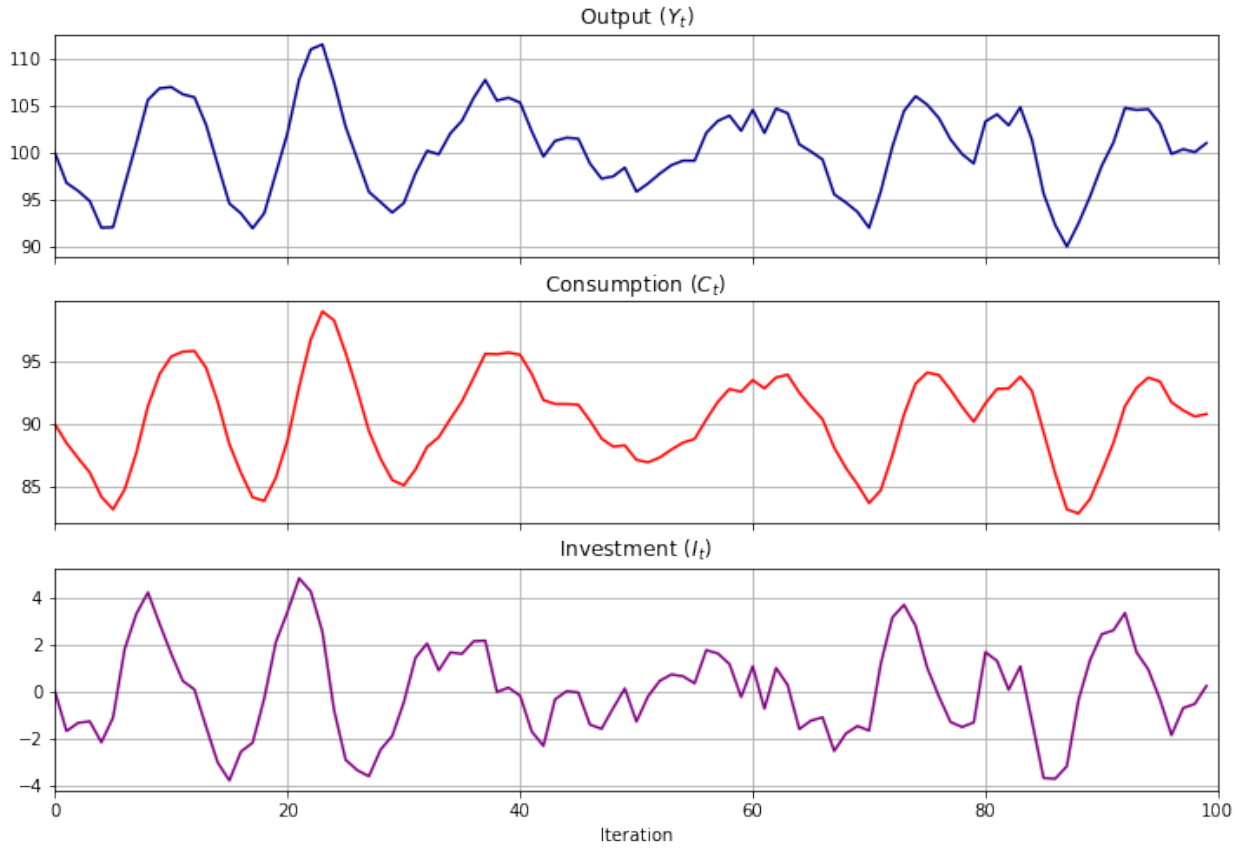
Let's show how we can use the `SamuelsonLSS`

```
samlss = SamuelsonLSS()
```

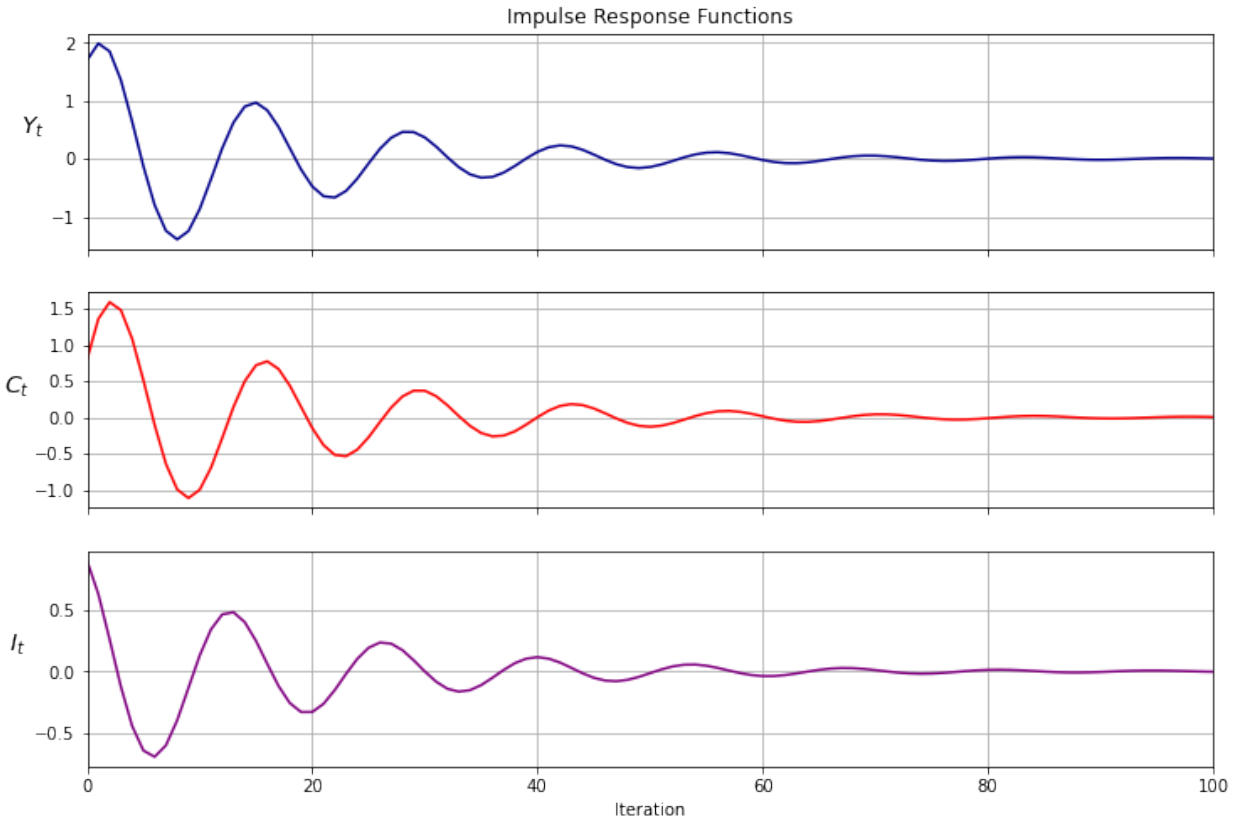
```
samlss.plot_simulation(100, stationary=False)
plt.show()
```




```
samlss.plot_simulation(100, stationary=True)
plt.show()
```




```
samlss.plot_irf(100)
plt.show()
```




```
samlss.multipliers()
```

```
array([7.414389, 6.835896, 0.578493])
```

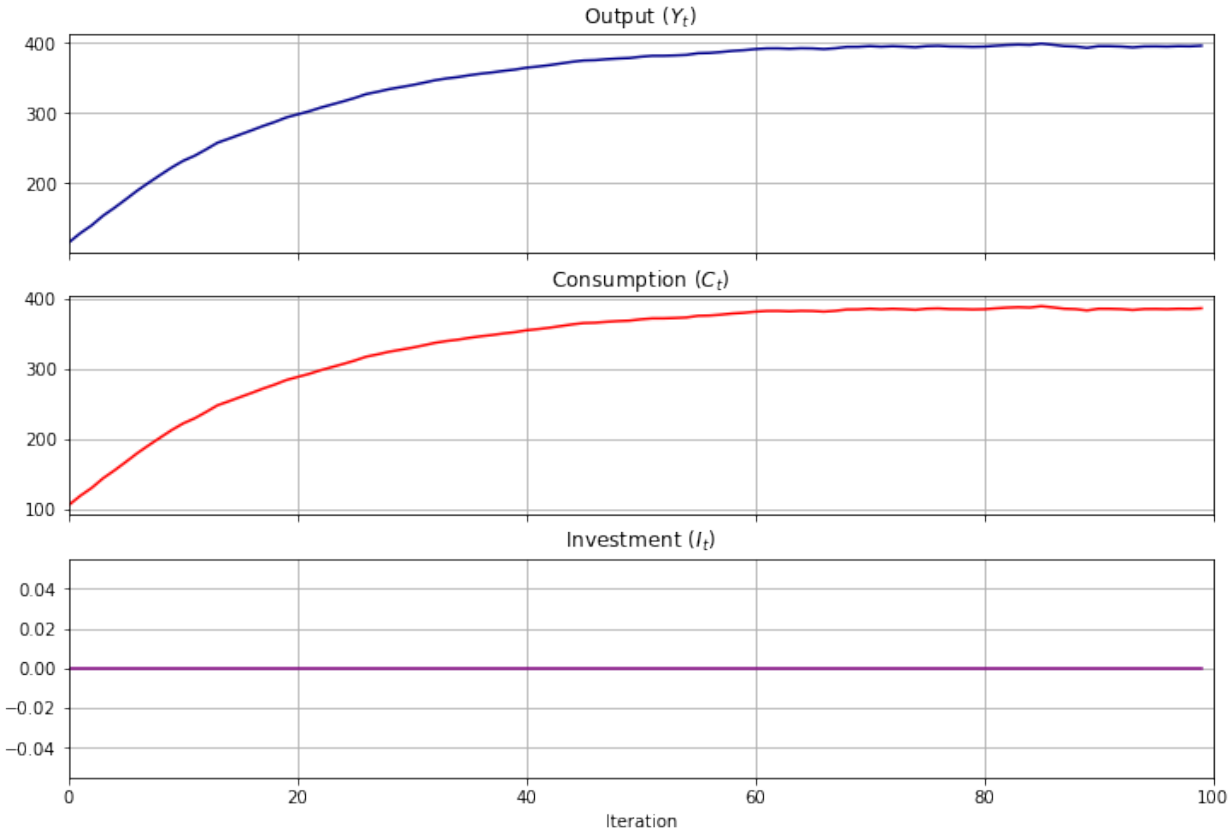
20.8 Pure Multiplier Model

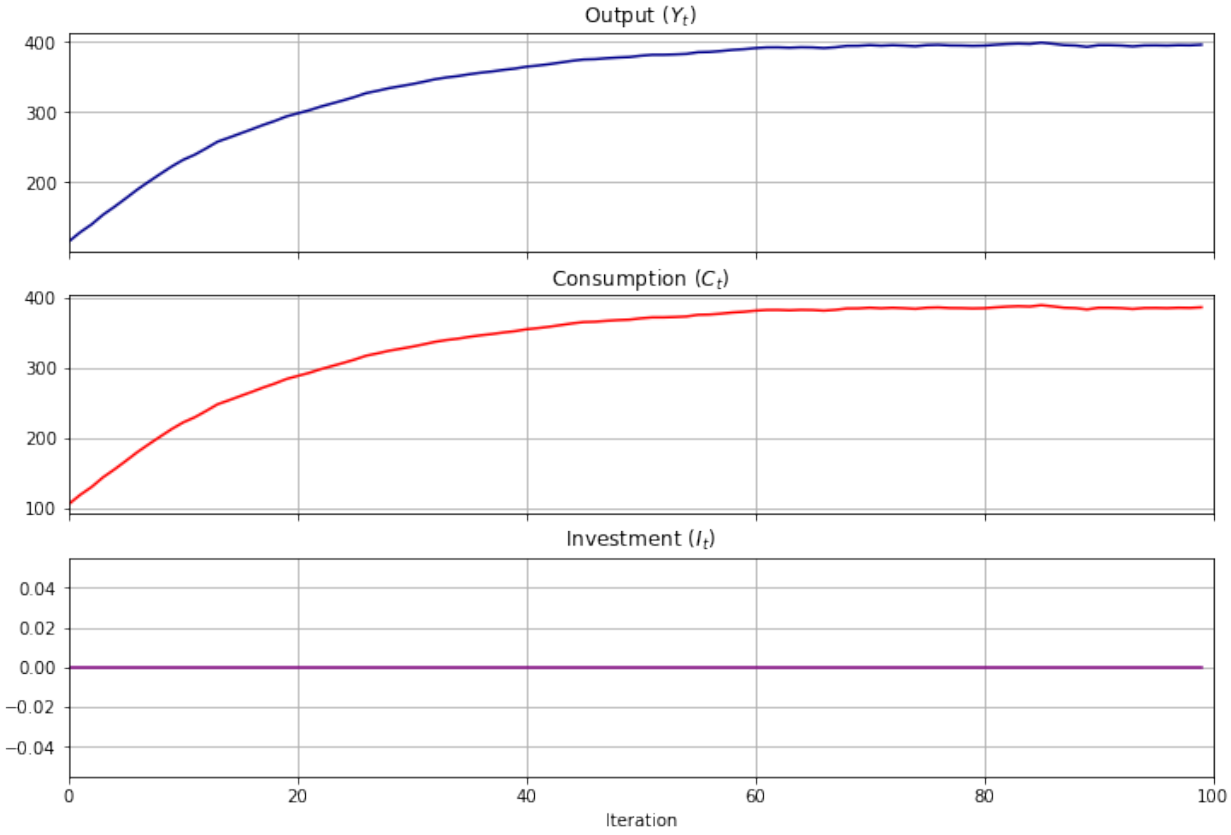
Let's shut down the accelerator by setting $b = 0$ to get a pure multiplier model

- the absence of cycles gives an idea about why Samuelson included the accelerator

```
pure_multiplier = SamuelsonLSS(α=0.95, β=0)
```

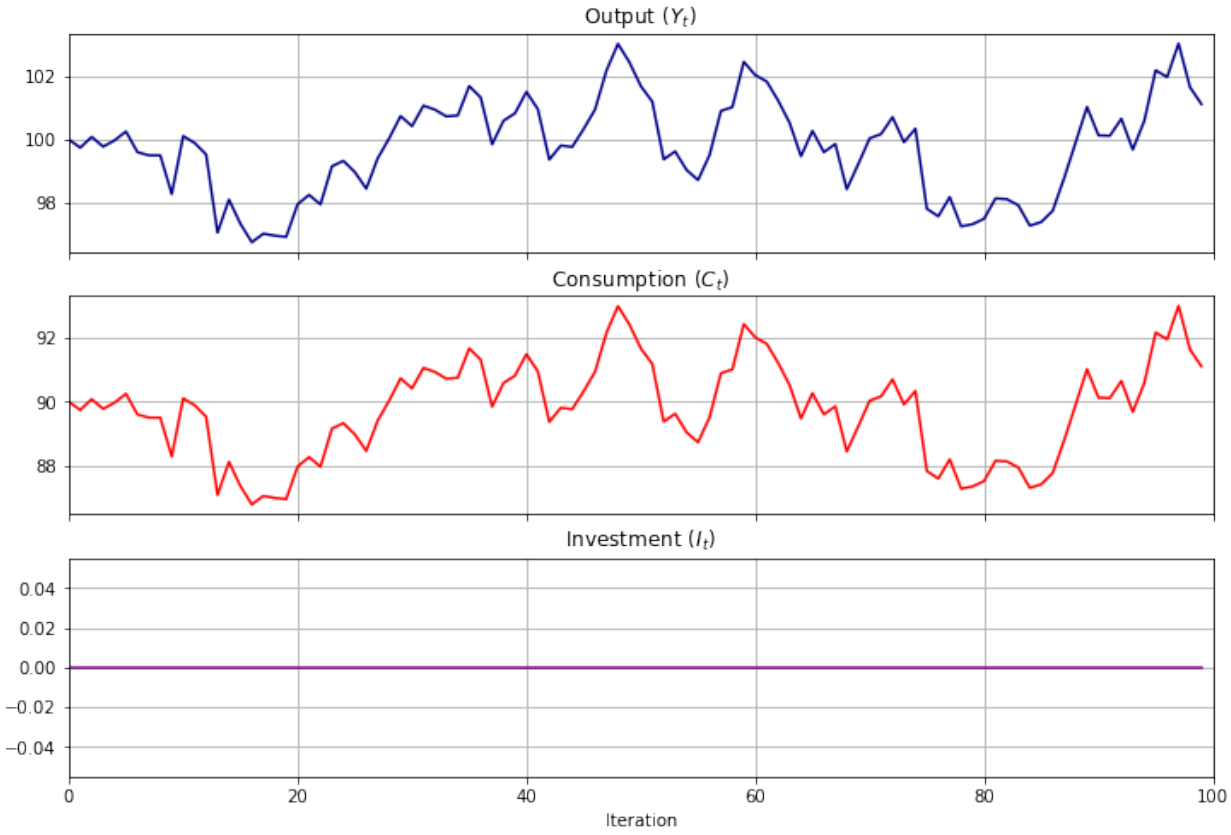
```
pure_multiplier.plot_simulation()
```

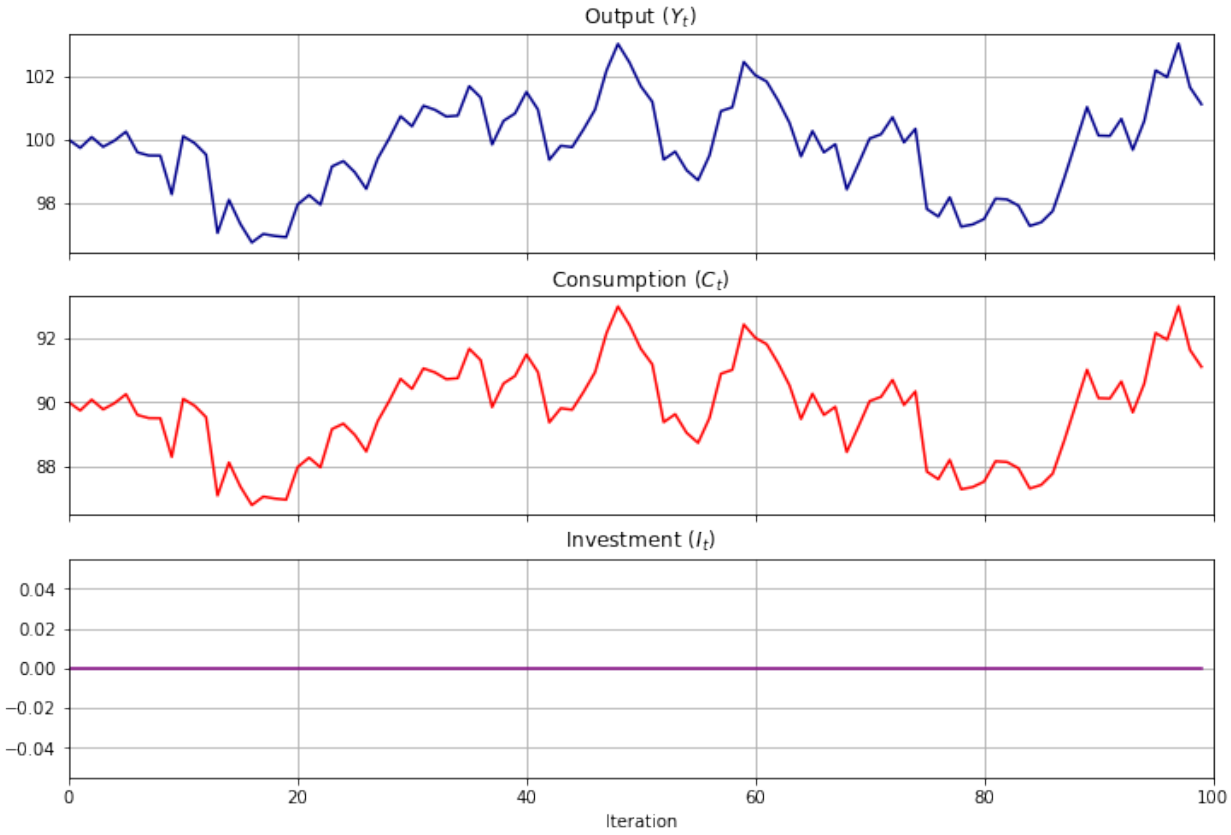

Output (Y_t)
400
300
200
Consumption (C_t)
400
300
200
100
Investment (I_t)
0.04
0.02
0.00
−0.02
−0.04
0
20
40
60
80
100
Iteration


Output (Y_t)
400
300
200
Consumption (C_t)
400
300
200
100
Investment (I_t)
0.04
0.02
0.00
−0.02
−0.04
0
20
40
60
80
100
Iteration

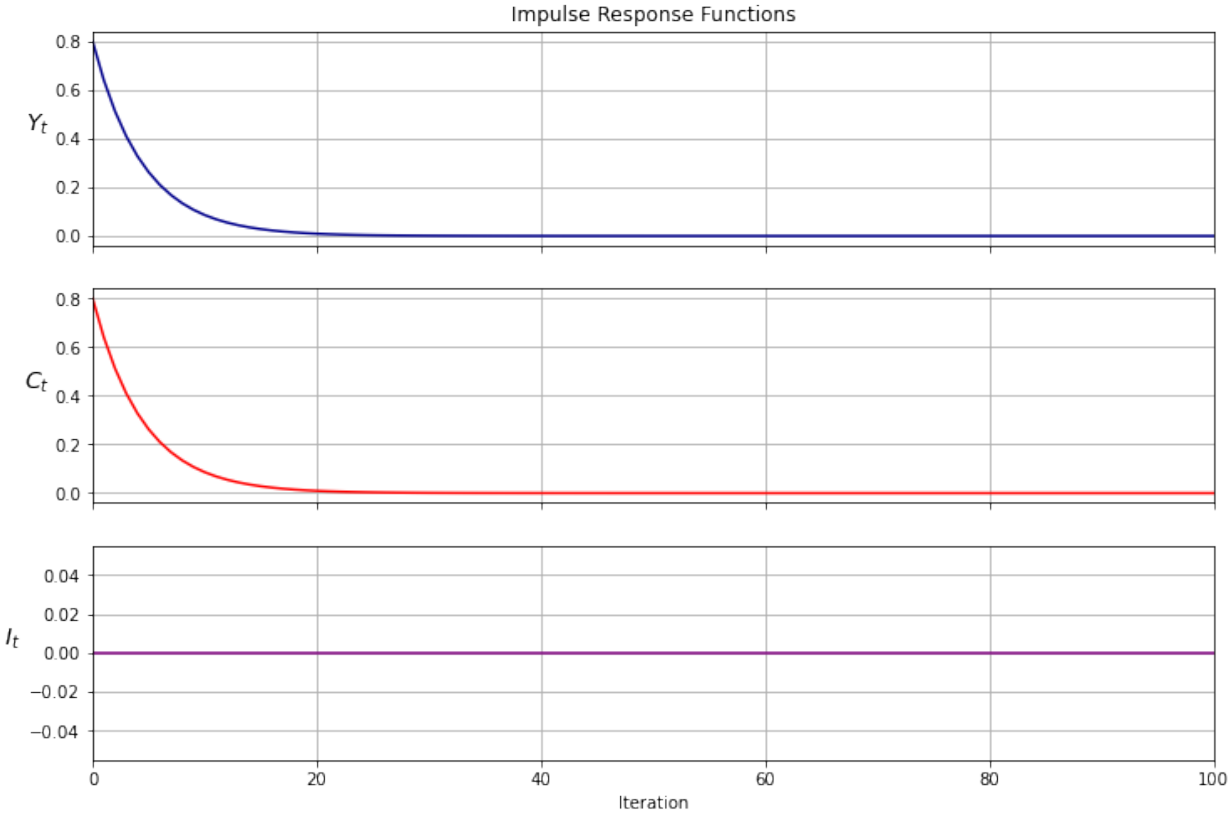
```
pure_multiplier = SamuelsonLSS(α=0.8, β=0)
```

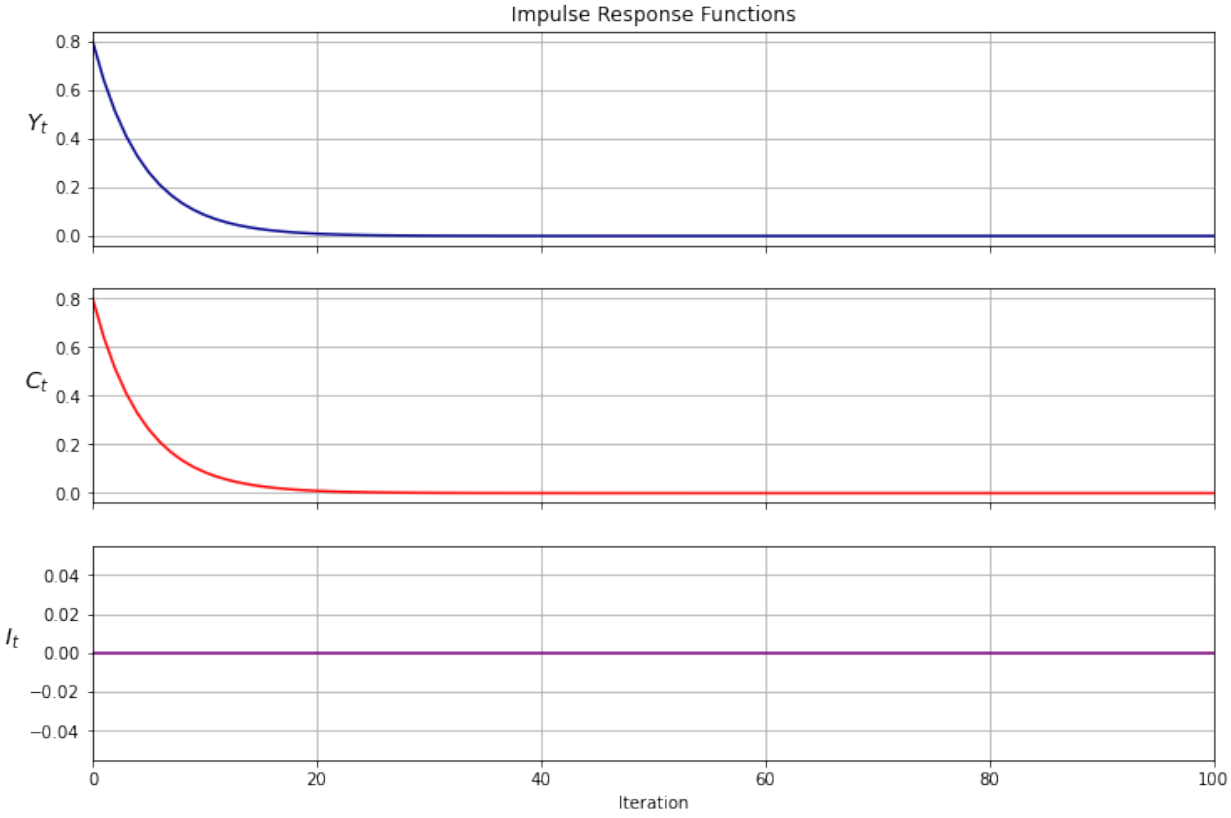
```
pure_multiplier.plot_simulation()
```






```
pure_multiplier.plot_irf(100)
```


Impulse Response Functions
Y_t
C_t
I_t
Iteration


Impulse Response Functions
Y_t
C_t
I_t
Iteration

20.9 Summary

In this lecture, we wrote functions and classes to represent non-stochastic and stochastic versions of the Samuelson (1939) multiplier-accelerator model, described in [Sam39].

We saw that different parameter values led to different output paths, which could either be stationary, explosive, or oscillating.

We also were able to represent the model using the QuantEcon.py LinearStateSpace class.

CHAPTER

TWENTYONE

KESTEN PROCESSES AND FIRM DYNAMICS

Contents

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

```
!conda install -y quantecon
!pip install --upgrade yfinance
```

21.1 Overview

Previously we learned about linear scalar-valued stochastic processes (AR(1) models).

Now we generalize these linear models slightly by allowing the multiplicative coefficient to be stochastic.

Such processes are known as Kesten processes after German–American mathematician Harry Kesten (1931–2019)

Although simple to write down, Kesten processes are interesting for at least two reasons:

1. A number of significant economic processes are or can be described as Kesten processes.
2. Kesten processes generate interesting dynamics, including, in some cases, heavy-tailed cross-sectional distributions.

We will discuss these issues as we go along.

Let's start with some imports:

```
%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 are only added to avoid a `FutureWarning` caused by compatibility issues between pandas and matplotlib.

```
from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters()
```

Additional technical background related to this lecture can be found in the monograph of [BDM+16].

21.2 Kesten Processes

A **Kesten process** is a stochastic process of the form

$$X_{t+1} = a_{t+1} X_t + \eta_{t+1} \tag{1}$$

where $\{a_t\}_{t \geq 1}$ and $\{\eta_t\}_{t \geq 1}$ are IID sequences.

We are interested in the dynamics of $\{X_t\}_{t \geq 0}$ when X_0 is given.

We will focus on the nonnegative scalar case, where X_t takes values in $\mathbb{R}_+$.

In particular, we will assume that

- the initial condition X_0 is nonnegative,
- $\{a_t\}_{t \geq 1}$ is a nonnegative IID stochastic process and
- $\{\eta_t\}_{t \geq 1}$ is another nonnegative IID stochastic process, independent of the first.

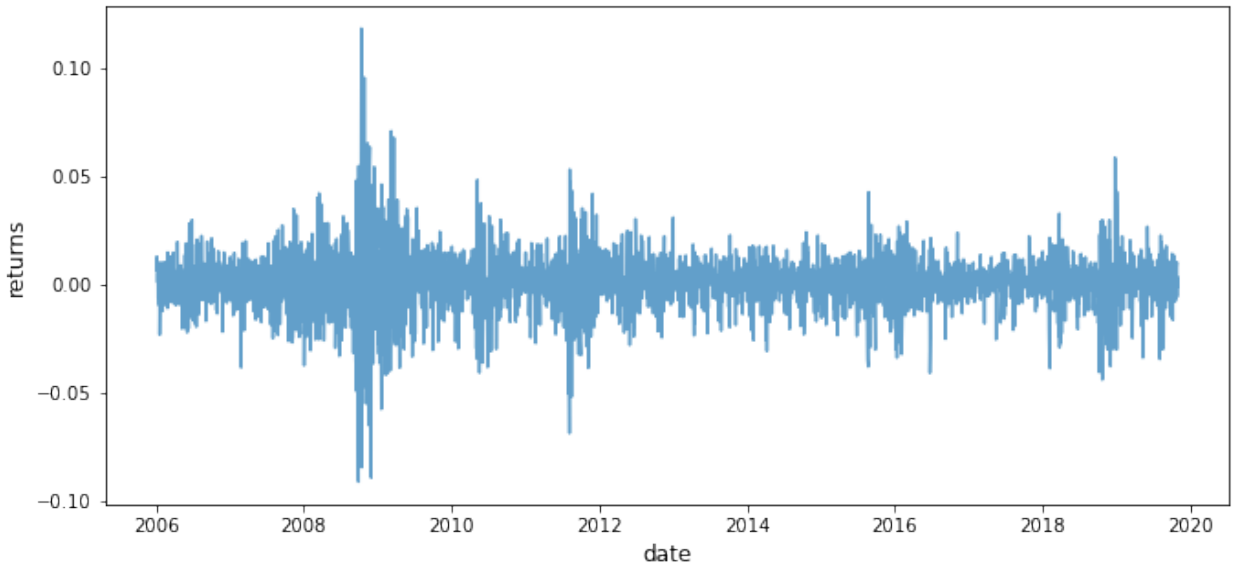
21.2.1 Example: GARCH Volatility

The GARCH model is common in financial applications, where time series such as asset returns exhibit time varying volatility.

For example, consider the following plot of daily returns on the Nasdaq Composite Index for the period 1st January 2006 to 1st November 2019.

```
import yfinance as yf
import pandas as pd

s = yf.download('^IXIC', '2006-1-1', '2019-11-1')['Adj Close']

r = s.pct_change()

fig, ax = plt.subplots()

ax.plot(r, alpha=0.7)

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

plt.show()
```

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



Notice how the series exhibits bursts of volatility (high variance) and then settles down again.

GARCH models can replicate this feature.

The GARCH(1, 1) volatility process takes the form

$$\sigma_{t+1}^2 = \alpha_0 + \sigma_t^2(\alpha_1 \xi_{t+1}^2 + \beta) \tag{2}$$

where $\{\xi_t\}$ is IID with $\mathbb{E}\xi_t^2 = 1$ and all parameters are positive.

Returns on a given asset are then modeled as

$$r_t = \sigma_t \zeta_t \tag{3}$$

where $\{\zeta_t\}$ is again IID and independent of $\{\xi_t\}$.

The volatility sequence $\{\sigma_t^2\}$, which drives the dynamics of returns, is a Kesten process.

21.2.2 Example: Wealth Dynamics

Suppose that a given household saves a fixed fraction s of its current wealth in every period.

The household earns labor income y_t at the start of time t.

Wealth then evolves according to

$$w_{t+1} = R_{t+1} s w_t + y_{t+1} \tag{4}$$

where $\{R_t\}$ is the gross rate of return on assets.

If $\{R_t\}$ and $\{y_t\}$ are both IID, then (4) is a Kesten process.

21.2.3 Stationarity

In earlier lectures, such as the one on *AR(1) processes*, we introduced the notion of a stationary distribution.

In the present context, we can define a stationary distribution as follows:

The distribution F^* on $\mathbb{R}$ is called **stationary** for the Kesten process (1) if

$$X_t \sim F^* \quad \Longrightarrow \quad a_{t+1} X_t + \eta_{t+1} \sim F^* \tag{5}$$

In other words, if the current state X_t has distribution F^*, then so does the next period state X_{t+1}.

We can write this alternatively as

$$F^*(y) = \int \mathbb{P}\{a_{t+1} x + \eta_{t+1} \leq y\} F^*(dx) \quad \text{for all } y \geq 0. \tag{6}$$

The left hand side is the distribution of the next period state when the current state is drawn from F^*.

The equality in (6) states that this distribution is unchanged.

21.2.4 Cross-Sectional Interpretation

There is an important cross-sectional interpretation of stationary distributions, discussed previously but worth repeating here.

Suppose, for example, that we are interested in the wealth distribution — that is, the current distribution of wealth across households in a given country.

Suppose further that

- the wealth of each household evolves independently according to (4),
- F^* is a stationary distribution for this stochastic process and
- there are many households.

Then F^* is a steady state for the cross-sectional wealth distribution in this country.

In other words, if F^* is the current wealth distribution then it will remain so in subsequent periods, *ceteris paribus*.

To see this, suppose that F^* is the current wealth distribution.

What is the fraction of households with wealth less than y next period?

To obtain this, we sum the probability that wealth is less than y tomorrow, given that current wealth is w, weighted by the fraction of households with wealth w.

Noting that the fraction of households with wealth in interval dw is $F^*(dw)$, we get

$$\int \mathbb{P}\{R_{t+1} sw + y_{t+1} \leq y\} F^*(dw)$$

By the definition of stationarity and the assumption that F^* is stationary for the wealth process, this is just $F^*(y)$.

Hence the fraction of households with wealth in $[0, y]$ is the same next period as it is this period.

Since y was chosen arbitrarily, the distribution is unchanged.

21.2.5 Conditions for Stationarity

The Kesten process $X_{t+1} = a_{t+1}X_t + \eta_{t+1}$ does not always have a stationary distribution.

For example, if $a_t \equiv \eta_t \equiv 1$ for all t, then $X_t = X_0 + t$, which diverges to infinity.

To prevent this kind of divergence, we require that $\{a_t\}$ is strictly less than 1 most of the time.

In particular, if

$$\mathbb{E} \ln a_t < 0 \quad \text{and} \quad \mathbb{E}\eta_t < \infty \tag{7}$$

then a unique stationary distribution exists on $\mathbb{R}_+$.

- See, for example, theorem 2.1.3 of [BDM+16], which provides slightly weaker conditions.

As one application of this result, we see that the wealth process (4) will have a unique stationary distribution whenever labor income has finite mean and $\mathbb{E} \ln R_t + \ln s < 0$.

21.3 Heavy Tails

Under certain conditions, the stationary distribution of a Kesten process has a Pareto tail.

(See our *earlier lecture* on heavy-tailed distributions for background.)

This fact is significant for economics because of the prevalence of Pareto-tailed distributions.

21.3.1 The Kesten–Goldie Theorem

To state the conditions under which the stationary distribution of a Kesten process has a Pareto tail, we first recall that a random variable is called **nonarithmetic** if its distribution is not concentrated on $\{\dots, -2t, -t, 0, t, 2t, \dots\}$ for any $t \geq 0$.

For example, any random variable with a density is nonarithmetic.

The famous Kesten–Goldie Theorem (see, e.g., [BDM+16], theorem 2.4.4) states that if

1. the stationarity conditions in (7) hold,
2. the random variable a_t is positive with probability one and nonarithmetic,
3. $\mathbb{P}\{a_t x + \eta_t = x\} < 1$ for all $x \in \mathbb{R}_+$ and
4. there exists a positive constant α such that

$$\mathbb{E}a_t^\alpha = 1, \quad \mathbb{E}\eta_t^\alpha < \infty, \quad \text{and} \quad \mathbb{E}[a_t^{\alpha+1}] < \infty$$

then the stationary distribution of the Kesten process has a Pareto tail with tail index α.

More precisely, if F^* is the unique stationary distribution and $X^* \sim F^*$, then

$$\lim_{x \to \infty} x^\alpha \mathbb{P}\{X^* > x\} = c$$

for some positive constant c.

21.3.2 Intuition

Later we will illustrate the Kesten–Goldie Theorem using rank-size plots.

Prior to doing so, we can give the following intuition for the conditions.

Two important conditions are that $\mathbb{E} \ln a_t < 0$, so the model is stationary, and $\mathbb{E} a_t^\alpha = 1$ for some $\alpha > 0$.

The first condition implies that the distribution of a_t has a large amount of probability mass below 1.

The second condition implies that the distribution of a_t has at least some probability mass at or above 1.

The first condition gives us existence of the stationary condition.

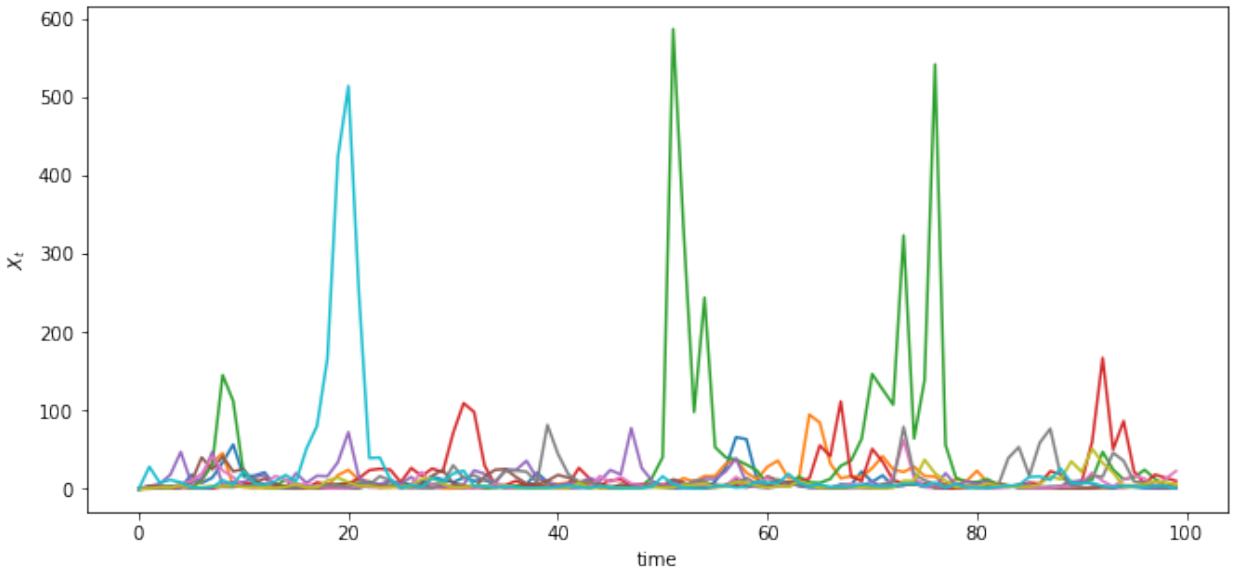
The second condition means that the current state can be expanded by a_t.

If this occurs for several concurrent periods, the effects compound each other, since a_t is multiplicative.

This leads to spikes in the time series, which fill out the extreme right hand tail of the distribution.

The spikes in the time series are visible in the following simulation, which generates of 10 paths when a_t and b_t are lognormal.

```
μ = -0.5
σ = 1.0

def kesten_ts(ts_length=100):
    x = np.zeros(ts_length)
    for t in range(ts_length-1):
        a = np.exp(μ + σ * np.random.randn())
        b = np.exp(np.random.randn())
        x[t+1] = a * x[t] + b
    return x

fig, ax = plt.subplots()

num_paths = 10
np.random.seed(12)

for i in range(num_paths):
    ax.plot(kesten_ts())

ax.set(xlabel='time', ylabel='$X_t$')
plt.show()
```



21.4 Application: Firm Dynamics

As noted in our *lecture on heavy tails*, for common measures of firm size such as revenue or employment, the US firm size distribution exhibits a Pareto tail (see, e.g., [Axt01], [Gab16]).

Let us try to explain this rather striking fact using the Kesten–Goldie Theorem.

21.4.1 Gibrat's Law

It was postulated many years ago by Robert Gibrat [Gib31] that firm size evolves according to a simple rule whereby size next period is proportional to current size.

This is now known as Gibrat's law of proportional growth.

We can express this idea by stating that a suitably defined measure s_t of firm size obeys

$$\frac{s_{t+1}}{s_t} = a_{t+1} \tag{8}$$

for some positive IID sequence $\{a_t\}$.

One implication of Gibrat's law is that the growth rate of individual firms does not depend on their size.

However, over the last few decades, research contradicting Gibrat's law has accumulated in the literature.

For example, it is commonly found that, on average,

1. small firms grow faster than large firms (see, e.g., [Eva87] and [Hal87]) and
2. the growth rate of small firms is more volatile than that of large firms [DRS89].

On the other hand, Gibrat's law is generally found to be a reasonable approximation for large firms [Eva87].

We can accommodate these empirical findings by modifying (8) to

$$s_{t+1} = a_{t+1} s_t + b_{t+1} \tag{9}$$

where $\{a_t\}$ and $\{b_t\}$ are both IID and independent of each other.

In the exercises you are asked to show that (9) is more consistent with the empirical findings presented above than Gibrat's law in (8).

21.4.2 Heavy Tails

So what has this to do with Pareto tails?

The answer is that (9) is a Kesten process.

If the conditions of the Kesten–Goldie Theorem are satisfied, then the firm size distribution is predicted to have heavy tails — which is exactly what we see in the data.

In the exercises below we explore this idea further, generalizing the firm size dynamics and examining the corresponding rank-size plots.

We also try to illustrate why the Pareto tail finding is significant for quantitative analysis.

21.5 Exercises

21.5.1 Exercise 1

Simulate and plot 15 years of daily returns (consider each year as having 250 working days) using the GARCH(1, 1) process in (2)–(3).

Take ξ_t and ζ_t to be independent and standard normal.

Set $\alpha_0 = 0.00001, \alpha_1 = 0.1, \beta = 0.9$ and $\sigma_0 = 0$.

Compare visually with the Nasdaq Composite Index returns *shown above*.

While the time path differs, you should see bursts of high volatility.

21.5.2 Exercise 2

In our discussion of firm dynamics, it was claimed that (9) is more consistent with the empirical literature than Gibrat's law in (8).

(The empirical literature was reviewed immediately above (9).)

In what sense is this true (or false)?

21.5.3 Exercise 3

Consider an arbitrary Kesten process as given in (1).

Suppose that $\{a_t\}$ is lognormal with parameters (μ, σ).

In other words, each a_t has the same distribution as $\exp(\mu + \sigma Z)$ when Z is standard normal.

Suppose further that $\mathbb{E}\eta_t^r < \infty$ for every $r > 0$, as would be the case if, say, η_t is also lognormal.

Show that the conditions of the Kesten–Goldie theorem are satisfied if and only if $\mu < 0$.

Obtain the value of α that makes the Kesten–Goldie conditions hold.

21.5.4 Exercise 4

One unrealistic aspect of the firm dynamics specified in (9) is that it ignores entry and exit.

In any given period and in any given market, we observe significant numbers of firms entering and exiting the market.

Empirical discussion of this can be found in a famous paper by Hugo Hopenhayn [Hop92].

In the same paper, Hopenhayn builds a model of entry and exit that incorporates profit maximization by firms and market clearing quantities, wages and prices.

In his model, a stationary equilibrium occurs when the number of entrants equals the number of exiting firms.

In this setting, firm dynamics can be expressed as

$$s_{t+1} = e_{t+1} \mathbb{1}\{s_t < \bar{s}\} + (a_{t+1} s_t + b_{t+1}) \mathbb{1}\{s_t \geq \bar{s}\} \tag{10}$$

Here

- the state variable s_t represents productivity (which is a proxy for output and hence firm size),
- the IID sequence $\{e_t\}$ is thought of as a productivity draw for a new entrant and
- the variable $\bar{s}$ is a threshold value that we take as given, although it is determined endogenously in Hopenhayn's model.

The idea behind (10) is that firms stay in the market as long as their productivity s_t remains at or above $\bar{s}$.

- In this case, their productivity updates according to (9).

Firms choose to exit when their productivity s_t falls below $\bar{s}$.

- In this case, they are replaced by a new firm with productivity e_{t+1}.

What can we say about dynamics?

Although (10) is not a Kesten process, it does update in the same way as a Kesten process when s_t is large.

So perhaps its stationary distribution still has Pareto tails?

Your task is to investigate this question via simulation and rank-size plots.

The approach will be to

1. generate M draws of s_T when M and T are large and
2. plot the largest 1,000 of the resulting draws in a rank-size plot.

(The distribution of s_T will be close to the stationary distribution when T is large.)

In the simulation, assume that

- each of a_t, b_t and e_t is lognormal,
- the parameters are

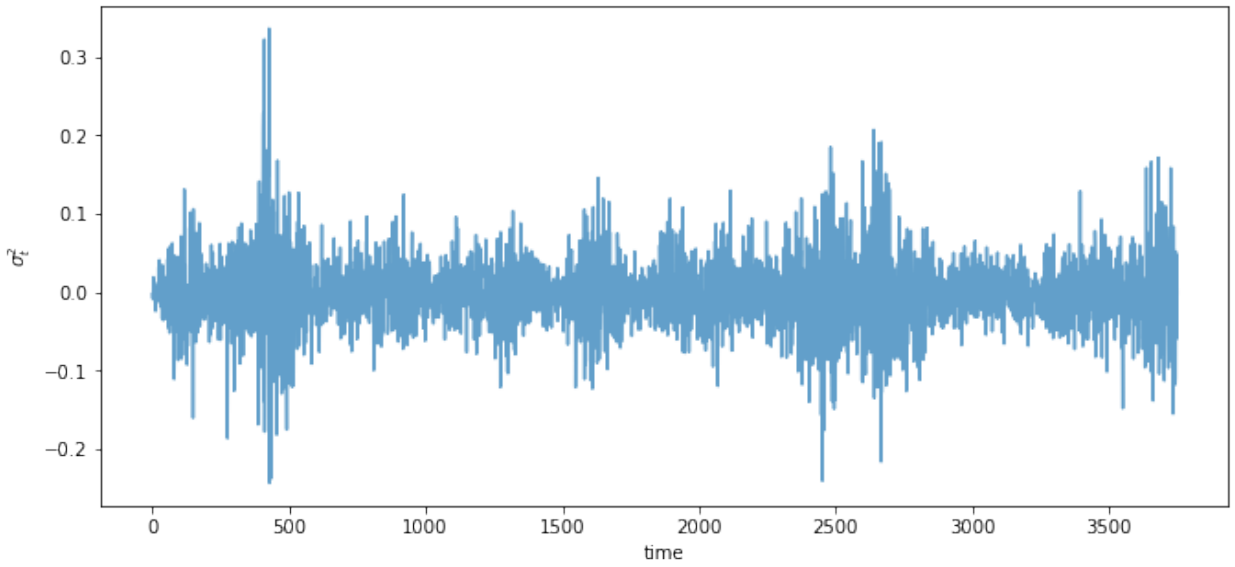
```
μ_a = -0.5        # location parameter for a
σ_a = 0.1         # scale parameter for a
μ_b = 0.0         # location parameter for b
σ_b = 0.5         # scale parameter for b
μ_e = 0.0         # location parameter for e
σ_e = 0.5         # scale parameter for e
s_bar = 1.0       # threshold
T = 500           # sampling date
M = 1_000_000     # number of firms
s_init = 1.0      # initial condition for each firm
```

21.6 Solutions

21.6.1 Exercise 1

Here is one solution:

```
α_0 = 1e-5
α_1 = 0.1
β = 0.9

years = 15
days = years * 250

def garch_ts(ts_length=days):
    σ2 = 0
    r = np.zeros(ts_length)
    for t in range(ts_length-1):
        ξ = np.random.randn()
        σ2 = α_0 + σ2 * (α_1 * ξ**2 + β)
        r[t] = np.sqrt(σ2) * np.random.randn()
    return r

fig, ax = plt.subplots()

np.random.seed(12)

ax.plot(garch_ts(), alpha=0.7)

ax.set(xlabel='time', ylabel='$\\sigma_t^2$')
plt.show()
```



21.6.2 Exercise 2

The empirical findings are that

1. small firms grow faster than large firms and
2. the growth rate of small firms is more volatile than that of large firms.

Also, Gibrat's law is generally found to be a reasonable approximation for large firms than for small firms

The claim is that the dynamics in (9) are more consistent with points 1-2 than Gibrat's law.

To see why, we rewrite (9) in terms of growth dynamics:

$$\frac{s_{t+1}}{s_t} = a_{t+1} + \frac{b_{t+1}}{s_t} \tag{11}$$

Taking $s_t = s$ as given, the mean and variance of firm growth are

$$\mathbb{E}a + \frac{\mathbb{E}b}{s} \quad \text{and} \quad \mathbb{V}a + \frac{\mathbb{V}b}{s^2}$$

Both of these decline with firm size s, consistent with the data.

Moreover, the law of motion (11) clearly approaches Gibrat's law (8) as s_t gets large.

21.6.3 Exercise 3

Since a_t has a density it is nonarithmetic.

Since a_t has the same density as $a = \exp(\mu + \sigma Z)$ when Z is standard normal, we have

$$\mathbb{E} \ln a_t = \mathbb{E}(\mu + \sigma Z) = \mu,$$

and since η_t has finite moments of all orders, the stationarity condition holds if and only if $\mu < 0$.

Given the properties of the lognormal distribution (which has finite moments of all orders), the only other condition in doubt is existence of a positive constant α such that $\mathbb{E}a_t^\alpha = 1$.

This is equivalent to the statement

$$\exp\left(\alpha\mu + \frac{\alpha^2\sigma^2}{2}\right) = 1.$$

Solving for α gives $\alpha = -2\mu/\sigma^2$.

21.6.4 Exercise 4

Here's one solution. First we generate the observations:

```
from numba import njit, prange
from numpy.random import randn


@njit(parallel=True)
def generate_draws(μ_a=-0.5,
                   σ_a=0.1,
                   μ_b=0.0,
```

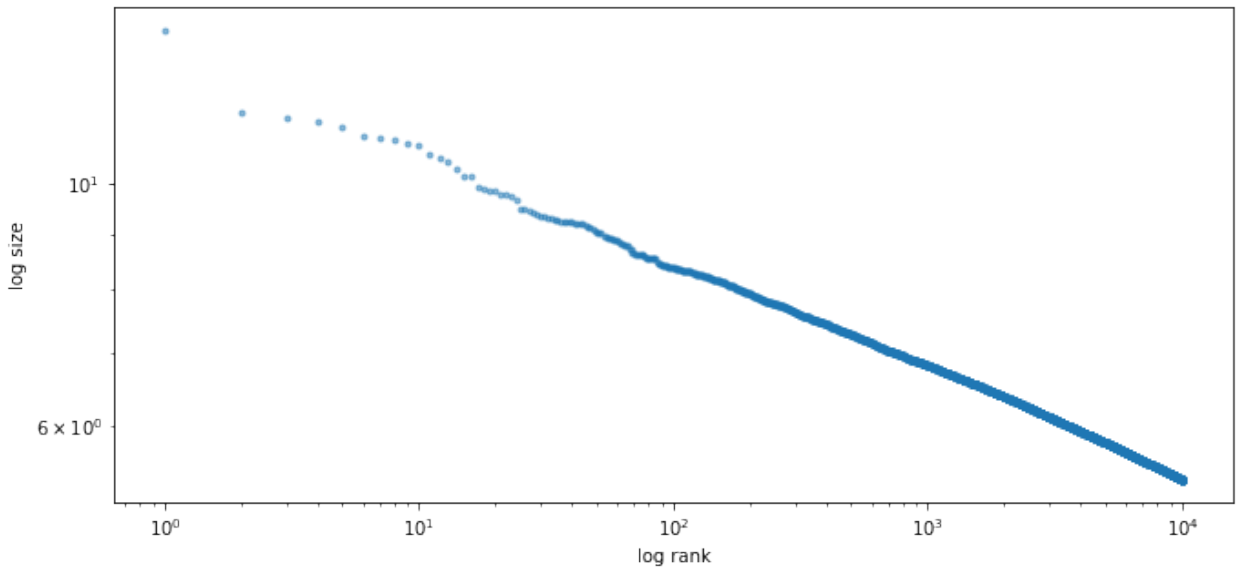
(continues on next page)

(continued from previous page)

```
                    σ_b=0.5,
                    μ_e=0.0,
                    σ_e=0.5,
                    s_bar=1.0,
                    T=500,
                    M=1_000_000,
                    s_init=1.0):

    draws = np.empty(M)
    for m in prange(M):
        s = s_init
        for t in range(T):
            if s < s_bar:
                new_s = np.exp(μ_e + σ_e *  randn())
            else:
                a = np.exp(μ_a + σ_a * randn())
                b = np.exp(μ_b + σ_b * randn())
                new_s = a * s + b
            s = new_s
        draws[m] = s

    return draws

data = generate_draws()
```

Now we produce the rank-size plot:

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

rank_data, size_data = qe.rank_size(data, c=0.01)
ax.loglog(rank_data, size_data, 'o', markersize=3.0, alpha=0.5)
ax.set_xlabel("log rank")
ax.set_ylabel("log size")

plt.show()
```



The plot produces a straight line, consistent with a Pareto tail.

CHAPTER

TWENTYTWO

WEALTH DISTRIBUTION DYNAMICS

Contents

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

```
!conda install -y quantecon
```

22.1 Overview

This notebook gives an introduction to wealth distribution dynamics, with a focus on

- modeling and computing the wealth distribution via simulation,
- measures of inequality such as the Lorenz curve and Gini coefficient, and
- how inequality is affected by the properties of wage income and returns on assets.

One interesting property of the wealth distribution we discuss is Pareto tails.

The wealth distribution in many countries exhibits a Pareto tail

- See *this lecture* for a definition.
- For a review of the empirical evidence, see, for example, [BB18].

This is consistent with high concentration of wealth amongst the richest households.

It also gives us a way to quantify such concentration, in terms of the tail index.

One question of interest is whether or not we can replicate Pareto tails from a relatively simple model.

22.1.1 A Note on Assumptions

The evolution of wealth for any given household depends on their savings behavior.

Modeling such behavior will form an important part of this lecture series.

However, in this particular lecture, we will be content with rather ad hoc (but plausible) savings rules.

We do this to more easily explore the implications of different specifications of income dynamics and investment returns.

At the same time, all of the techniques discussed here can be plugged into models that use optimization to obtain savings rules.

We will use the following imports.

```
%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
from numba import njit, float64, prange
from numba.experimental import jitclass
```

22.2 Lorenz Curves and the Gini Coefficient

Before we investigate wealth dynamics, we briefly review some measures of inequality.

22.2.1 Lorenz Curves

One popular graphical measure of inequality is the Lorenz curve.

The package QuantEcon.py, already imported above, contains a function to compute Lorenz curves.

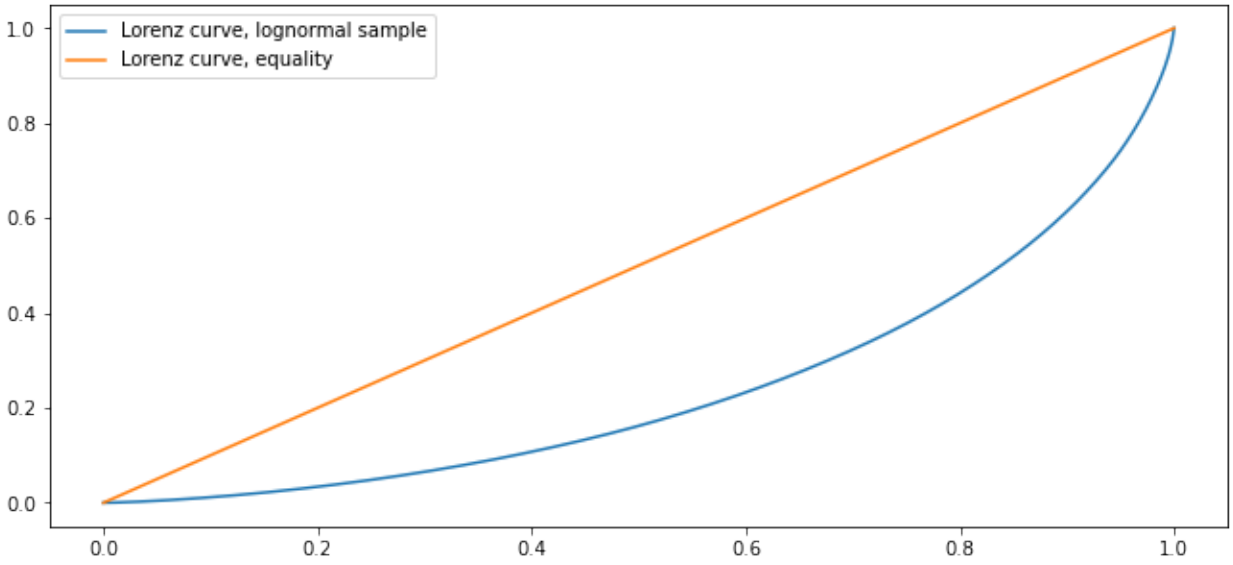
To illustrate, suppose that

```
n = 10_000                       # size of sample
w = np.exp(np.random.randn(n))   # lognormal draws
```

is data representing the wealth of 10,000 households.

We can compute and plot the Lorenz curve as follows:

```
f_vals, l_vals = qe.lorenz_curve(w)

fig, ax = plt.subplots()
ax.plot(f_vals, l_vals, label='Lorenz curve, lognormal sample')
ax.plot(f_vals, f_vals, label='Lorenz curve, equality')
ax.legend()
plt.show()
```



This curve can be understood as follows: if point (x, y) lies on the curve, it means that, collectively, the bottom $(100x)\%$ of the population holds $(100y)\%$ of the wealth.

The "equality" line is the 45 degree line (which might not be exactly 45 degrees in the figure, depending on the aspect ratio).

A sample that produces this line exhibits perfect equality.

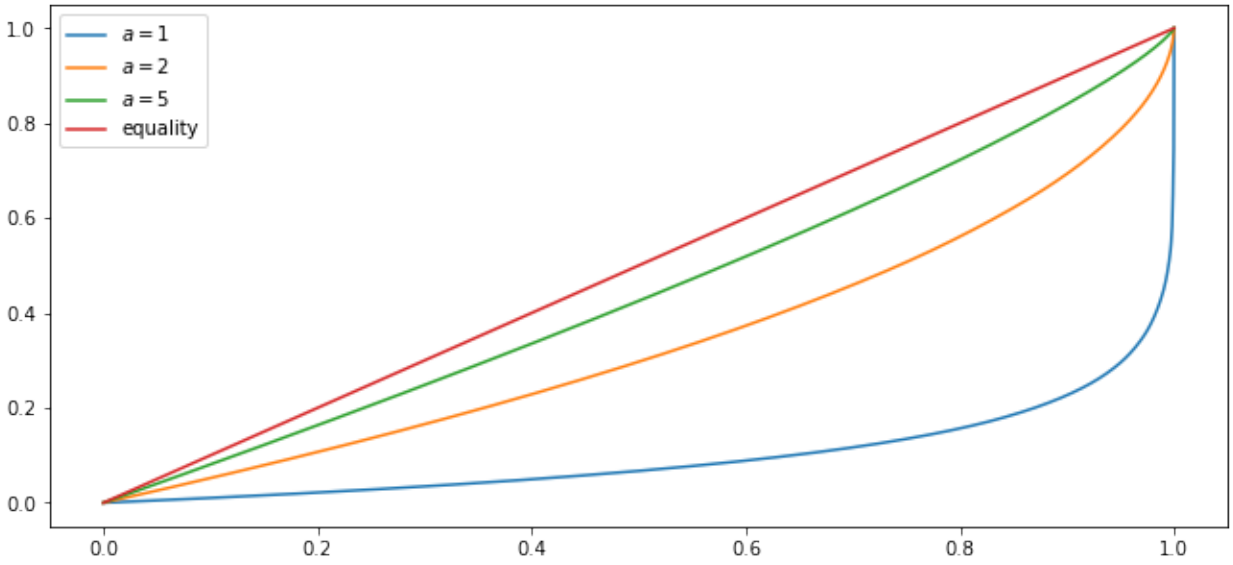
The other line in the figure is the Lorenz curve for the lognormal sample, which deviates significantly from perfect equality.

For example, the bottom 80% of the population holds around 40% of total wealth.

Here is another example, which shows how the Lorenz curve shifts as the underlying distribution changes.

We generate 10,000 observations using the Pareto distribution with a range of parameters, and then compute the Lorenz curve corresponding to each set of observations.

```
a_vals = (1, 2, 5)              # Pareto tail index
n = 10_000                      # size of each sample
fig, ax = plt.subplots()
for a in a_vals:
    u = np.random.uniform(size=n)
    y = u**(-1/a)               # distributed as Pareto with tail index a
    f_vals, l_vals = qe.lorenz_curve(y)
    ax.plot(f_vals, l_vals, label=f'$a = {a}$')
ax.plot(f_vals, f_vals, label='equality')
ax.legend()
plt.show()
```



You can see that, as the tail parameter of the Pareto distribution increases, inequality decreases.

This is to be expected, because a higher tail index implies less weight in the tail of the Pareto distribution.

22.2.2 The Gini Coefficient

The definition and interpretation of the Gini coefficient can be found on the corresponding Wikipedia page.

A value of 0 indicates perfect equality (corresponding the case where the Lorenz curve matches the 45 degree line) and a value of 1 indicates complete inequality (all wealth held by the richest household).

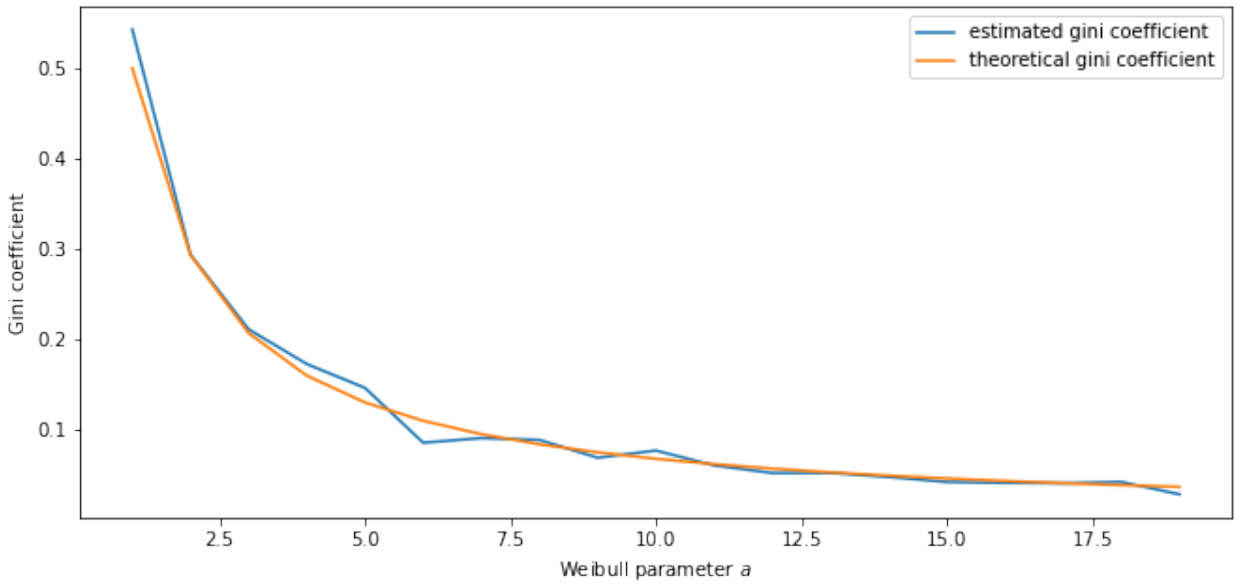
The QuantEcon.py library contains a function to calculate the Gini coefficient.

We can test it on the Weibull distribution with parameter a, where the Gini coefficient is known to be

$$G = 1 - 2^{-1/a}$$

Let's see if the Gini coefficient computed from a simulated sample matches this at each fixed value of a.

```
a_vals = range(1, 20)
ginis = []
ginis_theoretical = []
n = 100

fig, ax = plt.subplots()
for a in a_vals:
    y = np.random.weibull(a, size=n)
    ginis.append(qe.gini_coefficient(y))
    ginis_theoretical.append(1 - 2**(-1/a))
ax.plot(a_vals, ginis, label='estimated gini coefficient')
ax.plot(a_vals, ginis_theoretical, label='theoretical gini coefficient')
ax.legend()
ax.set_xlabel("Weibull parameter $a$")
ax.set_ylabel("Gini coefficient")
plt.show()
```



The simulation shows that the fit is good.

22.3 A Model of Wealth Dynamics

Having discussed inequality measures, let us now turn to wealth dynamics.

The model we will study is

$$w_{t+1} = (1 + r_{t+1})s(w_t) + y_{t+1} \tag{1}$$

where

- w_t is wealth at time t for a given household,
- r_t is the rate of return of financial assets,
- y_t is current non-financial (e.g., labor) income and
- $s(w_t)$ is current wealth net of consumption

Letting $\{z_t\}$ be a correlated state process of the form

$$z_{t+1} = az_t + b + \sigma_z \epsilon_{t+1}$$

we'll assume that

$$R_t := 1 + r_t = c_r \exp(z_t) + \exp(\mu_r + \sigma_r \xi_t)$$

and

$$y_t = c_y \exp(z_t) + \exp(\mu_y + \sigma_y \zeta_t)$$

Here $\{(\epsilon_t, \xi_t, \zeta_t)\}$ is IID and standard normal in $\mathbb{R}^3$.

The value of c_r should be close to zero, since rates of return on assets do not exhibit large trends.

When we simulate a population of households, we will assume all shocks are idiosyncratic (i.e., specific to individual households and independent across them).

Regarding the savings function s, our default model will be

$$s(w) = s_0 w \cdot \mathbb{1}\{w \geq \hat{w}\} \tag{2}$$

where s_0 is a positive constant.

Thus, for $w < \hat{w}$, the household saves nothing. For $w \geq \hat{w}$, the household saves a fraction s_0 of their wealth.

We are using something akin to a fixed savings rate model, while acknowledging that low wealth households tend to save very little.

22.4 Implementation

Here's some type information to help Numba.

```
wealth_dynamics_data = [
    ('w_hat',  float64),     # savings parameter
    ('s_0',    float64),     # savings parameter
    ('c_y',    float64),     # labor income parameter
    ('μ_y',    float64),     # labor income paraemter
    ('σ_y',    float64),     # labor income parameter
    ('c_r',    float64),     # rate of return parameter
    ('μ_r',    float64),     # rate of return parameter
    ('σ_r',    float64),     # rate of return parameter
    ('a',      float64),     # aggregate shock parameter
    ('b',      float64),     # aggregate shock parameter
    ('σ_z',    float64),     # aggregate shock parameter
    ('z_mean', float64),     # mean of z process
    ('z_var', float64),      # variance of z process
    ('y_mean', float64),     # mean of y process
    ('R_mean', float64)      # mean of R process
]
```

Here's a class that stores instance data and implements methods that update the aggregate state and household wealth.

```
@jitclass(wealth_dynamics_data)
class WealthDynamics:

    def __init__(self,
                 w_hat=1.0,
                 s_0=0.75,
                 c_y=1.0,
                 μ_y=1.0,
                 σ_y=0.2,
                 c_r=0.05,
                 μ_r=0.1,
                 σ_r=0.5,
                 a=0.5,
                 b=0.0,
                 σ_z=0.1):

        self.w_hat, self.s_0 = w_hat, s_0
        self.c_y, self.μ_y, self.σ_y = c_y, μ_y, σ_y
        self.c_r, self.μ_r, self.σ_r = c_r, μ_r, σ_r
        self.a, self.b, self.σ_z = a, b, σ_z
```

(continues on next page)

(continued from previous page)

```
        # Record stationary moments
        self.z_mean = b / (1 - a)
        self.z_var = σ_z**2 / (1 - a**2)
        exp_z_mean = np.exp(self.z_mean + self.z_var / 2)
        self.R_mean = c_r * exp_z_mean + np.exp(μ_r + σ_r**2 / 2)
        self.y_mean = c_y * exp_z_mean + np.exp(μ_y + σ_y**2 / 2)

        # Test a stability condition that ensures wealth does not diverge
        # to infinity.
        α = self.R_mean * self.s_0
        if α >= 1:
            raise ValueError("Stability condition failed.")

    def parameters(self):
        """
        Collect and return parameters.
        """
        parameters = (self.w_hat, self.s_0,
                      self.c_y, self.μ_y, self.σ_y,
                      self.c_r, self.μ_r, self.σ_r,
                      self.a, self.b, self.σ_z)
        return parameters

    def update_states(self, w, z):
        """
        Update one period, given current wealth w and persistent
        state z.
        """

        # Simplify names
        params = self.parameters()
        w_hat, s_0, c_y, μ_y, σ_y, c_r, μ_r, σ_r, a, b, σ_z = params
        zp = a * z + b + σ_z * np.random.randn()

        # Update wealth
        y = c_y * np.exp(zp) + np.exp(μ_y + σ_y * np.random.randn())
        wp = y
        if w >= w_hat:
            R = c_r * np.exp(zp) + np.exp(μ_r + σ_r * np.random.randn())
            wp += R * s_0 * w
        return wp, zp
```

Here's function to simulate the time series of wealth for in individual households.

```
@njit
def wealth_time_series(wdy, w_0, n):
    """
    Generate a single time series of length n for wealth given
    initial value w_0.

    The initial persistent state z_0 for each household is drawn from
    the stationary distribution of the AR(1) process.

        * wdy: an instance of WealthDynamics
        * w_0: scalar
        * n: int
```

(continues on next page)

(continued from previous page)

```
    """
    z = wdy.z_mean + np.sqrt(wdy.z_var) * np.random.randn()
    w = np.empty(n)
    w[0] = w_0
    for t in range(n-1):
        w[t+1], z = wdy.update_states(w[t], z)
    return w
```

Now here's function to simulate a cross section of households forward in time.

Note the use of parallelization to speed up computation.

```
@njit(parallel=True)
def update_cross_section(wdy, w_distribution, shift_length=500):
    """
    Shifts a cross-section of household forward in time

    * wdy: an instance of WealthDynamics
    * w_distribution: array_like, represents current cross-section

    Takes a current distribution of wealth values as w_distribution
    and updates each w_t in w_distribution to w_{t+j}, where
    j = shift_length.

    Returns the new distribution.

    """
    new_distribution = np.empty_like(w_distribution)

    # Update each household
    for i in prange(len(new_distribution)):
        z = wdy.z_mean + np.sqrt(wdy.z_var) * np.random.randn()
        w = w_distribution[i]
        for t in range(shift_length-1):
            w, z = wdy.update_states(w, z)
        new_distribution[i] = w
    return new_distribution
```

Parallelization is very effective in the function above because the time path of each household can be calculated independently once the path for the aggregate state is known.

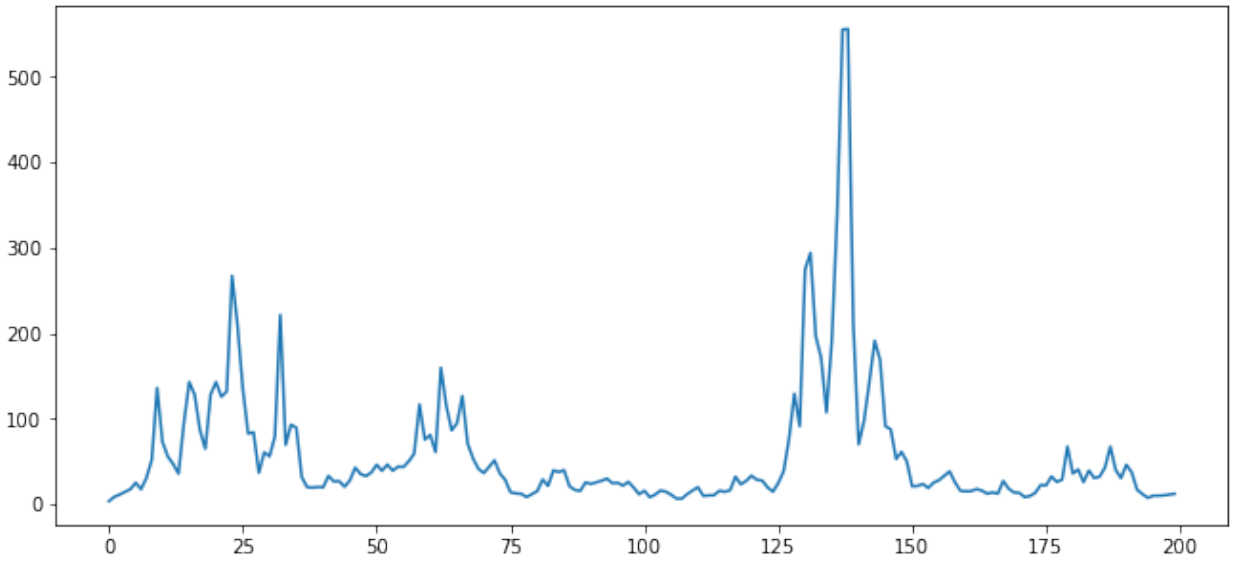
22.5 Applications

Let's try simulating the model at different parameter values and investigate the implications for the wealth distribution.

22.5.1 Time Series

Let's look at the wealth dynamics of an individual household.

```
wdy = WealthDynamics()

ts_length = 200
w = wealth_time_series(wdy, wdy.y_mean, ts_length)

fig, ax = plt.subplots()
ax.plot(w)
plt.show()
```



Notice the large spikes in wealth over time.

Such spikes are similar to what we observed in time series when *we studied Kesten processes*.

22.5.2 Inequality Measures

Let's look at how inequality varies with returns on financial assets.

The next function generates a cross section and then computes the Lorenz curve and Gini coefficient.

```
def generate_lorenz_and_gini(wdy, num_households=100_000, T=500):
    """
    Generate the Lorenz curve data and gini coefficient corresponding to a
    WealthDynamics mode by simulating num_households forward to time T.
    """
    ψ_0 = np.full(num_households, wdy.y_mean)
    z_0 = wdy.z_mean
```

(continues on next page)

(continued from previous page)

```
    ψ_star = update_cross_section(wdy, ψ_0, shift_length=T)
    return qe.gini_coefficient(ψ_star), qe.lorenz_curve(ψ_star)
```

Now we investigate how the Lorenz curves associated with the wealth distribution change as return to savings varies.

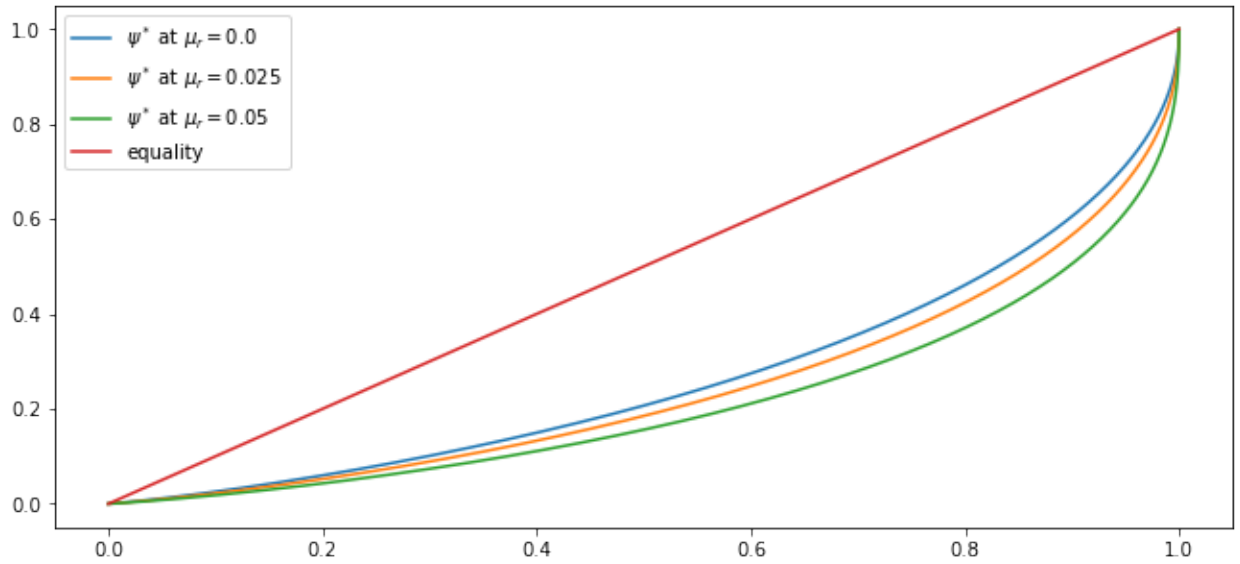
The code below plots Lorenz curves for three different values of μ_r.

If you are running this yourself, note that it will take one or two minutes to execute.

This is unavoidable because we are executing a CPU intensive task.

In fact the code, which is JIT compiled and parallelized, runs extremely fast relative to the number of computations.

```
fig, ax = plt.subplots()
μ_r_vals = (0.0, 0.025, 0.05)
gini_vals = []

for μ_r in μ_r_vals:
    wdy = WealthDynamics(μ_r=μ_r)
    gv, (f_vals, l_vals) = generate_lorenz_and_gini(wdy)
    ax.plot(f_vals, l_vals, label=f'$\psi^*$ at $\mu_r = {μ_r:0.2}$')
    gini_vals.append(gv)

ax.plot(f_vals, f_vals, label='equality')
ax.legend(loc="upper left")
plt.show()
```



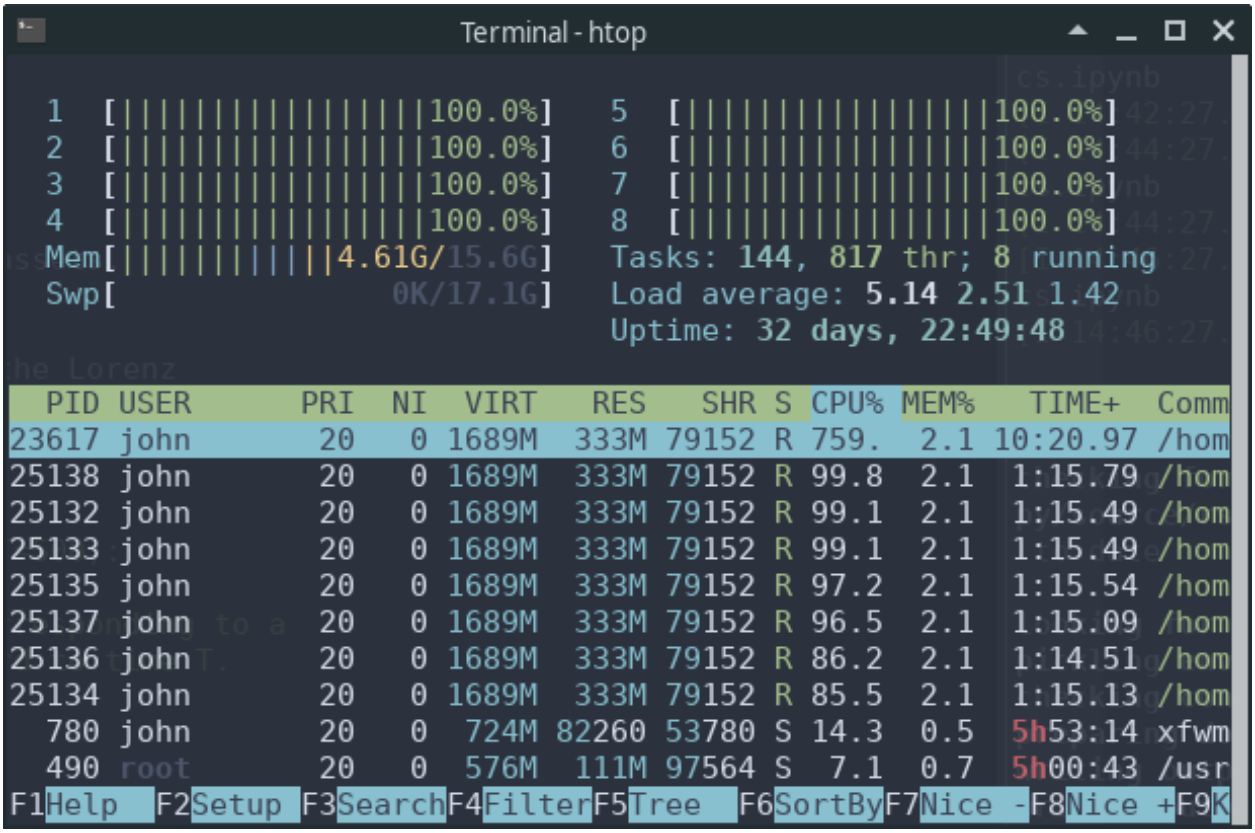
The Lorenz curve shifts downwards as returns on financial income rise, indicating a rise in inequality.

We will look at this again via the Gini coefficient immediately below, but first consider the following image of our system resources when the code above is executing:

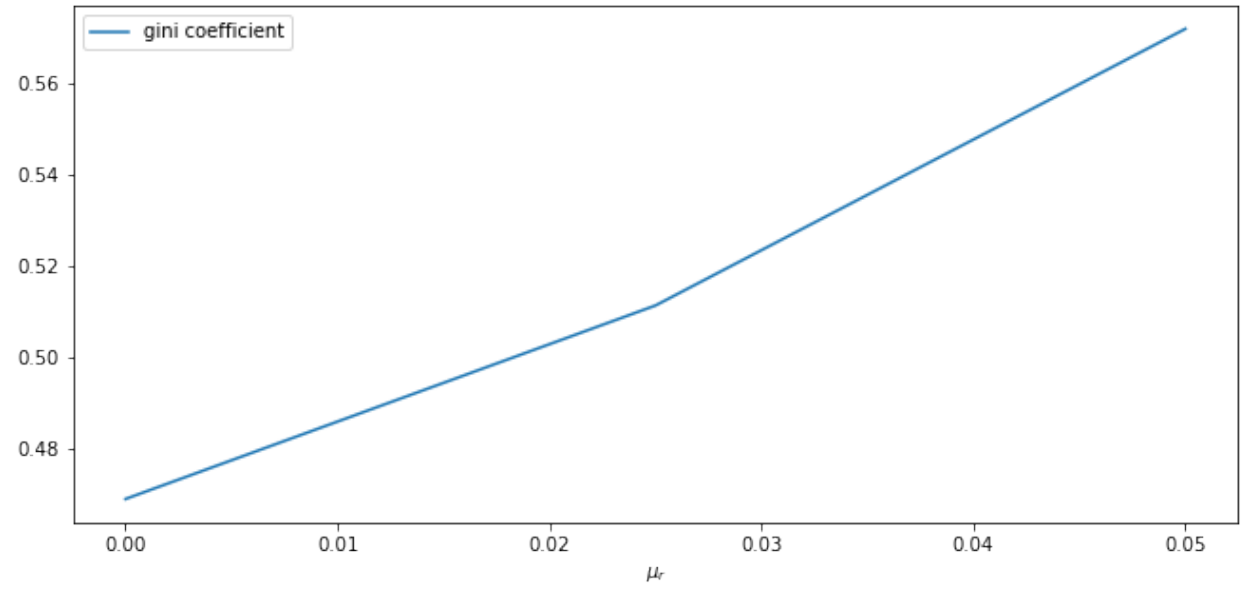
Notice how effectively Numba has implemented multithreading for this routine: all 8 CPUs on our workstation are running at maximum capacity (even though four of them are virtual).

Since the code is both efficiently JIT compiled and fully parallelized, it's close to impossible to make this sequence of tasks run faster without changing hardware.

Now let's check the Gini coefficient.



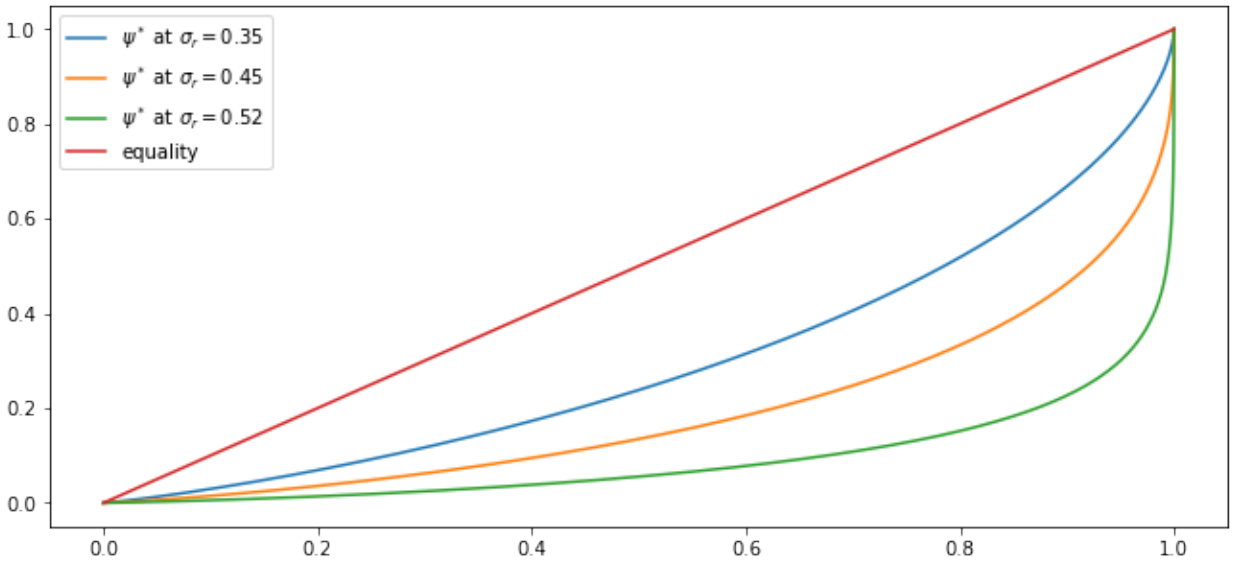

```
fig, ax = plt.subplots()
ax.plot(μ_r_vals, gini_vals, label='gini coefficient')
ax.set_xlabel("$\mu_r$")
ax.legend()
plt.show()
```



Once again, we see that inequality increases as returns on financial income rise.

Let's finish this section by investigating what happens when we change the volatility term σ_r in financial returns.

```
fig, ax = plt.subplots()
σ_r_vals = (0.35, 0.45, 0.52)
gini_vals = []

for σ_r in σ_r_vals:
    wdy = WealthDynamics(σ_r=σ_r)
    gv, (f_vals, l_vals) = generate_lorenz_and_gini(wdy)
    ax.plot(f_vals, l_vals, label=f'$\psi^*$ at $\sigma_r = {σ_r:0.2}$')
    gini_vals.append(gv)

ax.plot(f_vals, f_vals, label='equality')
ax.legend(loc="upper left")
plt.show()
```



We see that greater volatility has the effect of increasing inequality in this model.

22.6 Exercises

22.6.1 Exercise 1

For a wealth or income distribution with Pareto tail, a higher tail index suggests lower inequality.

Indeed, it is possible to prove that the Gini coefficient of the Pareto distribution with tail index a is $1/(2a-1)$.

To the extent that you can, confirm this by simulation.

In particular, generate a plot of the Gini coefficient against the tail index using both the theoretical value just given and the value computed from a sample via `qe.gini_coefficient`.

For the values of the tail index, use `a_vals = np.linspace(1, 10, 25)`.

Use sample of size 1,000 for each a and the sampling method for generating Pareto draws employed in the discussion of Lorenz curves for the Pareto distribution.

To the extent that you can, interpret the monotone relationship between the Gini index and a.

22.6.2 Exercise 2

The wealth process (1) is similar to a *Kesten process*.

This is because, according to (2), savings is constant for all wealth levels above $\hat{w}$.

When savings is constant, the wealth process has the same quasi-linear structure as a Kesten process, with multiplicative and additive shocks.

The Kesten–Goldie theorem tells us that Kesten processes have Pareto tails under a range of parameterizations.

The theorem does not directly apply here, since savings is not always constant and since the multiplicative and additive terms in (1) are not IID.

At the same time, given the similarities, perhaps Pareto tails will arise.

To test this, run a simulation that generates a cross-section of wealth and generate a rank-size plot.

If you like, you can use the function `rank_size` from the `quantecon` library (documentation here).

In viewing the plot, remember that Pareto tails generate a straight line. Is this what you see?

For sample size and initial conditions, use

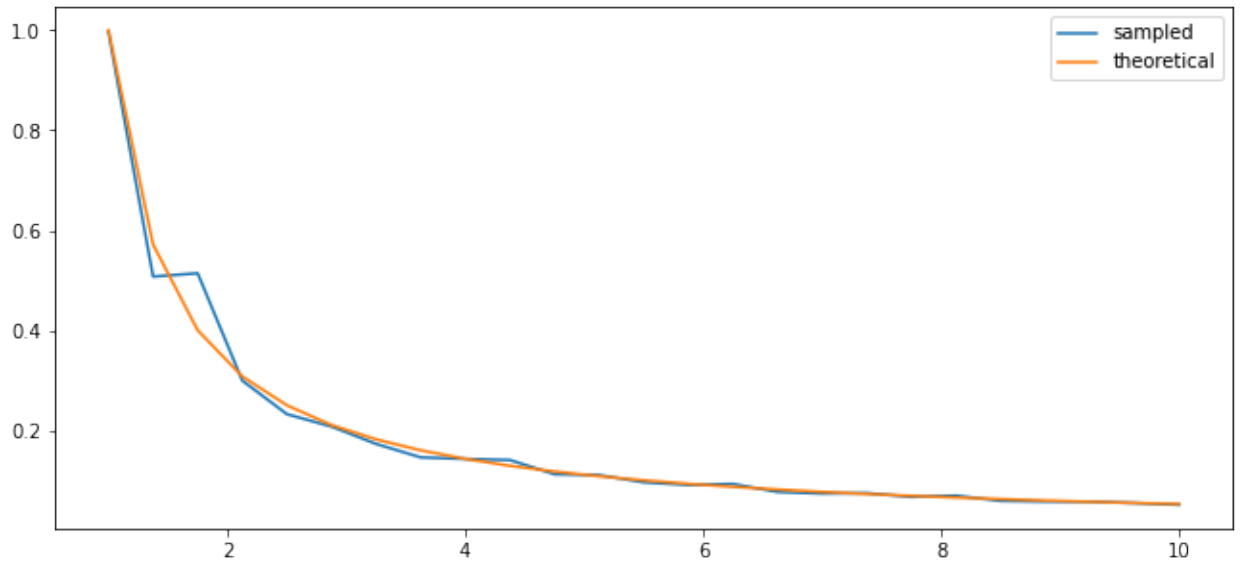
```
num_households = 250_000
T = 500                                    # shift forward T periods
ψ_0 = np.full(num_households, wdy.y_mean)  # initial distribution
z_0 = wdy.z_mean
```

22.7 Solutions

Here is one solution, which produces a good match between theory and simulation.

22.7.1 Exercise 1

```
a_vals = np.linspace(1, 10, 25)  # Pareto tail index
ginis = np.empty_like(a_vals)

n = 1000                         # size of each sample
fig, ax = plt.subplots()
for i, a in enumerate(a_vals):
    y = np.random.uniform(size=n)**(-1/a)
    ginis[i] = qe.gini_coefficient(y)
ax.plot(a_vals, ginis, label='sampled')
ax.plot(a_vals, 1/(2*a_vals - 1), label='theoretical')
ax.legend()
plt.show()
```



In general, for a Pareto distribution, a higher tail index implies less weight in the right hand tail.

This means less extreme values for wealth and hence more equality.

More equality translates to a lower Gini index.

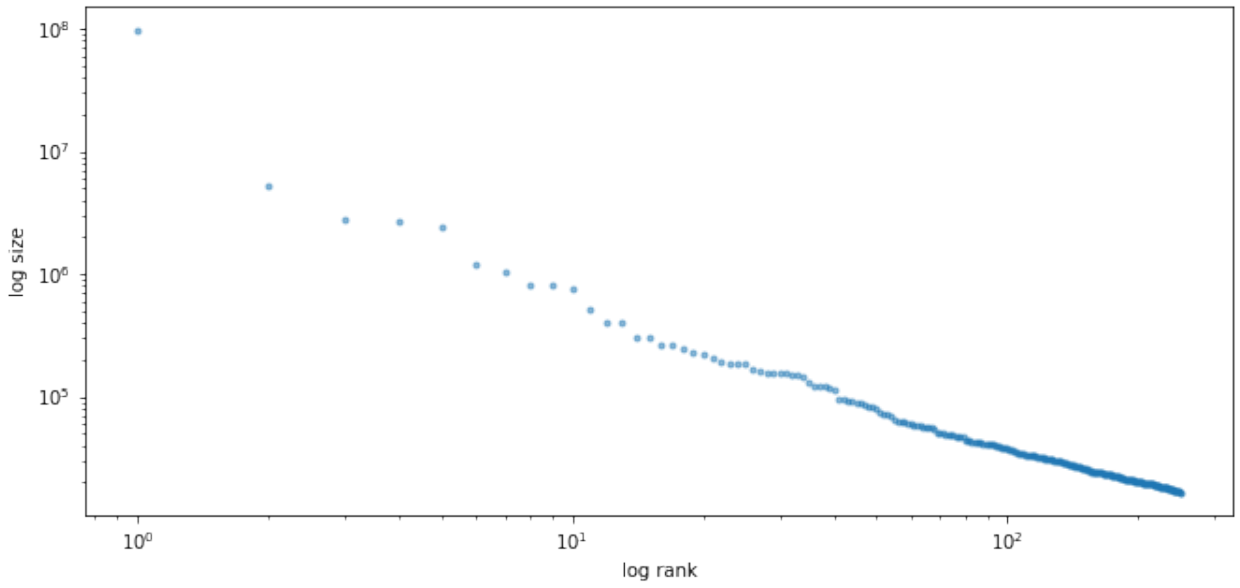
22.7.2 Exercise 2

First let's generate the distribution:

```
num_households = 250_000
T = 500  # how far to shift forward in time
ψ_0 = np.full(num_households, wdy.y_mean)
z_0 = wdy.z_mean

ψ_star = update_cross_section(wdy, ψ_0, shift_length=T)
```

Now let's see the rank-size plot:

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

rank_data, size_data = qe.rank_size(ψ_star, c=0.001)
ax.loglog(rank_data, size_data, 'o', markersize=3.0, alpha=0.5)
ax.set_xlabel("log rank")
ax.set_ylabel("log size")

plt.show()
```


10^8
10^7
10^6
10^5
log size
10^0
10^1
10^2
log rank

CHAPTER

TWENTYTHREE

A FIRST LOOK AT THE KALMAN FILTER

Contents

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

```
!conda install -y quantecon
```

23.1 Overview

This lecture provides a simple and intuitive introduction to the Kalman filter, for those who either

- have heard of the Kalman filter but don't know how it works, or
- know the Kalman filter equations, but don't know where they come from

For additional (more advanced) reading on the Kalman filter, see

- [LS18], section 2.7
- [AM05]

The second reference presents a comprehensive treatment of the Kalman filter.

Required knowledge: Familiarity with matrix manipulations, multivariate normal distributions, covariance matrices, etc.

We'll need the following imports:

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

(continues on next page)

(continued from previous page)

```
import numpy as np
import matplotlib.cm as cm
from quantecon import Kalman, LinearStateSpace
from scipy.stats import norm
from scipy.integrate import quad
from numpy.random import multivariate_normal
from scipy.linalg import eigvals
```

23.2 The Basic Idea

The Kalman filter has many applications in economics, but for now let's pretend that we are rocket scientists.

A missile has been launched from country Y and our mission is to track it.

Let $x \in \mathbb{R}^2$ denote the current location of the missile—a pair indicating latitude-longitude coordinates on a map.

At the present moment in time, the precise location x is unknown, but we do have some beliefs about x.

One way to summarize our knowledge is a point prediction $\hat{x}$

- But what if the President wants to know the probability that the missile is currently over the Sea of Japan?
- Then it is better to summarize our initial beliefs with a bivariate probability density p
  - $\int_E p(x)dx$ indicates the probability that we attach to the missile being in region E.

The density p is called our *prior* for the random variable x.

To keep things tractable in our example, we assume that our prior is Gaussian.

In particular, we take

$$p = N(\hat{x}, \Sigma) \tag{1}$$

where $\hat{x}$ is the mean of the distribution and Σ is a 2×2 covariance matrix. In our simulations, we will suppose that

$$\hat{x} = \begin{pmatrix} 0.2 \\ -0.2 \end{pmatrix}, \qquad \Sigma = \begin{pmatrix} 0.4 & 0.3 \\ 0.3 & 0.45 \end{pmatrix} \tag{2}$$

This density $p(x)$ is shown below as a contour map, with the center of the red ellipse being equal to $\hat{x}$.

```
# Set up the Gaussian prior density p
Σ = [[0.4, 0.3], [0.3, 0.45]]
Σ = np.matrix(Σ)
x_hat = np.matrix([0.2, -0.2]).T
# Define the matrices G and R from the equation y = G x + N(0, R)
G = [[1, 0], [0, 1]]
G = np.matrix(G)
R = 0.5 * Σ
# The matrices A and Q
A = [[1.2, 0], [0, -0.2]]
A = np.matrix(A)
Q = 0.3 * Σ
# The observed value of y
y = np.matrix([2.3, -1.9]).T

# Set up grid for plotting
x_grid = np.linspace(-1.5, 2.9, 100)
```

(continues on next page)

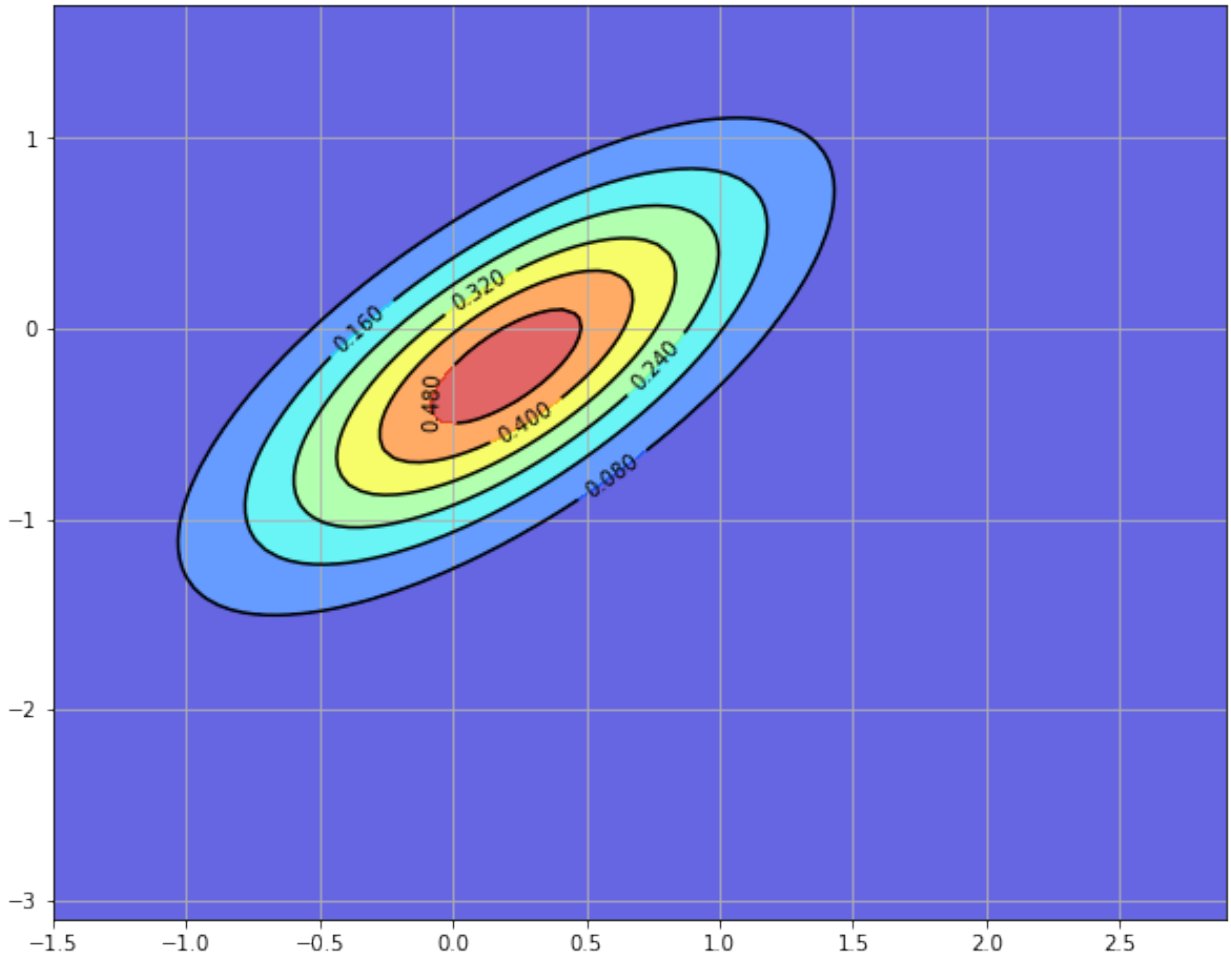
(continued from previous page)

```
y_grid = np.linspace(-3.1, 1.7, 100)
X, Y = np.meshgrid(x_grid, y_grid)

def bivariate_normal(x, y, σ_x=1.0, σ_y=1.0, μ_x=0.0, μ_y=0.0, σ_xy=0.0):
    """
    Compute and return the probability density function of bivariate normal
    distribution of normal random variables x and y

    Parameters
    ----------
    x : array_like(float)
        Random variable

    y : array_like(float)
        Random variable

    σ_x : array_like(float)
          Standard deviation of random variable x

    σ_y : array_like(float)
          Standard deviation of random variable y

    μ_x : scalar(float)
          Mean value of random variable x

    μ_y : scalar(float)
          Mean value of random variable y

    σ_xy : array_like(float)
           Covariance of random variables x and y

    """

    x_μ = x - μ_x
    y_μ = y - μ_y

    ρ = σ_xy / (σ_x * σ_y)
    z = x_μ**2 / σ_x**2 + y_μ**2 / σ_y**2 - 2 * ρ * x_μ * y_μ / (σ_x * σ_y)
    denom = 2 * np.pi * σ_x * σ_y * np.sqrt(1 - ρ**2)
    return np.exp(-z / (2 * (1 - ρ**2))) / denom

def gen_gaussian_plot_vals(μ, C):
    "Z values for plotting the bivariate Gaussian N(μ, C)"
    m_x, m_y = float(μ[0]), float(μ[1])
    s_x, s_y = np.sqrt(C[0, 0]), np.sqrt(C[1, 1])
    s_xy = C[0, 1]
    return bivariate_normal(X, Y, s_x, s_y, m_x, m_y, s_xy)

# Plot the figure

fig, ax = plt.subplots(figsize=(10, 8))
ax.grid()

Z = gen_gaussian_plot_vals(x_hat, Σ)
ax.contourf(X, Y, Z, 6, alpha=0.6, cmap=cm.jet)
cs = ax.contour(X, Y, Z, 6, colors="black")
ax.clabel(cs, inline=1, fontsize=10)
```

(continues on next page)

(continued from previous page)

```
plt.show()
```



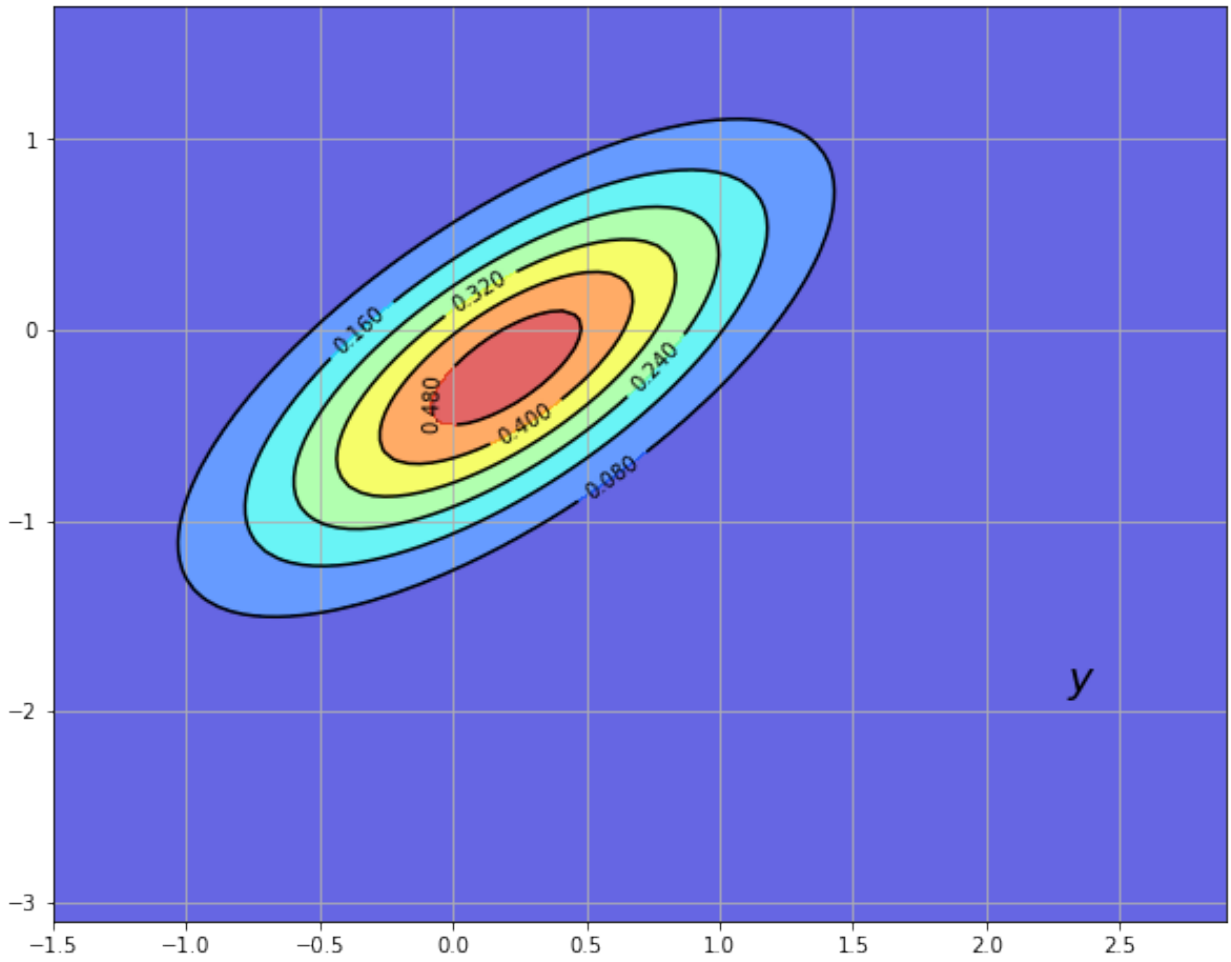
23.2.1 The Filtering Step

We are now presented with some good news and some bad news.

The good news is that the missile has been located by our sensors, which report that the current location is $y = (2.3, -1.9)$.

The next figure shows the original prior $p(x)$ and the new reported location y

```
fig, ax = plt.subplots(figsize=(10, 8))
ax.grid()

Z = gen_gaussian_plot_vals(x_hat, Σ)
ax.contourf(X, Y, Z, 6, alpha=0.6, cmap=cm.jet)
cs = ax.contour(X, Y, Z, 6, colors="black")
ax.clabel(cs, inline=1, fontsize=10)
ax.text(float(y[0]), float(y[1]), "$y$", fontsize=20, color="black")

plt.show()
```



The bad news is that our sensors are imprecise.

In particular, we should interpret the output of our sensor not as $y = x$, but rather as

$$y = Gx + v, \quad \text{where} \quad v \sim N(0, R) \tag{3}$$

Here G and R are 2×2 matrices with R positive definite. Both are assumed known, and the noise term v is assumed to be independent of x.

How then should we combine our prior $p(x) = N(\hat{x}, \Sigma)$ and this new information y to improve our understanding of the location of the missile?

As you may have guessed, the answer is to use Bayes' theorem, which tells us to update our prior $p(x)$ to $p(x \mid y)$ via

$$p(x \mid y) = \frac{p(y \mid x)\, p(x)}{p(y)}$$

where $p(y) = \int p(y \mid x)\, p(x) dx$.

In solving for $p(x \mid y)$, we observe that

- $p(x) = N(\hat{x}, \Sigma)$.
- In view of (3), the conditional density $p(y \mid x)$ is $N(Gx, R)$.
- $p(y)$ does not depend on x, and enters into the calculations only as a normalizing constant.

Because we are in a linear and Gaussian framework, the updated density can be computed by calculating population linear regressions.

In particular, the solution is known[1] to be

$$p(x \mid y) = N(\hat{x}^F, \Sigma^F)$$

where

$$\hat{x}^F := \hat{x} + \Sigma G'(G\Sigma G' + R)^{-1}(y - G\hat{x}) \quad \text{and} \quad \Sigma^F := \Sigma - \Sigma G'(G\Sigma G' + R)^{-1}G\Sigma \tag{4}$$

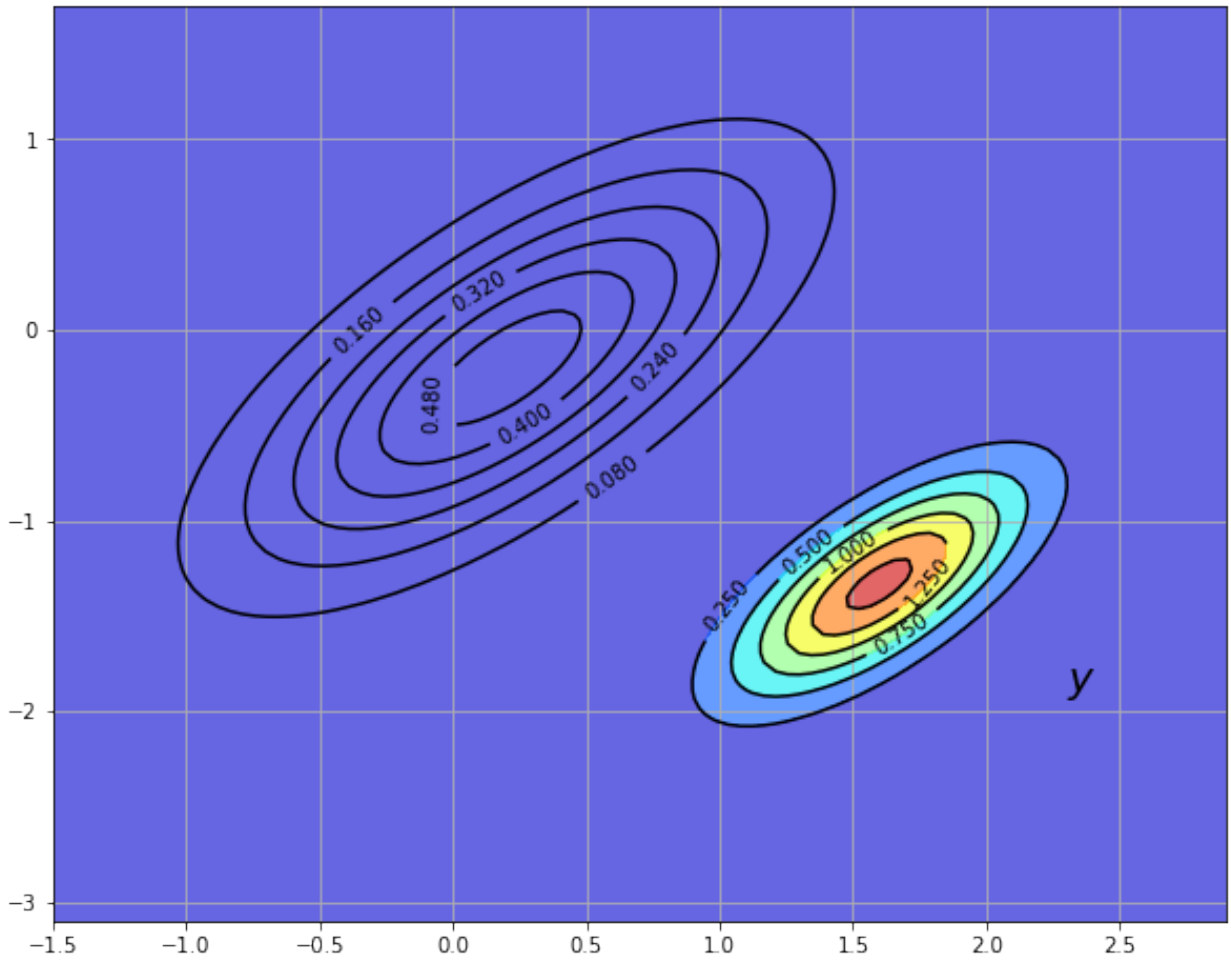
Here $\Sigma G'(G\Sigma G' + R)^{-1}$ is the matrix of population regression coefficients of the hidden object $x - \hat{x}$ on the surprise $y - G\hat{x}$.

This new density $p(x \mid y) = N(\hat{x}^F, \Sigma^F)$ is shown in the next figure via contour lines and the color map.

The original density is left in as contour lines for comparison

```
fig, ax = plt.subplots(figsize=(10, 8))
ax.grid()

Z = gen_gaussian_plot_vals(x_hat, Σ)
cs1 = ax.contour(X, Y, Z, 6, colors="black")
ax.clabel(cs1, inline=1, fontsize=10)
M = Σ * G.T * linalg.inv(G * Σ * G.T + R)
x_hat_F = x_hat + M * (y - G * x_hat)
Σ_F = Σ - M * G * Σ
new_Z = gen_gaussian_plot_vals(x_hat_F, Σ_F)
cs2 = ax.contour(X, Y, new_Z, 6, colors="black")
ax.clabel(cs2, inline=1, fontsize=10)
ax.contourf(X, Y, new_Z, 6, alpha=0.6, cmap=cm.jet)
ax.text(float(y[0]), float(y[1]), "$y$", fontsize=20, color="black")

plt.show()
```

[1] See, for example, page 93 of [Bis06]. To get from his expressions to the ones used above, you will also need to apply the Woodbury matrix identity.



Our new density twists the prior $p(x)$ in a direction determined by the new information $y - G\hat{x}$.

In generating the figure, we set G to the identity matrix and $R = 0.5\Sigma$ for Σ defined in (2).

23.2.2 The Forecast Step

What have we achieved so far?

We have obtained probabilities for the current location of the state (missile) given prior and current information.

This is called "filtering" rather than forecasting because we are filtering out noise rather than looking into the future.

- $p(x \mid y) = N(\hat{x}^F, \Sigma^F)$ is called the *filtering distribution*

But now let's suppose that we are given another task: to predict the location of the missile after one unit of time (whatever that may be) has elapsed.

To do this we need a model of how the state evolves.

Let's suppose that we have one, and that it's linear and Gaussian. In particular,

$$x_{t+1} = Ax_t + w_{t+1}, \quad \text{where} \quad w_t \sim N(0, Q) \tag{5}$$

Our aim is to combine this law of motion and our current distribution $p(x \mid y) = N(\hat{x}^F, \Sigma^F)$ to come up with a new *predictive* distribution for the location in one unit of time.

In view of (5), all we have to do is introduce a random vector $x^F \sim N(\hat{x}^F, \Sigma^F)$ and work out the distribution of $Ax^F + w$ where w is independent of x^F and has distribution $N(0, Q)$.

Since linear combinations of Gaussians are Gaussian, $Ax^F + w$ is Gaussian.

Elementary calculations and the expressions in (4) tell us that

$$\mathbb{E}[Ax^F + w] = A\mathbb{E}x^F + \mathbb{E}w = A\hat{x}^F = A\hat{x} + A\Sigma G'(G\Sigma G' + R)^{-1}(y - G\hat{x})$$

and

$$\text{Var}[Ax^F + w] = A\,\text{Var}[x^F]A' + Q = A\Sigma^F A' + Q = A\Sigma A' - A\Sigma G'(G\Sigma G' + R)^{-1}G\Sigma A' + Q$$

The matrix $A\Sigma G'(G\Sigma G' + R)^{-1}$ is often written as K_Σ and called the *Kalman gain*.

- The subscript Σ has been added to remind us that K_Σ depends on Σ, but not y or $\hat{x}$.

Using this notation, we can summarize our results as follows.

Our updated prediction is the density $N(\hat{x}_{new}, \Sigma_{new})$ where

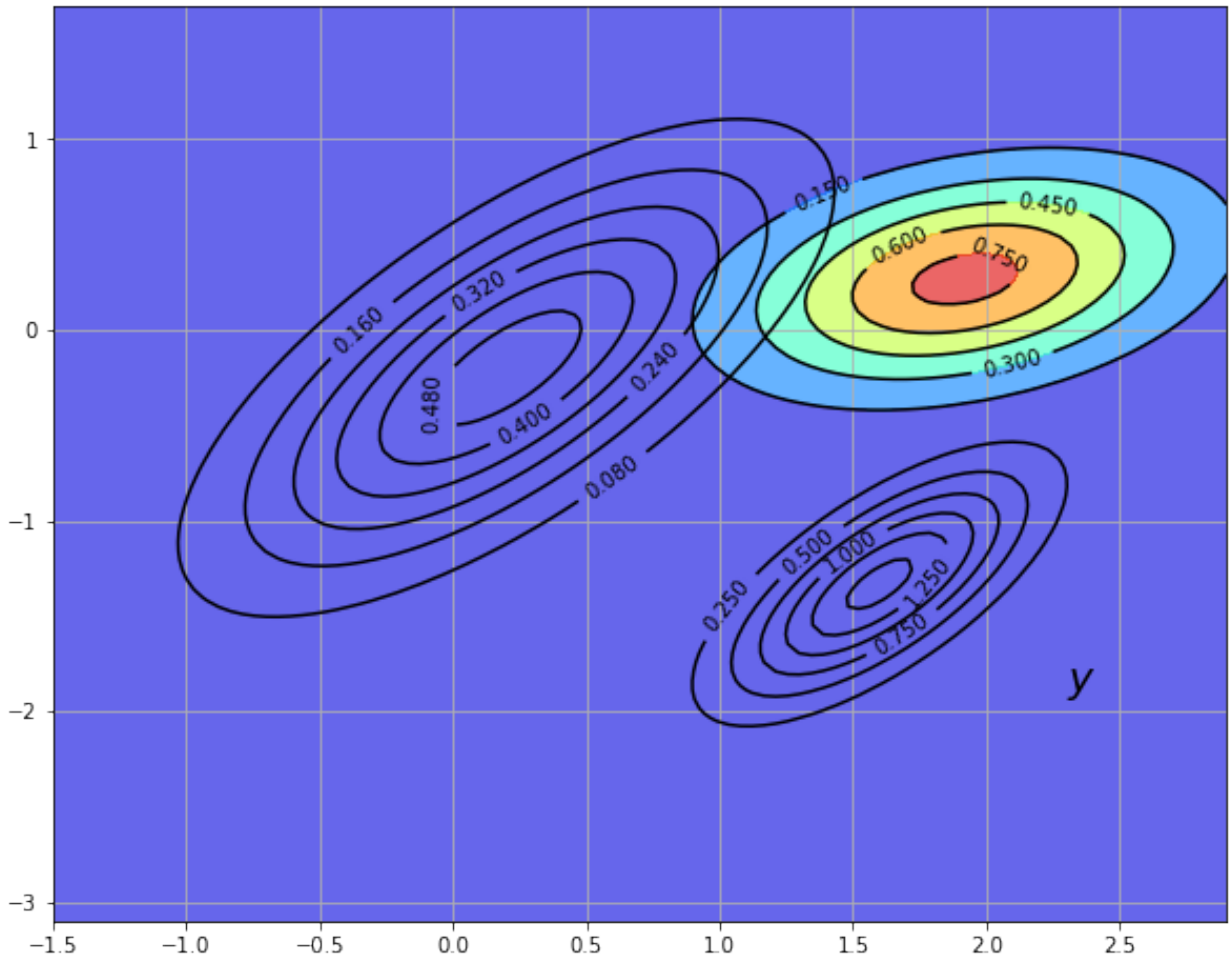
$$\begin{aligned}\hat{x}_{new} &:= A\hat{x} + K_\Sigma(y - G\hat{x}) \\ \Sigma_{new} &:= A\Sigma A' - K_\Sigma G\Sigma A' + Q\end{aligned}$$

- The density $p_{new}(x) = N(\hat{x}_{new}, \Sigma_{new})$ is called the *predictive distribution*

The predictive distribution is the new density shown in the following figure, where the update has used parameters.

$$A = \begin{pmatrix} 1.2 & 0.0 \\ 0.0 & -0.2 \end{pmatrix}, \qquad Q = 0.3 * \Sigma$$

```
fig, ax = plt.subplots(figsize=(10, 8))
ax.grid()

# Density 1
Z = gen_gaussian_plot_vals(x_hat, Σ)
cs1 = ax.contour(X, Y, Z, 6, colors="black")
ax.clabel(cs1, inline=1, fontsize=10)

# Density 2
M = Σ * G.T * linalg.inv(G * Σ * G.T + R)
x_hat_F = x_hat + M * (y - G * x_hat)
Σ_F = Σ - M * G * Σ
Z_F = gen_gaussian_plot_vals(x_hat_F, Σ_F)
cs2 = ax.contour(X, Y, Z_F, 6, colors="black")
ax.clabel(cs2, inline=1, fontsize=10)

# Density 3
new_x_hat = A * x_hat_F
new_Σ = A * Σ_F * A.T + Q
new_Z = gen_gaussian_plot_vals(new_x_hat, new_Σ)
cs3 = ax.contour(X, Y, new_Z, 6, colors="black")
ax.clabel(cs3, inline=1, fontsize=10)
ax.contourf(X, Y, new_Z, 6, alpha=0.6, cmap=cm.jet)
ax.text(float(y[0]), float(y[1]), "$y$", fontsize=20, color="black")

plt.show()
```



23.2.3 The Recursive Procedure

Let's look back at what we've done.

We started the current period with a prior $p(x)$ for the location x of the missile.

We then used the current measurement y to update to $p(x \mid y)$.

Finally, we used the law of motion (5) for $\{x_t\}$ to update to $p_{new}(x)$.

If we now step into the next period, we are ready to go round again, taking $p_{new}(x)$ as the current prior.

Swapping notation $p_t(x)$ for $p(x)$ and $p_{t+1}(x)$ for $p_{new}(x)$, the full recursive procedure is:

1. Start the current period with prior $p_t(x) = N(\hat{x}_t, \Sigma_t)$.
2. Observe current measurement y_t.
3. Compute the filtering distribution $p_t(x \mid y) = N(\hat{x}_t^F, \Sigma_t^F)$ from $p_t(x)$ and y_t, applying Bayes rule and the conditional distribution (3).
4. Compute the predictive distribution $p_{t+1}(x) = N(\hat{x}_{t+1}, \Sigma_{t+1})$ from the filtering distribution and (5).
5. Increment t by one and go to step 1.

Repeating (6), the dynamics for $\hat{x}_t$ and Σ_t are as follows

$$\hat{x}_{t+1} = A\hat{x}_t + K_{\Sigma_t}(y_t - G\hat{x}_t)$$
$$\Sigma_{t+1} = A\Sigma_t A' - K_{\Sigma_t} G\Sigma_t A' + Q$$

These are the standard dynamic equations for the Kalman filter (see, for example, [LS18], page 58).

23.3 Convergence

The matrix Σ_t is a measure of the uncertainty of our prediction $\hat{x}_t$ of x_t.

Apart from special cases, this uncertainty will never be fully resolved, regardless of how much time elapses.

One reason is that our prediction $\hat{x}_t$ is made based on information available at $t-1$, not t.

Even if we know the precise value of x_{t-1} (which we don't), the transition equation (5) implies that $x_t = Ax_{t-1} + w_t$.

Since the shock w_t is not observable at $t-1$, any time $t-1$ prediction of x_t will incur some error (unless w_t is degenerate).

However, it is certainly possible that Σ_t converges to a constant matrix as $t \to \infty$.

To study this topic, let's expand the second equation in (6):

$$\Sigma_{t+1} = A\Sigma_t A' - A\Sigma_t G'(G\Sigma_t G' + R)^{-1} G\Sigma_t A' + Q \tag{6}$$

This is a nonlinear difference equation in Σ_t.

A fixed point of (6) is a constant matrix Σ such that

$$\Sigma = A\Sigma A' - A\Sigma G'(G\Sigma G' + R)^{-1} G\Sigma A' + Q \tag{7}$$

Equation (6) is known as a discrete-time Riccati difference equation.

Equation (7) is known as a discrete-time algebraic Riccati equation.

Conditions under which a fixed point exists and the sequence $\{\Sigma_t\}$ converges to it are discussed in [AHMS96] and [AM05], chapter 4.

A sufficient (but not necessary) condition is that all the eigenvalues λ_i of A satisfy $|\lambda_i| < 1$ (cf. e.g., [AM05], p. 77).

(This strong condition assures that the unconditional distribution of x_t converges as $t \to +\infty$.)

In this case, for any initial choice of Σ_0 that is both non-negative and symmetric, the sequence $\{\Sigma_t\}$ in (6) converges to a non-negative symmetric matrix Σ that solves (7).

23.4 Implementation

The class `Kalman` from the QuantEcon.py package implements the Kalman filter

- Instance data consists of:
  - the moments $(\hat{x}_t, \Sigma_t)$ of the current prior.
  - An instance of the LinearStateSpace class from QuantEcon.py.

The latter represents a linear state space model of the form

$$x_{t+1} = Ax_t + Cw_{t+1}$$
$$y_t = Gx_t + Hv_t$$

where the shocks w_t and v_t are IID standard normals.

To connect this with the notation of this lecture we set

$$Q := CC' \quad \text{and} \quad R := HH'$$

- The class `Kalman` from the QuantEcon.py package has a number of methods, some that we will wait to use until we study more advanced applications in subsequent lectures.
- Methods pertinent for this lecture are:
  - `prior_to_filtered`, which updates $(\hat{x}_t, \Sigma_t)$ to $(\hat{x}_t^F, \Sigma_t^F)$
  - `filtered_to_forecast`, which updates the filtering distribution to the predictive distribution – which becomes the new prior $(\hat{x}_{t+1}, \Sigma_{t+1})$
  - `update`, which combines the last two methods
  - a `stationary_values`, which computes the solution to (7) and the corresponding (stationary) Kalman gain

You can view the program on GitHub.

23.5 Exercises

23.5.1 Exercise 1

Consider the following simple application of the Kalman filter, loosely based on [LS18], section 2.9.2.

Suppose that

- all variables are scalars
- the hidden state $\{x_t\}$ is in fact constant, equal to some $\theta \in \mathbb{R}$ unknown to the modeler

State dynamics are therefore given by (5) with $A = 1$, $Q = 0$ and $x_0 = \theta$.

The measurement equation is $y_t = \theta + v_t$ where v_t is $N(0, 1)$ and IID.

The task of this exercise to simulate the model and, using the code from `kalman.py`, plot the first five predictive densities $p_t(x) = N(\hat{x}_t, \Sigma_t)$.

As shown in [LS18], sections 2.9.1–2.9.2, these distributions asymptotically put all mass on the unknown value θ.

In the simulation, take $\theta = 10$, $\hat{x}_0 = 8$ and $\Sigma_0 = 1$.

Your figure should – modulo randomness – look something like this

23.5.2 Exercise 2

The preceding figure gives some support to the idea that probability mass converges to θ.

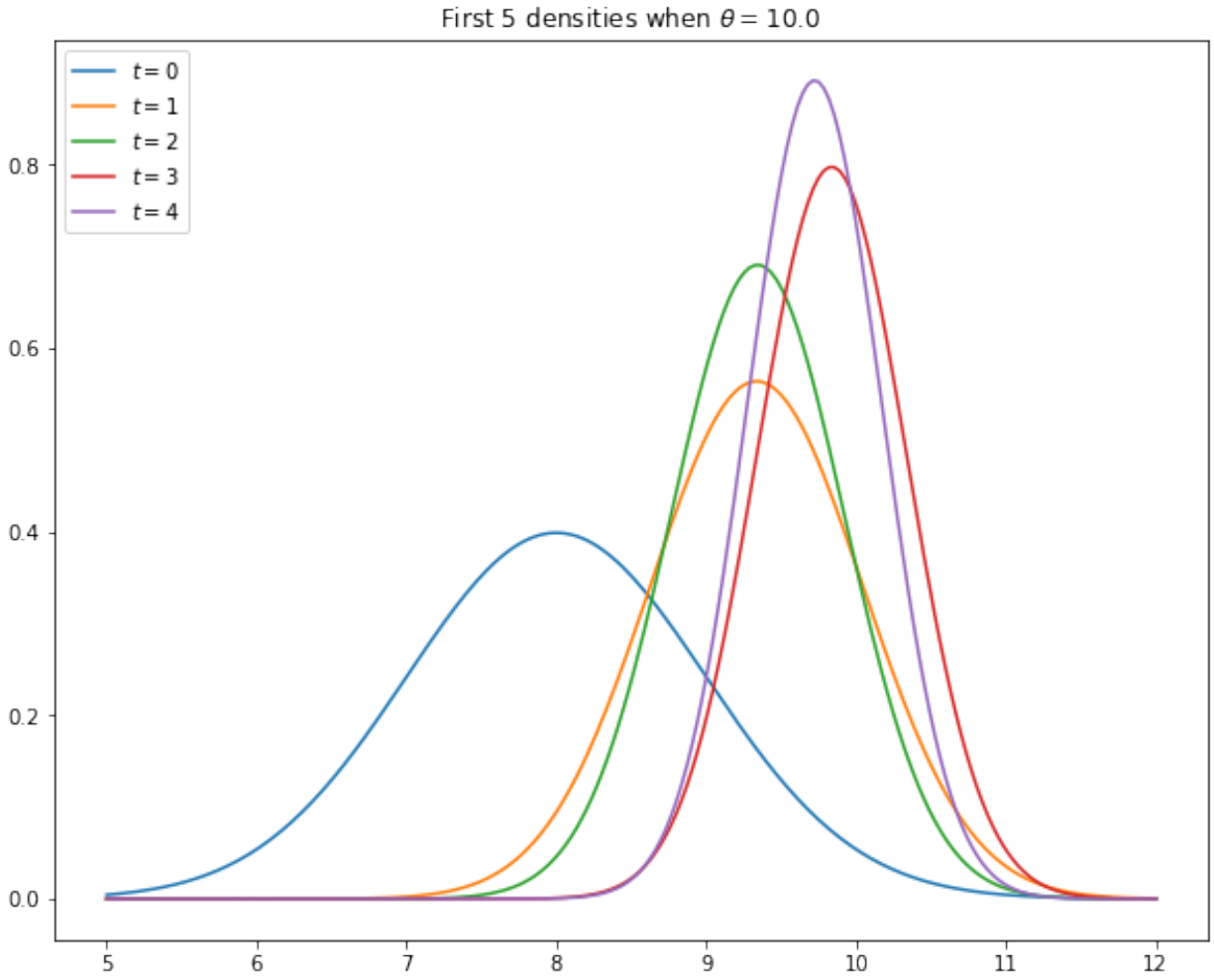
To get a better idea, choose a small $\epsilon > 0$ and calculate

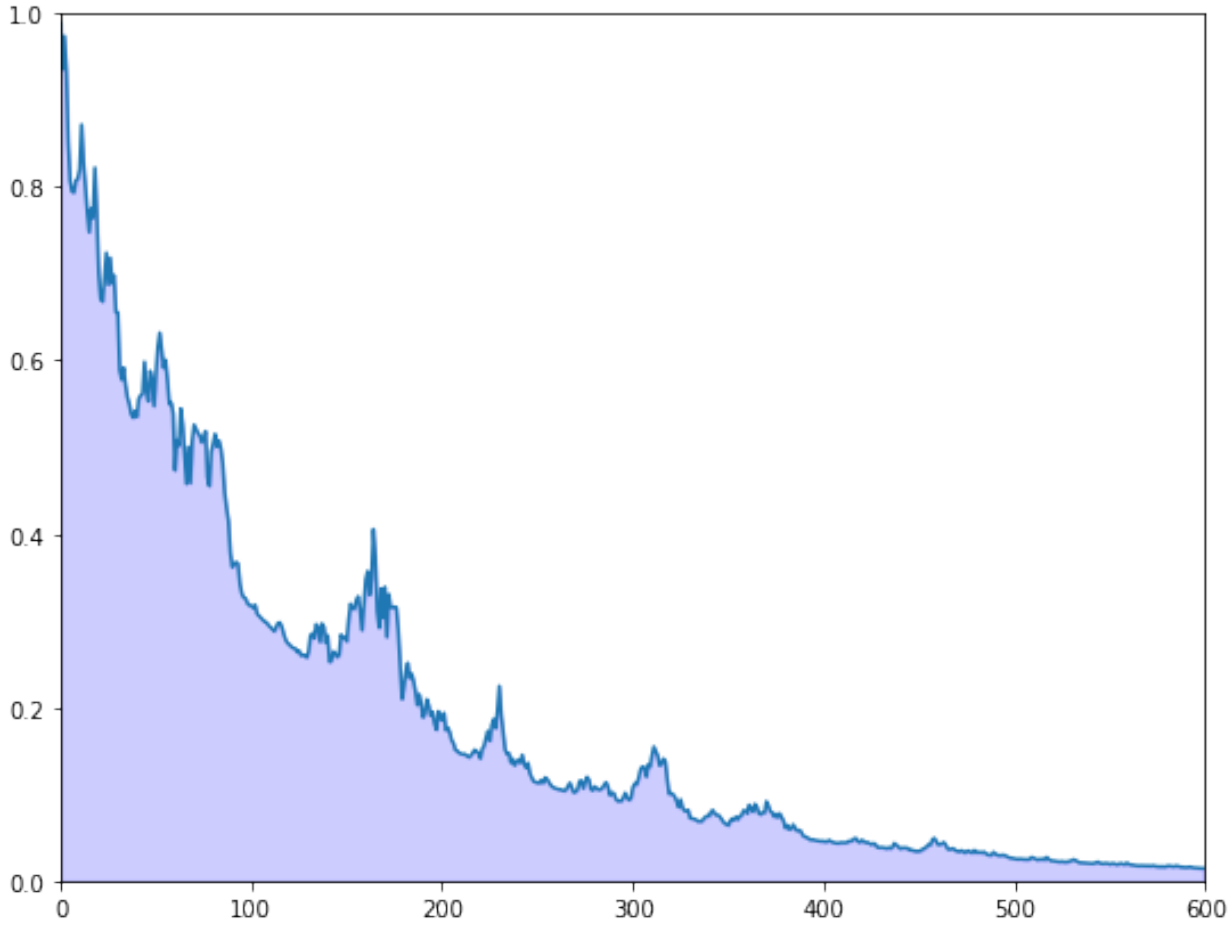
$$z_t := 1 - \int_{\theta-\epsilon}^{\theta+\epsilon} p_t(x)dx$$

for $t = 0, 1, 2, \ldots, T$.

Plot z_t against T, setting $\epsilon = 0.1$ and $T = 600$.

Your figure should show error erratically declining something like this


First 5 densities when $\theta = 10.0$
$t = 0$
$t = 1$
$t = 2$
$t = 3$
$t = 4$
0.0
0.2
0.4
0.6
0.8
5
6
7
8
9
10
11
12


1.0
0.8
0.6
0.4
0.2
0.0
0
100
200
300
400
500
600

23.5.3 Exercise 3

As discussed *above*, if the shock sequence $\{w_t\}$ is not degenerate, then it is not in general possible to predict x_t without error at time $t-1$ (and this would be the case even if we could observe x_{t-1}).

Let's now compare the prediction $\hat{x}_t$ made by the Kalman filter against a competitor who **is** allowed to observe x_{t-1}.

This competitor will use the conditional expectation $\mathbb{E}[x_t \mid x_{t-1}]$, which in this case is Ax_{t-1}.

The conditional expectation is known to be the optimal prediction method in terms of minimizing mean squared error.

(More precisely, the minimizer of $\mathbb{E}\,\|x_t - g(x_{t-1})\|^2$ with respect to g is $g^*(x_{t-1}) := \mathbb{E}[x_t \mid x_{t-1}]$)

Thus we are comparing the Kalman filter against a competitor who has more information (in the sense of being able to observe the latent state) and behaves optimally in terms of minimizing squared error.

Our horse race will be assessed in terms of squared error.

In particular, your task is to generate a graph plotting observations of both $\|x_t - Ax_{t-1}\|^2$ and $\|x_t - \hat{x}_t\|^2$ against t for $t = 1, \ldots, 50$.

For the parameters, set $G = I, R = 0.5I$ and $Q = 0.3I$, where I is the 2×2 identity.

Set

$$A = \begin{pmatrix} 0.5 & 0.4 \\ 0.6 & 0.3 \end{pmatrix}$$

To initialize the prior density, set

$$\Sigma_0 = \begin{pmatrix} 0.9 & 0.3 \\ 0.3 & 0.9 \end{pmatrix}$$

and $\hat{x}_0 = (8, 8)$.

Finally, set $x_0 = (0, 0)$.

You should end up with a figure similar to the following (modulo randomness)

Observe how, after an initial learning period, the Kalman filter performs quite well, even relative to the competitor who predicts optimally with knowledge of the latent state.

23.5.4 Exercise 4

Try varying the coefficient 0.3 in $Q = 0.3I$ up and down.

Observe how the diagonal values in the stationary solution Σ (see (7)) increase and decrease in line with this coefficient.

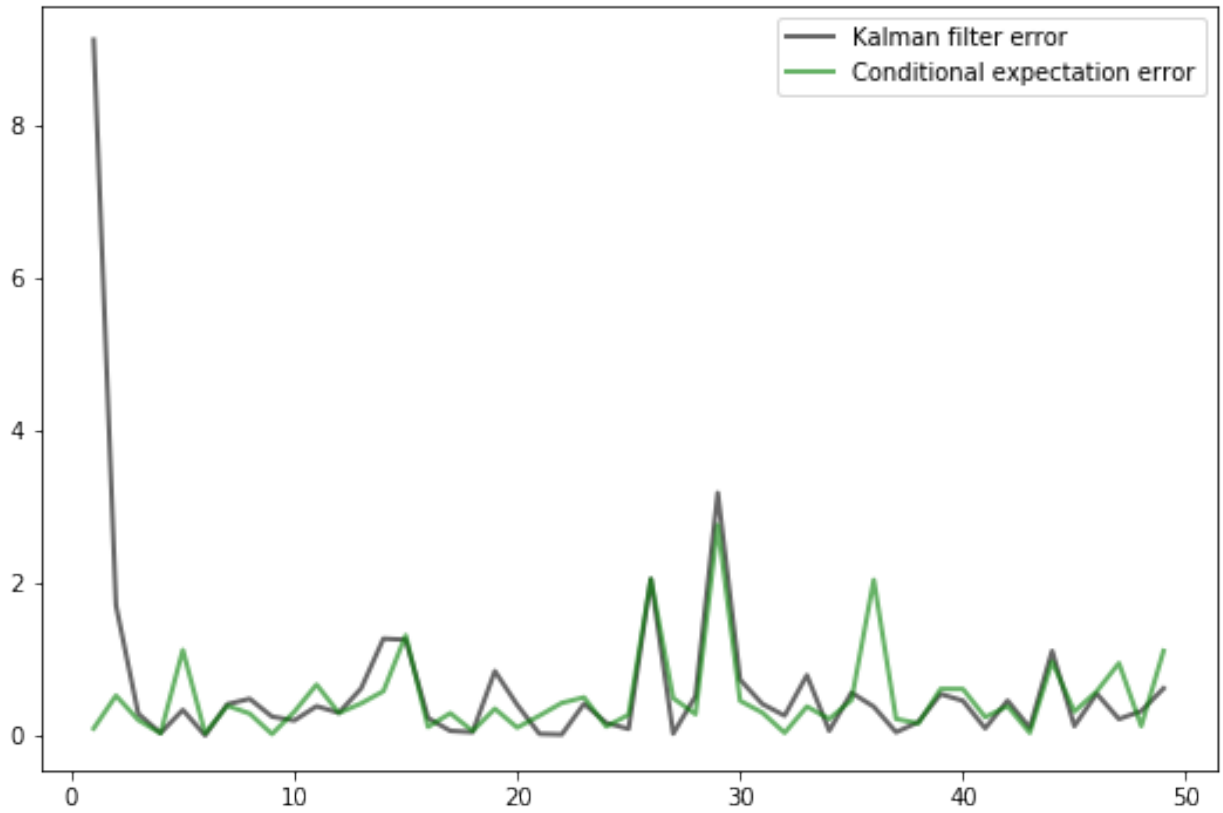
The interpretation is that more randomness in the law of motion for x_t causes more (permanent) uncertainty in prediction.

23.6 Solutions

23.6.1 Exercise 1

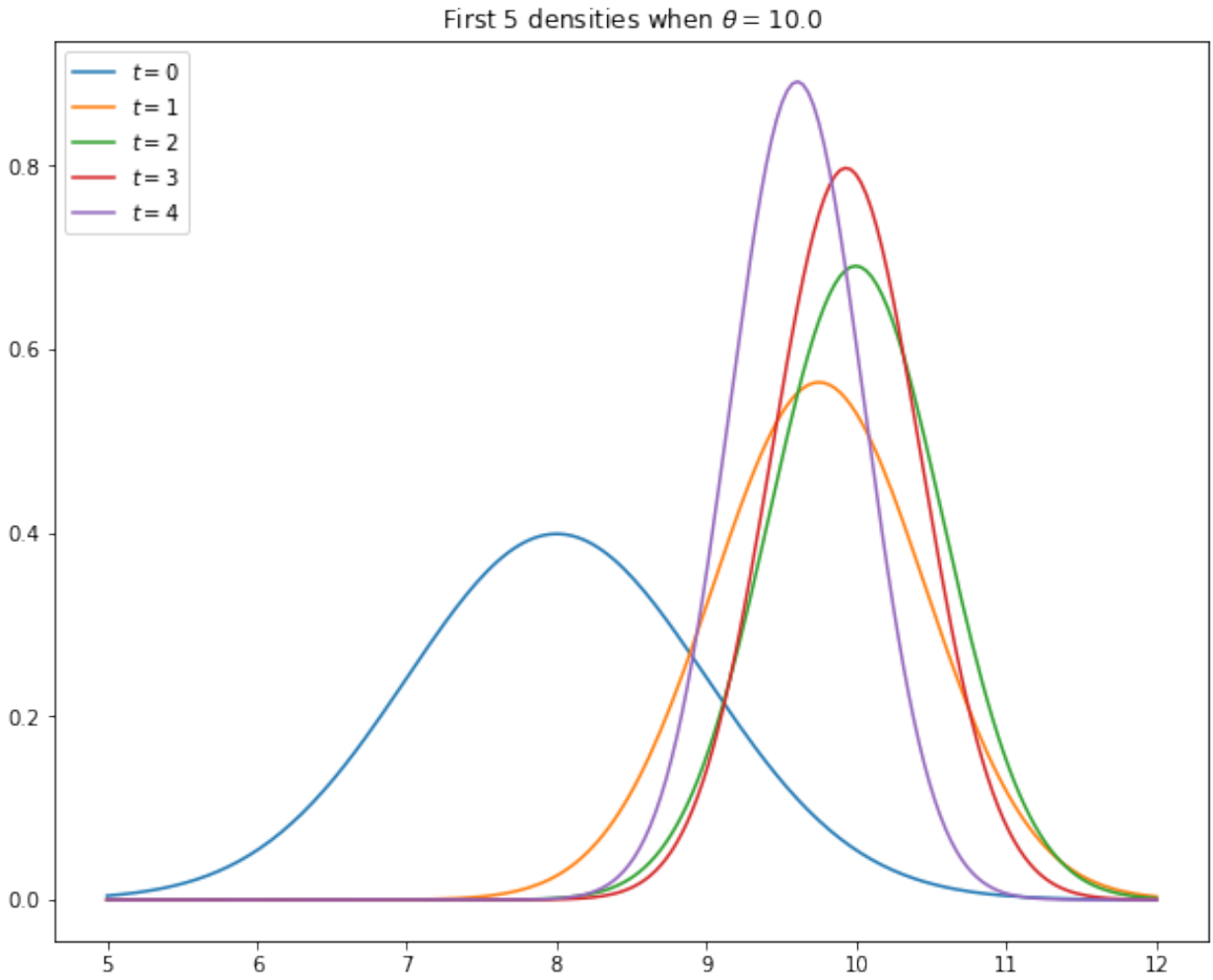
```
# Parameters
θ = 10  # Constant value of state x_t
A, C, G, H = 1, 0, 1, 1
ss = LinearStateSpace(A, C, G, H, mu_0=θ)
```

(continues on next page)



(continued from previous page)

```
# Set prior, initialize kalman filter
x_hat_0, Σ_0 = 8, 1
kalman = Kalman(ss, x_hat_0, Σ_0)

# Draw observations of y from state space model
N = 5
x, y = ss.simulate(N)
y = y.flatten()

# Set up plot
fig, ax = plt.subplots(figsize=(10,8))
xgrid = np.linspace(θ - 5, θ + 2, 200)

for i in range(N):
    # Record the current predicted mean and variance
    m, v = [float(z) for z in (kalman.x_hat, kalman.Sigma)]
    # Plot, update filter
    ax.plot(xgrid, norm.pdf(xgrid, loc=m, scale=np.sqrt(v)), label=f'$t={i}$')
    kalman.update(y[i])

ax.set_title(f'First {N} densities when $\\theta = {θ:.1f}$')
ax.legend(loc='upper left')
plt.show()
```



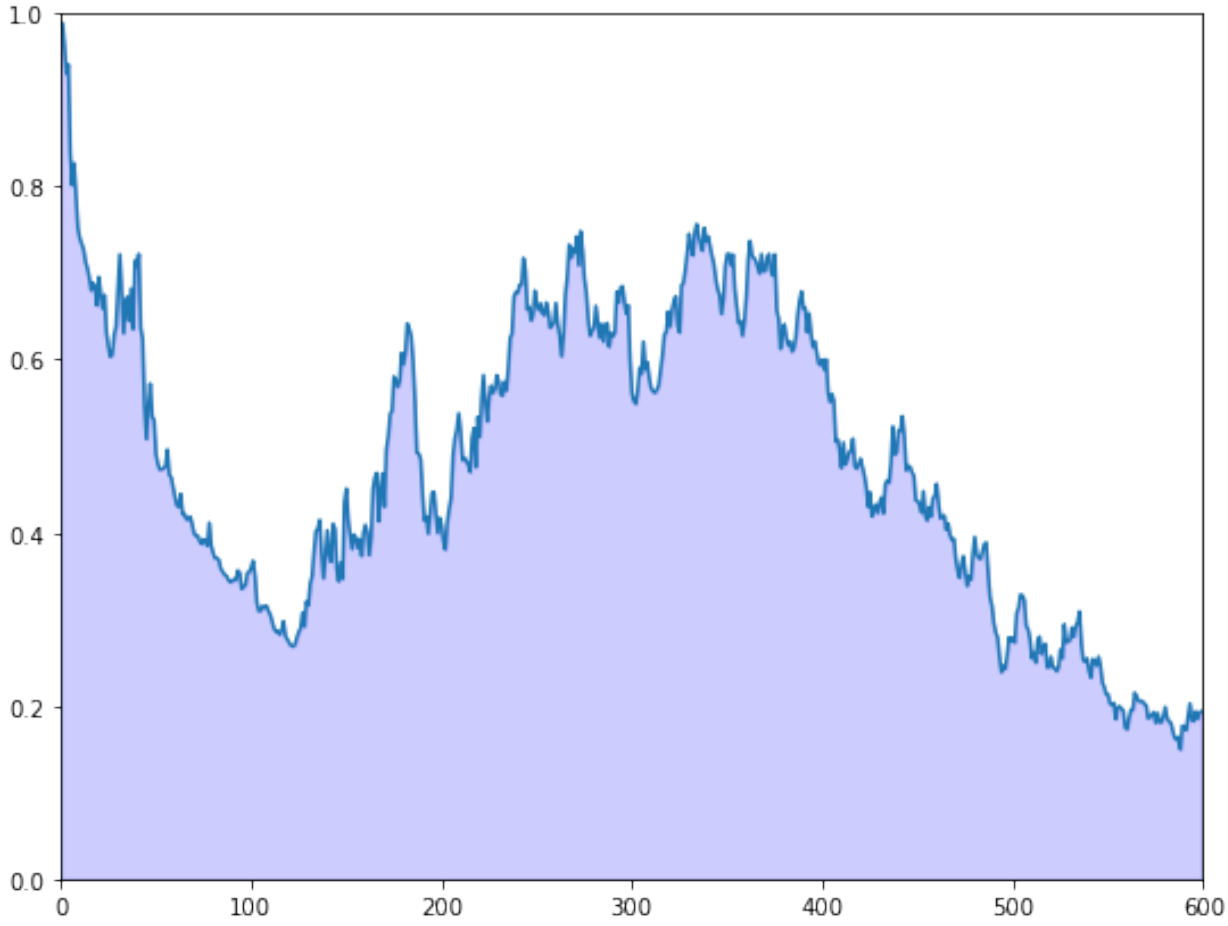
23.6.2 Exercise 2

```
ϵ = 0.1
θ = 10  # Constant value of state x_t
A, C, G, H = 1, 0, 1, 1
ss = LinearStateSpace(A, C, G, H, mu_0=θ)

x_hat_0, Σ_0 = 8, 1
kalman = Kalman(ss, x_hat_0, Σ_0)

T = 600
z = np.empty(T)
x, y = ss.simulate(T)
y = y.flatten()

for t in range(T):
    # Record the current predicted mean and variance and plot their densities
    m, v = [float(temp) for temp in (kalman.x_hat, kalman.Sigma)]

    f = lambda x: norm.pdf(x, loc=m, scale=np.sqrt(v))
    integral, error = quad(f, θ - ϵ, θ + ϵ)
```

(continues on next page)

(continued from previous page)

```
    z[t] = 1 - integral

    kalman.update(y[t])

fig, ax = plt.subplots(figsize=(9, 7))
ax.set_ylim(0, 1)
ax.set_xlim(0, T)
ax.plot(range(T), z)
ax.fill_between(range(T), np.zeros(T), z, color="blue", alpha=0.2)
plt.show()
```



23.6.3 Exercise 3

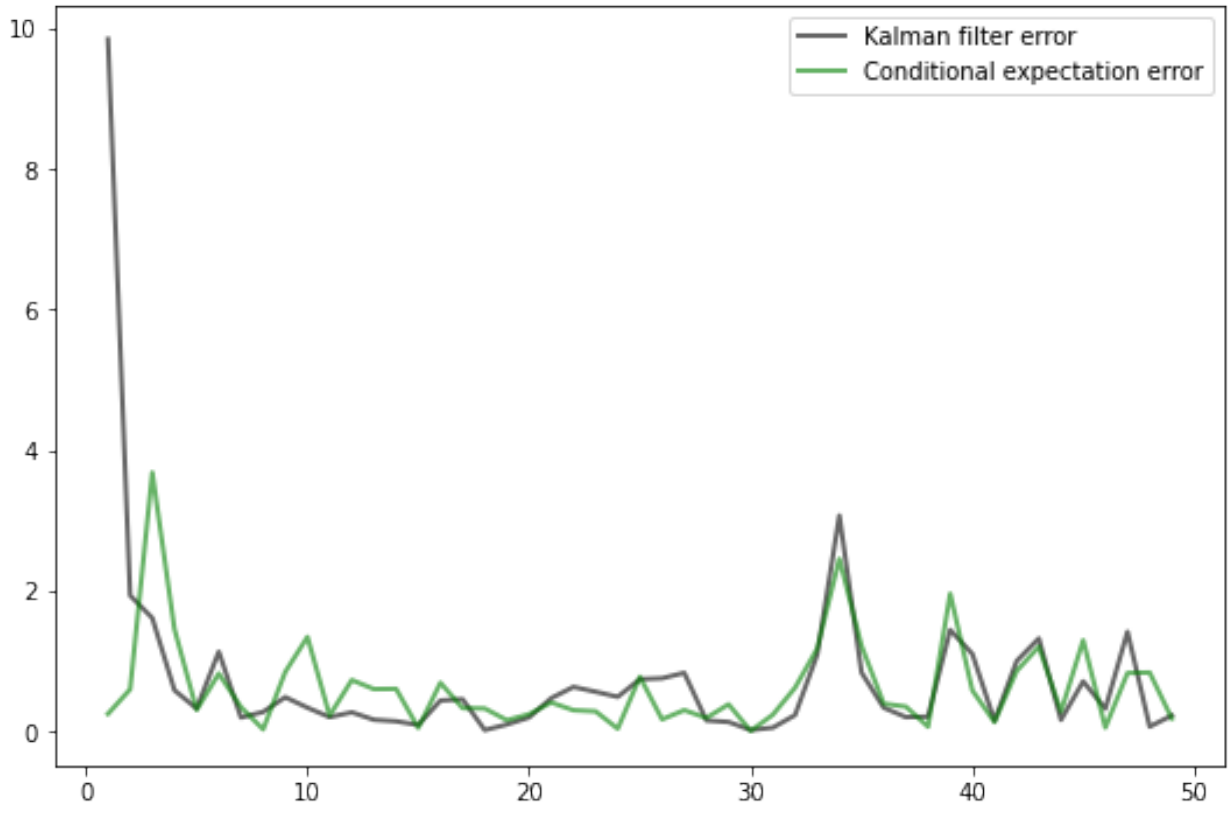
```
# Define A, C, G, H
G = np.identity(2)
H = np.sqrt(0.5) * np.identity(2)

A = [[0.5, 0.4],
     [0.6, 0.3]]
C = np.sqrt(0.3) * np.identity(2)

# Set up state space mode, initial value x_0 set to zero
```

(continues on next page)

(continued from previous page)

```
ss = LinearStateSpace(A, C, G, H, mu_0 = np.zeros(2))

# Define the prior density
Σ = [[0.9, 0.3],
     [0.3, 0.9]]
Σ = np.array(Σ)
x_hat = np.array([8, 8])

# Initialize the Kalman filter
kn = Kalman(ss, x_hat, Σ)

# Print eigenvalues of A
print("Eigenvalues of A:")
print(eigvals(A))

# Print stationary Σ
S, K = kn.stationary_values()
print("Stationary prediction error variance:")
print(S)

# Generate the plot
T = 50
x, y = ss.simulate(T)

e1 = np.empty(T-1)
e2 = np.empty(T-1)

for t in range(1, T):
    kn.update(y[:,t])
    e1[t-1] = np.sum((x[:, t] - kn.x_hat.flatten())**2)
    e2[t-1] = np.sum((x[:, t] - A @ x[:, t-1])**2)

fig, ax = plt.subplots(figsize=(9,6))
ax.plot(range(1, T), e1, 'k-', lw=2, alpha=0.6,
        label='Kalman filter error')
ax.plot(range(1, T), e2, 'g-', lw=2, alpha=0.6,
        label='Conditional expectation error')
ax.legend()
plt.show()
```

```
Eigenvalues of A:
[ 0.9+0.j -0.1+0.j]
Stationary prediction error variance:
[[0.40329108 0.1050718 ]
 [0.1050718  0.41061709]]
```


Kalman filter error
Conditional expectation error
10
8
6
4
2
0
0
10
20
30
40
50

CHAPTER

TWENTYFOUR

SHORTEST PATHS

Contents

24.1 Overview

The shortest path problem is a classic problem in mathematics and computer science with applications in

- Economics (sequential decision making, analysis of social networks, etc.)
- Operations research and transportation
- Robotics and artificial intelligence
- Telecommunication network design and routing
- etc., etc.

Variations of the methods we discuss in this lecture are used millions of times every day, in applications such as

- Google Maps
- routing packets on the internet

For us, the shortest path problem also provides a nice introduction to the logic of **dynamic programming**.

Dynamic programming is an extremely powerful optimization technique that we apply in many lectures on this site.

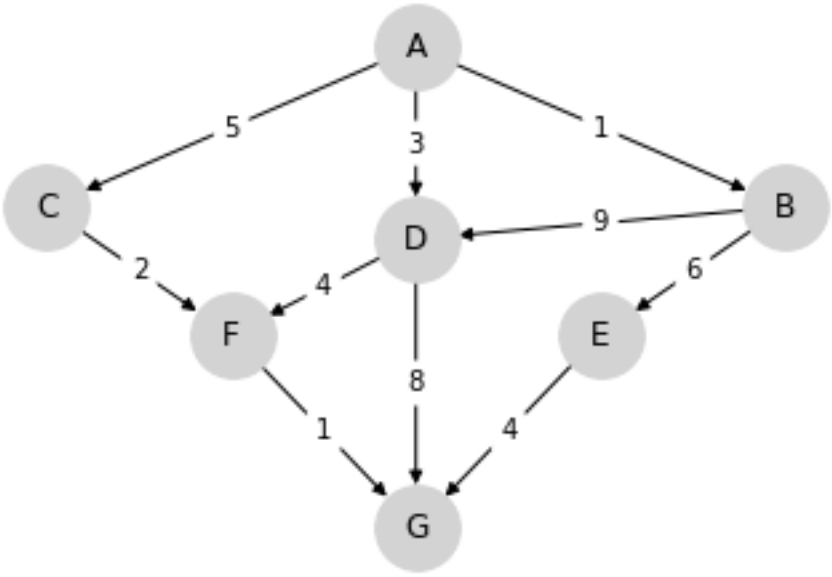
The only scientific library we'll need in what follows is NumPy:

```
import numpy as np
```

24.2 Outline of the Problem

The shortest path problem is one of finding how to traverse a graph from one specified node to another at minimum cost.

Consider the following graph



We wish to travel from node (vertex) A to node G at minimum cost

- Arrows (edges) indicate the movements we can take.
- Numbers on edges indicate the cost of traveling that edge.

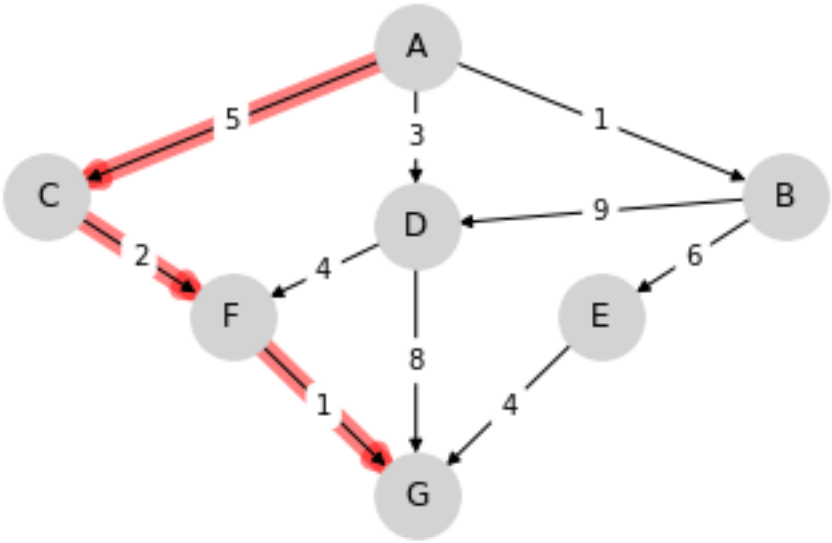
(Graphs such as the one above are called weighted directed graphs.)

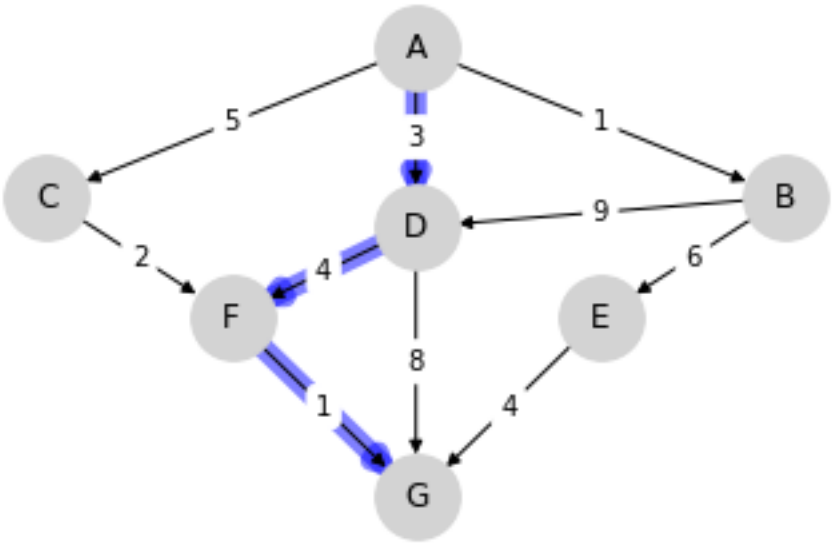
Possible interpretations of the graph include

- Minimum cost for supplier to reach a destination.
- Routing of packets on the internet (minimize time).
- Etc., etc.

For this simple graph, a quick scan of the edges shows that the optimal paths are

- A, C, F, G at cost 8
- A, D, F, G at cost 8


A
C
B
D
F
E
G
5
1
3
9
2
4
6
8
1
4

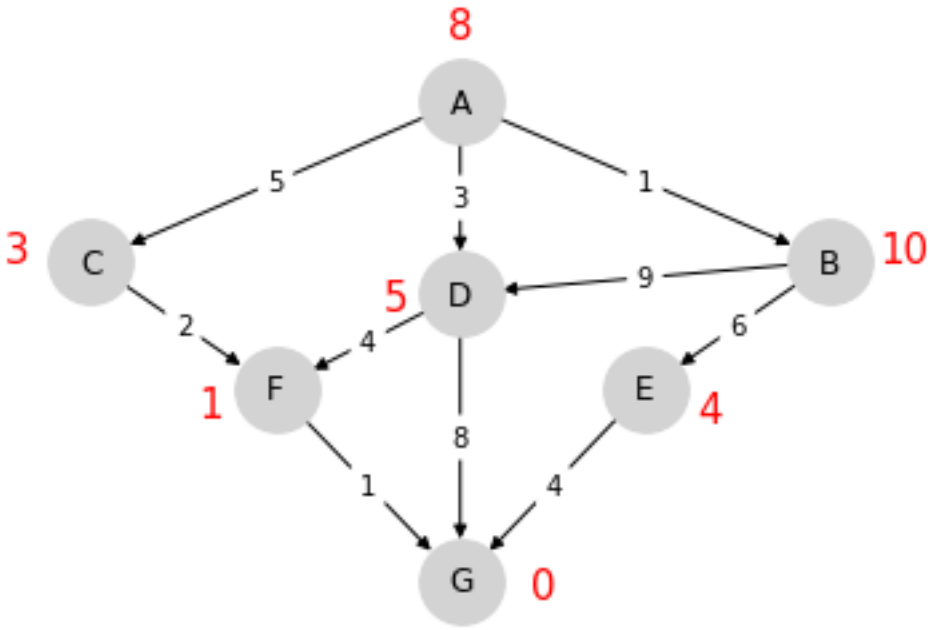

A
C
B
D
F
E
G
5
1
3
9
2
4
6
8
1
4

24.3 Finding Least-Cost Paths

For large graphs, we need a systematic solution.

Let $J(v)$ denote the minimum cost-to-go from node v, understood as the total cost from v if we take the best route.

Suppose that we know $J(v)$ for each node v, as shown below for the graph from the preceding example



Note that $J(G) = 0$.

The best path can now be found as follows

1. Start at node $v = A$
2. From current node v, move to any node that solves

$$\min_{w \in F_v} \{c(v, w) + J(w)\} \tag{1}$$

where

- F_v is the set of nodes that can be reached from v in one step.
- $c(v, w)$ is the cost of traveling from v to w.

Hence, if we know the function J, then finding the best path is almost trivial.

But how can we find the cost-to-go function J?

Some thought will convince you that, for every node v, the function J satisfies

$$J(v) = \min_{w \in F_v} \{c(v, w) + J(w)\} \tag{2}$$

This is known as the *Bellman equation*, after the mathematician Richard Bellman.

The Bellman equation can be thought of as a restriction that J must satisfy.

What we want to do now is use this restriction to compute J.

24.4 Solving for Minimum Cost-to-Go

Let's look at an algorithm for computing J and then think about how to implement it.

24.4.1 The Algorithm

The standard algorithm for finding J is to start an initial guess and then iterate.

This is a standard approach to solving nonlinear equations, often called the method of **successive approximations**.

Our initial guess will be

$$J_0(v) = 0 \text{ for all } v \tag{3}$$

Now

1. Set $n = 0$
2. Set $J_{n+1}(v) = \min_{w \in F_v}\{c(v, w) + J_n(w)\}$ for all v
3. If J_{n+1} and J_n are not equal then increment n, go to 2

This sequence converges to J.

Although we omit the proof, we'll prove similar claims in our other lectures on dynamic programming.

24.4.2 Implementation

Having an algorithm is a good start, but we also need to think about how to implement it on a computer.

First, for the cost function c, we'll implement it as a matrix Q, where a typical element is

$$Q(v, w) = \begin{cases} c(v, w) \text{ if } w \in F_v \\ +\infty \text{ otherwise} \end{cases}$$

In this context Q is usually called the **distance matrix**.

We're also numbering the nodes now, with $A = 0$, so, for example

$$Q(1, 2) = \text{ the cost of traveling from B to C}$$

For example, for the simple graph above, we set

```
from numpy import inf

Q = np.array([[inf, 1,   5,   3,   inf, inf, inf],
              [inf, inf, inf, 9,   6,   inf, inf],
              [inf, inf, inf, inf, inf, 2,   inf],
              [inf, inf, inf, inf, inf, 4,   8],
              [inf, inf, inf, inf, inf, inf, 4],
              [inf, inf, inf, inf, inf, inf, 1],
              [inf, inf, inf, inf, inf, inf, 0]])
```

Notice that the cost of staying still (on the principle diagonal) is set to

- np.inf for non-destination nodes — moving on is required.
- 0 for the destination node — here is where we stop.

For the sequence of approximations $\{J_n\}$ of the cost-to-go functions, we can use NumPy arrays.

Let's try with this example and see how we go:

```
nodes = range(7)                              # Nodes = 0, 1, ..., 6
J = np.zeros_like(nodes, dtype=int)         # Initial guess
next_J = np.empty_like(nodes, dtype=int)    # Stores updated guess

max_iter = 500
i = 0

while i < max_iter:
    for v in nodes:
        # minimize Q[v, w] + J[w] over all choices of w
        lowest_cost = inf
        for w in nodes:
            cost = Q[v, w] + J[w]
            if cost < lowest_cost:
                lowest_cost = cost
        next_J[v] = lowest_cost
    if np.equal(next_J, J).all():
        break
    else:
        J[:] = next_J    # Copy contents of next_J to J
        i += 1

print("The cost-to-go function is", J)
```

```
The cost-to-go function is [ 8 10  3  5  4  1  0]
```

This matches with the numbers we obtained by inspection above.

But, importantly, we now have a methodology for tackling large graphs.

24.5 Exercises

24.5.1 Exercise 1

The text below describes a weighted directed graph.

The line `node0, node1 0.04, node8 11.11, node14 72.21` means that from node0 we can go to

- node1 at cost 0.04
- node8 at cost 11.11
- node14 at cost 72.21

No other nodes can be reached directly from node0.

Other lines have a similar interpretation.

Your task is to use the algorithm given above to find the optimal path and its cost.

Note: You will be dealing with floating point numbers now, rather than integers, so consider replacing `np.equal()` with `np.allclose()`.

```
%%file graph.txt
node0, node1 0.04, node8 11.11, node14 72.21
node1, node46 1247.25, node6 20.59, node13 64.94
node2, node66 54.18, node31 166.80, node45 1561.45
node3, node20 133.65, node6 2.06, node11 42.43
node4, node75 3706.67, node5 0.73, node7 1.02
node5, node45 1382.97, node7 3.33, node11 34.54
node6, node31 63.17, node9 0.72, node10 13.10
node7, node50 478.14, node9 3.15, node10 5.85
node8, node69 577.91, node11 7.45, node12 3.18
node9, node70 2454.28, node13 4.42, node20 16.53
node10, node89 5352.79, node12 1.87, node16 25.16
node11, node94 4961.32, node18 37.55, node20 65.08
node12, node84 3914.62, node24 34.32, node28 170.04
node13, node60 2135.95, node38 236.33, node40 475.33
node14, node67 1878.96, node16 2.70, node24 38.65
node15, node91 3597.11, node17 1.01, node18 2.57
node16, node36 392.92, node19 3.49, node38 278.71
node17, node76 783.29, node22 24.78, node23 26.45
node18, node91 3363.17, node23 16.23, node28 55.84
node19, node26 20.09, node20 0.24, node28 70.54
node20, node98 3523.33, node24 9.81, node33 145.80
node21, node56 626.04, node28 36.65, node31 27.06
node22, node72 1447.22, node39 136.32, node40 124.22
node23, node52 336.73, node26 2.66, node33 22.37
node24, node66 875.19, node26 1.80, node28 14.25
node25, node70 1343.63, node32 36.58, node35 45.55
node26, node47 135.78, node27 0.01, node42 122.00
node27, node65 480.55, node35 48.10, node43 246.24
node28, node82 2538.18, node34 21.79, node36 15.52
node29, node64 635.52, node32 4.22, node33 12.61
node30, node98 2616.03, node33 5.61, node35 13.95
node31, node98 3350.98, node36 20.44, node44 125.88
node32, node97 2613.92, node34 3.33, node35 1.46
node33, node81 1854.73, node41 3.23, node47 111.54
node34, node73 1075.38, node42 51.52, node48 129.45
node35, node52 17.57, node41 2.09, node50 78.81
node36, node71 1171.60, node54 101.08, node57 260.46
node37, node75 269.97, node38 0.36, node46 80.49
node38, node93 2767.85, node40 1.79, node42 8.78
node39, node50 39.88, node40 0.95, node41 1.34
node40, node75 548.68, node47 28.57, node54 53.46
node41, node53 18.23, node46 0.28, node54 162.24
node42, node59 141.86, node47 10.08, node72 437.49
node43, node98 2984.83, node54 95.06, node60 116.23
node44, node91 807.39, node46 1.56, node47 2.14
node45, node58 79.93, node47 3.68, node49 15.51
node46, node52 22.68, node57 27.50, node67 65.48
node47, node50 2.82, node56 49.31, node61 172.64
node48, node99 2564.12, node59 34.52, node60 66.44
node49, node78 53.79, node50 0.51, node56 10.89
node50, node85 251.76, node53 1.38, node55 20.10
node51, node98 2110.67, node59 23.67, node60 73.79
node52, node94 1471.80, node64 102.41, node66 123.03
node53, node72 22.85, node56 4.33, node67 88.35
node54, node88 967.59, node59 24.30, node73 238.61
node55, node84 86.09, node57 2.13, node64 60.80
```

(continues on next page)

(continued from previous page)

```
node56, node76 197.03, node57 0.02, node61 11.06
node57, node86 701.09, node58 0.46, node60 7.01
node58, node83 556.70, node64 29.85, node65 34.32
node59, node90 820.66, node60 0.72, node71 0.67
node60, node76 48.03, node65 4.76, node67 1.63
node61, node98 1057.59, node63 0.95, node64 4.88
node62, node91 132.23, node64 2.94, node76 38.43
node63, node66 4.43, node72 70.08, node75 56.34
node64, node80 47.73, node65 0.30, node76 11.98
node65, node94 594.93, node66 0.64, node73 33.23
node66, node98 395.63, node68 2.66, node73 37.53
node67, node82 153.53, node68 0.09, node70 0.98
node68, node94 232.10, node70 3.35, node71 1.66
node69, node99 247.80, node70 0.06, node73 8.99
node70, node76 27.18, node72 1.50, node73 8.37
node71, node89 104.50, node74 8.86, node91 284.64
node72, node76 15.32, node84 102.77, node92 133.06
node73, node83 52.22, node76 1.40, node90 243.00
node74, node81 1.07, node76 0.52, node78 8.08
node75, node92 68.53, node76 0.81, node77 1.19
node76, node85 13.18, node77 0.45, node78 2.36
node77, node80 8.94, node78 0.98, node86 64.32
node78, node98 355.90, node81 2.59
node79, node81 0.09, node85 1.45, node91 22.35
node80, node92 121.87, node88 28.78, node98 264.34
node81, node94 99.78, node89 39.52, node92 99.89
node82, node91 47.44, node88 28.05, node93 11.99
node83, node94 114.95, node86 8.75, node88 5.78
node84, node89 19.14, node94 30.41, node98 121.05
node85, node97 94.51, node87 2.66, node89 4.90
node86, node97 85.09
node87, node88 0.21, node91 11.14, node92 21.23
node88, node93 1.31, node91 6.83, node98 6.12
node89, node97 36.97, node99 82.12
node90, node96 23.53, node94 10.47, node99 50.99
node91, node97 22.17
node92, node96 10.83, node97 11.24, node99 34.68
node93, node94 0.19, node97 6.71, node99 32.77
node94, node98 5.91, node96 2.03
node95, node98 6.17, node99 0.27
node96, node98 3.32, node97 0.43, node99 5.87
node97, node98 0.30
node98, node99 0.33
node99,
```

```
Overwriting graph.txt
```

24.6 Solutions

24.6.1 Exercise 1

First let's write a function that reads in the graph data above and builds a distance matrix.

```
num_nodes = 100
destination_node = 99

def map_graph_to_distance_matrix(in_file):

    # First let's set of the distance matrix Q with inf everywhere
    Q = np.full((num_nodes, num_nodes), np.inf)

    # Now we read in the data and modify Q
    infile = open(in_file)
    for line in infile:
        elements = line.split(',')
        node = elements.pop(0)
        node = int(node[4:])     # convert node description to integer
        if node != destination_node:
            for element in elements:
                destination, cost = element.split()
                destination = int(destination[4:])
                Q[node, destination] = float(cost)
        Q[destination_node, destination_node] = 0

    infile.close()
    return Q
```

In addition, let's write

1. a "Bellman operator" function that takes a distance matrix and current guess of J and returns an updated guess of J, and
2. a function that takes a distance matrix and returns a cost-to-go function.

We'll use the algorithm described above.

The minimization step is vectorized to make it faster.

```
def bellman(J, Q):
    num_nodes = Q.shape[0]
    next_J = np.empty_like(J)
    for v in range(num_nodes):
        next_J[v] = np.min(Q[v, :] + J)
    return next_J


def compute_cost_to_go(Q):
    num_nodes = Q.shape[0]
    J = np.zeros(num_nodes)       # Initial guess
    max_iter = 500
    i = 0

    while i < max_iter:
        next_J = bellman(J, Q)
        if np.allclose(next_J, J):
```

(continues on next page)

(continued from previous page)

```
            break
        else:
            J[:] = next_J    # Copy contents of next_J to J
            i += 1

    return(J)
```

We used np.allclose() rather than testing exact equality because we are dealing with floating point numbers now.

Finally, here's a function that uses the cost-to-go function to obtain the optimal path (and its cost).

```
def print_best_path(J, Q):
    sum_costs = 0
    current_node = 0
    while current_node != destination_node:
        print(current_node)
        # Move to the next node and increment costs
        next_node = np.argmin(Q[current_node, :] + J)
        sum_costs += Q[current_node, next_node]
        current_node = next_node

    print(destination_node)
    print('Cost: ', sum_costs)
```

Okay, now we have the necessary functions, let's call them to do the job we were assigned.

```
Q = map_graph_to_distance_matrix('graph.txt')
J = compute_cost_to_go(Q)
print_best_path(J, Q)
```

```
0
8
11
18
23
33
41
53
56
57
60
67
70
73
76
85
87
88
93
94
96
97
98
99
Cost:  160.55000000000007
```

The total cost of the path should agree with $J[0]$ so let's check this.

```
J[0]
```

```
160.55
```

CHAPTER

TWENTYFIVE

CASS-KOOPMANS PLANNING PROBLEM

Contents

25.1 Overview

This lecture and lecture *Cass-Koopmans Competitive Equilibrium* describe a model that Tjalling Koopmans [Koo65] and David Cass [Cas65] used to analyze optimal growth.

The model can be viewed as an extension of the model of Robert Solow described in an earlier lecture but adapted to make the saving rate the outcome of an optimal choice.

(Solow assumed a constant saving rate determined outside the model.)

We describe two versions of the model, one in this lecture and the other in *Cass-Koopmans Competitive Equilibrium*.

Together, the two lectures illustrate what is, in fact, a more general connection between a **planned economy** and a decentralized economy organized as a **competitive equilibrium**.

This lecture is devoted to the planned economy version.

The lecture uses important ideas including

- A min-max problem for solving a planning problem.
- A **shooting algorithm** for solving difference equations subject to initial and terminal conditions.
- A **turnpike** property that describes optimal paths for long but finite-horizon economies.

Let's start with some standard imports:

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
from numba import njit, float64
from numba.experimental import jitclass
import numpy as np
```

25.2 The Model

Time is discrete and takes values $t = 0, 1, \ldots, T$ where T is finite.

(We'll study a limiting case in which $T = +\infty$ before concluding).

A single good can either be consumed or invested in physical capital.

The consumption good is not durable and depreciates completely if not consumed immediately.

The capital good is durable but depreciates some each period.

We let C_t be a nondurable consumption good at time t.

Let K_t be the stock of physical capital at time t.

Let $\vec{C} = \{C_0, \ldots, C_T\}$ and $\vec{K} = \{K_0, \ldots, K_{T+1}\}$.

A representative household is endowed with one unit of labor at each t and likes the consumption good at each t.

The representative household inelastically supplies a single unit of labor N_t at each t, so that $N_t = 1$ for all $t \in [0, T]$.

The representative household has preferences over consumption bundles ordered by the utility functional:

$$U(\vec{C}) = \sum_{t=0}^{T} \beta^t \frac{C_t^{1-\gamma}}{1-\gamma} \tag{1}$$

where $\beta \in (0, 1)$ is a discount factor and $\gamma > 0$ governs the curvature of the one-period utility function with larger γ implying more curvature.

Note that

$$u(C_t) = \frac{C_t^{1-\gamma}}{1-\gamma} \tag{2}$$

satisfies $u' > 0, u'' < 0$.

$u' > 0$ asserts that the consumer prefers more to less.

$u'' < 0$ asserts that marginal utility declines with increases in C_t.

We assume that $K_0 > 0$ is an exogenous initial capital stock.

There is an economy-wide production function

$$F(K_t, N_t) = AK_t^{\alpha} N_t^{1-\alpha} \tag{3}$$

with $0 < \alpha < 1$, $A > 0$.

A feasible allocation $\vec{C}, \vec{K}$ satisfies

$$C_t + K_{t+1} \leq F(K_t, N_t) + (1-\delta)K_t, \quad \text{for all } t \in [0, T] \tag{4}$$

where $\delta \in (0, 1)$ is a depreciation rate of capital.

25.3 Planning Problem

A planner chooses an allocation $\{\vec{C}, \vec{K}\}$ to maximize (1) subject to (4).

Let $\vec{\mu} = \{\mu_0, \dots, \mu_T\}$ be a sequence of nonnegative **Lagrange multipliers**.

To find an optimal allocation, form a Lagrangian

$$\mathcal{L}(\vec{C}, \vec{K}, \vec{\mu}) = \sum_{t=0}^{T} \beta^t \left\{ u(C_t) + \mu_t \left(F(K_t, 1) + (1-\delta)K_t - C_t - K_{t+1} \right) \right\}$$

and then pose the following min-max problem:

$$\min_{\vec{\mu}} \max_{\vec{C}, \vec{K}} \mathcal{L}(\vec{C}, \vec{K}, \vec{\mu}) \tag{5}$$

- **Extremization** means maximization with respect to $\vec{C}, \vec{K}$ and minimization with respect to $\vec{\mu}$.
- Our problem satisfies conditions that assure that required second-order conditions are satisfied at an allocation that satisfies the first-order conditions that we are about to compute.

Before computing first-order conditions, we present some handy formulas.

25.3.1 Useful Properties of Linearly Homogeneous Production Function

The following technicalities will help us.

Notice that

$$F(K_t, N_t) = AK_t^{\alpha} N_t^{1-\alpha} = N_t A \left(\frac{K_t}{N_t} \right)^{\alpha}$$

Define the **output per-capita production function**

$$\frac{F(K_t, N_t)}{N_t} \equiv f\left(\frac{K_t}{N_t} \right) = A \left(\frac{K_t}{N_t} \right)^{\alpha}$$

whose argument is **capital per-capita**.

It is useful to recall the following calculations for the marginal product of capital

$$\begin{aligned} \frac{\partial F(K_t, N_t)}{\partial K_t} &= \frac{\partial N_t f\left(\frac{K_t}{N_t}\right)}{\partial K_t} \\ &= N_t f'\left(\frac{K_t}{N_t} \right) \frac{1}{N_t} \quad \text{(Chain rule)} \\ &= f'\left(\frac{K_t}{N_t} \right) \Big|_{N_t = 1} \\ &= f'(K_t) \end{aligned} \tag{6}$$

and the marginal product of labor

$$\begin{aligned} \frac{\partial F(K_t, N_t)}{\partial N_t} &= \frac{\partial N_t f\left(\frac{K_t}{N_t}\right)}{\partial N_t} \quad \text{(Product rule)} \\ &= f\left(\frac{K_t}{N_t} \right) + N_t f'\left(\frac{K_t}{N_t} \right) \frac{-K_t}{N_t^2} \quad \text{(Chain rule)} \\ &= f\left(\frac{K_t}{N_t} \right) - \frac{K_t}{N_t} f'\left(\frac{K_t}{N_t} \right) \Big|_{N_t = 1} \\ &= f(K_t) - f'(K_t) K_t \end{aligned}$$

25.3.2 First-order necessary conditions

We now compute **first order necessary conditions** for extremization of the Lagrangian:

$$C_t: \quad u'(C_t) - \mu_t = 0 \quad \text{for all} \quad t = 0, 1, \ldots, T \tag{7}$$

$$K_t: \quad \beta\mu_t \left[(1-\delta) + f'(K_t)\right] - \mu_{t-1} = 0 \quad \text{for all} \quad t = 1, 2, \ldots, T \tag{8}$$

$$\mu_t: \quad F(K_t, 1) + (1-\delta)K_t - C_t - K_{t+1} = 0 \quad \text{for all} \quad t = 0, 1, \ldots, T \tag{9}$$

$$K_{T+1}: \quad -\mu_T \leq 0, \ \leq 0 \text{ if } K_{T+1} = 0; \ = 0 \text{ if } K_{T+1} > 0 \tag{10}$$

In computing (9) we recognize that K_t appears in both the time t and time $t-1$ feasibility constraints.

(10) comes from differentiating with respect to K_{T+1} and applying the following **Karush-Kuhn-Tucker condition** (KKT) (see Karush-Kuhn-Tucker conditions):

$$\mu_T K_{T+1} = 0 \tag{11}$$

Combining (7) and (8) gives

$$u'\left(C_t\right)\left[(1-\delta) + f'\left(K_t\right)\right] - u'\left(C_{t-1}\right) = 0 \quad \text{for all } t = 1, 2, \ldots, T+1$$

which can be rearranged to become

$$u'\left(C_{t+1}\right)\left[(1-\delta) + f'\left(K_{t+1}\right)\right] = u'\left(C_t\right) \quad \text{for all } t = 0, 1, \ldots, T \tag{12}$$

Applying the inverse of the utility function on both sides of the above equation gives

$$C_{t+1} = u'^{-1}\left(\left(\frac{\beta}{u'(C_t)}[f'(K_{t+1}) + (1-\delta)]\right)^{-1}\right)$$

which for our utility function (2) becomes the consumption **Euler equation**

$$\begin{aligned} C_{t+1} &= \left(\beta C_t^{\gamma}[f'(K_{t+1}) + (1-\delta)]\right)^{1/\gamma} \\ &= C_t\left(\beta[f'(K_{t+1}) + (1-\delta)]\right)^{1/\gamma} \end{aligned}$$

Below we define a `jitclass` that stores parameters and functions that define our economy.

```
planning_data = [
    ('γ', float64),     # Coefficient of relative risk aversion
    ('β', float64),     # Discount factor
    ('δ', float64),     # Depreciation rate on capital
    ('α', float64),     # Return to capital per capita
    ('A', float64)      # Technology
]
```

```
@jitclass(planning_data)
class PlanningProblem():

    def __init__(self, γ=2, β=0.95, δ=0.02, α=0.33, A=1):

        self.γ, self.β = γ, β
        self.δ, self.α, self.A = δ, α, A

    def u(self, c):
```

(continues on next page)

(continued from previous page)

```
        '''
        Utility function
        ASIDE: If you have a utility function that is hard to solve by hand
        you can use automatic or symbolic differentiation
        See https://github.com/HIPS/autograd
        '''
        γ = self.γ

        return c ** (1 - γ) / (1 - γ) if γ!= 1 else np.log(c)

    def u_prime(self, c):
        'Derivative of utility'
        γ = self.γ

        return c ** (-γ)

    def u_prime_inv(self, c):
        'Inverse of derivative of utility'
        γ = self.γ

        return c ** (-1 / γ)

    def f(self, k):
        'Production function'
        α, A = self.α, self.A

        return A * k ** α

    def f_prime(self, k):
        'Derivative of production function'
        α, A = self.α, self.A

        return α * A * k ** (α - 1)

    def f_prime_inv(self, k):
        'Inverse of derivative of production function'
        α, A = self.α, self.A

        return (k / (A * α)) ** (1 / (α - 1))

    def next_k_c(self, k, c):
        ''''
        Given the current capital Kt and an arbitrary feasible
        consumption choice Ct, computes Kt+1 by state transition law
        and optimal Ct+1 by Euler equation.
        '''
        β, δ = self.β, self.δ
        u_prime, u_prime_inv = self.u_prime, self.u_prime_inv
        f, f_prime = self.f, self.f_prime

        k_next = f(k) + (1 - δ) * k - c
        c_next = u_prime_inv(u_prime(c) / (β * (f_prime(k_next) + (1 - δ))))

        return k_next, c_next
```

We can construct an economy with the Python code:

```
pp = PlanningProblem()
```

25.4 Shooting Algorithm

We use **shooting** to compute an optimal allocation $\vec{C}, \vec{K}$ and an associated Lagrange multiplier sequence $\vec{\mu}$.

The first-order necessary conditions (7), (8), and (9) for the planning problem form a system of **difference equations** with two boundary conditions:

- K_0 is a given **initial condition** for capital
- $K_{T+1} = 0$ is a **terminal condition** for capital that we deduced from the first-order necessary condition for K_{T+1} the KKT condition (11)

We have no initial condition for the Lagrange multiplier μ_0.

If we did, our job would be easy:

- Given μ_0 and k_0, we could compute c_0 from equation (7) and then k_1 from equation (9) and μ_1 from equation (8).
- We could continue in this way to compute the remaining elements of $\vec{C}, \vec{K}, \vec{\mu}$.

But we don't have an initial condition for μ_0, so this won't work.

Indeed, part of our task is to compute the optimal value of μ_0.

To compute μ_0 and the other objects we want, a simple modification of the above procedure will work.

It is called the **shooting algorithm**.

It is an instance of a **guess and verify** algorithm that consists of the following steps:

- Guess an initial Lagrange multiplier μ_0.
- Apply the **simple algorithm** described above.
- Compute k_{T+1} and check whether it equals zero.
- If $K_{T+1} = 0$, we have solved the problem.
- If $K_{T+1} > 0$, lower μ_0 and try again.
- If $K_{T+1} < 0$, raise μ_0 and try again.

The following Python code implements the shooting algorithm for the planning problem.

We actually modify the algorithm slightly by starting with a guess for c_0 instead of μ_0 in the following code.

```
@njit
def shooting(pp, c0, k0, T=10):
    '''
    Given the initial condition of capital k0 and an initial guess
    of consumption c0, computes the whole paths of c and k
    using the state transition law and Euler equation for T periods.
    '''
    if c0 > pp.f(k0):
        print("initial consumption is not feasible")

        return None

    # initialize vectors of c and k
```

(continues on next page)

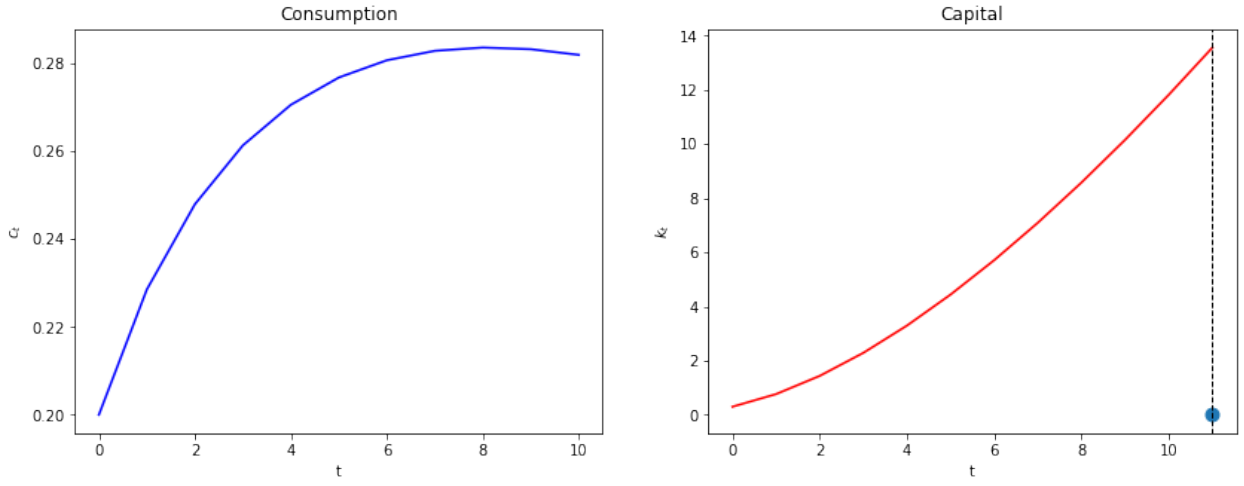
(continued from previous page)

```
    c_vec = np.empty(T+1)
    k_vec = np.empty(T+2)

    c_vec[0] = c0
    k_vec[0] = k0

    for t in range(T):
        k_vec[t+1], c_vec[t+1] = pp.next_k_c(k_vec[t], c_vec[t])

    k_vec[T+1] = pp.f(k_vec[T]) + (1 - pp.δ) * k_vec[T] - c_vec[T]

    return c_vec, k_vec
```

We'll start with an incorrect guess.

```
paths = shooting(pp, 0.2, 0.3, T=10)
```

```
fig, axs = plt.subplots(1, 2, figsize=(14, 5))

colors = ['blue', 'red']
titles = ['Consumption', 'Capital']
ylabels = ['$c_t$', '$k_t$']

T = paths[0].size - 1
for i in range(2):
    axs[i].plot(paths[i], c=colors[i])
    axs[i].set(xlabel='t', ylabel=ylabels[i], title=titles[i])

axs[1].scatter(T+1, 0, s=80)
axs[1].axvline(T+1, color='k', ls='--', lw=1)

plt.show()
```



Evidently, our initial guess for μ_0 is too high, so initial consumption too low.

We know this because we miss our $K_{T+1} = 0$ target on the high side.

Now we automate things with a search-for-a-good μ_0 algorithm that stops when we hit the target $K_{t+1} = 0$.

We use a **bisection method**.

We make an initial guess for C_0 (we can eliminate μ_0 because C_0 is an exact function of μ_0).

We know that the lowest C_0 can ever be is 0 and the largest it can be is initial output $f(K_0)$.

Guess C_0 and shoot forward to $T+1$.

If $K_{T+1} > 0$, we take it to be our new **lower** bound on C_0.

If $K_{T+1} < 0$, we take it to be our new **upper** bound.

Make a new guess for C_0 that is halfway between our new upper and lower bounds.

Shoot forward again, iterating on these steps until we converge.

When K_{T+1} gets close enough to 0 (i.e., within an error tolerance bounds), we stop.

```
@njit
def bisection(pp, c0, k0, T=10, tol=1e-4, max_iter=500, k_ter=0, verbose=True):

    # initial boundaries for guess c0
    c0_upper = pp.f(k0)
    c0_lower = 0

    i = 0
    while True:
        c_vec, k_vec = shooting(pp, c0, k0, T)
        error = k_vec[-1] - k_ter

        # check if the terminal condition is satisfied
        if np.abs(error) < tol:
            if verbose:
                print('Converged successfully on iteration ', i+1)
            return c_vec, k_vec

        i += 1
        if i == max_iter:
            if verbose:
                print('Convergence failed.')
            return c_vec, k_vec

        # if iteration continues, updates boundaries and guess of c0
        if error > 0:
            c0_lower = c0
        else:
            c0_upper = c0

        c0 = (c0_lower + c0_upper) / 2
```

```
def plot_paths(pp, c0, k0, T_arr, k_ter=0, k_ss=None, axs=None):

    if axs is None:
        fix, axs = plt.subplots(1, 3, figsize=(16, 4))
    ylabels = ['$c_t$', '$k_t$', '$\mu_t$']
    titles = ['Consumption', 'Capital', 'Lagrange Multiplier']

    c_paths = []
    k_paths = []
    for T in T_arr:
        c_vec, k_vec = bisection(pp, c0, k0, T, k_ter=k_ter, verbose=False)
        c_paths.append(c_vec)
```

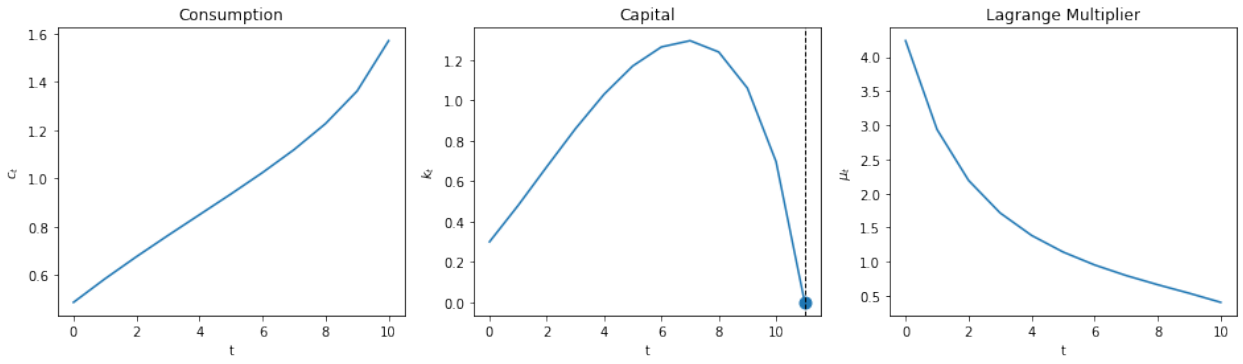
(continues on next page)

(continued from previous page)

```
        k_paths.append(k_vec)

        μ_vec = pp.u_prime(c_vec)
        paths = [c_vec, k_vec, μ_vec]

        for i in range(3):
            axs[i].plot(paths[i])
            axs[i].set(xlabel='t', ylabel=ylabels[i], title=titles[i])

        # Plot steady state value of capital
        if k_ss is not None:
            axs[1].axhline(k_ss, c='k', ls='--', lw=1)

        axs[1].axvline(T+1, c='k', ls='--', lw=1)
        axs[1].scatter(T+1, paths[1][-1], s=80)

    return c_paths, k_paths
```

Now we can solve the model and plot the paths of consumption, capital, and Lagrange multiplier.

```
plot_paths(pp, 0.3, 0.3, [10]);
```



25.5 Setting Initial Capital to Steady State Capital

When $T \to +\infty$, the optimal allocation converges to steady state values of C_t and K_t.

It is instructive to set K_0 equal to the $\lim_{T \to +\infty} K_t$, which we'll call steady state capital.

In a steady state $K_{t+1} = K_t = \bar{K}$ for all very large t.

Evalauating the feasibility constraint (4) at $\bar{K}$ gives

$$f(\bar{K}) - \delta \bar{K} = \bar{C} \tag{13}$$

Substituting $K_t = \bar{K}$ and $C_t = \bar{C}$ for all t into (12) gives

$$1 = \beta \frac{u'(\bar{C})}{u'(\bar{C})} [f'(\bar{K}) + (1 - \delta)]$$

Defining $\beta = \frac{1}{1+\rho}$, and cancelling gives

$$1 + \rho = 1[f'(\bar{K}) + (1 - \delta)]$$

Simplifying gives

$$f'(\bar{K}) = \rho + \delta$$

and

$$\bar{K} = f'^{-1}(\rho + \delta)$$

For the production function (3) this becomes

$$\alpha \bar{K}^{\alpha-1} = \rho + \delta$$

As an example, after setting $\alpha = .33$, $\rho = 1/\beta - 1 = 1/(19/20) - 1 = 20/19 - 19/19 = 1/19$, $\delta = 1/50$, we get

$$\bar{K} = \left(\frac{\frac{33}{100}}{\frac{1}{50} + \frac{1}{19}} \right)^{\frac{67}{100}} \approx 9.57583$$

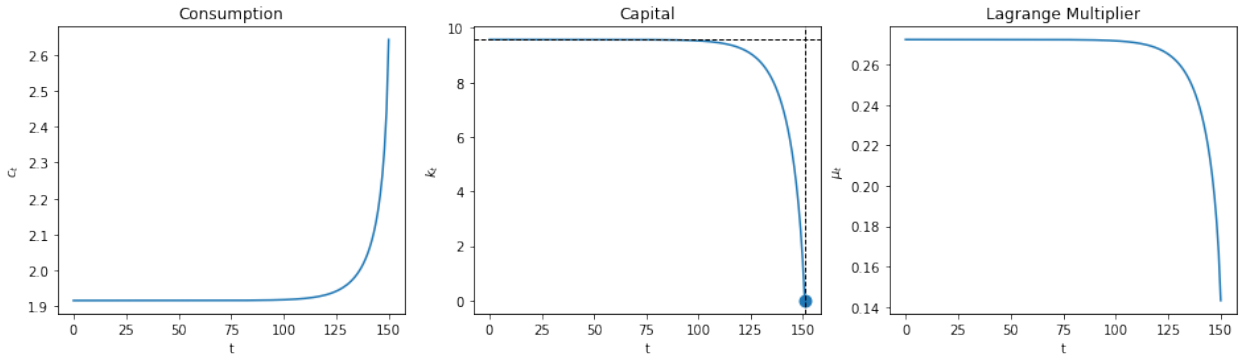
Let's verify this with Python and then use this steady state $\bar{K}$ as our initial capital stock K_0.

```
ρ = 1 / pp.β - 1
k_ss = pp.f_prime_inv(ρ+pp.δ)

print(f'steady state for capital is: {k_ss}')
```

```
steady state for capital is: 9.57583816331462
```

Now we plot

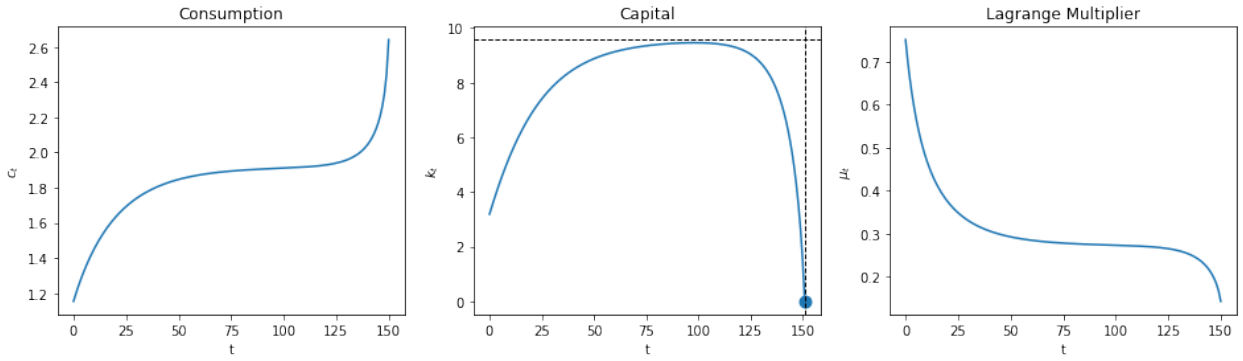
```
plot_paths(pp, 0.3, k_ss, [150], k_ss=k_ss);
```



Evidently, with a large value of T, K_t stays near K_0 until t approaches T closely.

Let's see what the planner does when we set K_0 below $\bar{K}$.

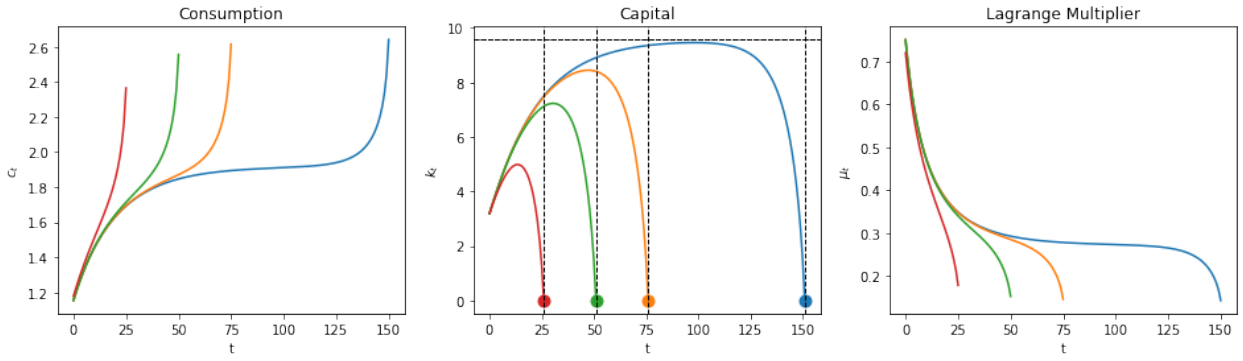
```
plot_paths(pp, 0.3, k_ss/3, [150], k_ss=k_ss);
```



Notice how the planner pushes capital toward the steady state, stays near there for a while, then pushes K_t toward the terminal value $K_{T+1} = 0$ when t closely approaches T.

The following graphs compare optimal outcomes as we vary T.

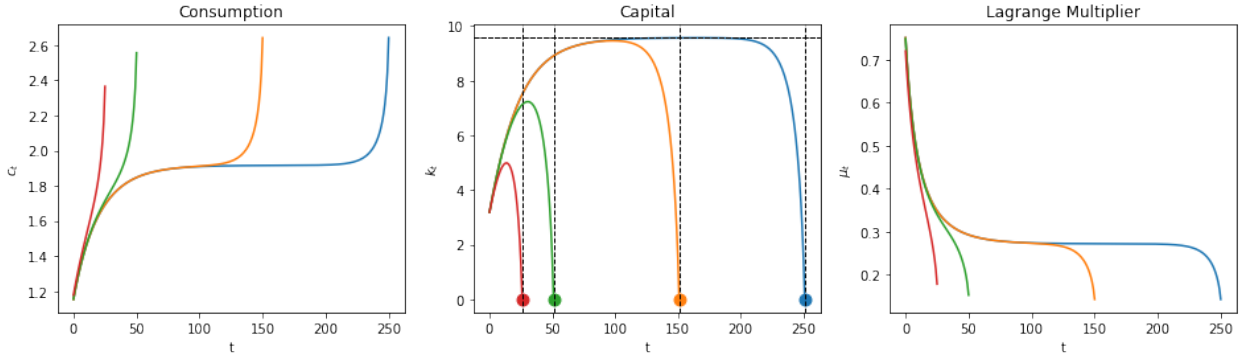
```
plot_paths(pp, 0.3, k_ss/3, [150, 75, 50, 25], k_ss=k_ss);
```



25.6 A Turnpike Property

The following calculation indicates that when T is very large, the optimal capital stock stays close to its steady state value most of the time.

```
plot_paths(pp, 0.3, k_ss/3, [250, 150, 50, 25], k_ss=k_ss);
```



Different colors in the above graphs are associated with different horizons T.

Notice that as the horizon increases, the planner puts K_t closer to the steady state value $\bar{K}$ for longer.

This pattern reflects a **turnpike** property of the steady state.

A rule of thumb for the planner is

- from K_0, push K_t toward the steady state and stay close to the steady state until time approaches T.

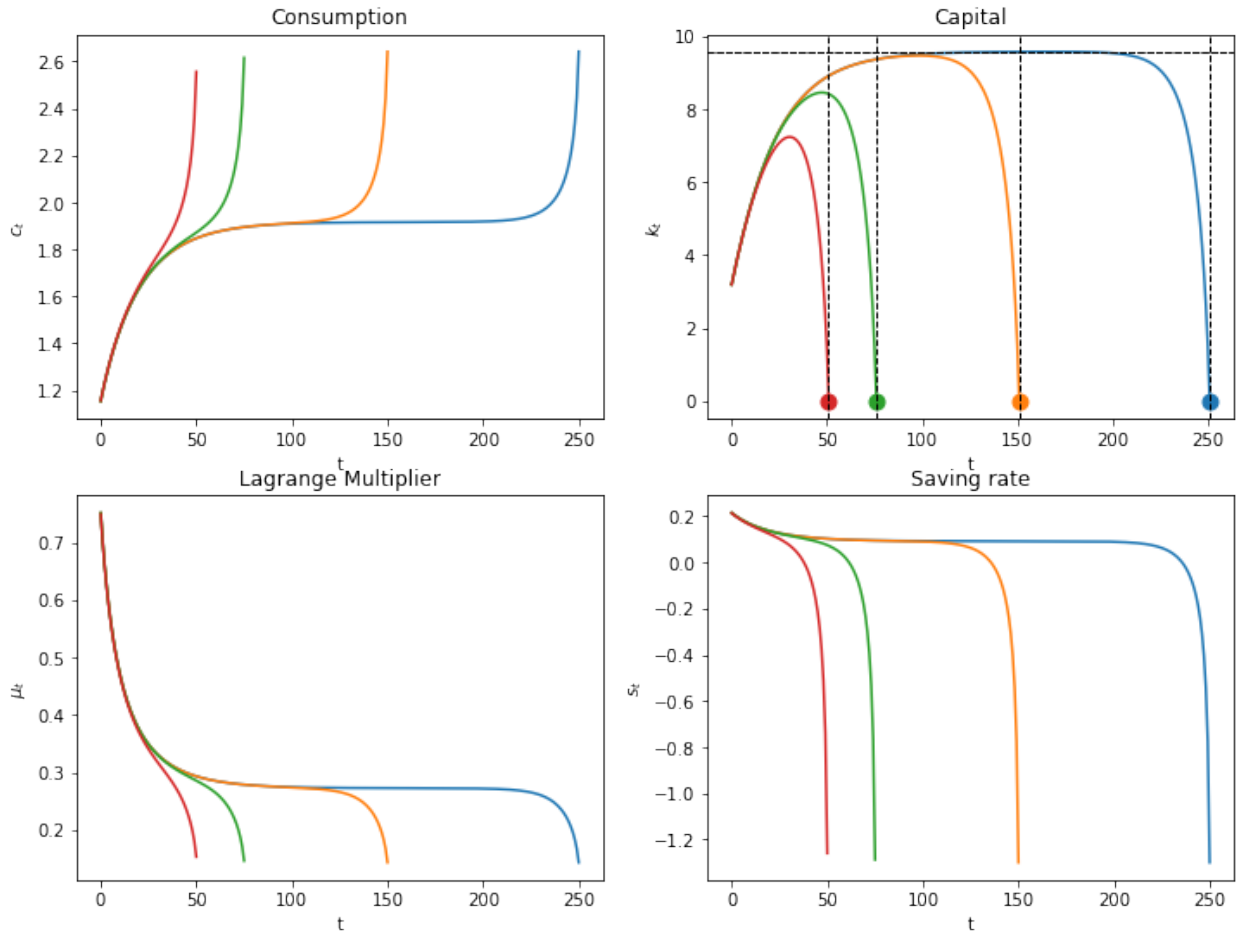
The planner accomplishes this by adjusting the saving rate $\frac{f(K_t)-C_t}{f(K_t)}$ over time.

Let's calculate and plot the saving rate.

```
@njit
def saving_rate(pp, c_path, k_path):
    'Given paths of c and k, computes the path of saving rate.'
    production = pp.f(k_path[:-1])

    return (production - c_path) / production
```

```
def plot_saving_rate(pp, c0, k0, T_arr, k_ter=0, k_ss=None, s_ss=None):

    fix, axs = plt.subplots(2, 2, figsize=(12, 9))

    c_paths, k_paths = plot_paths(pp, c0, k0, T_arr, k_ter=k_ter, k_ss=k_ss, axs=axs.
↪flatten())

    for i, T in enumerate(T_arr):
        s_path = saving_rate(pp, c_paths[i], k_paths[i])
        axs[1, 1].plot(s_path)

    axs[1, 1].set(xlabel='t', ylabel='$s_t$', title='Saving rate')

    if s_ss is not None:
        axs[1, 1].hlines(s_ss, 0, np.max(T_arr), linestyle='--')
```

```
plot_saving_rate(pp, 0.3, k_ss/3, [250, 150, 75, 50], k_ss=k_ss)
```



25.7 A Limiting Economy

We want to set $T = +\infty$.

The appropriate thing to do is to replace terminal condition (10) with

$$\lim_{T \to +\infty} \beta^T u'(C_T) K_{T+1} = 0,$$

a condition that will be satisfied by a path that converges to an optimal steady state.

We can approximate the optimal path by starting from an arbitrary initial K_0 and shooting towards the optimal steady state K at a large but finite $T + 1$.

In the following code, we do this for a large T and plot consumption, capital, and the saving rate.

We know that in the steady state that the saving rate is constant and that $\bar{s} = \frac{f(\bar{K}) - \bar{C}}{f(\bar{K})}$.

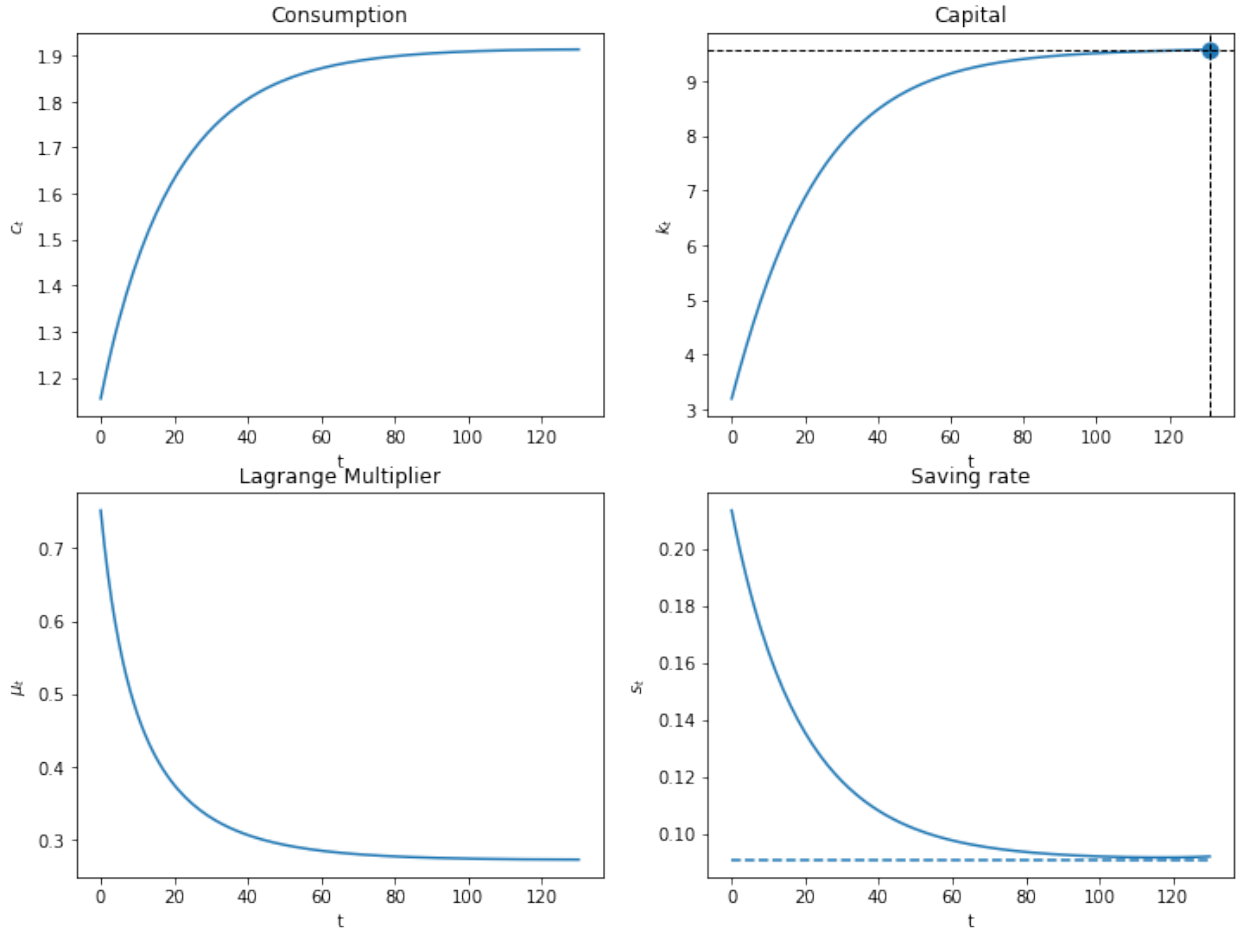
From (13) the steady state saving rate equals

$$\bar{s} = \frac{\delta \bar{K}}{f(\bar{K})}$$

The steady state saving rate $\bar{S} = \bar{s} f(\bar{K})$ is the amount required to offset capital depreciation each period.

We first study optimal capital paths that start below the steady state.

```
# steady state of saving rate
s_ss = pp.δ * k_ss / pp.f(k_ss)

plot_saving_rate(pp, 0.3, k_ss/3, [130], k_ter=k_ss, k_ss=k_ss, s_ss=s_ss)
```



Since $K_0 < \bar{K}$, $f'(K_0) > \rho + \delta$.

The planner chooses a positive saving rate that is higher than the steady state saving rate.

Note, $f''(K) < 0$, so as K rises, $f'(K)$ declines.

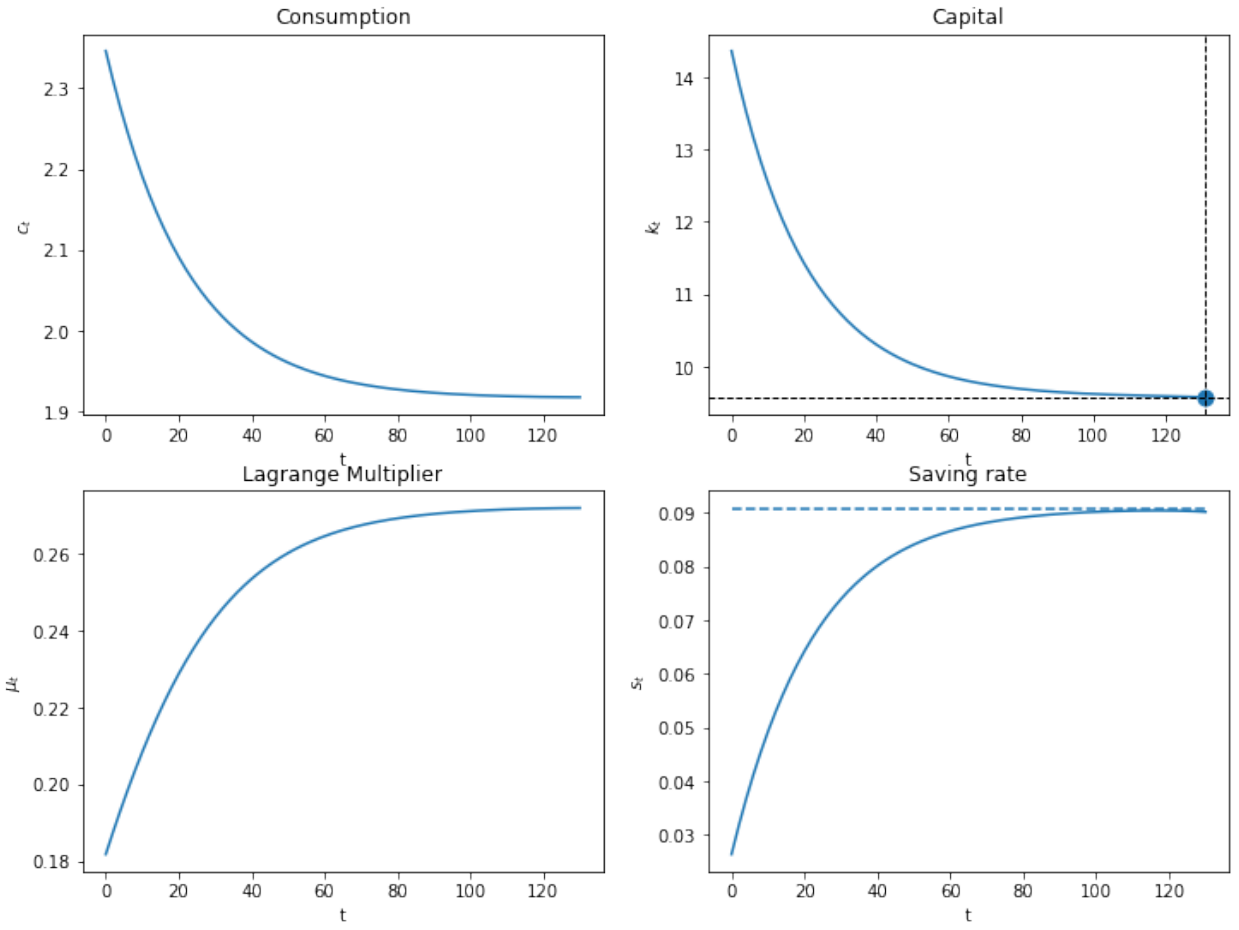
The planner slowly lowers the saving rate until reaching a steady state in which $f'(K) = \rho + \delta$.

25.7.1 Exercise

- Plot the optimal consumption, capital, and saving paths when the initial capital level begins at 1.5 times the steady state level as we shoot towards the steady state at $T = 130$.
- Why does the saving rate respond as it does?

25.7.2 Solution

```
plot_saving_rate(pp, 0.3, k_ss*1.5, [130], k_ter=k_ss, k_ss=k_ss, s_ss=s_ss)
```



25.8 Concluding Remarks

In *Cass-Koopmans Competitive Equilibrium*, we study a decentralized version of an economy with exactly the same technology and preference structure as deployed here.

In that lecture, we replace the planner of this lecture with Adam Smith's **invisible hand**

In place of quantity choices made by the planner, there are market prices somewhat produced by the invisible hand.

Market prices must adjust to reconcile distinct decisions that are made independently by a representative household and a representative firm.

The relationship between a command economy like the one studied in this lecture and a market economy like that studied in *Cass-Koopmans Competitive Equilibrium* is a foundational topic in general equilibrium theory and welfare economics.

CHAPTER

TWENTYSIX

CASS-KOOPMANS COMPETITIVE EQUILIBRIUM

Contents

26.1 Overview

This lecture continues our analysis in this lecture *Cass-Koopmans Planning Model* about the model that Tjalling Koopmans [Koo65] and David Cass [Cas65] used to study optimal growth.

This lecture illustrates what is, in fact, a more general connection between a **planned economy** and an economy organized as a **competitive equilibrium**.

The earlier lecture *Cass-Koopmans Planning Model* studied a planning problem and used ideas including

- A min-max problem for solving the planning problem.
- A **shooting algorithm** for solving difference equations subject to initial and terminal conditions.
- A **turnpike** property that describes optimal paths for long-but-finite horizon economies.

The present lecture uses additional ideas including

- Hicks-Arrow prices named after John R. Hicks and Kenneth Arrow.
- A connection between some Lagrange multipliers in the min-max problem and the Hicks-Arrow prices.
- A **Big** K **, little** k trick widely used in macroeconomic dynamics.
  - We shall encounter this trick in this lecture and also in this lecture.
- A non-stochastic version of a theory of the **term structure of interest rates**.

- An intimate connection between the cases for the optimality of two competing visions of good ways to organize an economy, namely:
  - **socialism** in which a central planner commands the allocation of resources, and
  - **capitalism** (also known as **a market economy**) in which competitive equilibrium **prices** induce individual consumers and producers to choose a socially optimal allocation as an unintended consequence of their selfish decisions

Let's start with some standard imports:

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
from numba import njit, float64
from numba.experimental import jitclass
import numpy as np
```

26.2 Review of Cass-Koopmans Model

The physical setting is identical with that in *Cass-Koopmans Planning Model*.

Time is discrete and takes values $t = 0, 1, \ldots, T$.

A single good can either be consumed or invested in physical capital.

The consumption good is not durable and depreciates completely if not consumed immediately.

The capital good is durable but partially depreciates each period at a constant rate.

We let C_t be a nondurable consumption good at time t.

Let K_t be the stock of physical capital at time t.

Let $\vec{C} = \{C_0, \ldots, C_T\}$ and $\vec{K} = \{K_0, \ldots, K_{T+1}\}$.

A representative household is endowed with one unit of labor at each t and likes the consumption good at each t.

The representative household inelastically supplies a single unit of labor N_t at each t, so that $N_t = 1$ for all $t \in [0, T]$.

The representative household has preferences over consumption bundles ordered by the utility functional:

$$U(\vec{C}) = \sum_{t=0}^{T} \beta^t \frac{C_t^{1-\gamma}}{1-\gamma}$$

where $\beta \in (0, 1)$ is a discount factor and $\gamma > 0$ governs the curvature of the one-period utility function.

We assume that $K_0 > 0$.

There is an economy-wide production function

$$F(K_t, N_t) = AK_t^{\alpha} N_t^{1-\alpha}$$

with $0 < \alpha < 1$, $A > 0$.

A feasible allocation $\vec{C}, \vec{K}$ satisfies

$$C_t + K_{t+1} \leq F(K_t, N_t) + (1-\delta)K_t, \quad \text{for all } t \in [0, T]$$

where $\delta \in (0, 1)$ is a depreciation rate of capital.

26.2.1 Planning Problem

In this lecture *Cass-Koopmans Planning Model*, we studied a problem in which a planner chooses an allocation $\{\vec{C}, \vec{K}\}$ to maximize (1) subject to (4).

The allocation that solves the planning problem plays an important role in a competitive equilibrium as we shall see below.

26.3 Competitive Equilibrium

We now study a decentralized version of the economy.

It shares the same technology and preference structure as the planned economy studied in this lecture *Cass-Koopmans Planning Model*.

But now there is no planner.

Market prices adjust to reconcile distinct decisions that are made separately by a representative household and a representative firm.

There is a representative consumer who has the same preferences over consumption plans as did the consumer in the planned economy.

Instead of being told what to consume and save by a planner, the household chooses for itself subject to a budget constraint

- At each time t, the household receives wages and rentals of capital from a firm – these comprise its **income** at time t.
- The consumer decides how much income to allocate to consumption or to savings.
- The household can save either by acquiring additional physical capital (it trades one for one with time t consumption) or by acquiring claims on consumption at dates other than t.
- The household owns all physical capital and labor and rents them to the firm.
- The household consumes, supplies labor, and invests in physical capital.
- A profit-maximizing representative firm operates the production technology.
- The firm rents labor and capital each period from the representative household and sells its output each period to the household.
- The representative household and the representative firm are both **price takers** who believe that prices are not affected by their choices

Note: We can think of there being a large number M of identical representative consumers and M identical representative firms.

26.4 Market Structure

The representative household and the representative firm are both price takers.

The household owns both factors of production, namely, labor and physical capital.

Each period, the firm rents both factors from the household.

There is a **single** grand competitive market in which a household can trade date 0 goods for goods at all other dates $t = 1, 2, \ldots, T$.

26.4.1 Prices

There are sequences of prices $\{w_t, \eta_t\}_{t=0}^T = \{\vec{w}, \vec{\eta}\}$ where w_t is a wage or rental rate for labor at time t and η_t is a rental rate for capital at time t.

In addition there is are intertemporal prices that work as follows.

Let q_t^0 be the price of a good at date t relative to a good at date 0.

We call $\{q_t^0\}_{t=0}^T$ a vector of **Hicks-Arrow prices**, named after the 1972 economics Nobel prize winners.

Evidently,

$$q_t^0 = \frac{\text{number of time 0 goods}}{\text{number of time t goods}}$$

Because q_t^0 is a **relative price**, the units in terms of which prices are quoted are arbitrary – we are free to normalize them.

26.5 Firm Problem

At time t a representative firm hires labor $\tilde{n}_t$ and capital $\tilde{k}_t$.

The firm's profits at time t are

$$F(\tilde{k}_t, \tilde{n}_t) - w_t \tilde{n}_t - \eta_t \tilde{k}_t$$

where w_t is a wage rate at t and η_t is the rental rate on capital at t.

As in the planned economy model

$$F(\tilde{k}_t, \tilde{n}_t) = A \tilde{k}_t^\alpha \tilde{n}_t^{1-\alpha}$$

26.5.1 Zero Profit Conditions

Zero-profits condition for capital and labor are

$$F_k(\tilde{k}_t, \tilde{n}_t) = \eta_t$$

and

$$F_n(\tilde{k}_t, \tilde{n}_t) = w_t \tag{1}$$

These conditions emerge from a no-arbitrage requirement.

To describe this no-arbitrage profits reasoning, we begin by applying a theorem of Euler about linearly homogenous functions.

The theorem applies to the Cobb-Douglas production function because it assumed displays constant returns to scale:

$$\alpha F(\tilde{k}_t, \tilde{n}_t) = F(\alpha \tilde{k}_t, \alpha \tilde{n}_t)$$

for $\alpha \in (0, 1)$.

Taking the partial derivative $\frac{\partial F}{\partial \alpha}$ on both sides of the above equation gives

$$F(\tilde{k}_t, \tilde{n}_t) =_{\text{chain rule}} \frac{\partial F}{\partial \tilde{k}_t} \tilde{k}_t + \frac{\partial F}{\partial \tilde{n}_t} \tilde{n}_t$$

Rewrite the firm's profits as

$$\frac{\partial F}{\partial \tilde{k}_t}\tilde{k}_t + \frac{\partial F}{\partial \tilde{n}_t}\tilde{n}_t - w_t\tilde{n}_t - \eta_t k_t$$

or

$$\left(\frac{\partial F}{\partial \tilde{k}_t} - \eta_t\right)\tilde{k}_t + \left(\frac{\partial F}{\partial \tilde{n}_t} - w_t\right)\tilde{n}_t$$

Because F is homogeneous of degree 1, it follows that $\frac{\partial F}{\partial \tilde{k}_t}$ and $\frac{\partial F}{\partial \tilde{n}_t}$ are homogeneous of degree 0 and therefore fixed with respect to $\tilde{k}_t$ and $\tilde{n}_t$.

If $\frac{\partial F}{\partial \tilde{k}_t} > \eta_t$, then the firm makes positive profits on each additional unit of $\tilde{k}_t$, so it will want to make $\tilde{k}_t$ arbitrarily large.

But setting $\tilde{k}_t = +\infty$ is not physically feasible, so presumably **equilibrium** prices will assume values that present the firm with no such arbitrage opportunity.

A similar argument applies if $\frac{\partial F}{\partial \tilde{n}_t} > w_t$.

If $\frac{\partial \tilde{k}_t}{\partial \tilde{k}_t} < \eta_t$, the firm will set $\tilde{k}_t$ to zero, something that is not feasible.

It is convenient to define $\vec{w} = \{w_0, \ldots, w_T\}$and $\vec{\eta} = \{\eta_0, \ldots, \eta_T\}$.

26.6 Household Problem

A representative household lives at $t = 0, 1, \ldots, T$.

At t, the household rents 1 unit of labor and k_t units of capital to a firm and receives income

$$w_t 1 + \eta_t k_t$$

At t the household allocates its income to the following purchases

$$(c_t + (k_{t+1} - (1-\delta)k_t))$$

Here $(k_{t+1} - (1-\delta)k_t)$ is the household's net investment in physical capital and $\delta \in (0,1)$ is again a depreciation rate of capital.

In period t is free to purchase more goods to be consumed and invested in physical capital than its income from supplying capital and labor to the firm, provided that in some other periods its income exceeds its purchases.

A household's net excess demand for time t consumption goods is the gap

$$e_t \equiv (c_t + (k_{t+1} - (1-\delta)k_t)) - (w_t 1 + \eta_t k_t)$$

Let $\vec{c} = \{c_0, \ldots, c_T\}$ and let $\vec{k} = \{k_1, \ldots, k_{T+1}\}$.

k_0 is given to the household.

The household faces a **single** budget constraint. that states that the present value of the household's net excess demands must be zero:

$$\sum_{t=0}^{T} q_t^0 e_t \leq 0$$

or

$$\sum_{t=0}^{T} q_t^0 \left(c_t + (k_{t+1} - (1-\delta)k_t) - (w_t 1 + \eta_t k_t)\right) \leq 0$$

The household chooses an allocation to solve the constrained optimization problem:

$$\begin{aligned} &\max_{\vec{c},\vec{k}} \sum_{t=0}^{T} \beta^t u(c_t) \\ \text{subject to } &\sum_{t=0}^{T} q_t^0 \left(c_t + (k_{t+1} - (1-\delta)k_t) - w_t - \eta_t k_t\right) \leq 0 \end{aligned}$$

26.6.1 Definitions

- A **price system** is a sequence $\{q_t^0, \eta_t, w_t\}_{t=0}^T = \{\vec{q}, \vec{\eta}, \vec{w}\}$.
- An **allocation** is a sequence $\{c_t, k_{t+1}, n_t = 1\}_{t=0}^T = \{\vec{c}, \vec{k}, \vec{n}\}$.
- A **competitive equilibrium** is a price system and an allocation for which
  - Given the price system, the allocation solves the household's problem.
  - Given the price system, the allocation solves the firm's problem.

26.7 Computing a Competitive Equilibrium

We compute a competitive equilibrium by using a **guess and verify** approach.

- We **guess** equilibrium price sequences $\{\vec{q}, \vec{\eta}, \vec{w}\}$.
- We then **verify** that at those prices, the household and the firm choose the same allocation.

26.7.1 Guess for Price System

In this lecture *Cass-Koopmans Planning Model*, we computed an allocation $\{\vec{C}, \vec{K}, \vec{N}\}$ that solves the planning problem.

(This allocation will constitute the **Big** K to be in the present instance of the **Big** K **, little** k trick that we'll apply to a competitive equilibrium in the spirit of this lecture and this lecture.)

We use that allocation to construct a guess for the equilibrium price system.

In particular, we guess that for $t = 0, \ldots, T$:

$$\lambda q_t^0 = \beta^t u'(K_t) = \beta^t \mu_t \tag{2}$$

$$w_t = f(K_t) - K_t f'(K_t) \tag{3}$$

$$\eta_t = f'(K_t) \tag{4}$$

At these prices, let the capital chosen by the household be

$$k_t^*(\vec{q}, \vec{w}, \vec{\eta}), \quad t \geq 0 \tag{5}$$

and let the allocation chosen by the firm be

$$\tilde{k}_t^*(\vec{q}, \vec{w}, \vec{\eta}), \quad t \geq 0$$

and so on.

If our guess for the equilibrium price system is correct, then it must occur that

$$k_t^* = \tilde{k}_t^* \tag{6}$$

$$1 = \tilde{n}_t^* \tag{7}$$

$$c_t^* + k_{t+1}^* - (1-\delta)k_t^* = F(\tilde{k}_t^*, \tilde{n}_t^*)$$

We shall verify that for $t = 0, \ldots, T$ the allocations chosen by the household and the firm both equal the allocation that solves the planning problem:

$$k_t^* = \tilde{k}_t^* = K_t, \tilde{n}_t = 1, c_t^* = C_t \tag{8}$$

26.7.2 Verification Procedure

Our approach is to stare at first-order necessary conditions for the optimization problems of the household and the firm.

At the price system we have guessed, we'll then verify that both sets of first-order conditions are satisfied at the allocation that solves the planning problem.

26.7.3 Household's Lagrangian

To solve the household's problem, we formulate the Lagrangian

$$\mathcal{L}(\vec{c}, \vec{k}, \lambda) = \sum_{t=0}^{T} \beta^t u(c_t) + \lambda \left(\sum_{t=0}^{T} q_t^0 \left(((1-\delta)k_t - w_t) + \eta_t k_t - c_t - k_{t+1} \right) \right)$$

and attack the min-max problem:

$$\min_{\lambda} \max_{\vec{c}, \vec{k}} \mathcal{L}(\vec{c}, \vec{k}, \lambda)$$

First-order conditions are

$$c_t: \quad \beta^t u'(c_t) - \lambda q_t^0 = 0 \quad t = 0, 1, \ldots, T \tag{9}$$

$$k_t: \quad -\lambda q_t^0 \left[(1-\delta) + \eta_t \right] + \lambda q_{t-1}^0 = 0 \quad t = 1, 2, \ldots, T+1 \tag{10}$$

$$\lambda: \quad \left(\sum_{t=0}^{T} q_t^0 \left(c_t + (k_{t+1} - (1-\delta)k_t) - w_t - \eta_t k_t \right) \right) \leq 0 \tag{11}$$

$$k_{T+1}: \quad -\lambda q_0^{T+1} \leq 0, \ \leq 0 \text{ if } k_{T+1} = 0; \ = 0 \text{ if } k_{T+1} > 0 \tag{12}$$

Now we plug in our guesses of prices and embark on some algebra in the hope of derived all first-order necessary conditions (7)-(10) for the planning problem from this lecture *Cass-Koopmans Planning Model*.

Combining (9) and (2), we get:

$$u'(C_t) = \mu_t$$

which is (7).

Combining (10), (2), and (4) we get:

$$-\lambda\beta^t\mu_t\left[(1-\delta)+f'(K_t)\right]+\lambda\beta^{t-1}\mu_{t-1}=0 \tag{13}$$

Rewriting (13) by dividing by λ on both sides (which is nonzero since u'>0) we get:

$$\beta^t\mu_t[(1-\delta+f'(K_t)]=\beta^{t-1}\mu_{t-1}$$

or

$$\beta\mu_t[(1-\delta+f'(K_t)]=\mu_{t-1}$$

which is (8).

Combining (11), (2), (3) and (4) after multiplying both sides of (11) by λ, we get

$$\sum_{t=0}^{T}\beta^t\mu_t\left(C_t+(K_{t+1}-(1-\delta)K_t)-f(K_t)+K_tf'(K_t)-f'(K_t)K_t\right)\leq 0$$

which simplifies

$$\sum_{t=0}^{T}\beta^t\mu_t\left(C_t+K_{t+1}-(1-\delta)K_t-F(K_t,1)\right)\leq 0$$

Since $\beta^t\mu_t>0$ for $t=0,\ldots,T$, it follows that

$$C_t+K_{t+1}-(1-\delta)K_t-F(K_t,1)=0 \quad \text{ for all } t \text{ in } 0,\ldots,T$$

which is (9).

Combining (12) and (2), we get:

$$-\beta^{T+1}\mu_{T+1}\leq 0$$

Dividing both sides by β^{T+1} gives

$$-\mu_{T+1}\leq 0$$

which is (10) for the planning problem.

Thus, at our guess of the equilibrium price system, the allocation that solves the planning problem also solves the problem faced by a representative household living in a competitive equilibrium.

We now turn to the problem faced by a firm in a competitive equilibrium:

If we plug (8) into (1) for all t, we get

$$\frac{\partial F(K_t,1)}{\partial K_t}=f'(K_t)=\eta_t$$

which is (4).

If we now plug (8) into (1) for all t, we get:

$$\frac{\partial F(\tilde{K}_t,1)}{\partial \tilde{L}_t}=f(K_t)-f'(K_t)K_t=w_t$$

which is exactly (5).

So at our guess for the equilibrium price system, the allocation that solves the planning problem also solves the problem faced by a firm within a competitive equilibrium.

By (6) and (7) this allocation is identical to the one that solves the consumer's problem.

Note: Because budget sets are affected only by relative prices, $\{q_0^t\}$ is determined only up to multiplication by a positive constant.

Normalization: We are free to choose a $\{q_0^t\}$ that makes $\lambda = 1$ so that we are measuring q_0^t in units of the marginal utility of time 0 goods.

We will plot q, w, η below to show these equilibrium prices induce the same aggregate movements that we saw earlier in the planning problem.

To proceed, we bring in Python code that *Cass-Koopmans Planning Model* used to solve the planning problem

First let's define a `jitclass` that stores parameters and functions the characterize an economy.

```
planning_data = [
    ('γ', float64),    # Coefficient of relative risk aversion
    ('β', float64),    # Discount factor
    ('δ', float64),    # Depreciation rate on capital
    ('α', float64),    # Return to capital per capita
    ('A', float64)     # Technology
]
```

```
@jitclass(planning_data)
class PlanningProblem():

    def __init__(self, γ=2, β=0.95, δ=0.02, α=0.33, A=1):

        self.γ, self.β = γ, β
        self.δ, self.α, self.A = δ, α, A

    def u(self, c):
        '''
        Utility function
        ASIDE: If you have a utility function that is hard to solve by hand
        you can use automatic or symbolic differentiation
        See https://github.com/HIPS/autograd
        '''
        γ = self.γ

        return c ** (1 - γ) / (1 - γ) if γ!= 1 else np.log(c)

    def u_prime(self, c):
        'Derivative of utility'
        γ = self.γ

        return c ** (-γ)

    def u_prime_inv(self, c):
        'Inverse of derivative of utility'
        γ = self.γ

        return c ** (-1 / γ)

    def f(self, k):
        'Production function'
```

(continues on next page)

(continued from previous page)

```
        α, A = self.α, self.A

        return A * k ** α

    def f_prime(self, k):
        'Derivative of production function'
        α, A = self.α, self.A

        return α * A * k ** (α - 1)

    def f_prime_inv(self, k):
        'Inverse of derivative of production function'
        α, A = self.α, self.A

        return (k / (A * α)) ** (1 / (α - 1))

    def next_k_c(self, k, c):
        ''''
        Given the current capital Kt and an arbitrary feasible
        consumption choice Ct, computes Kt+1 by state transition law
        and optimal Ct+1 by Euler equation.
        '''
        β, δ = self.β, self.δ
        u_prime, u_prime_inv = self.u_prime, self.u_prime_inv
        f, f_prime = self.f, self.f_prime

        k_next = f(k) + (1 - δ) * k - c
        c_next = u_prime_inv(u_prime(c) / (β * (f_prime(k_next) + (1 - δ))))

        return k_next, c_next
```

```
@njit
def shooting(pp, c0, k0, T=10):
    '''
    Given the initial condition of capital k0 and an initial guess
    of consumption c0, computes the whole paths of c and k
    using the state transition law and Euler equation for T periods.
    '''
    if c0 > pp.f(k0):
        print("initial consumption is not feasible")

        return None

    # initialize vectors of c and k
    c_vec = np.empty(T+1)
    k_vec = np.empty(T+2)

    c_vec[0] = c0
    k_vec[0] = k0

    for t in range(T):
        k_vec[t+1], c_vec[t+1] = pp.next_k_c(k_vec[t], c_vec[t])

    k_vec[T+1] = pp.f(k_vec[T]) + (1 - pp.δ) * k_vec[T] - c_vec[T]

    return c_vec, k_vec
```

```
@njit
def bisection(pp, c0, k0, T=10, tol=1e-4, max_iter=500, k_ter=0, verbose=True):

    # initial boundaries for guess c0
    c0_upper = pp.f(k0)
    c0_lower = 0

    i = 0
    while True:
        c_vec, k_vec = shooting(pp, c0, k0, T)
        error = k_vec[-1] - k_ter

        # check if the terminal condition is satisfied
        if np.abs(error) < tol:
            if verbose:
                print('Converged successfully on iteration ', i+1)
            return c_vec, k_vec

        i += 1
        if i == max_iter:
            if verbose:
                print('Convergence failed.')
            return c_vec, k_vec

        # if iteration continues, updates boundaries and guess of c0
        if error > 0:
            c0_lower = c0
        else:
            c0_upper = c0

        c0 = (c0_lower + c0_upper) / 2
```

```
pp = PlanningProblem()

# Steady states
ρ = 1 / pp.β - 1
k_ss = pp.f_prime_inv(ρ+pp.δ)
c_ss = pp.f(k_ss) - pp.δ * k_ss
```

The above code from this lecture *Cass-Koopmans Planning Model* lets us compute an optimal allocation for the planning problem that turns out to be the allocation associated with a competitive equilibium.

Now we're ready to bring in Python code that we require to compute additional objects that appear in a competitive equilibrium.

```
@njit
def q(pp, c_path):
    # Here we choose numeraire to be u'(c_0) -- this is q^(t_0)_t
    T = len(c_path) - 1
    q_path = np.ones(T+1)
    q_path[0] = 1
    for t in range(1, T+1):
        q_path[t] = pp.β ** t * pp.u_prime(c_path[t])
    return q_path

@njit
def w(pp, k_path):
```

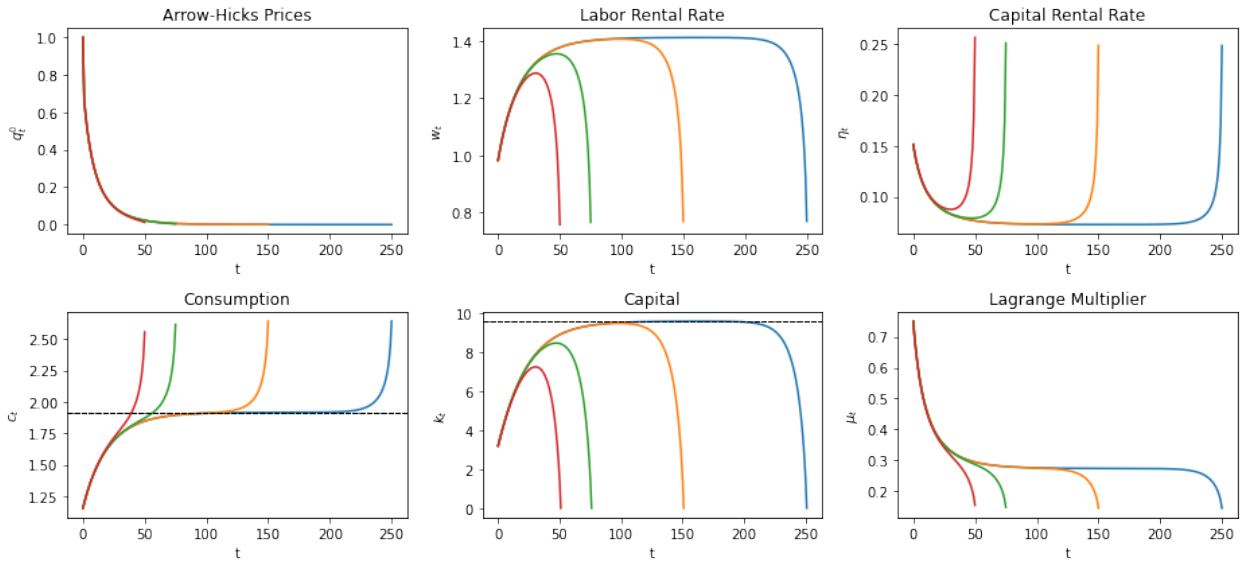
(continues on next page)

(continued from previous page)

```
    w_path = pp.f(k_path) - k_path * pp.f_prime(k_path)
    return w_path

@njit
def η(pp, k_path):
    η_path = pp.f_prime(k_path)
    return η_path
```

Now we calculate and plot for each T

```
T_arr = [250, 150, 75, 50]

fix, axs = plt.subplots(2, 3, figsize=(13, 6))
titles = ['Arrow-Hicks Prices', 'Labor Rental Rate', 'Capital Rental Rate',
          'Consumption', 'Capital', 'Lagrange Multiplier']
ylabels = ['$q_t^0$', '$w_t$', '$\eta_t$', '$c_t$', '$k_t$', '$\mu_t$']

for T in T_arr:
    c_path, k_path = bisection(pp, 0.3, k_ss/3, T, verbose=False)
    μ_path = pp.u_prime(c_path)

    q_path = q(pp, c_path)
    w_path = w(pp, k_path)[:-1]
    η_path = η(pp, k_path)[:-1]
    paths = [q_path, w_path, η_path, c_path, k_path, μ_path]

    for i, ax in enumerate(axs.flatten()):
        ax.plot(paths[i])
        ax.set(title=titles[i], ylabel=ylabels[i], xlabel='t')
        if titles[i] == 'Capital':
            ax.axhline(k_ss, lw=1, ls='--', c='k')
        if titles[i] == 'Consumption':
            ax.axhline(c_ss, lw=1, ls='--', c='k')

plt.tight_layout()
plt.show()
```

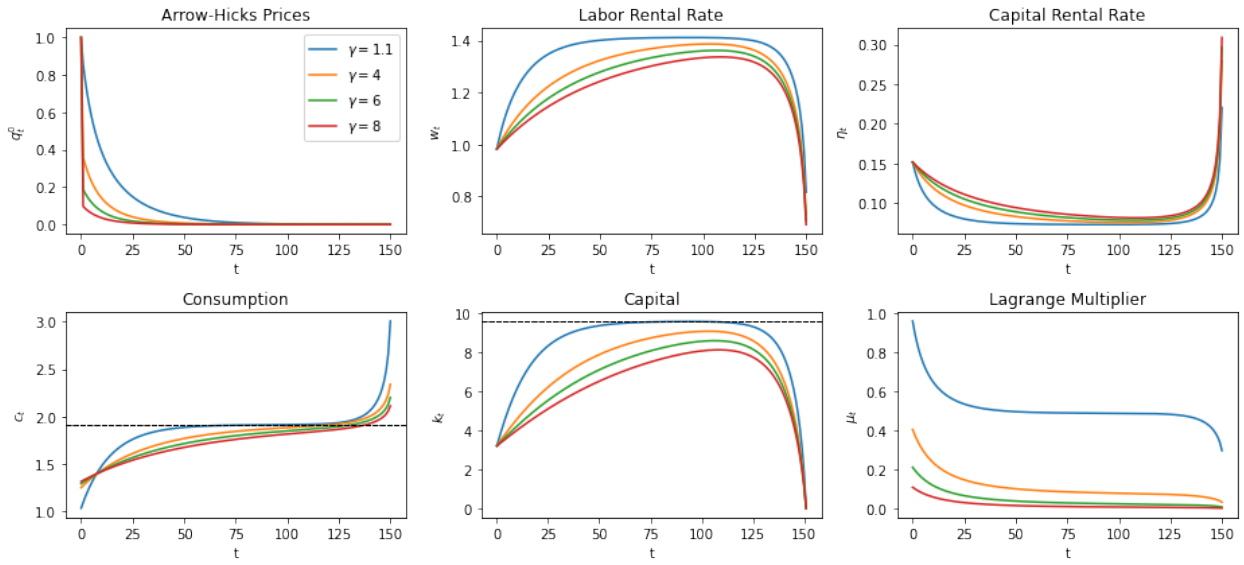


Varying Curvature

Now we see how our results change if we keep T constant, but allow the curvature parameter, γ to vary, starting with K_0 below the steady state.

We plot the results for $T = 150$

```
T = 150
γ_arr = [1.1, 4, 6, 8]

fix, axs = plt.subplots(2, 3, figsize=(13, 6))

for γ in γ_arr:
    pp_γ = PlanningProblem(γ=γ)
    c_path, k_path = bisection(pp_γ, 0.3, k_ss/3, T, verbose=False)
    μ_path = pp_γ.u_prime(c_path)

    q_path = q(pp_γ, c_path)
    w_path = w(pp_γ, k_path)[:-1]
    η_path = η(pp_γ, k_path)[:-1]
    paths = [q_path, w_path, η_path, c_path, k_path, μ_path]

    for i, ax in enumerate(axs.flatten()):
        ax.plot(paths[i], label=f'$\gamma = {γ}$')
        ax.set(title=titles[i], ylabel=ylabels[i], xlabel='t')
        if titles[i] == 'Capital':
            ax.axhline(k_ss, lw=1, ls='--', c='k')
        if titles[i] == 'Consumption':
            ax.axhline(c_ss, lw=1, ls='--', c='k')

axs[0, 0].legend()
plt.tight_layout()
plt.show()
```



Adjusting γ means adjusting how much individuals prefer to smooth consumption.

Higher γ means individuals prefer to smooth more resulting in slower adjustments to the steady state allocations.

Vice-versa for lower γ.

26.8 Yield Curves and Hicks-Arrow Prices

We return to Hicks-Arrow prices and calculate how they are related to **yields** on loans of alternative maturities.

This will let us plot a **yield curve** that graphs yields on bonds of maturities $j = 1, 2, ...$ against $j = 1, 2, ...$

The formulas we want are:

A **yield to maturity** on a loan made at time t_0 that matures at time $t > t_0$

$$r_{t_0,t} = -\frac{\log q_t^{t_0}}{t - t_0}$$

A Hicks-Arrow price for a base-year $t_0 \leq t$

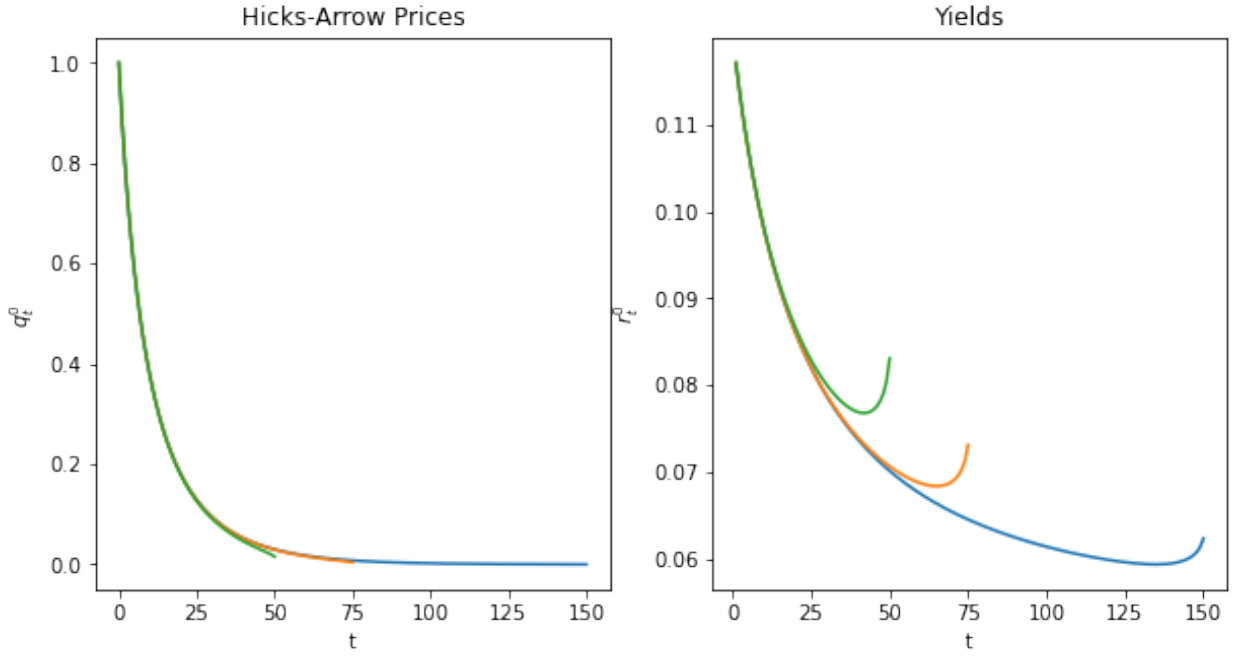
$$q_t^{t_0} = \beta^{t-t_0} \frac{u'(c_t)}{u'(c_{t_0})} = \beta^{t-t_0} \frac{c_t^{-\gamma}}{c_{t_0}^{-\gamma}}$$

We redefine our function for q to allow arbitrary base years, and define a new function for r, then plot both.

We begin by continuing to assume that $t_0 = 0$ and plot things for different maturities $t = T$, with K_0 below the steady state

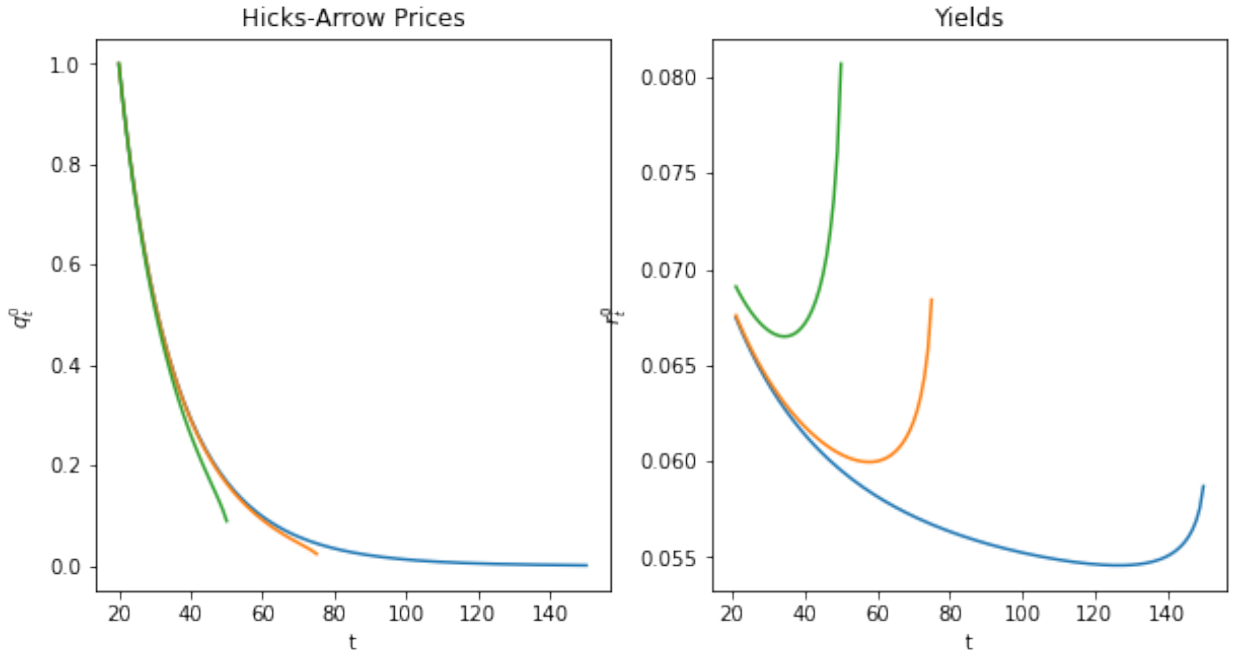
```
@njit
def q_generic(pp, t0, c_path):
    # simplify notations
    β = pp.β
    u_prime = pp.u_prime

    T = len(c_path) - 1
    q_path = np.zeros(T+1-t0)
    q_path[0] = 1
    for t in range(t0+1, T+1):
        q_path[t-t0] = β ** (t-t0) * u_prime(c_path[t]) / u_prime(c_path[t0])
    return q_path

@njit
def r(pp, t0, q_path):
    '''Yield to maturity'''
    r_path = - np.log(q_path[1:]) / np.arange(1, len(q_path))
    return r_path

def plot_yield_curves(pp, t0, c0, k0, T_arr):

    fig, axs = plt.subplots(1, 2, figsize=(10, 5))

    for T in T_arr:
        c_path, k_path = bisection(pp, c0, k0, T, verbose=False)
        q_path = q_generic(pp, t0, c_path)
        r_path = r(pp, t0, q_path)

        axs[0].plot(range(t0, T+1), q_path)
        axs[0].set(xlabel='t', ylabel='$q_t^0$', title='Hicks-Arrow Prices')

        axs[1].plot(range(t0+1, T+1), r_path)
        axs[1].set(xlabel='t', ylabel='$r_t^0$', title='Yields')
```

```
T_arr = [150, 75, 50]
plot_yield_curves(pp, 0, 0.3, k_ss/3, T_arr)
```



Now we plot when $t_0 = 20$

```
plot_yield_curves(pp, 20, 0.3, k_ss/3, T_arr)
```



We aim to have more to say about the term structure of interest rates in a planned lecture on the topic.