Part VIII

Multiple Agent Models

CHAPTER

FIFTYTWO

SCHELLING'S SEGREGATION MODEL

Contents

52.1 Outline

In 1969, Thomas C. Schelling developed a simple but striking model of racial segregation [Sch69].

His model studies the dynamics of racially mixed neighborhoods.

Like much of Schelling's work, the model shows how local interactions can lead to surprising aggregate structure.

In particular, it shows that relatively mild preference for neighbors of similar race can lead in aggregate to the collapse of mixed neighborhoods, and high levels of segregation.

In recognition of this and other research, Schelling was awarded the 2005 Nobel Prize in Economic Sciences (joint with Robert Aumann).

In this lecture, we (in fact you) will build and run a version of Schelling's model.

Let's start with some imports:

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
from random import uniform, seed
from math import sqrt
```

52.2 The Model

We will cover a variation of Schelling's model that is easy to program and captures the main idea.

52.2.1 Set-Up

Suppose we have two types of people: orange people and green people.

For the purpose of this lecture, we will assume there are 250 of each type.

These agents all live on a single unit square.

The location of an agent is just a point (x, y), where $0 < x, y < 1$.

52.2.2 Preferences

We will say that an agent is *happy* if half or more of her 10 nearest neighbors are of the same type.

Here 'nearest' is in terms of Euclidean distance.

An agent who is not happy is called *unhappy*.

An important point here is that agents are not averse to living in mixed areas.

They are perfectly happy if half their neighbors are of the other color.

52.2.3 Behavior

Initially, agents are mixed together (integrated).

In particular, the initial location of each agent is an independent draw from a bivariate uniform distribution on $S = (0, 1)^2$.

Now, cycling through the set of all agents, each agent is now given the chance to stay or move.

We assume that each agent will stay put if they are happy and move if unhappy.

The algorithm for moving is as follows

1. Draw a random location in S
2. If happy at new location, move there
3. Else, go to step 1

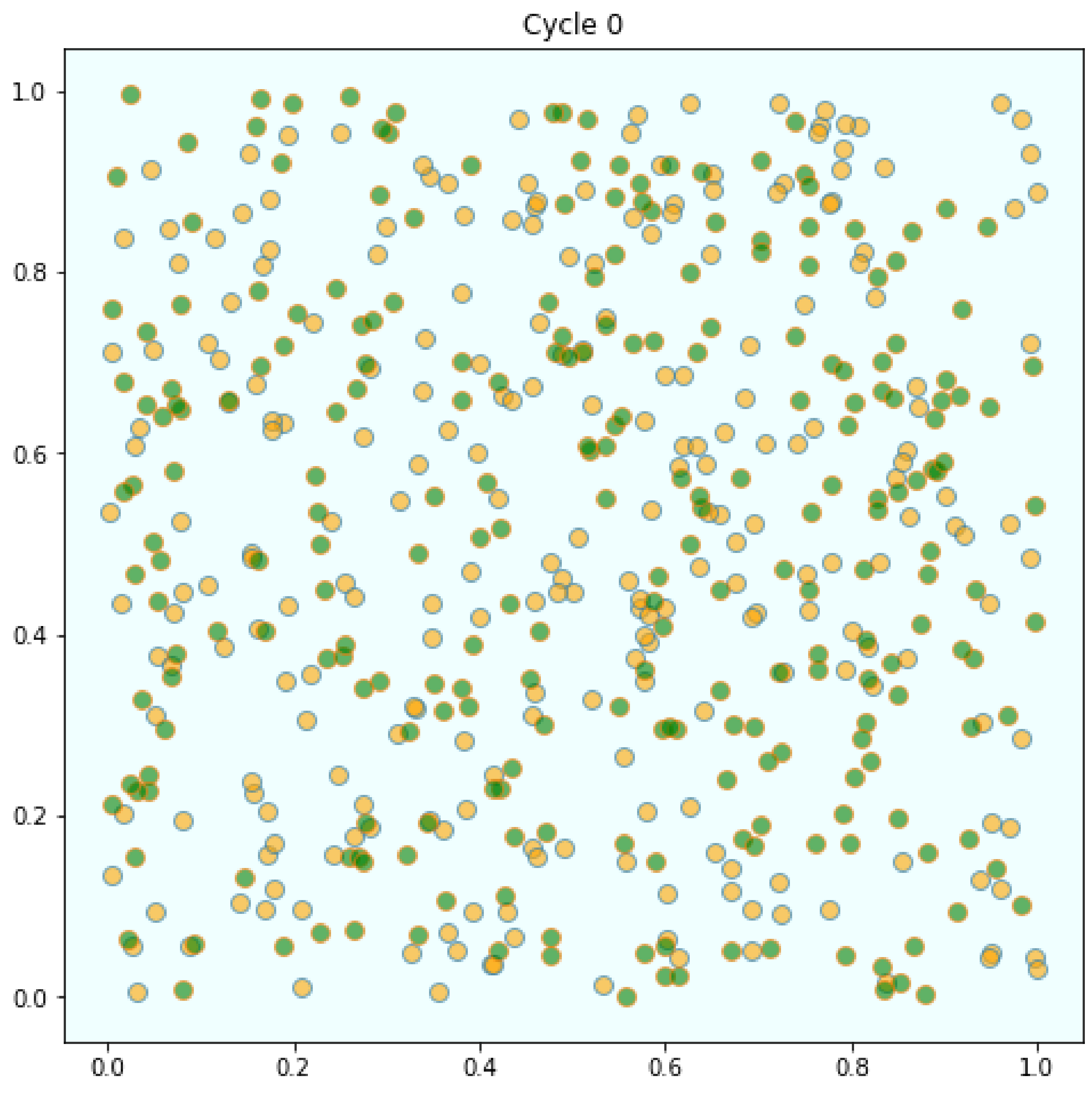
In this way, we cycle continuously through the agents, moving as required.

We continue to cycle until no one wishes to move.

52.3 Results

Let's have a look at the results we got when we coded and ran this model.

As discussed above, agents are initially mixed randomly together.



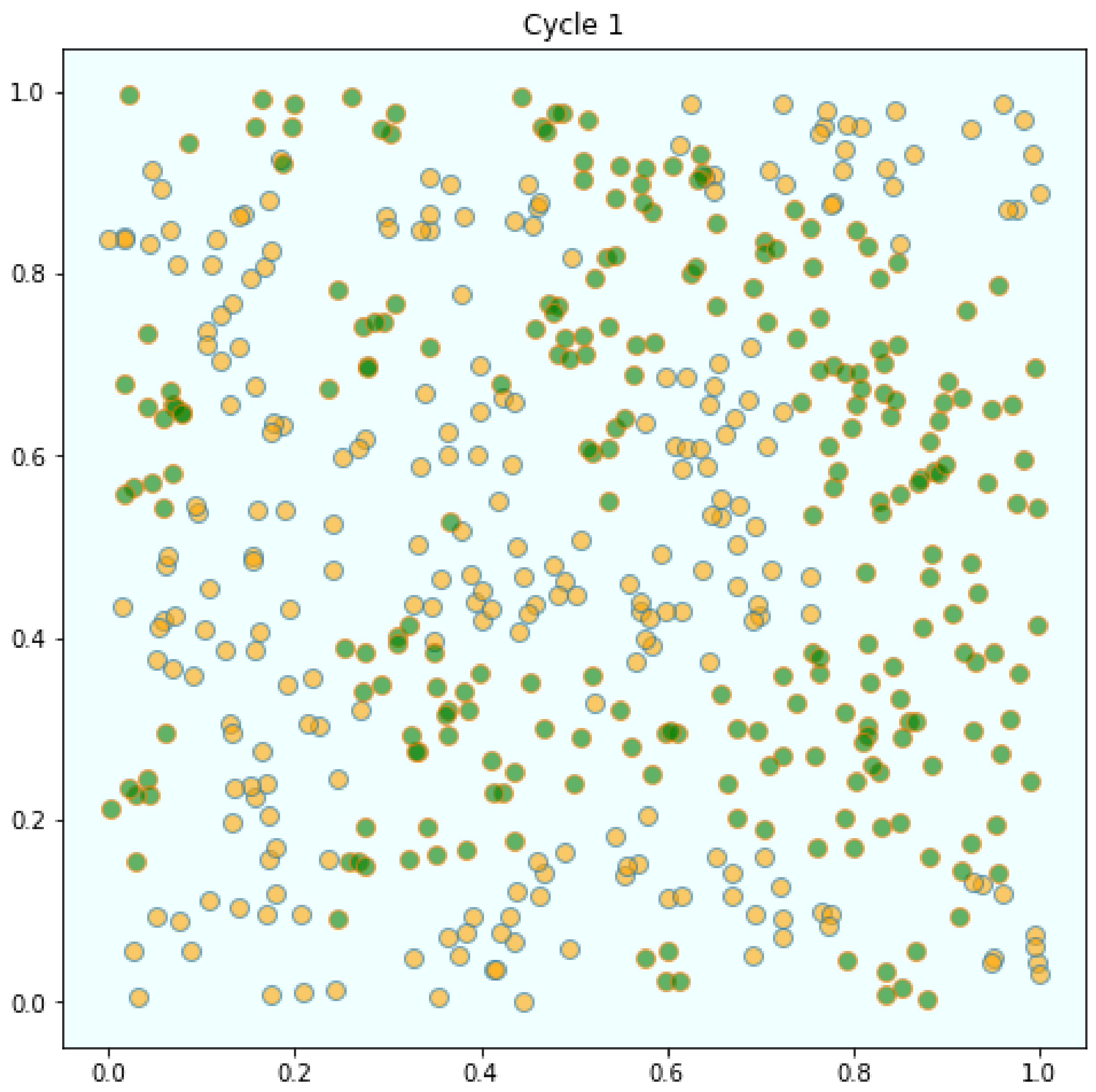
But after several cycles, they become segregated into distinct regions.

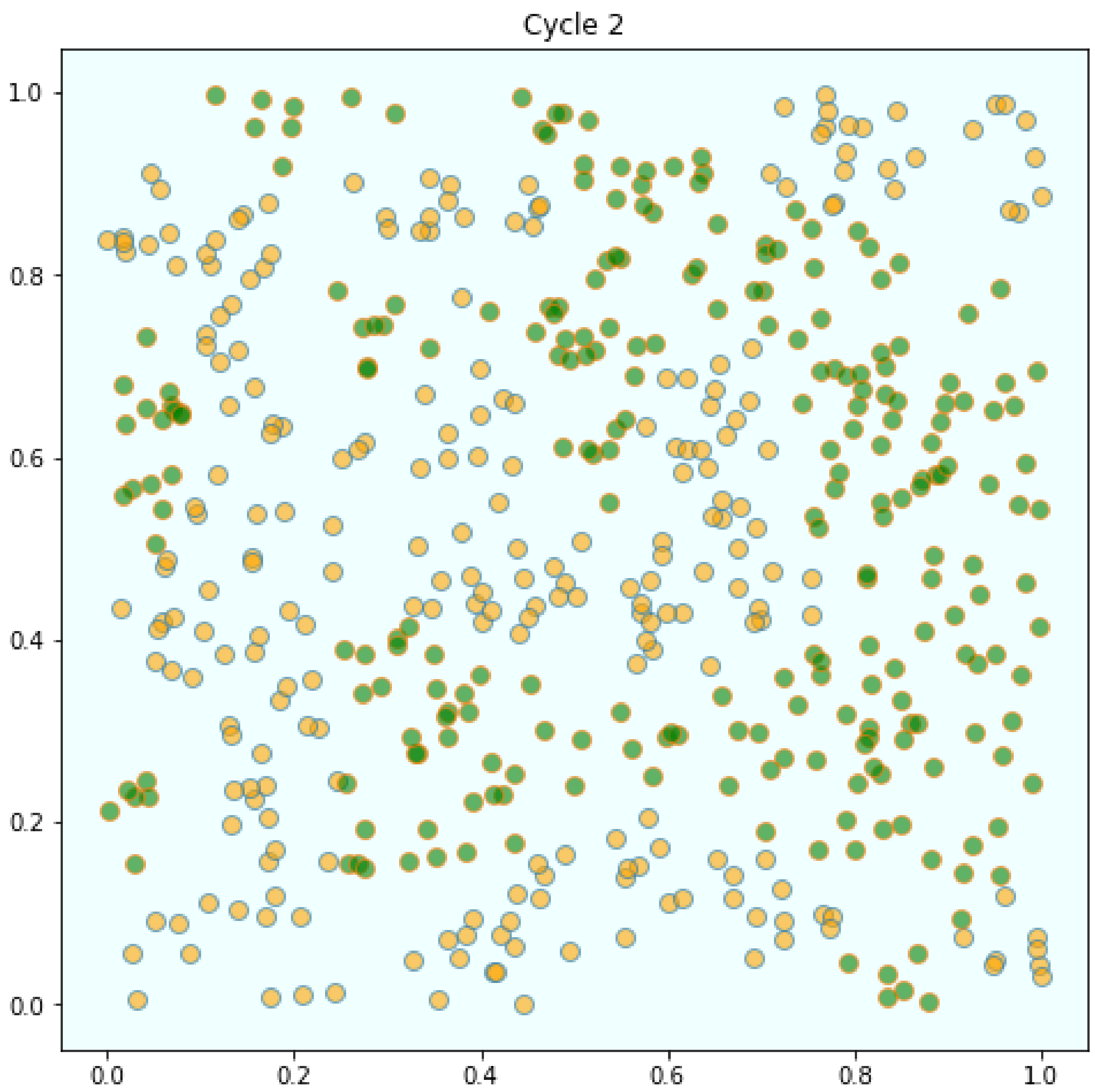
In this instance, the program terminated after 4 cycles through the set of agents, indicating that all agents had reached a state of happiness.

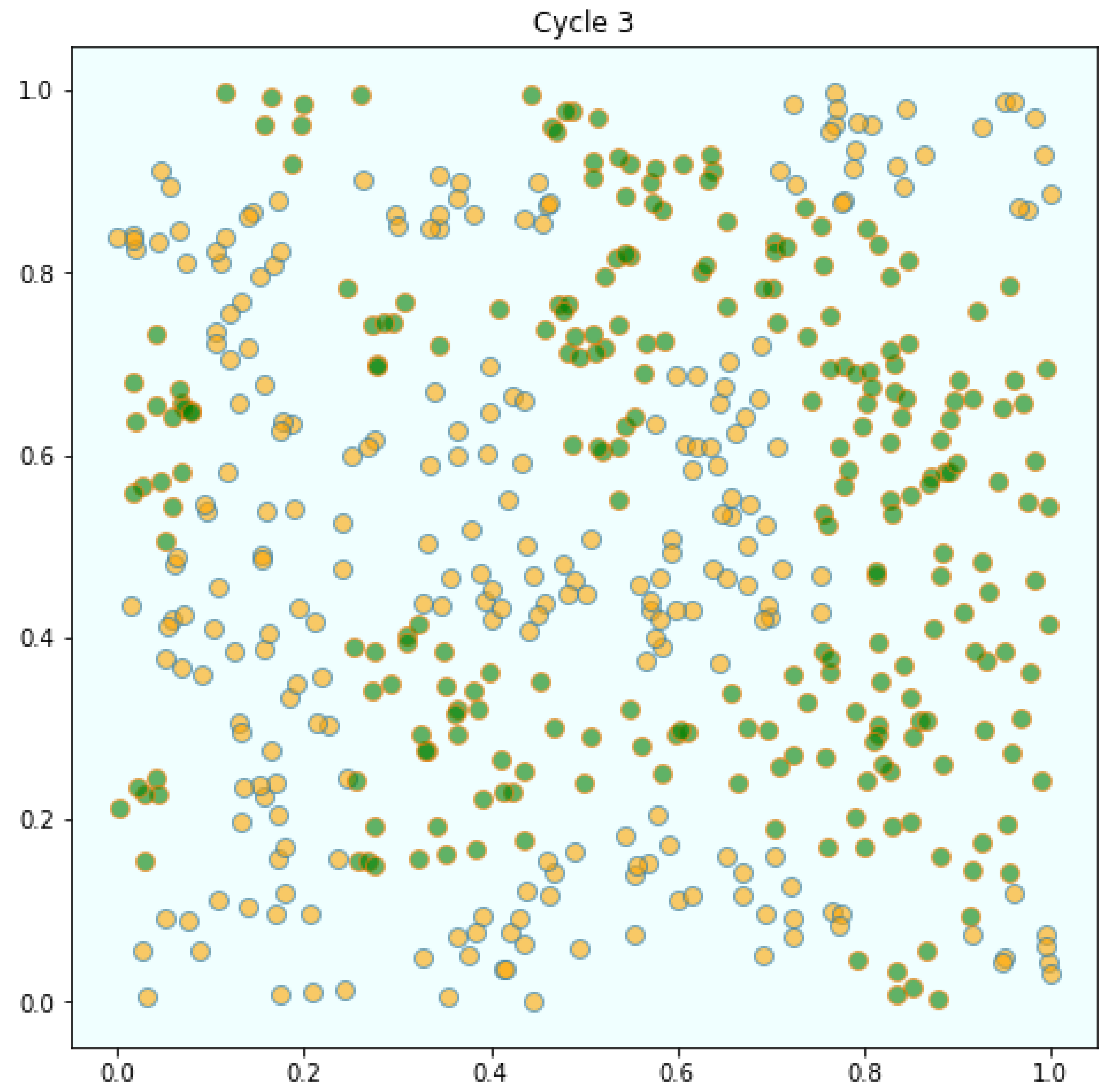
What is striking about the pictures is how rapidly racial integration breaks down.

This is despite the fact that people in the model don't actually mind living mixed with the other type.

Even with these preferences, the outcome is a high degree of segregation.


Cycle 1
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0


Cycle 2
0.0
0.2
0.4
0.6
0.8
1.0
0.0
0.2
0.4
0.6
0.8
1.0


Cycle 3
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

52.4 Exercises

52.4.1 Exercise 1

Implement and run this simulation for yourself.

Consider the following structure for your program.

Agents can be modeled as objects.

Here's an indication of how they might look

```
* Data:

    * type (green or orange)
    * location

* Methods:

    * determine whether happy or not given locations of other agents

    * If not happy, move

        * find a new location where happy
```

And here's some pseudocode for the main loop

```
while agents are still moving
    for agent in agents
        give agent the opportunity to move
```

Use 250 agents of each type.

52.5 Solutions

52.5.1 Exercise 1

Here's one solution that does the job we want.

If you feel like a further exercise, you can probably speed up some of the computations and then increase the number of agents.

```
seed(10)  # For reproducible random numbers

class Agent:

    def __init__(self, type):
        self.type = type
        self.draw_location()

    def draw_location(self):
        self.location = uniform(0, 1), uniform(0, 1)

    def get_distance(self, other):
        "Computes the euclidean distance between self and other agent."
```

(continues on next page)

(continued from previous page)

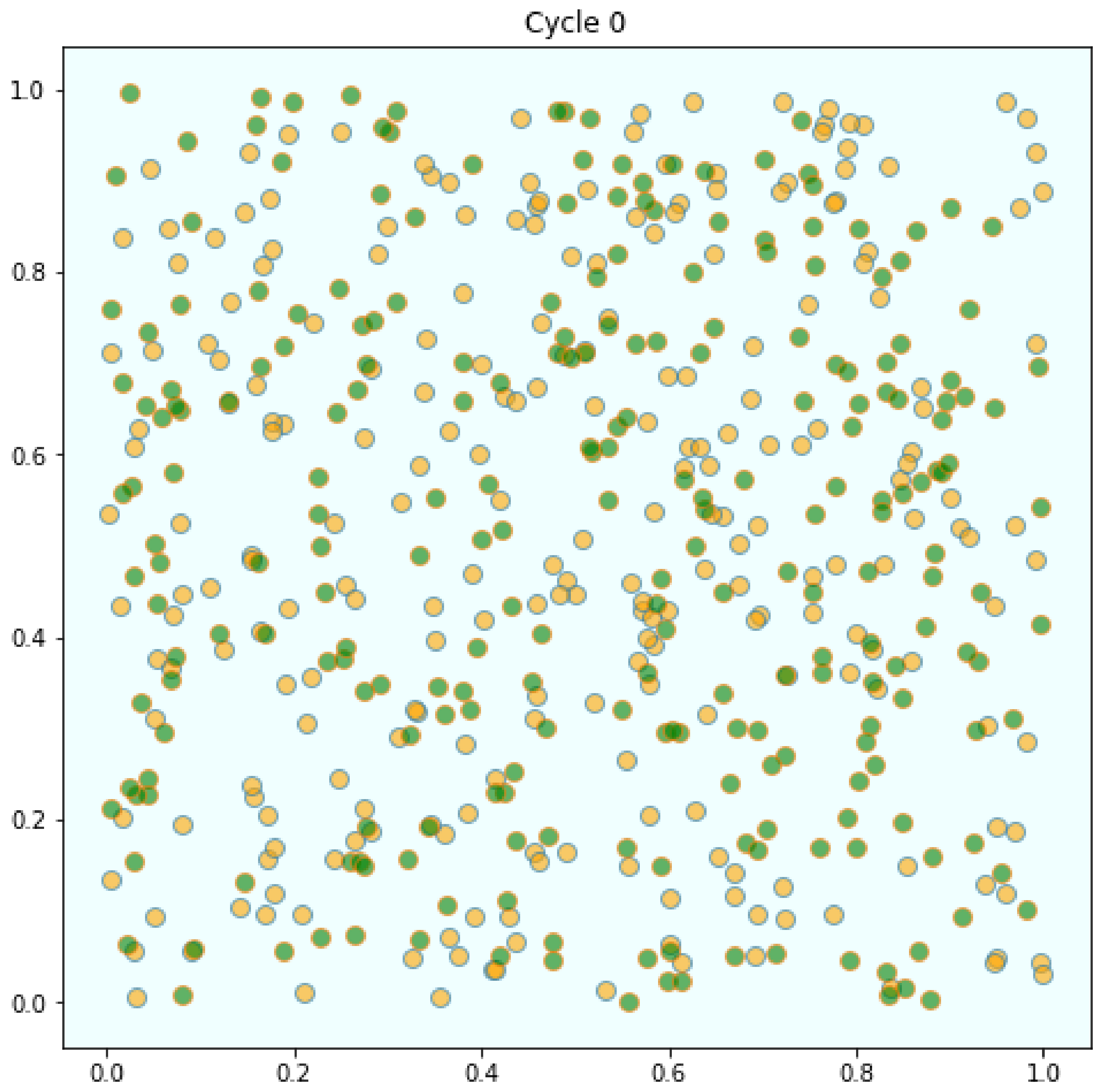
```
        a = (self.location[0] - other.location[0])**2
        b = (self.location[1] - other.location[1])**2
        return sqrt(a + b)

    def happy(self, agents):
        "True if sufficient number of nearest neighbors are of the same type."
        distances = []
        # distances is a list of pairs (d, agent), where d is distance from
        # agent to self
        for agent in agents:
            if self != agent:
                distance = self.get_distance(agent)
                distances.append((distance, agent))
        # == Sort from smallest to largest, according to distance == #
        distances.sort()
        # == Extract the neighboring agents == #
        neighbors = [agent for d, agent in distances[:num_neighbors]]
        # == Count how many neighbors have the same type as self == #
        num_same_type = sum(self.type == agent.type for agent in neighbors)
        return num_same_type >= require_same_type

    def update(self, agents):
        "If not happy, then randomly choose new locations until happy."
        while not self.happy(agents):
            self.draw_location()


def plot_distribution(agents, cycle_num):
    "Plot the distribution of agents after cycle_num rounds of the loop."
    x_values_0, y_values_0 = [], []
    x_values_1, y_values_1 = [], []
    # == Obtain locations of each type == #
    for agent in agents:
        x, y = agent.location
        if agent.type == 0:
            x_values_0.append(x)
            y_values_0.append(y)
        else:
            x_values_1.append(x)
            y_values_1.append(y)
    fig, ax = plt.subplots(figsize=(8, 8))
    plot_args = {'markersize': 8, 'alpha': 0.6}
    ax.set_facecolor('azure')
    ax.plot(x_values_0, y_values_0, 'o', markerfacecolor='orange', **plot_args)
    ax.plot(x_values_1, y_values_1, 'o', markerfacecolor='green', **plot_args)
    ax.set_title(f'Cycle {cycle_num-1}')
    plt.show()

# == Main == #

num_of_type_0 = 250
num_of_type_1 = 250
num_neighbors = 10      # Number of agents regarded as neighbors
require_same_type = 5   # Want at least this many neighbors to be same type

# == Create a list of agents == #
agents = [Agent(0) for i in range(num_of_type_0)]
```

(continues on next page)

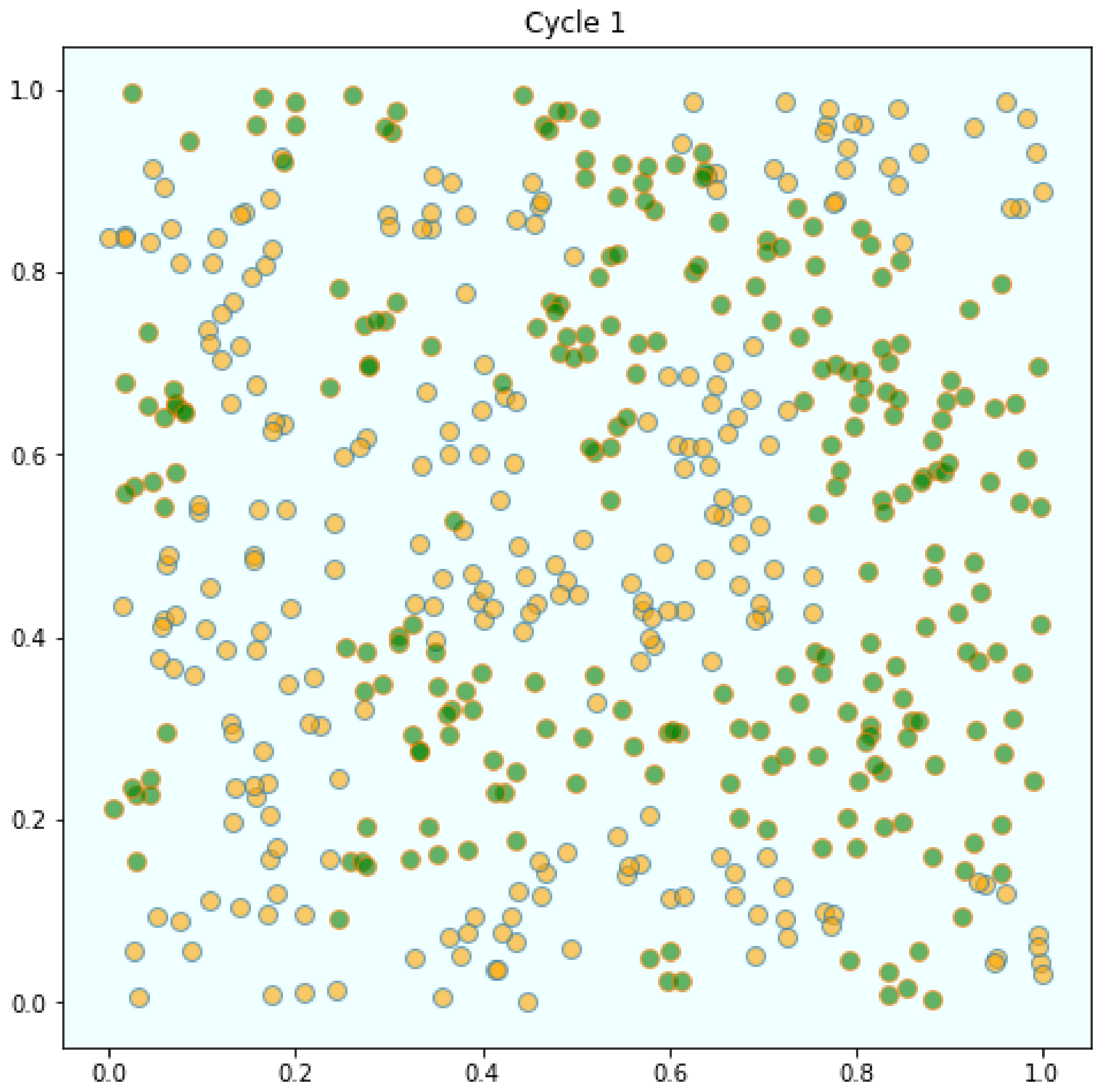
(continued from previous page)

```python
agents.extend(Agent(1) for i in range(num_of_type_1))


count = 1
# ==  Loop until none wishes to move == #
while True:
    print('Entering loop ', count)
    plot_distribution(agents, count)
    count += 1
    no_one_moved = True
    for agent in agents:
        old_location = agent.location
        agent.update(agents)
        if agent.location != old_location:
            no_one_moved = False
    if no_one_moved:
        break

print('Converged, terminating.')
```

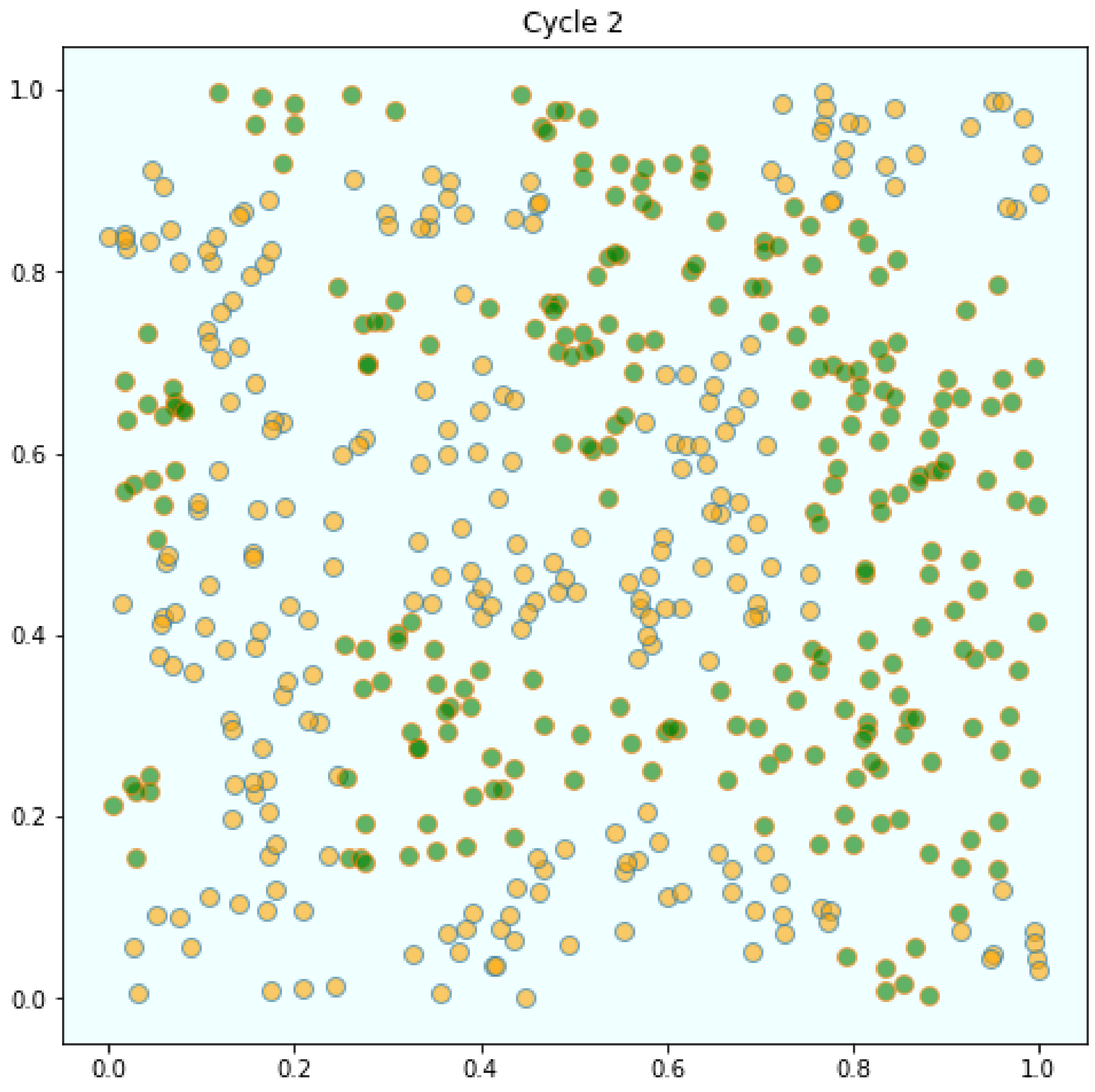
```
Entering loop  1
```



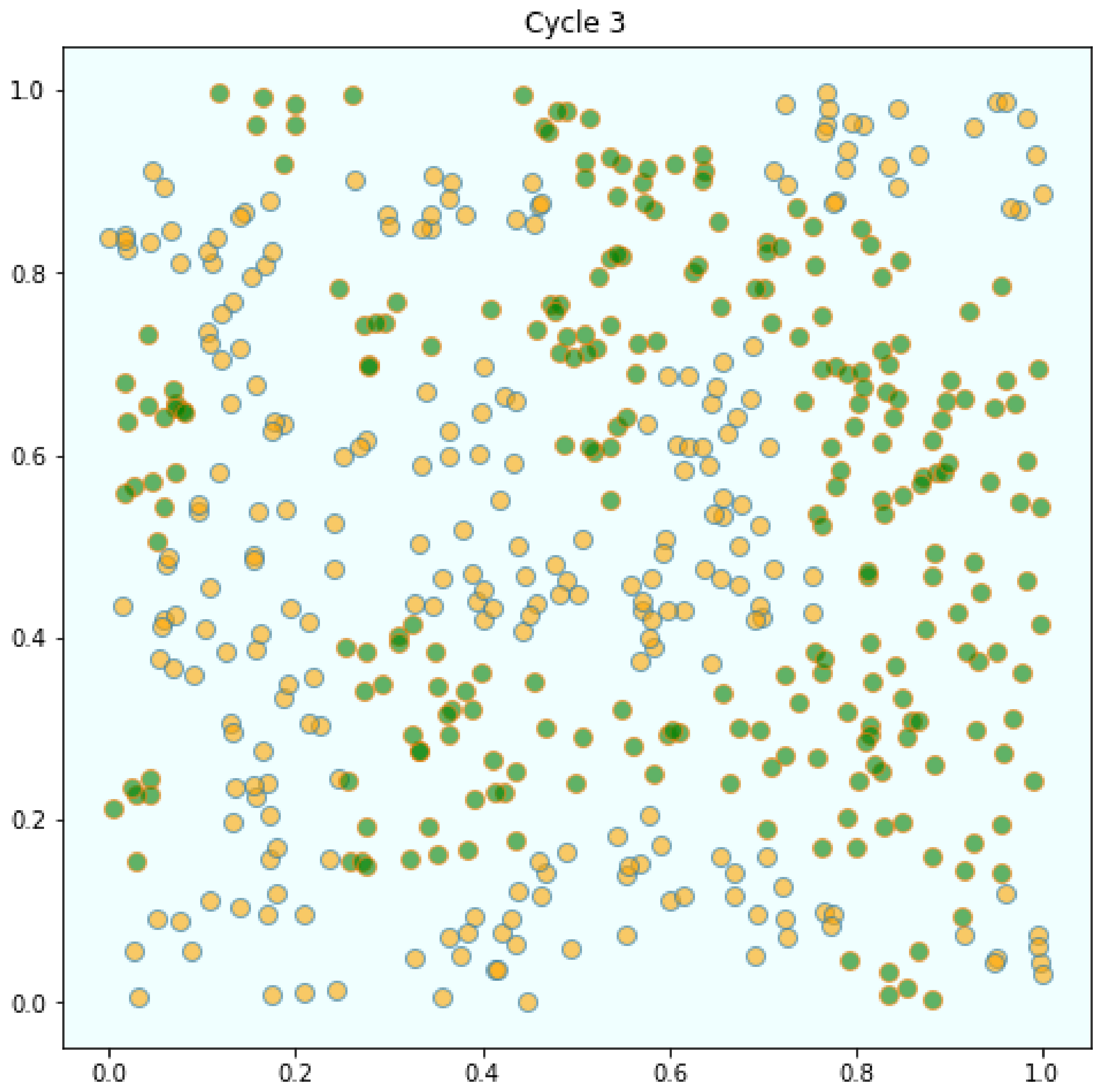
```
Entering loop  2
```



```
Entering loop  3
```



```
Entering loop  4
```



```
Converged, terminating.
```

CHAPTER

FIFTYTHREE

A LAKE MODEL OF EMPLOYMENT AND UNEMPLOYMENT

Contents

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

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

53.1 Overview

This lecture describes what has come to be called a *lake model*.

The lake model is a basic tool for modeling unemployment.

It allows us to analyze

- flows between unemployment and employment.
- how these flows influence steady state employment and unemployment rates.

It is a good model for interpreting monthly labor department reports on gross and net jobs created and jobs destroyed.

The "lakes" in the model are the pools of employed and unemployed.

The "flows" between the lakes are caused by

- firing and hiring
- entry and exit from the labor force

For the first part of this lecture, the parameters governing transitions into and out of unemployment and employment are exogenous.

Later, we'll determine some of these transition rates endogenously using the *McCall search model*.

We'll also use some nifty concepts like ergodicity, which provides a fundamental link between *cross-sectional* and *long run time series* distributions.

These concepts will help us build an equilibrium model of ex-ante homogeneous workers whose different luck generates variations in their ex post experiences.

Let's start with some imports:

```python
%matplotlib inline
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from quantecon import MarkovChain
from scipy.stats import norm
from scipy.optimize import brentq
from quantecon.distributions import BetaBinomial
from numba import jit
```

53.1.1 Prerequisites

Before working through what follows, we recommend you read the *lecture on finite Markov chains*.

You will also need some basic *linear algebra* and probability.

53.2 The Model

The economy is inhabited by a very large number of ex-ante identical workers.

The workers live forever, spending their lives moving between unemployment and employment.

Their rates of transition between employment and unemployment are governed by the following parameters:

- λ, the job finding rate for currently unemployed workers
- α, the dismissal rate for currently employed workers
- b, the entry rate into the labor force
- d, the exit rate from the labor force

The growth rate of the labor force evidently equals $g = b - d$.

53.2.1 Aggregate Variables

We want to derive the dynamics of the following aggregates

- E_t, the total number of employed workers at date t
- U_t, the total number of unemployed workers at t
- N_t, the number of workers in the labor force at t

We also want to know the values of the following objects

- The employment rate $e_t := E_t/N_t$.
- The unemployment rate $u_t := U_t/N_t$.

(Here and below, capital letters represent aggregates and lowercase letters represent rates)

53.2.2 Laws of Motion for Stock Variables

We begin by constructing laws of motion for the aggregate variables E_t, U_t, N_t.

Of the mass of workers E_t who are employed at date t,

- $(1-d)E_t$ will remain in the labor force
- of these, $(1-\alpha)(1-d)E_t$ will remain employed

Of the mass of workers U_t workers who are currently unemployed,

- $(1-d)U_t$ will remain in the labor force
- of these, $(1-d)\lambda U_t$ will become employed

Therefore, the number of workers who will be employed at date $t+1$ will be

$$E_{t+1} = (1-d)(1-\alpha)E_t + (1-d)\lambda U_t$$

A similar analysis implies

$$U_{t+1} = (1-d)\alpha E_t + (1-d)(1-\lambda)U_t + b(E_t + U_t)$$

The value $b(E_t + U_t)$ is the mass of new workers entering the labor force unemployed.

The total stock of workers $N_t = E_t + U_t$ evolves as

$$N_{t+1} = (1+b-d)N_t = (1+g)N_t$$

Letting $X_t := \begin{pmatrix} U_t \\ E_t \end{pmatrix}$, the law of motion for X is

$$X_{t+1} = AX_t \quad \text{where} \quad A := \begin{pmatrix} (1-d)(1-\lambda)+b & (1-d)\alpha + b \\ (1-d)\lambda & (1-d)(1-\alpha) \end{pmatrix}$$

This law tells us how total employment and unemployment evolve over time.

53.2.3 Laws of Motion for Rates

Now let's derive the law of motion for rates.

To get these we can divide both sides of $X_{t+1} = AX_t$ by N_{t+1} to get

$$\begin{pmatrix} U_{t+1}/N_{t+1} \\ E_{t+1}/N_{t+1} \end{pmatrix} = \frac{1}{1+g} A \begin{pmatrix} U_t/N_t \\ E_t/N_t \end{pmatrix}$$

Letting

$$x_t := \begin{pmatrix} u_t \\ e_t \end{pmatrix} = \begin{pmatrix} U_t/N_t \\ E_t/N_t \end{pmatrix}$$

we can also write this as

$$x_{t+1} = \hat{A} x_t \quad \text{where} \quad \hat{A} := \frac{1}{1+g} A$$

You can check that $e_t + u_t = 1$ implies that $e_{t+1} + u_{t+1} = 1$.

This follows from the fact that the columns of $\hat{A}$ sum to 1.

53.3 Implementation

Let's code up these equations.

To do this we're going to use a class that we'll call `LakeModel`.

This class will

1. store the primitives α, λ, b, d
2. compute and store the implied objects $g, A, \hat{A}$
3. provide methods to simulate dynamics of the stocks and rates
4. provide a method to compute the steady state of the rate

Please be careful because the implied objects $g, A, \hat{A}$ will not change if you only change the primitives.

For example, if you would like to update a primitive like $\alpha = 0.03$, you need to create an instance and update it by `lm = LakeModel(α=0.03)`.

In the exercises, we show how to avoid this issue by using getter and setter methods.

```python
class LakeModel:
    """
    Solves the lake model and computes dynamics of unemployment stocks and
    rates.

    Parameters:
    ------------
    λ : scalar
        The job finding rate for currently unemployed workers
    α : scalar
        The dismissal rate for currently employed workers
    b : scalar
        Entry rate into the labor force
    d : scalar
        Exit rate from the labor force

    """
    def __init__(self, λ=0.283, α=0.013, b=0.0124, d=0.00822):
        self.λ, self.α, self.b, self.d = λ, α, b, d

        λ, α, b, d = self.λ, self.α, self.b, self.d
        self.g = b - d
        self.A = np.array([[(1-d) * (1-λ) + b,      (1 - d) * α + b],
                           [        (1-d) * λ,      (1 - d) * (1 - α)]])

        self.A_hat = self.A / (1 + self.g)


    def rate_steady_state(self, tol=1e-6):
        """
        Finds the steady state of the system :math:`x_{t+1} = \hat A x_{t}`

        Returns
        ---------
        xbar : steady state vector of employment and unemployment rates
        """
        x = np.full(2, 0.5)
```

(continues on next page)

(continued from previous page)

```
        error = tol + 1
        while error > tol:
            new_x = self.A_hat @ x
            error = np.max(np.abs(new_x - x))
            x = new_x
        return x

    def simulate_stock_path(self, X0, T):
        """
        Simulates the sequence of Employment and Unemployment stocks

        Parameters
        ------------
        X0 : array
            Contains initial values (E0, U0)
        T : int
            Number of periods to simulate

        Returns
        ---------
        X : iterator
            Contains sequence of employment and unemployment stocks
        """

        X = np.atleast_1d(X0)  # Recast as array just in case
        for t in range(T):
            yield X
            X = self.A @ X

    def simulate_rate_path(self, x0, T):
        """
        Simulates the sequence of employment and unemployment rates

        Parameters
        ------------
        x0 : array
            Contains initial values (e0,u0)
        T : int
            Number of periods to simulate

        Returns
        ---------
        x : iterator
            Contains sequence of employment and unemployment rates

        """
        x = np.atleast_1d(x0)  # Recast as array just in case
        for t in range(T):
            yield x
            x = self.A_hat @ x
```

As explained, if we create an instance and update it by `lm = LakeModel(α=0.03)`, derived objects like A will also change.

```
lm = LakeModel()
lm.α
```

```
0.013
```

```
lm.A
```

```
array([[0.72350626, 0.02529314],
       [0.28067374, 0.97888686]])
```

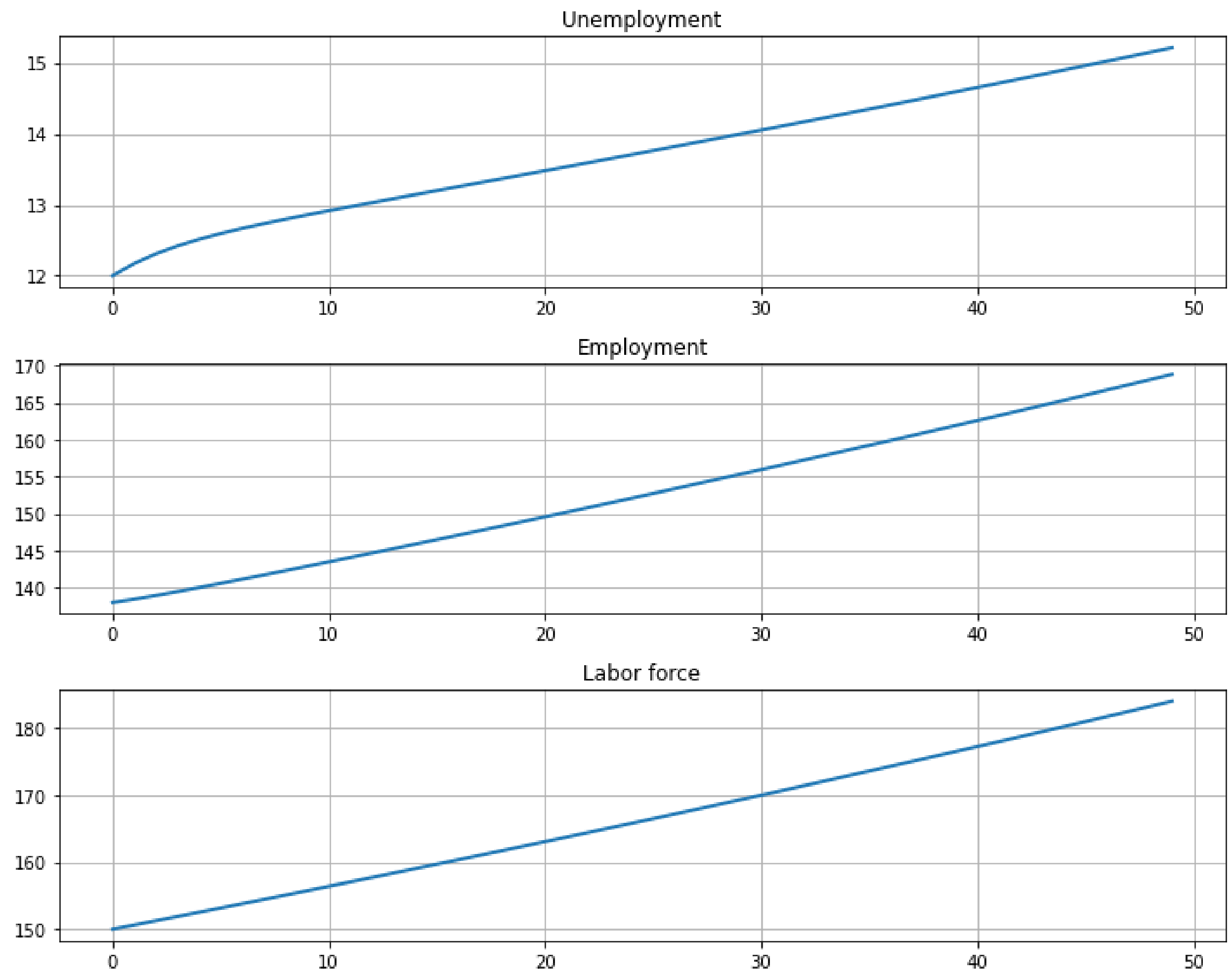
```
lm = LakeModel(α = 0.03)
lm.A
```

```
array([[0.72350626, 0.0421534 ],
       [0.28067374, 0.9620266 ]])
```

53.3.1 Aggregate Dynamics

Let's run a simulation under the default parameters (see above) starting from $X_0 = (12, 138)$

```
lm = LakeModel()
N_0 = 150      # Population
e_0 = 0.92     # Initial employment rate
u_0 = 1 - e_0  # Initial unemployment rate
T = 50         # Simulation length

U_0 = u_0 * N_0
E_0 = e_0 * N_0

fig, axes = plt.subplots(3, 1, figsize=(10, 8))
X_0 = (U_0, E_0)
X_path = np.vstack(tuple(lm.simulate_stock_path(X_0, T)))

axes[0].plot(X_path[:, 0], lw=2)
axes[0].set_title('Unemployment')

axes[1].plot(X_path[:, 1], lw=2)
axes[1].set_title('Employment')

axes[2].plot(X_path.sum(1), lw=2)
axes[2].set_title('Labor force')

for ax in axes:
    ax.grid()

plt.tight_layout()
plt.show()
```



The aggregates E_t and U_t don't converge because their sum $E_t + U_t$ grows at rate g.

On the other hand, the vector of employment and unemployment rates x_t can be in a steady state $\bar{x}$ if there exists an $\bar{x}$ such that

- $\bar{x} = \hat{A}\bar{x}$
- the components satisfy $\bar{e} + \bar{u} = 1$

This equation tells us that a steady state level $\bar{x}$ is an eigenvector of $\hat{A}$ associated with a unit eigenvalue.

We also have $x_t \to \bar{x}$ as $t \to \infty$ provided that the remaining eigenvalue of $\hat{A}$ has modulus less than 1.

This is the case for our default parameters:

```
lm = LakeModel()
e, f = np.linalg.eigvals(lm.A_hat)
abs(e), abs(f)
```

```
(0.6953067378358462, 1.0)
```

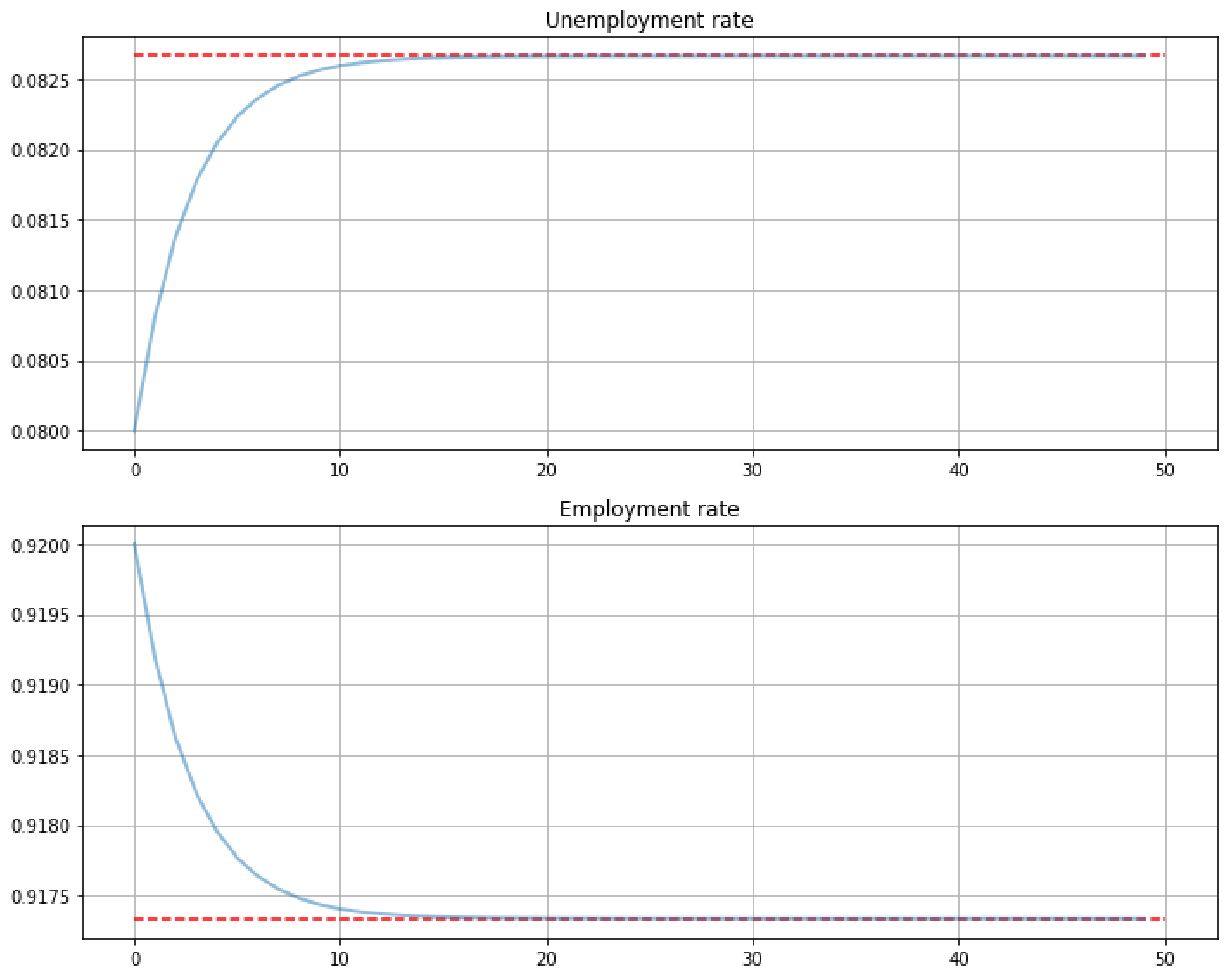
Let's look at the convergence of the unemployment and employment rate to steady state levels (dashed red line)

```
lm = LakeModel()
e_0 = 0.92      # Initial employment rate
```

(continues on next page)

(continued from previous page)

```
u_0 = 1 - e_0   # Initial unemployment rate
T = 50          # Simulation length

xbar = lm.rate_steady_state()

fig, axes = plt.subplots(2, 1, figsize=(10, 8))
x_0 = (u_0, e_0)
x_path = np.vstack(tuple(lm.simulate_rate_path(x_0, T)))

titles = ['Unemployment rate', 'Employment rate']

for i, title in enumerate(titles):
    axes[i].plot(x_path[:, i], lw=2, alpha=0.5)
    axes[i].hlines(xbar[i], 0, T, 'r', '--')
    axes[i].set_title(title)
    axes[i].grid()

plt.tight_layout()
plt.show()
```



53.4 Dynamics of an Individual Worker

An individual worker's employment dynamics are governed by a *finite state Markov process*.

The worker can be in one of two states:

- $s_t = 0$ means unemployed
- $s_t = 1$ means employed

Let's start off under the assumption that $b = d = 0$.

The associated transition matrix is then

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

Let ψ_t denote the *marginal distribution* over employment/unemployment states for the worker at time t.

As usual, we regard it as a row vector.

We know *from an earlier discussion* that ψ_t follows the law of motion

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

We also know from the *lecture on finite Markov chains* that if $\alpha \in (0, 1)$ and $\lambda \in (0, 1)$, then P has a unique stationary distribution, denoted here by ψ^*.

The unique stationary distribution satisfies

$$\psi^*[0] = \frac{\alpha}{\alpha + \lambda}$$

Not surprisingly, probability mass on the unemployment state increases with the dismissal rate and falls with the job finding rate.

53.4.1 Ergodicity

Let's look at a typical lifetime of employment-unemployment spells.

We want to compute the average amounts of time an infinitely lived worker would spend employed and unemployed.

Let

$$\bar{s}_{u,T} := \frac{1}{T} \sum_{t=1}^{T} \mathbb{1}\{s_t = 0\}$$

and

$$\bar{s}_{e,T} := \frac{1}{T} \sum_{t=1}^{T} \mathbb{1}\{s_t = 1\}$$

(As usual, $\mathbb{1}\{Q\} = 1$ if statement Q is true and 0 otherwise)

These are the fraction of time a worker spends unemployed and employed, respectively, up until period T.

If $\alpha \in (0, 1)$ and $\lambda \in (0, 1)$, then P is *ergodic*, and hence we have

$$\lim_{T \to \infty} \bar{s}_{u,T} = \psi^*[0] \quad \text{and} \quad \lim_{T \to \infty} \bar{s}_{e,T} = \psi^*[1]$$

with probability one.

Inspection tells us that P is exactly the transpose of $\hat{A}$ under the assumption $b = d = 0$.

Thus, the percentages of time that an infinitely lived worker spends employed and unemployed equal the fractions of workers employed and unemployed in the steady state distribution.

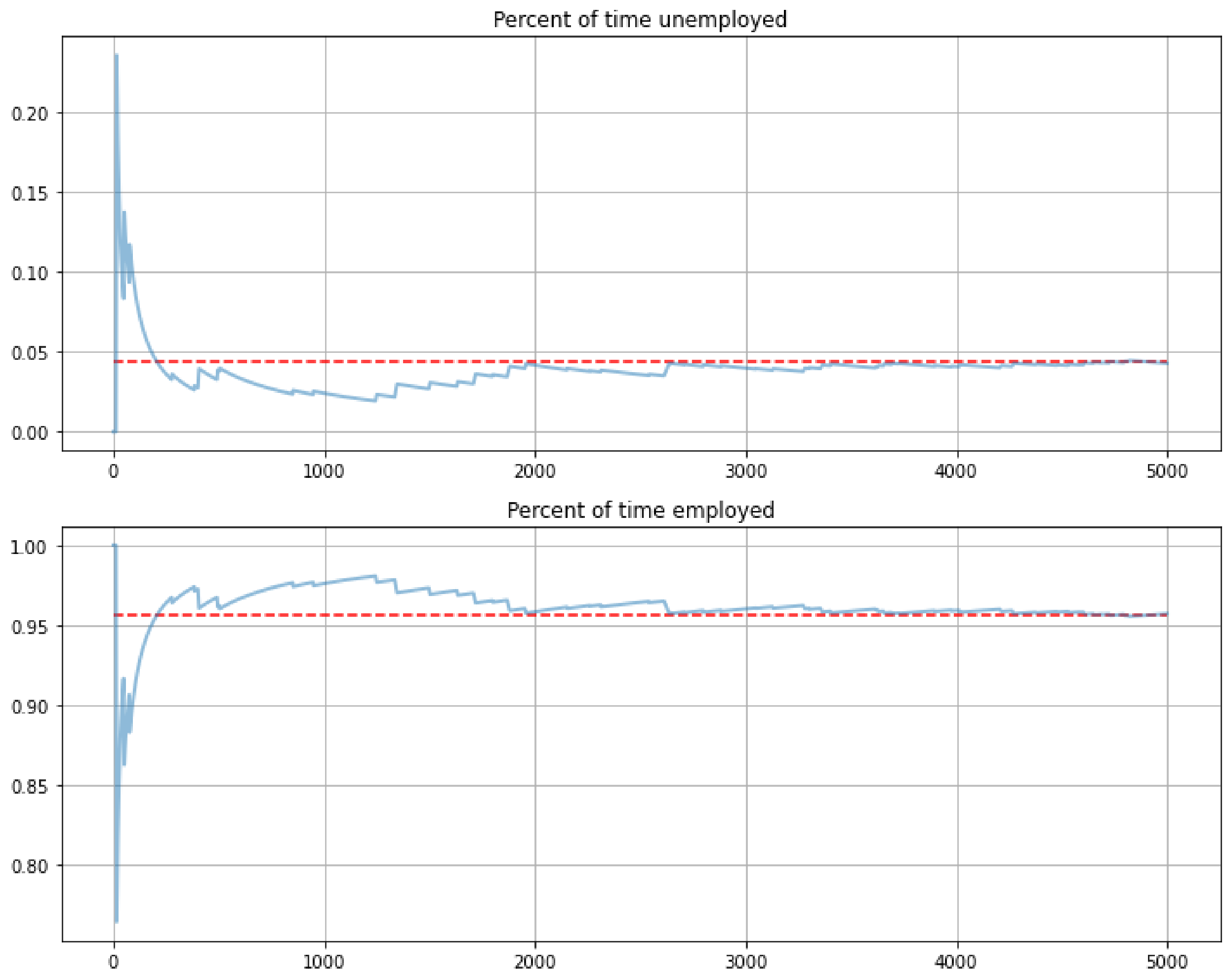
53.4.2 Convergence Rate

How long does it take for time series sample averages to converge to cross-sectional averages?

We can use QuantEcon.py's MarkovChain class to investigate this.

Let's plot the path of the sample averages over 5,000 periods

```
lm = LakeModel(d=0, b=0)
T = 5000  # Simulation length

α, λ = lm.α, lm.λ

P = [[1 - λ,        λ],
    [    α,    1 - α]]

mc = MarkovChain(P)

xbar = lm.rate_steady_state()

fig, axes = plt.subplots(2, 1, figsize=(10, 8))
s_path = mc.simulate(T, init=1)
s_bar_e = s_path.cumsum() / range(1, T+1)
s_bar_u = 1 - s_bar_e

to_plot = [s_bar_u, s_bar_e]
titles = ['Percent of time unemployed', 'Percent of time employed']

for i, plot in enumerate(to_plot):
    axes[i].plot(plot, lw=2, alpha=0.5)
    axes[i].hlines(xbar[i], 0, T, 'r', '--')
    axes[i].set_title(titles[i])
    axes[i].grid()

plt.tight_layout()
plt.show()
```



The stationary probabilities are given by the dashed red line.

In this case it takes much of the sample for these two objects to converge.

This is largely due to the high persistence in the Markov chain.

53.5 Endogenous Job Finding Rate

We now make the hiring rate endogenous.

The transition rate from unemployment to employment will be determined by the McCall search model [McC70].

All details relevant to the following discussion can be found in *our treatment* of that model.

53.5.1 Reservation Wage

The most important thing to remember about the model is that optimal decisions are characterized by a reservation wage $\bar{w}$

- If the wage offer w in hand is greater than or equal to $\bar{w}$, then the worker accepts.
- Otherwise, the worker rejects.

As we saw in *our discussion of the model*, the reservation wage depends on the wage offer distribution and the parameters

- α, the separation rate
- β, the discount factor
- γ, the offer arrival rate
- c, unemployment compensation

53.5.2 Linking the McCall Search Model to the Lake Model

Suppose that all workers inside a lake model behave according to the McCall search model.

The exogenous probability of leaving employment remains α.

But their optimal decision rules determine the probability λ of leaving unemployment.

This is now

$$\lambda = \gamma \mathbb{P}\{w_t \geq \bar{w}\} = \gamma \sum_{w' \geq \bar{w}} p(w') \tag{1}$$

53.5.3 Fiscal Policy

We can use the McCall search version of the Lake Model to find an optimal level of unemployment insurance.

We assume that the government sets unemployment compensation c.

The government imposes a lump-sum tax τ sufficient to finance total unemployment payments.

To attain a balanced budget at a steady state, taxes, the steady state unemployment rate u, and the unemployment compensation rate must satisfy

$$\tau = uc$$

The lump-sum tax applies to everyone, including unemployed workers.

Thus, the post-tax income of an employed worker with wage w is $w - \tau$.

The post-tax income of an unemployed worker is $c - \tau$.

For each specification (c, τ) of government policy, we can solve for the worker's optimal reservation wage.

This determines λ via (1) evaluated at post tax wages, which in turn determines a steady state unemployment rate $u(c, \tau)$.

For a given level of unemployment benefit c, we can solve for a tax that balances the budget in the steady state

$$\tau = u(c, \tau)c$$

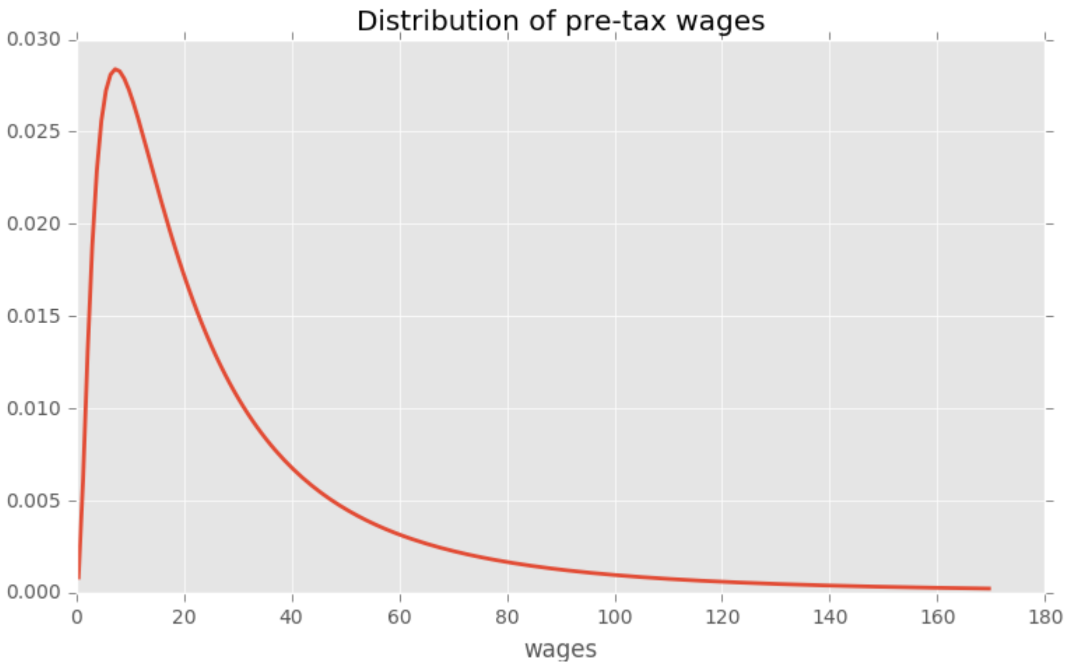
To evaluate alternative government tax-unemployment compensation pairs, we require a welfare criterion.

We use a steady state welfare criterion

$$W := e \, \mathbb{E}[V \mid \text{employed}] + u \, U$$

where the notation V and U is as defined in the *McCall search model lecture*.

The wage offer distribution will be a discretized version of the lognormal distribution $LN(\log(20), 1)$, as shown in the next figure



We take a period to be a month.

We set b and d to match monthly birth and death rates, respectively, in the U.S. population

- $b = 0.0124$
- $d = 0.00822$

Following [DFH06], we set α, the hazard rate of leaving employment, to

- $\alpha = 0.013$

53.5.4 Fiscal Policy Code

We will make use of techniques from the *McCall model lecture*

The first piece of code implements value function iteration

```
# A default utility function

@jit
def u(c, σ):
    if c > 0:
        return (c**(1 - σ) - 1) / (1 - σ)
    else:
```

(continues on next page)

(continued from previous page)

```
        return -10e6


class McCallModel:
    """
    Stores the parameters and functions associated with a given model.
    """

    def __init__(self,
                 α=0.2,       # Job separation rate
                 β=0.98,      # Discount rate
                 γ=0.7,       # Job offer rate
                 c=6.0,       # Unemployment compensation
                 σ=2.0,       # Utility parameter
                 w_vec=None,  # Possible wage values
                 p_vec=None): # Probabilities over w_vec

        self.α, self.β, self.γ, self.c = α, β, γ, c
        self.σ = σ

        # Add a default wage vector and probabilities over the vector using
        # the beta-binomial distribution
        if w_vec is None:
            n = 60  # Number of possible outcomes for wage
            # Wages between 10 and 20
            self.w_vec = np.linspace(10, 20, n)
            a, b = 600, 400  # Shape parameters
            dist = BetaBinomial(n-1, a, b)
            self.p_vec = dist.pdf()
        else:
            self.w_vec = w_vec
            self.p_vec = p_vec

@jit
def _update_bellman(α, β, γ, c, σ, w_vec, p_vec, V, V_new, U):
    """
    A jitted function to update the Bellman equations.  Note that V_new is
    modified in place (i.e, modified by this function).  The new value of U
    is returned.

    """
    for w_idx, w in enumerate(w_vec):
        # w_idx indexes the vector of possible wages
        V_new[w_idx] = u(w, σ) + β * ((1 - α) * V[w_idx] + α * U)

    U_new = u(c, σ) + β * (1 - γ) * U + \
                    β * γ * np.sum(np.maximum(U, V) * p_vec)

    return U_new


def solve_mccall_model(mcm, tol=1e-5, max_iter=2000):
    """
    Iterates to convergence on the Bellman equations

    Parameters
    ----------
```

(continues on next page)

(continued from previous page)

```python
    mcm : an instance of McCallModel
    tol : float
        error tolerance
    max_iter : int
        the maximum number of iterations
    """

    V = np.ones(len(mcm.w_vec))  # Initial guess of V
    V_new = np.empty_like(V)     # To store updates to V
    U = 1                        # Initial guess of U
    i = 0
    error = tol + 1

    while error > tol and i < max_iter:
        U_new = _update_bellman(mcm.α, mcm.β, mcm.γ,
                mcm.c, mcm.σ, mcm.w_vec, mcm.p_vec, V, V_new, U)
        error_1 = np.max(np.abs(V_new - V))
        error_2 = np.abs(U_new - U)
        error = max(error_1, error_2)
        V[:] = V_new
        U = U_new
        i += 1

    return V, U
```

The second piece of code is used to complete the reservation wage:

```python
def compute_reservation_wage(mcm, return_values=False):
    """
    Computes the reservation wage of an instance of the McCall model
    by finding the smallest w such that V(w) > U.

    If V(w) > U for all w, then the reservation wage w_bar is set to
    the lowest wage in mcm.w_vec.

    If v(w) < U for all w, then w_bar is set to np.inf.

    Parameters
    ----------
    mcm : an instance of McCallModel
    return_values : bool (optional, default=False)
        Return the value functions as well

    Returns
    -------
    w_bar : scalar
        The reservation wage

    """

    V, U = solve_mccall_model(mcm)
    w_idx = np.searchsorted(V - U, 0)

    if w_idx == len(V):
        w_bar = np.inf
    else:
        w_bar = mcm.w_vec[w_idx]
```

(continues on next page)

(continued from previous page)

```python
    if return_values == False:
        return w_bar
    else:
        return w_bar, V, U
```

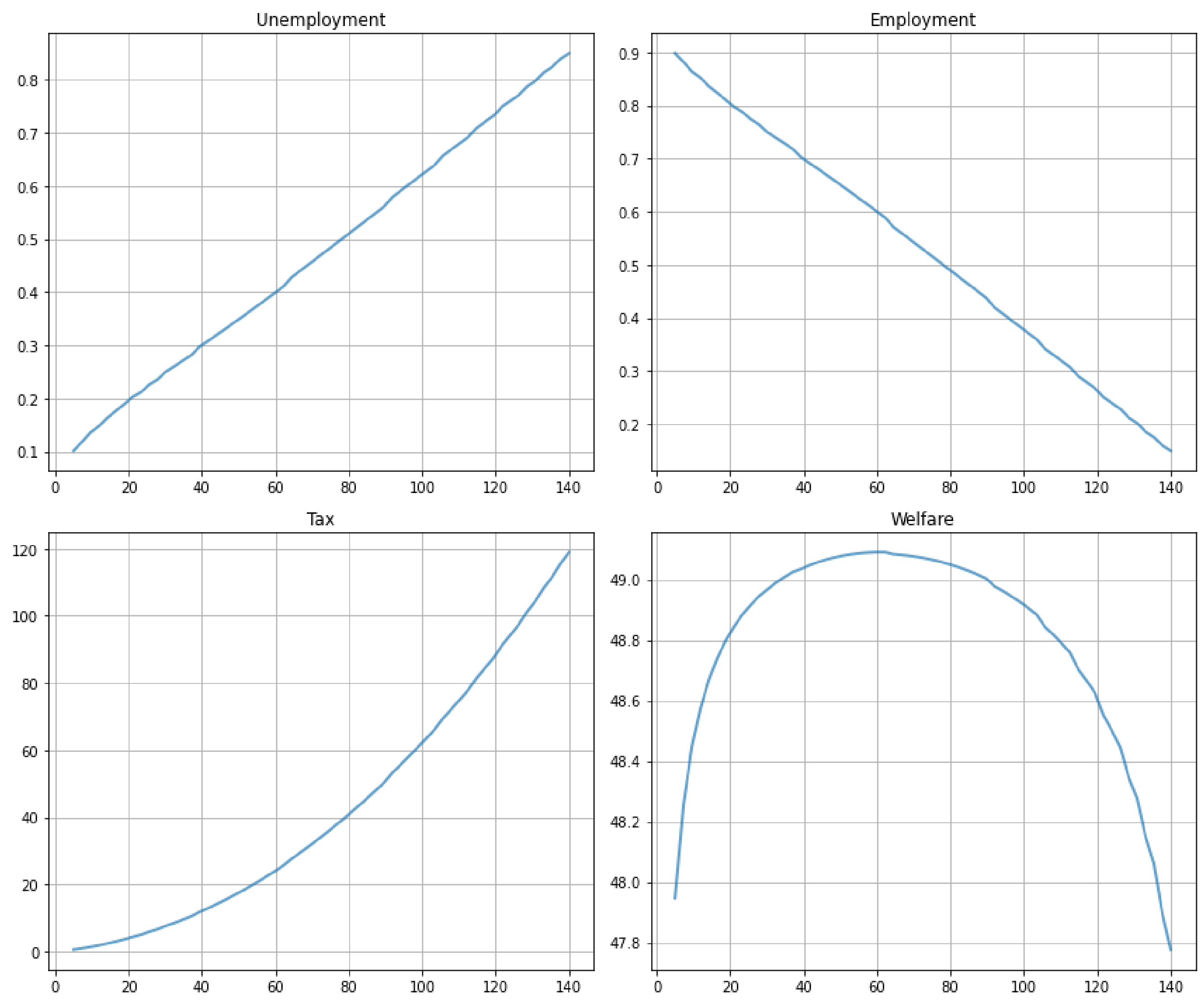
Now let's compute and plot welfare, employment, unemployment, and tax revenue as a function of the unemployment compensation rate

```python
# Some global variables that will stay constant
α = 0.013
α_q = (1-(1-α)**3)    # Quarterly (α is monthly)
b = 0.0124
d = 0.00822
β = 0.98
γ = 1.0
σ = 2.0

# The default wage distribution --- a discretized lognormal
log_wage_mean, wage_grid_size, max_wage = 20, 200, 170
logw_dist = norm(np.log(log_wage_mean), 1)
w_vec = np.linspace(1e-8, max_wage, wage_grid_size + 1)
cdf = logw_dist.cdf(np.log(w_vec))
pdf = cdf[1:] - cdf[:-1]
p_vec = pdf / pdf.sum()
w_vec = (w_vec[1:] + w_vec[:-1]) / 2


def compute_optimal_quantities(c, τ):
    """
    Compute the reservation wage, job finding rate and value functions
    of the workers given c and τ.

    """

    mcm = McCallModel(α=α_q,
                      β=β,
                      γ=γ,
                      c=c-τ,          # Post tax compensation
                      σ=σ,
                      w_vec=w_vec-τ,  # Post tax wages
                      p_vec=p_vec)

    w_bar, V, U = compute_reservation_wage(mcm, return_values=True)
    λ = γ * np.sum(p_vec[w_vec - τ > w_bar])
    return w_bar, λ, V, U

def compute_steady_state_quantities(c, τ):
    """
    Compute the steady state unemployment rate given c and τ using optimal
    quantities from the McCall model and computing corresponding steady
    state quantities

    """
    w_bar, λ, V, U = compute_optimal_quantities(c, τ)
```

(continues on next page)

(continued from previous page)

```
    # Compute steady state employment and unemployment rates
    lm = LakeModel(α=α_q, λ=λ, b=b, d=d)
    x = lm.rate_steady_state()
    u, e = x

    # Compute steady state welfare
    w = np.sum(V * p_vec * (w_vec - τ > w_bar)) / np.sum(p_vec * (w_vec -
            τ > w_bar))
    welfare = e * w + u * U

    return e, u, welfare


def find_balanced_budget_tax(c):
    """
    Find the tax level that will induce a balanced budget.

    """
    def steady_state_budget(t):
        e, u, w = compute_steady_state_quantities(c, t)
        return t - u * c

    τ = brentq(steady_state_budget, 0.0, 0.9 * c)
    return τ


# Levels of unemployment insurance we wish to study
c_vec = np.linspace(5, 140, 60)

tax_vec = []
unempl_vec = []
empl_vec = []
welfare_vec = []

for c in c_vec:
    t = find_balanced_budget_tax(c)
    e_rate, u_rate, welfare = compute_steady_state_quantities(c, t)
    tax_vec.append(t)
    unempl_vec.append(u_rate)
    empl_vec.append(e_rate)
    welfare_vec.append(welfare)

fig, axes = plt.subplots(2, 2, figsize=(12, 10))

plots = [unempl_vec, empl_vec, tax_vec, welfare_vec]
titles = ['Unemployment', 'Employment', 'Tax', 'Welfare']

for ax, plot, title in zip(axes.flatten(), plots, titles):
    ax.plot(c_vec, plot, lw=2, alpha=0.7)
    ax.set_title(title)
    ax.grid()

plt.tight_layout()
plt.show()
```



Welfare first increases and then decreases as unemployment benefits rise.

The level that maximizes steady state welfare is approximately 62.

53.6 Exercises

53.6.1 Exercise 1

In the Lake Model, there is derived data such as A which depends on primitives like α and λ.

So, when a user alters these primitives, we need the derived data to update automatically.

(For example, if a user changes the value of b for a given instance of the class, we would like $g = b - d$ to update automatically)

In the code above, we took care of this issue by creating new instances every time we wanted to change parameters.

That way the derived data is always matched to current parameter values.

However, we can use descriptors instead, so that derived data is updated whenever parameters are changed.

This is safer and means we don't need to create a fresh instance for every new parameterization.

(On the other hand, the code becomes denser, which is why we don't always use the descriptor approach in our lectures.)

In this exercise, your task is to arrange the `LakeModel` class by using descriptors and decorators such as `@property`.

(If you need to refresh your understanding of how these work, consult this lecture.)

53.6.2 Exercise 2

Consider an economy with an initial stock of workers $N_0 = 100$ at the steady state level of employment in the baseline parameterization

- $\alpha = 0.013$
- $\lambda = 0.283$
- $b = 0.0124$
- $d = 0.00822$

(The values for α and λ follow [DFH06])

Suppose that in response to new legislation the hiring rate reduces to $\lambda = 0.2$.

Plot the transition dynamics of the unemployment and employment stocks for 50 periods.

Plot the transition dynamics for the rates.

How long does the economy take to converge to its new steady state?

What is the new steady state level of employment?

Note: it may be easier to use the class created in exercise 1 to help with changing variables.

53.6.3 Exercise 3

Consider an economy with an initial stock of workers $N_0 = 100$ at the steady state level of employment in the baseline parameterization.

Suppose that for 20 periods the birth rate was temporarily high ($b = 0.025$) and then returned to its original level.

Plot the transition dynamics of the unemployment and employment stocks for 50 periods.

Plot the transition dynamics for the rates.

How long does the economy take to return to its original steady state?

53.7 Solutions

53.7.1 Exercise 1

```python
class LakeModelModified:
    """
    Solves the lake model and computes dynamics of unemployment stocks and
    rates.

    Parameters:
    ------------
    λ : scalar
```

(continues on next page)

(continued from previous page)

```
        The job finding rate for currently unemployed workers
    α : scalar
        The dismissal rate for currently employed workers
    b : scalar
        Entry rate into the labor force
    d : scalar
        Exit rate from the labor force

    """
    def __init__(self, λ=0.283, α=0.013, b=0.0124, d=0.00822):
        self._λ, self._α, self._b, self._d = λ, α, b, d
        self.compute_derived_values()

    def compute_derived_values(self):
        # Unpack names to simplify expression
        λ, α, b, d = self._λ, self._α, self._b, self._d

        self._g = b - d
        self._A = np.array([[(1-d) * (1-λ) + b,      (1 - d) * α + b],
                            [        (1-d) * λ,   (1 - d) * (1 - α)]])

        self._A_hat = self._A / (1 + self._g)

    @property
    def g(self):
        return self._g

    @property
    def A(self):
        return self._A

    @property
    def A_hat(self):
        return self._A_hat

    @property
    def λ(self):
        return self._λ

    @λ.setter
    def λ(self, new_value):
        self._λ = new_value
        self.compute_derived_values()

    @property
    def α(self):
        return self._α

    @α.setter
    def α(self, new_value):
        self._α = new_value
        self.compute_derived_values()

    @property
    def b(self):
        return self._b
```

(continues on next page)

(continued from previous page)

```
    @b.setter
    def b(self, new_value):
        self._b = new_value
        self.compute_derived_values()

    @property
    def d(self):
        return self._d

    @d.setter
    def d(self, new_value):
        self._d = new_value
        self.compute_derived_values()


    def rate_steady_state(self, tol=1e-6):
        """
        Finds the steady state of the system :math:`x_{t+1} = \hat A x_{t}`

        Returns
        --------
        xbar : steady state vector of employment and unemployment rates
        """
        x = np.full(2, 0.5)
        error = tol + 1
        while error > tol:
            new_x = self.A_hat @ x
            error = np.max(np.abs(new_x - x))
            x = new_x
        return x

    def simulate_stock_path(self, X0, T):
        """
        Simulates the sequence of Employment and Unemployment stocks

        Parameters
        ------------
        X0 : array
            Contains initial values (E0, U0)
        T : int
            Number of periods to simulate

        Returns
        ---------
        X : iterator
            Contains sequence of employment and unemployment stocks
        """

        X = np.atleast_1d(X0)  # Recast as array just in case
        for t in range(T):
            yield X
            X = self.A @ X

    def simulate_rate_path(self, x0, T):
        """
        Simulates the sequence of employment and unemployment rates
```

(continues on next page)

(continued from previous page)

```
        Parameters
        ------------
        x0 : array
            Contains initial values (e0,u0)
        T : int
            Number of periods to simulate

        Returns
        ---------
        x : iterator
            Contains sequence of employment and unemployment rates

        """
        x = np.atleast_1d(x0)  # Recast as array just in case
        for t in range(T):
            yield x
            x = self.A_hat @ x
```

53.7.2 Exercise 2

We begin by constructing the class containing the default parameters and assigning the steady state values to `x0`

```
lm = LakeModelModified()
x0 = lm.rate_steady_state()
print(f"Initial Steady State: {x0}")
```

```
Initial Steady State: [0.08266806 0.91733194]
```

Initialize the simulation values

```
N0 = 100
T = 50
```

New legislation changes λ to 0.2

```
lm.λ = 0.2

xbar = lm.rate_steady_state()  # new steady state
X_path = np.vstack(tuple(lm.simulate_stock_path(x0 * N0, T)))
x_path = np.vstack(tuple(lm.simulate_rate_path(x0, T)))
print(f"New Steady State: {xbar}")
```

```
New Steady State: [0.11309573 0.88690427]
```

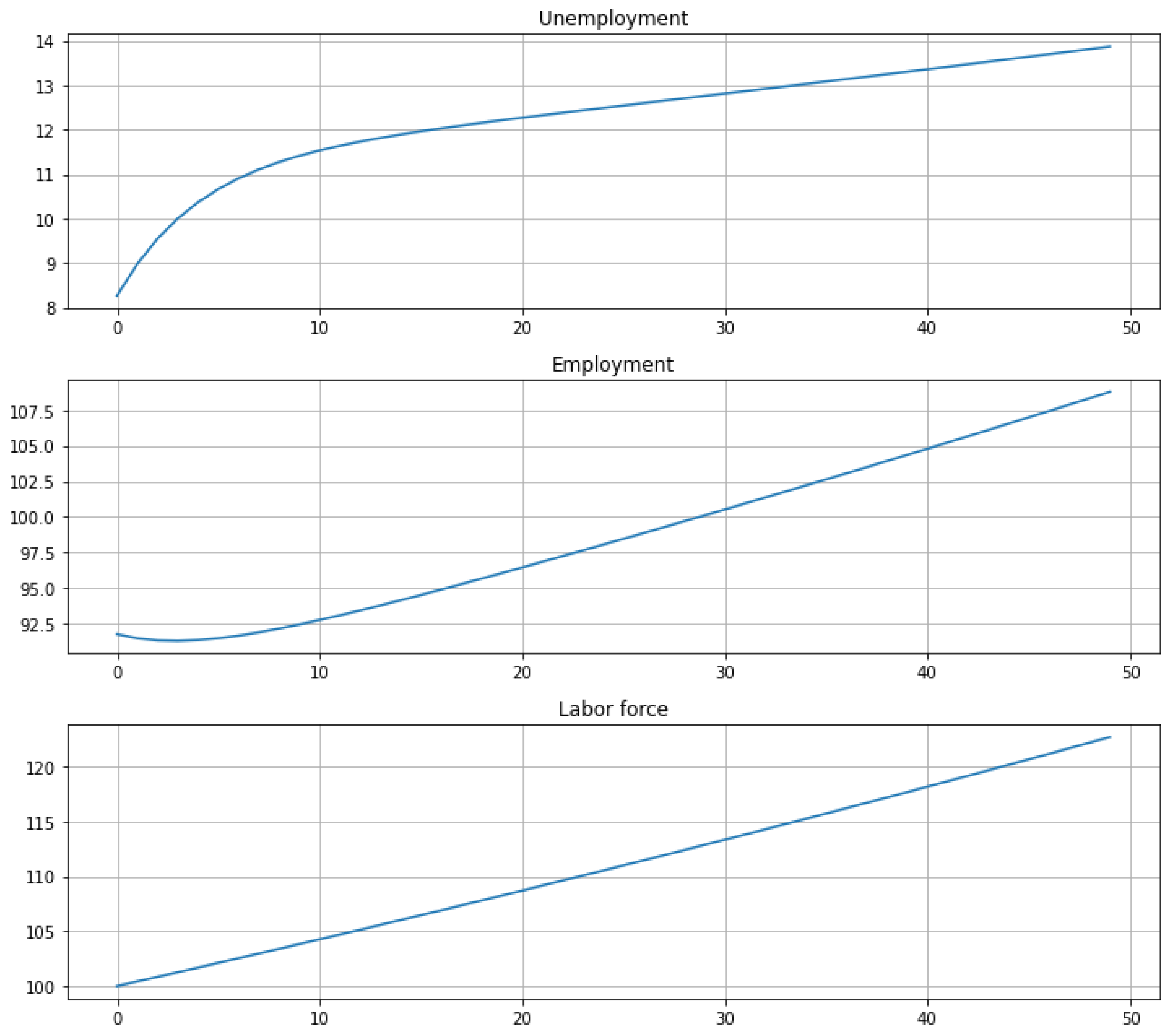
Now plot stocks

```
fig, axes = plt.subplots(3, 1, figsize=[10, 9])

axes[0].plot(X_path[:, 0])
axes[0].set_title('Unemployment')

axes[1].plot(X_path[:, 1])
axes[1].set_title('Employment')
```

(continues on next page)

(continued from previous page)

```python
axes[2].plot(X_path.sum(1))
axes[2].set_title('Labor force')

for ax in axes:
    ax.grid()

plt.tight_layout()
plt.show()
```



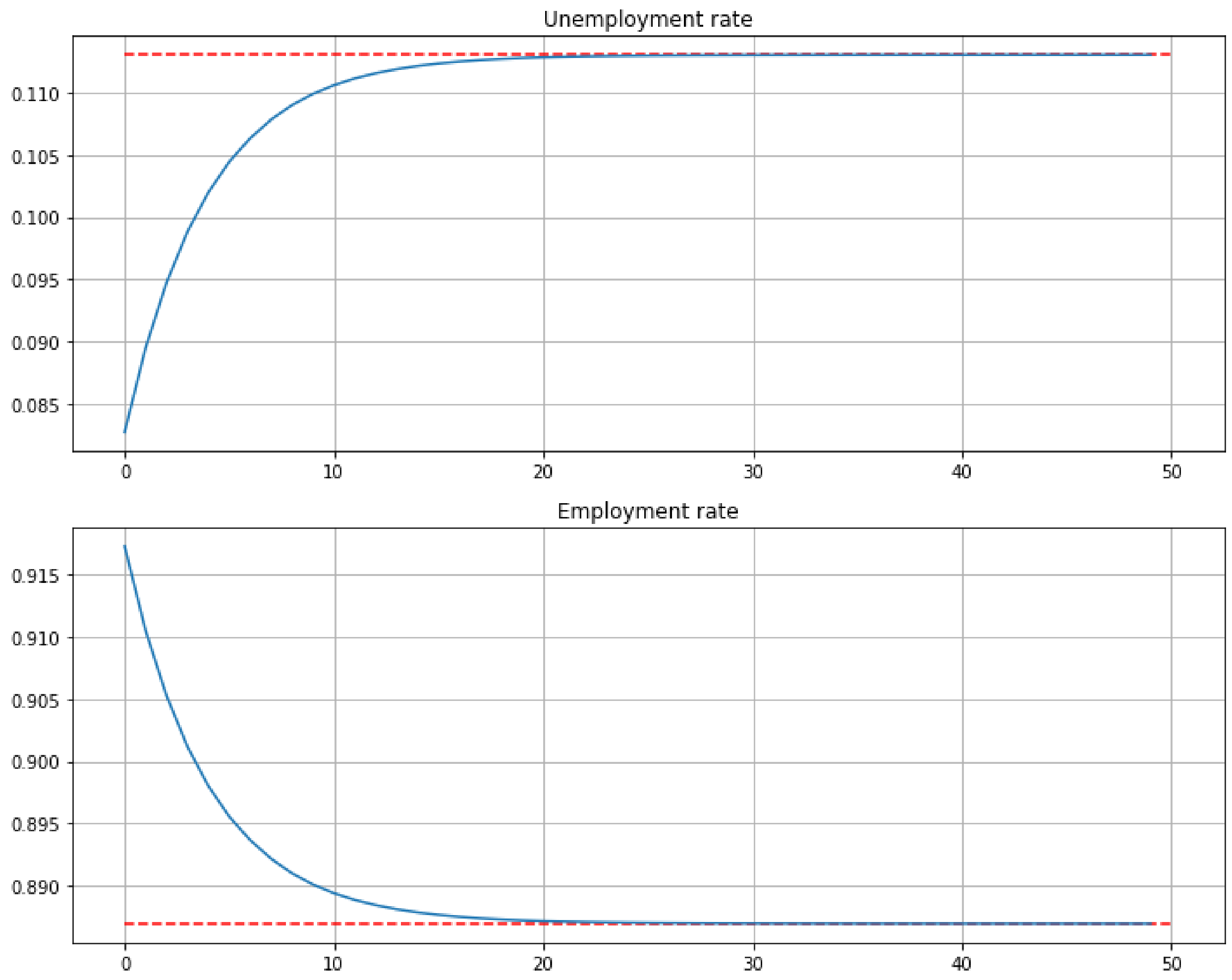
And how the rates evolve

```python
fig, axes = plt.subplots(2, 1, figsize=(10, 8))

titles = ['Unemployment rate', 'Employment rate']

for i, title in enumerate(titles):
    axes[i].plot(x_path[:, i])
    axes[i].hlines(xbar[i], 0, T, 'r', '--')
```

(continues on next page)

(continued from previous page)

```
    axes[i].set_title(title)
    axes[i].grid()

plt.tight_layout()
plt.show()
```



We see that it takes 20 periods for the economy to converge to its new steady state levels.

53.7.3 Exercise 3

This next exercise has the economy experiencing a boom in entrances to the labor market and then later returning to the original levels.

For 20 periods the economy has a new entry rate into the labor market.

Let’s start off at the baseline parameterization and record the steady state

```
lm = LakeModelModified()
x0 = lm.rate_steady_state()
```

Here are the other parameters:

```
b_hat = 0.025
T_hat = 20
```

Let's increase b to the new value and simulate for 20 periods

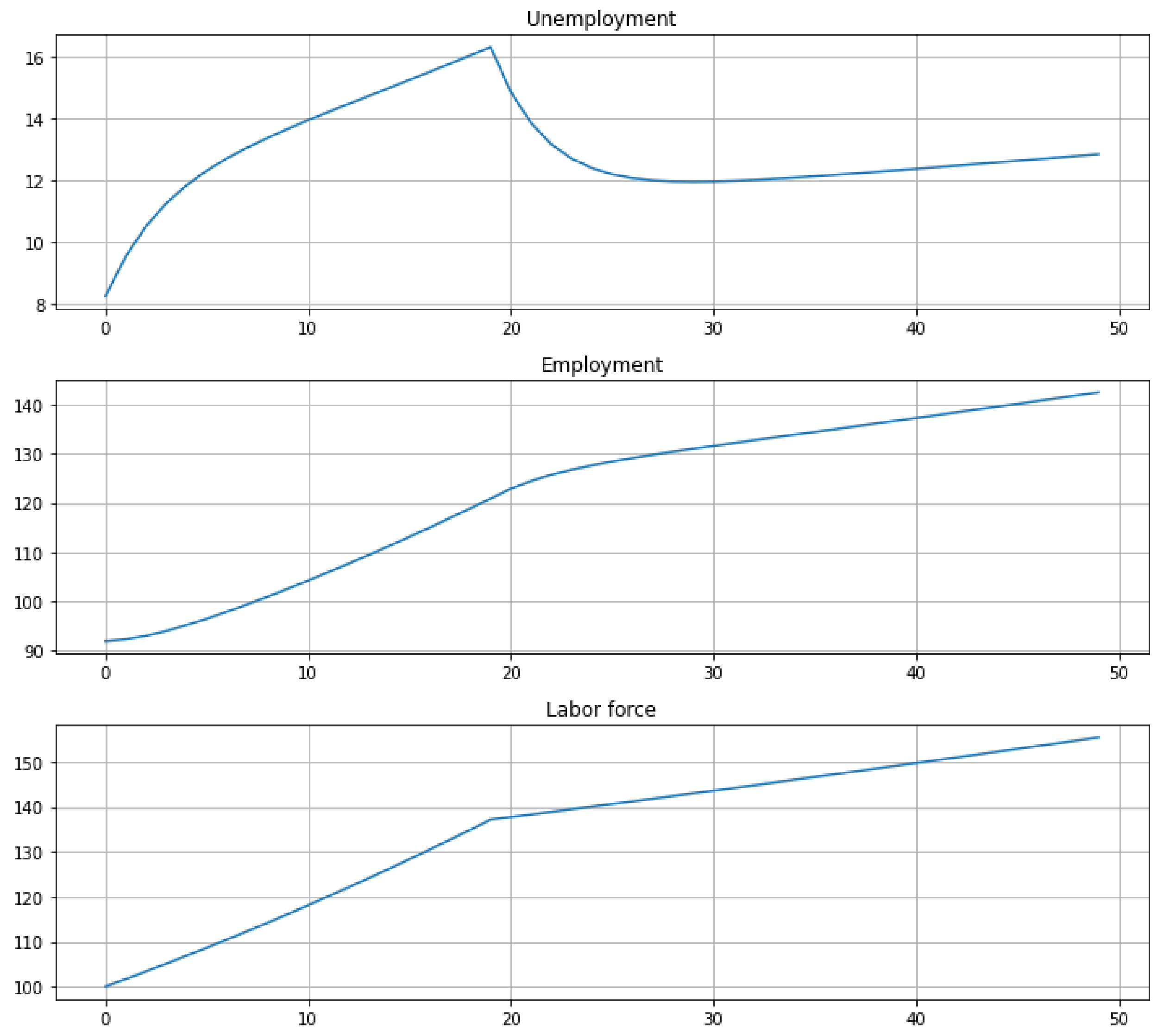
```
lm.b = b_hat
# Simulate stocks
X_path1 = np.vstack(tuple(lm.simulate_stock_path(x0 * N0, T_hat)))
# Simulate rates
x_path1 = np.vstack(tuple(lm.simulate_rate_path(x0, T_hat)))
```

Now we reset b to the original value and then, using the state after 20 periods for the new initial conditions, we simulate for the additional 30 periods

```
lm.b = 0.0124
# Simulate stocks
X_path2 = np.vstack(tuple(lm.simulate_stock_path(X_path1[-1, :2], T-T_hat+1)))
# Simulate rates
x_path2 = np.vstack(tuple(lm.simulate_rate_path(x_path1[-1, :2], T-T_hat+1)))
```

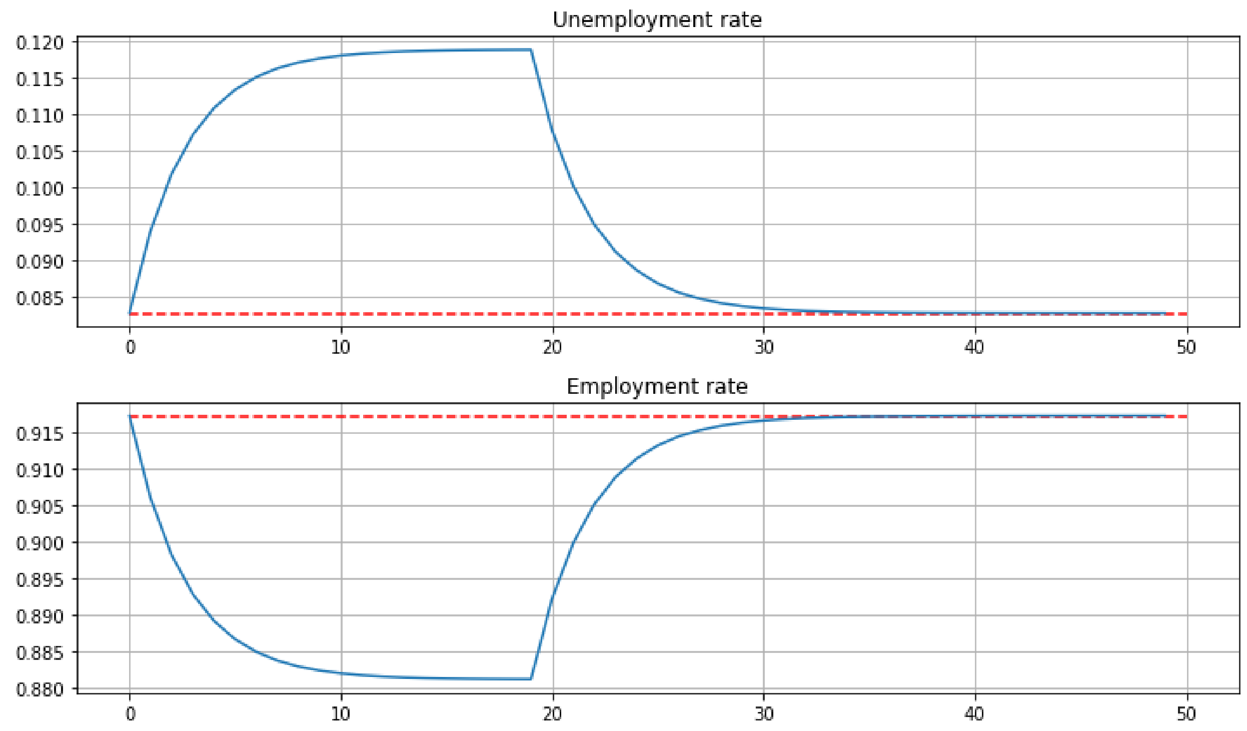
Finally, we combine these two paths and plot

```
# note [1:] to avoid doubling period 20
x_path = np.vstack([x_path1, x_path2[1:]])
X_path = np.vstack([X_path1, X_path2[1:]])

fig, axes = plt.subplots(3, 1, figsize=[10, 9])

axes[0].plot(X_path[:, 0])
axes[0].set_title('Unemployment')

axes[1].plot(X_path[:, 1])
axes[1].set_title('Employment')

axes[2].plot(X_path.sum(1))
axes[2].set_title('Labor force')

for ax in axes:
    ax.grid()

plt.tight_layout()
plt.show()
```



And the rates

```
fig, axes = plt.subplots(2, 1, figsize=[10, 6])

titles = ['Unemployment rate', 'Employment rate']

for i, title in enumerate(titles):
    axes[i].plot(x_path[:, i])
    axes[i].hlines(x0[i], 0, T, 'r', '--')
    axes[i].set_title(title)
    axes[i].grid()

plt.tight_layout()
plt.show()
```


Unemployment rate
0.120
0.115
0.110
0.105
0.100
0.095
0.090
0.085
0
10
20
30
40
50
Employment rate
0.915
0.910
0.905
0.900
0.895
0.890
0.885
0.880
0
10
20
30
40
50

CHAPTER

FIFTYFOUR

RATIONAL EXPECTATIONS EQUILIBRIUM

Contents

"If you're so smart, why aren't you rich?"

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

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

54.1 Overview

This lecture introduces the concept of *rational expectations equilibrium*.

To illustrate it, we describe a linear quadratic version of a famous and important model due to Lucas and Prescott [LP71].

This 1971 paper is one of a small number of research articles that kicked off the *rational expectations revolution*.

We follow Lucas and Prescott by employing a setting that is readily "Bellmanized" (i.e., capable of being formulated in terms of dynamic programming problems).

Because we use linear quadratic setups for demand and costs, we can adapt the LQ programming techniques described in *this lecture*.

We will learn about how a representative agent's problem differs from a planner's, and how a planning problem can be used to compute rational expectations quantities.

We will also learn about how a rational expectations equilibrium can be characterized as a fixed point of a mapping from a *perceived law of motion* to an *actual law of motion*.

Equality between a perceived and an actual law of motion for endogenous market-wide objects captures in a nutshell what the rational expectations equilibrium concept is all about.

Finally, we will learn about the important "Big K, little k" trick, a modeling device widely used in macroeconomics.

Except that for us

- Instead of "Big K" it will be "Big Y".
- Instead of "little k" it will be "little y".

Let's start with some standard imports:

```
%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 LQ class from QuantEcon.py.

```
from quantecon import LQ
```

54.1.1 The Big Y, Little y Trick

This widely used method applies in contexts in which a "representative firm" or agent is a "price taker" operating within a competitive equilibrium.

We want to impose that

- The representative firm or individual takes *aggregate* Y as given when it chooses individual y, but ….
- At the end of the day, $Y = y$, so that the representative firm is indeed representative.

The Big Y, little y trick accomplishes these two goals by

- Taking Y as beyond control when posing the choice problem of who chooses y; but ….
- Imposing $Y = y$ *after* having solved the individual's optimization problem.

Please watch for how this strategy is applied as the lecture unfolds.

We begin by applying the Big Y, little y trick in a very simple static context.

A Simple Static Example of the Big Y, Little y Trick

Consider a static model in which a collection of n firms produce a homogeneous good that is sold in a competitive market.

Each of these n firms sells output y.

The price p of the good lies on an inverse demand curve

$$p = a_0 - a_1 Y \tag{1}$$

where

- $a_i > 0$ for $i = 0, 1$
- $Y = ny$ is the market-wide level of output

Each firm has a total cost function

$$c(y) = c_1 y + 0.5 c_2 y^2, \qquad c_i > 0 \text{ for } i = 1, 2$$

The profits of a representative firm are $py - c(y)$.

Using (1), we can express the problem of the representative firm as

$$\max_y \Big[(a_0 - a_1 Y) y - c_1 y - 0.5 c_2 y^2 \Big] \tag{2}$$

In posing problem (2), we want the firm to be a *price taker*.

We do that by regarding p and therefore Y as exogenous to the firm.

The essence of the Big Y, little y trick is *not* to set $Y = ny$ *before* taking the first-order condition with respect to y in problem (2).

This assures that the firm is a price taker.

The first-order condition for problem (2) is

$$a_0 - a_1 Y - c_1 - c_2 y = 0 \tag{3}$$

At this point, *but not before*, we substitute $Y = ny$ into (3) to obtain the following linear equation

$$a_0 - c_1 - (a_1 + n^{-1} c_2) Y = 0 \tag{4}$$

to be solved for the competitive equilibrium market-wide output Y.

After solving for Y, we can compute the competitive equilibrium price p from the inverse demand curve (1).

54.1.2 Further Reading

References for this lecture include

- [LP71]
- [Sar87], chapter XIV
- [LS18], chapter 7

54.2 Defining Rational Expectations Equilibrium

Our first illustration of a rational expectations equilibrium involves a market with n firms, each of which seeks to maximize the discounted present value of profits in the face of adjustment costs.

The adjustment costs induce the firms to make gradual adjustments, which in turn requires consideration of future prices.

Individual firms understand that, via the inverse demand curve, the price is determined by the amounts supplied by other firms.

Hence each firm wants to forecast future total industry supplies.

In our context, a forecast is generated by a belief about the law of motion for the aggregate state.

Rational expectations equilibrium prevails when this belief coincides with the actual law of motion generated by production choices induced by this belief.

We formulate a rational expectations equilibrium in terms of a fixed point of an operator that maps beliefs into optimal beliefs.

54.2.1 Competitive Equilibrium with Adjustment Costs

To illustrate, consider a collection of n firms producing a homogeneous good that is sold in a competitive market.

Each of these n firms sell output y_t.

The price p_t of the good lies on the inverse demand curve

$$p_t = a_0 - a_1 Y_t \tag{5}$$

where

- $a_i > 0$ for $i = 0, 1$
- $Y_t = n y_t$ is the market-wide level of output

The Firm's Problem

Each firm is a price taker.

While it faces no uncertainty, it does face adjustment costs

In particular, it chooses a production plan to maximize

$$\sum_{t=0}^{\infty} \beta^t r_t \tag{6}$$

where

$$r_t := p_t y_t - \frac{\gamma (y_{t+1} - y_t)^2}{2}, \qquad y_0 \text{ given} \tag{7}$$

Regarding the parameters,

- $\beta \in (0, 1)$ is a discount factor
- $\gamma > 0$ measures the cost of adjusting the rate of output

Regarding timing, the firm observes p_t and y_t when it chooses y_{t+1} at time t.

To state the firm's optimization problem completely requires that we specify dynamics for all state variables.

This includes ones that the firm cares about but does not control like p_t.

We turn to this problem now.

Prices and Aggregate Output

In view of (5), the firm's incentive to forecast the market price translates into an incentive to forecast aggregate output Y_t.

Aggregate output depends on the choices of other firms.

We assume that n is such a large number that the output of any single firm has a negligible effect on aggregate output.

That justifies firms in regarding their forecasts of aggregate output as being unaffected by their own output decisions.

The Firm's Beliefs

We suppose the firm believes that market-wide output Y_t follows the law of motion

$$Y_{t+1} = H(Y_t) \tag{8}$$

where Y_0 is a known initial condition.

The *belief function* H is an equilibrium object, and hence remains to be determined.

Optimal Behavior Given Beliefs

For now, let's fix a particular belief H in (8) and investigate the firm's response to it.

Let v be the optimal value function for the firm's problem given H.

The value function satisfies the Bellman equation

$$v(y, Y) = \max_{y'} \left\{ a_0 y - a_1 y Y - \frac{\gamma(y' - y)^2}{2} + \beta v(y', H(Y)) \right\} \tag{9}$$

Let's denote the firm's optimal policy function by h, so that

$$y_{t+1} = h(y_t, Y_t) \tag{10}$$

where

$$h(y, Y) := \arg\max_{y'} \left\{ a_0 y - a_1 y Y - \frac{\gamma(y' - y)^2}{2} + \beta v(y', H(Y)) \right\} \tag{11}$$

Evidently v and h both depend on H.

A First-Order Characterization

In what follows it will be helpful to have a second characterization of h, based on first-order conditions.

The first-order necessary condition for choosing y' is

$$-\gamma(y' - y) + \beta v_y(y', H(Y)) = 0 \tag{12}$$

An important useful envelope result of Benveniste-Scheinkman [BS79] implies that to differentiate v with respect to y we can naively differentiate the right side of (9), giving

$$v_y(y, Y) = a_0 - a_1 Y + \gamma(y' - y)$$

Substituting this equation into (12) gives the *Euler equation*

$$-\gamma(y_{t+1} - y_t) + \beta[a_0 - a_1 Y_{t+1} + \gamma(y_{t+2} - y_{t+1})] = 0 \tag{13}$$

The firm optimally sets an output path that satisfies (13), taking (8) as given, and subject to

- the initial conditions for (y_0, Y_0).
- the terminal condition $\lim_{t \to \infty} \beta^t y_t v_y(y_t, Y_t) = 0$.

This last condition is called the *transversality condition*, and acts as a first-order necessary condition "at infinity".

The firm's decision rule solves the difference equation (13) subject to the given initial condition y_0 and the transversality condition.

Note that solving the Bellman equation (9) for v and then h in (11) yields a decision rule that automatically imposes both the Euler equation (13) and the transversality condition.

The Actual Law of Motion for Output

As we've seen, a given belief translates into a particular decision rule h.

Recalling that $Y_t = ny_t$, the *actual law of motion* for market-wide output is then

$$Y_{t+1} = nh(Y_t/n, Y_t) \tag{14}$$

Thus, when firms believe that the law of motion for market-wide output is (8), their optimizing behavior makes the actual law of motion be (14).

54.2.2 Definition of Rational Expectations Equilibrium

A *rational expectations equilibrium* or *recursive competitive equilibrium* of the model with adjustment costs is a decision rule h and an aggregate law of motion H such that

1. Given belief H, the map h is the firm's optimal policy function.
2. The law of motion H satisfies $H(Y) = nh(Y/n, Y)$ for all Y.

Thus, a rational expectations equilibrium equates the perceived and actual laws of motion (8) and (14).

Fixed Point Characterization

As we've seen, the firm's optimum problem induces a mapping Φ from a perceived law of motion H for market-wide output to an actual law of motion $\Phi(H)$.

The mapping Φ is the composition of two operations, taking a perceived law of motion into a decision rule via (9)–(11), and a decision rule into an actual law via (14).

The H component of a rational expectations equilibrium is a fixed point of Φ.

54.3 Computation of an Equilibrium

Now let's consider the problem of computing the rational expectations equilibrium.

54.3.1 Failure of Contractivity

Readers accustomed to dynamic programming arguments might try to address this problem by choosing some guess H_0 for the aggregate law of motion and then iterating with Φ.

Unfortunately, the mapping Φ is not a contraction.

In particular, there is no guarantee that direct iterations on Φ converge[1].

Furthermore, there are examples in which these iterations diverge.

Fortunately, there is another method that works here.

The method exploits a connection between equilibrium and Pareto optimality expressed in the fundamental theorems of welfare economics (see, e.g, [MCWG95]).

[1] A literature that studies whether models populated with agents who learn can converge to rational expectations equilibria features iterations on a modification of the mapping Φ that can be approximated as $\gamma\Phi + (1-\gamma)I$. Here I is the identity operator and $\gamma \in (0,1)$ is a *relaxation parameter*. See [MS89] and [EH01] for statements and applications of this approach to establish conditions under which collections of adaptive agents who use least squares learning to converge to a rational expectations equilibrium.

Lucas and Prescott [LP71] used this method to construct a rational expectations equilibrium.

The details follow.

54.3.2 A Planning Problem Approach

Our plan of attack is to match the Euler equations of the market problem with those for a single-agent choice problem.

As we'll see, this planning problem can be solved by LQ control (*linear regulator*).

The optimal quantities from the planning problem are rational expectations equilibrium quantities.

The rational expectations equilibrium price can be obtained as a shadow price in the planning problem.

For convenience, in this section, we set $n = 1$.

We first compute a sum of consumer and producer surplus at time t

$$s(Y_t, Y_{t+1}) := \int_0^{Y_t} (a_0 - a_1 x) \, dx - \frac{\gamma (Y_{t+1} - Y_t)^2}{2} \tag{15}$$

The first term is the area under the demand curve, while the second measures the social costs of changing output.

The *planning problem* is to choose a production plan $\{Y_t\}$ to maximize

$$\sum_{t=0}^{\infty} \beta^t s(Y_t, Y_{t+1})$$

subject to an initial condition for Y_0.

54.3.3 Solution of the Planning Problem

Evaluating the integral in (15) yields the quadratic form $a_0 Y_t - a_1 Y_t^2 / 2$.

As a result, the Bellman equation for the planning problem is

$$V(Y) = \max_{Y'} \left\{ a_0 Y - \frac{a_1}{2} Y^2 - \frac{\gamma (Y' - Y)^2}{2} + \beta V(Y') \right\} \tag{16}$$

The associated first-order condition is

$$-\gamma (Y' - Y) + \beta V'(Y') = 0 \tag{17}$$

Applying the same Benveniste-Scheinkman formula gives

$$V'(Y) = a_0 - a_1 Y + \gamma (Y' - Y)$$

Substituting this into equation (17) and rearranging leads to the Euler equation

$$\beta a_0 + \gamma Y_t - [\beta a_1 + \gamma (1 + \beta)] Y_{t+1} + \gamma \beta Y_{t+2} = 0 \tag{18}$$

54.3.4 The Key Insight

Return to equation (13) and set $y_t = Y_t$ for all t.

(Recall that for this section we've set $n = 1$ to simplify the calculations)

A small amount of algebra will convince you that when $y_t = Y_t$, equations (18) and (13) are identical.

Thus, the Euler equation for the planning problem matches the second-order difference equation that we derived by

1. finding the Euler equation of the representative firm and
2. substituting into it the expression $Y_t = ny_t$ that "makes the representative firm be representative".

If it is appropriate to apply the same terminal conditions for these two difference equations, which it is, then we have verified that a solution of the planning problem is also a rational expectations equilibrium quantity sequence.

It follows that for this example we can compute equilibrium quantities by forming the optimal linear regulator problem corresponding to the Bellman equation (16).

The optimal policy function for the planning problem is the aggregate law of motion H that the representative firm faces within a rational expectations equilibrium.

Structure of the Law of Motion

As you are asked to show in the exercises, the fact that the planner's problem is an LQ problem implies an optimal policy — and hence aggregate law of motion — taking the form

$$Y_{t+1} = \kappa_0 + \kappa_1 Y_t \tag{19}$$

for some parameter pair κ_0, κ_1.

Now that we know the aggregate law of motion is linear, we can see from the firm's Bellman equation (9) that the firm's problem can also be framed as an LQ problem.

As you're asked to show in the exercises, the LQ formulation of the firm's problem implies a law of motion that looks as follows

$$y_{t+1} = h_0 + h_1 y_t + h_2 Y_t \tag{20}$$

Hence a rational expectations equilibrium will be defined by the parameters $(\kappa_0, \kappa_1, h_0, h_1, h_2)$ in (19)–(20).

54.4 Exercises

54.4.1 Exercise 1

Consider the firm problem *described above*.

Let the firm's belief function H be as given in (19).

Formulate the firm's problem as a discounted optimal linear regulator problem, being careful to describe all of the objects needed.

Use the class `LQ` from the QuantEcon.py package to solve the firm's problem for the following parameter values:

$$a_0 = 100, a_1 = 0.05, \beta = 0.95, \gamma = 10, \kappa_0 = 95.5, \kappa_1 = 0.95$$

Express the solution of the firm's problem in the form (20) and give the values for each h_j.

If there were n identical competitive firms all behaving according to (20), what would (20) imply for the *actual* law of motion (8) for market supply.

54.4.2 Exercise 2

Consider the following κ_0, κ_1 pairs as candidates for the aggregate law of motion component of a rational expectations equilibrium (see (19)).

Extending the program that you wrote for exercise 1, determine which if any satisfy *the definition* of a rational expectations equilibrium

- (94.0886298678, 0.923409232937)
- (93.2119845412, 0.984323478873)
- (95.0818452486, 0.952459076301)

Describe an iterative algorithm that uses the program that you wrote for exercise 1 to compute a rational expectations equilibrium.

(You are not being asked actually to use the algorithm you are suggesting)

54.4.3 Exercise 3

Recall the planner's problem *described above*

1. Formulate the planner's problem as an LQ problem.
2. Solve it using the same parameter values in exercise 1
   - $a_0 = 100, a_1 = 0.05, \beta = 0.95, \gamma = 10$
3. Represent the solution in the form $Y_{t+1} = \kappa_0 + \kappa_1 Y_t$.
4. Compare your answer with the results from exercise 2.

54.4.4 Exercise 4

A monopolist faces the industry demand curve (5) and chooses $\{Y_t\}$ to maximize $\sum_{t=0}^{\infty} \beta^t r_t$ where

$$r_t = p_t Y_t - \frac{\gamma (Y_{t+1} - Y_t)^2}{2}$$

Formulate this problem as an LQ problem.

Compute the optimal policy using the same parameters as the previous exercise.

In particular, solve for the parameters in

$$Y_{t+1} = m_0 + m_1 Y_t$$

Compare your results with the previous exercise – comment.

54.5 Solutions

54.5.1 Exercise 1

To map a problem into a discounted optimal linear control problem, we need to define

- state vector x_t and control vector u_t
- matrices A, B, Q, R that define preferences and the law of motion for the state

For the state and control vectors, we choose

$$x_t = \begin{bmatrix} y_t \\ Y_t \\ 1 \end{bmatrix}, \qquad u_t = y_{t+1} - y_t$$

For B, Q, R we set

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \kappa_1 & \kappa_0 \\ 0 & 0 & 1 \end{bmatrix}, \quad B = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad R = \begin{bmatrix} 0 & a_1/2 & -a_0/2 \\ a_1/2 & 0 & 0 \\ -a_0/2 & 0 & 0 \end{bmatrix}, \quad Q = \gamma/2$$

By multiplying out you can confirm that

- $x_t' R x_t + u_t' Q u_t = -r_t$
- $x_{t+1} = A x_t + B u_t$

We'll use the module `lqcontrol.py` to solve the firm's problem at the stated parameter values.

This will return an LQ policy F with the interpretation $u_t = -F x_t$, or

$$y_{t+1} - y_t = -F_0 y_t - F_1 Y_t - F_2$$

Matching parameters with $y_{t+1} = h_0 + h_1 y_t + h_2 Y_t$ leads to

$$h_0 = -F_2, \quad h_1 = 1 - F_0, \quad h_2 = -F_1$$

Here's our solution

```
# Model parameters

a0 = 100
a1 = 0.05
β = 0.95
γ = 10.0

# Beliefs

κ0 = 95.5
κ1 = 0.95

# Formulate the LQ problem

A = np.array([[1, 0, 0], [0, κ1, κ0], [0, 0, 1]])
B = np.array([1, 0, 0])
B.shape = 3, 1
R = np.array([[0, a1/2, -a0/2], [a1/2, 0, 0], [-a0/2, 0, 0]])
Q = 0.5 * γ
```

(continues on next page)

(continued from previous page)

```
# Solve for the optimal policy

lq = LQ(Q, R, A, B, beta=β)
P, F, d = lq.stationary_values()
F = F.flatten()
out1 = f"F = [{F[0]:.3f}, {F[1]:.3f}, {F[2]:.3f}]"
h0, h1, h2 = -F[2], 1 - F[0], -F[1]
out2 = f"(h0, h1, h2) = ({h0:.3f}, {h1:.3f}, {h2:.3f})"

print(out1)
print(out2)
```

```
F = [-0.000, 0.046, -96.949]
(h0, h1, h2) = (96.949, 1.000, -0.046)
```

The implication is that

$$y_{t+1} = 96.949 + y_t - 0.046\, Y_t$$

For the case $n > 1$, recall that $Y_t = n y_t$, which, combined with the previous equation, yields

$$Y_{t+1} = n\,(96.949 + y_t - 0.046\, Y_t) = n96.949 + (1 - n0.046) Y_t$$

54.5.2 Exercise 2

To determine whether a κ_0, κ_1 pair forms the aggregate law of motion component of a rational expectations equilibrium, we can proceed as follows:

- Determine the corresponding firm law of motion $y_{t+1} = h_0 + h_1 y_t + h_2 Y_t$.
- Test whether the associated aggregate law :$Y_{t+1} = n h(Y_t/n, Y_t)$ evaluates to $Y_{t+1} = \kappa_0 + \kappa_1 Y_t$.

In the second step, we can use $Y_t = n y_t = y_t$, so that $Y_{t+1} = n h(Y_t/n, Y_t)$ becomes

$$Y_{t+1} = h(Y_t, Y_t) = h_0 + (h_1 + h_2) Y_t$$

Hence to test the second step we can test $\kappa_0 = h_0$ and $\kappa_1 = h_1 + h_2$.

The following code implements this test

```
candidates = ((94.0886298678, 0.923409232937),
              (93.2119845412, 0.984323478873),
              (95.0818452486, 0.952459076301))

for κ0, κ1 in candidates:

    # Form the associated law of motion
    A = np.array([[1, 0, 0], [0, κ1, κ0], [0, 0, 1]])

    # Solve the LQ problem for the firm
    lq = LQ(Q, R, A, B, beta=β)
    P, F, d = lq.stationary_values()
    F = F.flatten()
    h0, h1, h2 = -F[2], 1 - F[0], -F[1]
```

(continues on next page)

(continued from previous page)

```
    # Test the equilibrium condition
    if np.allclose((κ0, κ1), (h0, h1 + h2)):
        print(f'Equilibrium pair = {κ0}, {κ1}')
        print('f(h0, h1, h2) = {h0}, {h1}, {h2}')
        break
```

```
Equilibrium pair = 95.0818452486, 0.952459076301
f(h0, h1, h2) = {h0}, {h1}, {h2}
```

The output tells us that the answer is pair (iii), which implies $(h_0, h_1, h_2) = (95.0819, 1.0000, -.0475)$.

(Notice we use `np.allclose` to test equality of floating-point numbers, since exact equality is too strict).

Regarding the iterative algorithm, one could loop from a given (κ_0, κ_1) pair to the associated firm law and then to a new (κ_0, κ_1) pair.

This amounts to implementing the operator Φ described in the lecture.

(There is in general no guarantee that this iterative process will converge to a rational expectations equilibrium)

54.5.3 Exercise 3

We are asked to write the planner problem as an LQ problem.

For the state and control vectors, we choose

$$x_t = \begin{bmatrix} Y_t \\ 1 \end{bmatrix}, \quad u_t = Y_{t+1} - Y_t$$

For the LQ matrices, we set

$$A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad B = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad R = \begin{bmatrix} a_1/2 & -a_0/2 \\ -a_0/2 & 0 \end{bmatrix}, \quad Q = \gamma/2$$

By multiplying out you can confirm that

- $x_t' R x_t + u_t' Q u_t = -s(Y_t, Y_{t+1})$
- $x_{t+1} = A x_t + B u_t$

By obtaining the optimal policy and using $u_t = -F x_t$ or

$$Y_{t+1} - Y_t = -F_0 Y_t - F_1$$

we can obtain the implied aggregate law of motion via $\kappa_0 = -F_1$ and $\kappa_1 = 1 - F_0$.

The Python code to solve this problem is below:

```
# Formulate the planner's LQ problem

A = np.array([[1, 0], [0, 1]])
B = np.array([[1], [0]])
R = np.array([[a1 / 2, -a0 / 2], [-a0 / 2, 0]])
Q = γ / 2

# Solve for the optimal policy

lq = LQ(Q, R, A, B, beta=β)
```

(continues on next page)

(continued from previous page)

```
P, F, d = lq.stationary_values()

# Print the results

F = F.flatten()
κ0, κ1 = -F[1], 1 - F[0]
print(κ0, κ1)
```

```
95.08187459215002 0.9524590627039248
```

The output yields the same (κ_0, κ_1) pair obtained as an equilibrium from the previous exercise.

54.5.4 Exercise 4

The monopolist's LQ problem is almost identical to the planner's problem from the previous exercise, except that

$$R = \begin{bmatrix} a_1 & -a_0/2 \\ -a_0/2 & 0 \end{bmatrix}$$

The problem can be solved as follows

```
A = np.array([[1, 0], [0, 1]])
B = np.array([[1], [0]])
R = np.array([[a1, -a0 / 2], [-a0 / 2, 0]])
Q = γ / 2

lq = LQ(Q, R, A, B, beta=β)
P, F, d = lq.stationary_values()

F = F.flatten()
m0, m1 = -F[1], 1 - F[0]
print(m0, m1)
```

```
73.47294403502818 0.9265270559649701
```

We see that the law of motion for the monopolist is approximately $Y_{t+1} = 73.4729 + 0.9265 Y_t$.

In the rational expectations case, the law of motion was approximately $Y_{t+1} = 95.0818 + 0.9525 Y_t$.

One way to compare these two laws of motion is by their fixed points, which give long-run equilibrium output in each case.

For laws of the form $Y_{t+1} = c_0 + c_1 Y_t$, the fixed point is $c_0/(1 - c_1)$.

If you crunch the numbers, you will see that the monopolist adopts a lower long-run quantity than obtained by the competitive market, implying a higher market price.

This is analogous to the elementary static-case results

CHAPTER

FIFTYFIVE

STABILITY IN LINEAR RATIONAL EXPECTATIONS MODELS

Contents

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

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

```
%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 sympy import *
init_printing()
```

55.1 Overview

This lecture studies stability in the context of an elementary rational expectations model.

We study a rational expectations version of Philip Cagan's model [Cag56] linking the price level to the money supply.

Cagan did not use a rational expectations version of his model, but Sargent [Sar77] did.

We study a rational expectations version of this model because it is intrinsically interesting and because it has a mathematical structure that appears in virtually all linear rational expectations model, namely, that a key endogenous variable equals a mathematical expectation of a geometric sum of future values of another variable.

The model determines the price level or rate of inflation as a function of the money supply or the rate of change in the money supply.

In this lecture, we'll encounter:

- a convenient formula for the expectation of geometric sum of future values of a variable
- a way of solving an expectational difference equation by mapping it into a vector first-order difference equation and appropriately manipulating an eigen decomposition of the transition matrix in order to impose stability
- a way to use a Big K, little k argument to allow apparent feedback from endogenous to exogenous variables within a rational expectations equilibrium
- a use of eigenvector decompositions of matrices that allowed Blanchard and Khan (1981) [BK80] and Whiteman (1983) [Whi83] to solve a class of linear rational expectations models
- how to use **SymPy** to get analytical formulas for some key objects comprising a rational expectations equilibrium

We formulate a version of Cagan's model under rational expectations as an **expectational difference equation** whose solution is a rational expectations equilibrium.

We'll start this lecture with a quick review of deterministic (i.e., non-random) first-order and second-order linear difference equations.

55.2 Linear difference equations

We'll use the *backward shift* or *lag* operator L.

The lag operator L maps a sequence $\{x_t\}_{t=0}^{\infty}$ into the sequence $\{x_{t-1}\}_{t=0}^{\infty}$

We'll deploy L by using the equality $Lx_t \equiv x_{t-1}$ in algebraic expressions.

Further, the inverse L^{-1} of the lag operator is the *forward shift* operator.

We'll often use the equality $L^{-1}x_t \equiv x_{t+1}$ below.

The algebra of lag and forward shift operators can simplify representing and solving linear difference equations.

55.2.1 First order

We want to solve a linear first-order scalar difference equation.

Let $|\lambda| < 1$ and let $\{u_t\}_{t=-\infty}^{\infty}$ be a bounded sequence of scalar real numbers.

Let L be the lag operator defined by $Lx_t \equiv x_{t-1}$ and let L^{-1} be the forward shift operator defined by $L^{-1}x_t \equiv x_{t+1}$.

Then

$$(1 - \lambda L)y_t = u_t, \forall t \tag{1}$$

has solutions

$$y_t = (1 - \lambda L)^{-1}u_t + k\lambda^t \tag{2}$$

or

$$y_t = \sum_{j=0}^{\infty} \lambda^j u_{t-j} + k\lambda^t$$

for any real number k.

You can verify this fact by applying $(1 - \lambda L)$ to both sides of equation (2) and noting that $(1 - \lambda L)\lambda^t = 0$.

To pin down k we need one condition imposed from outside (e.g., an initial or terminal condition) on the path of y.

Now let $|\lambda| > 1$.

Rewrite equation (1) as

$$y_{t-1} = \lambda^{-1}y_t - \lambda^{-1}u_t, \forall t \tag{3}$$

or

$$(1 - \lambda^{-1}L^{-1})y_t = -\lambda^{-1}u_{t+1}. \tag{4}$$

A solution is

$$y_t = -\lambda^{-1}\left(\frac{1}{1 - \lambda^{-1}L^{-1}}\right)u_{t+1} + k\lambda^t \tag{5}$$

for any k.

To verify that this is a solution, check the consequences of operating on both sides of equation (5) by $(1 - \lambda L)$ and compare to equation (1).

For any bounded $\{u_t\}$ sequence, solution (2) exists for $|\lambda| < 1$ because the **distributed lag** in u converges.

Solution (5) exists when $|\lambda| > 1$ because the **distributed lead** in u converges.

When $|\lambda| > 1$, the distributed lag in u in (2) may diverge, in which case a solution of this form does not exist.

The distributed lead in u in (5) need not converge when $|\lambda| < 1$.

55.2.2 Second order

Now consider the second order difference equation

$$(1 - \lambda_1 L)(1 - \lambda_2 L)y_{t+1} = u_t \tag{6}$$

where $\{u_t\}$ is a bounded sequence, y_0 is an initial condition, $|\lambda_1| < 1$ and $|\lambda_2| > 1$.

We seek a bounded sequence $\{y_t\}_{t=0}^{\infty}$ that satisfies (6). Using insights from our analysis of the first-order equation, operate on both sides of (6) by the forward inverse of $(1 - \lambda_2 L)$ to rewrite equation (6) as

$$(1 - \lambda_1 L) y_{t+1} = -\frac{\lambda_2^{-1}}{1 - \lambda_2^{-1} L^{-1}} u_{t+1}$$

or

$$y_{t+1} = \lambda_1 y_t - \lambda_2^{-1} \sum_{j=0}^{\infty} \lambda_2^{-j} u_{t+j+1}. \tag{7}$$

Thus, we obtained equation (7) by solving a stable root (in this case λ_1) **backward**, and an unstable root (in this case λ_2) **forward**.

Equation (7) has a form that we shall encounter often.

- $\lambda_1 y_t$ is called the **feedback part**
- $-\frac{\lambda_2^{-1}}{1-\lambda_2^{-1}L^{-1}} u_{t+1}$ is called the **feedforward part**

55.3 Illustration: Cagan's Model

Now let's use linear difference equations to represent and solve Sargent's [Sar77] rational expectations version of Cagan's model [Cag56] that connects the price level to the public's anticipations of future money supplies.

Cagan did not use a rational expectations version of his model, but Sargent [Sar77]

Let

- m_t^d be the log of the demand for money
- m_t be the log of the supply of money
- p_t be the log of the price level

It follows that $p_{t+1} - p_t$ is the rate of inflation.

The logarithm of the demand for real money balances $m_t^d - p_t$ is an inverse function of the expected rate of inflation $p_{t+1} - p_t$ for $t \geq 0$:

$$m_t^d - p_t = -\beta (p_{t+1} - p_t), \quad \beta > 0$$

Equate the demand for log money m_t^d to the supply of log money m_t in the above equation and rearrange to deduce that the logarithm of the price level p_t is related to the logarithm of the money supply m_t by

$$p_t = (1 - \lambda) m_t + \lambda p_{t+1} \tag{8}$$

where $\lambda \equiv \frac{\beta}{1+\beta} \in (0, 1)$.

(We note that the characteristic polynomial if $1 - \lambda^{-1} z^{-1} = 0$ so that the zero of the characteristic polynomial in this case is $\lambda \in (0, 1)$ which here is **inside** the unit circle.)

Solving the first order difference equation (8) forward gives

$$p_t = (1 - \lambda) \sum_{j=0}^{\infty} \lambda^j m_{t+j}, \tag{9}$$

which is the unique **stable** solution of difference equation (8) among a class of more general solutions

$$p_t = (1-\lambda)\sum_{j=0}^{\infty} \lambda^j m_{t+j} + c\lambda^{-t} \tag{10}$$

that is indexed by the real number $c \in \mathbf{R}$.

Because we want to focus on stable solutions, we set $c = 0$.

Equation (10) attributes **perfect foresight** about the money supply sequence to the holders of real balances.

We begin by assuming that the log of the money supply is **exogenous** in the sense that it is an autonomous process that does not feed back on the log of the price level.

In particular, we assume that the log of the money supply is described by the linear state space system

$$\begin{aligned} m_t &= G x_t \\ x_{t+1} &= A x_t \end{aligned} \tag{11}$$

where x_t is an $n \times 1$ vector that does not include p_t or lags of p_t, A is an $n \times n$ matrix with eigenvalues that are less than λ^{-1} in absolute values, and G is a $1 \times n$ selector matrix.

Variables appearing in the vector x_t contain information that might help predict future values of the money supply.

We'll start with an example in which x_t includes only m_t, possibly lagged values of m, and a constant.

An example of such an $\{m_t\}$ process that fits info state space system (11) is one that satisfies the second order linear difference equation

$$m_{t+1} = \alpha + \rho_1 m_t + \rho_2 m_{t-1}$$

where the zeros of the characteristic polynomial $(1 - \rho_1 z - \rho_2 z^2)$ are strictly greater than 1 in modulus.

(Please see *this* QuantEcon lecture for more about characteristic polynomials and their role in solving linear difference equations.)

We seek a stable or non-explosive solution of the difference equation (8) that obeys the system comprised of (8)-(11).

By stable or non-explosive, we mean that neither m_t nor p_t diverges as $t \rightarrow +\infty$.

This requires that we shut down the term $c\lambda^{-t}$ in equation (10) above by setting $c = 0$

The solution we are after is

$$p_t = F x_t \tag{12}$$

where

$$F = (1-\lambda) G (I - \lambda A)^{-1} \tag{13}$$

Note: As mentioned above, an *explosive solution* of difference equation (8) can be constructed by adding to the right hand of (12) a sequence $c\lambda^{-t}$ where c is an arbitrary positive constant.

55.4 Some Python code

We'll construct examples that illustrate (11).

Our first example takes as the law of motion for the log money supply the second order difference equation

$$m_{t+1} = \alpha + \rho_1 m_t + \rho_2 m_{t-1} \tag{14}$$

that is parameterized by ρ_1, ρ_2, α

To capture this parameterization with system (9) we set

$$x_t = \begin{bmatrix} 1 \\ m_t \\ m_{t-1} \end{bmatrix}, \quad A = \begin{bmatrix} 1 & 0 & 0 \\ \alpha & \rho_1 & \rho_2 \\ 0 & 1 & 0 \end{bmatrix}, \quad G = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}$$

Here is Python code

```
λ = .9

α = 0
ρ1 = .9
ρ2 = .05

A = np.array([[1,  0,  0],
              [α, ρ1, ρ2],
              [0,  1,  0]])
G = np.array([[0, 1, 0]])
```

The matrix A has one eigenvalue equal to unity.

It is associated with the A_{11} component that captures a constant component of the state x_t.

We can verify that the two eigenvalues of A not associated with the constant in the state x_t are strictly less than unity in modulus.

```
eigvals = np.linalg.eigvals(A)
print(eigvals)
```

```
[-0.05249378  0.95249378  1.        ]
```

```
(abs(eigvals) <= 1).all()
```

```
True
```

Now let's compute F in formulas (12) and (13).

```
# compute the solution, i.e. forumula (3)
F = (1 - λ) * G @ np.linalg.inv(np.eye(A.shape[0]) - λ * A)
print("F= ",F)
```

```
F=  [[0.         0.66889632 0.03010033]]
```

Now let's simulate paths of m_t and p_t starting from an initial value x_0.

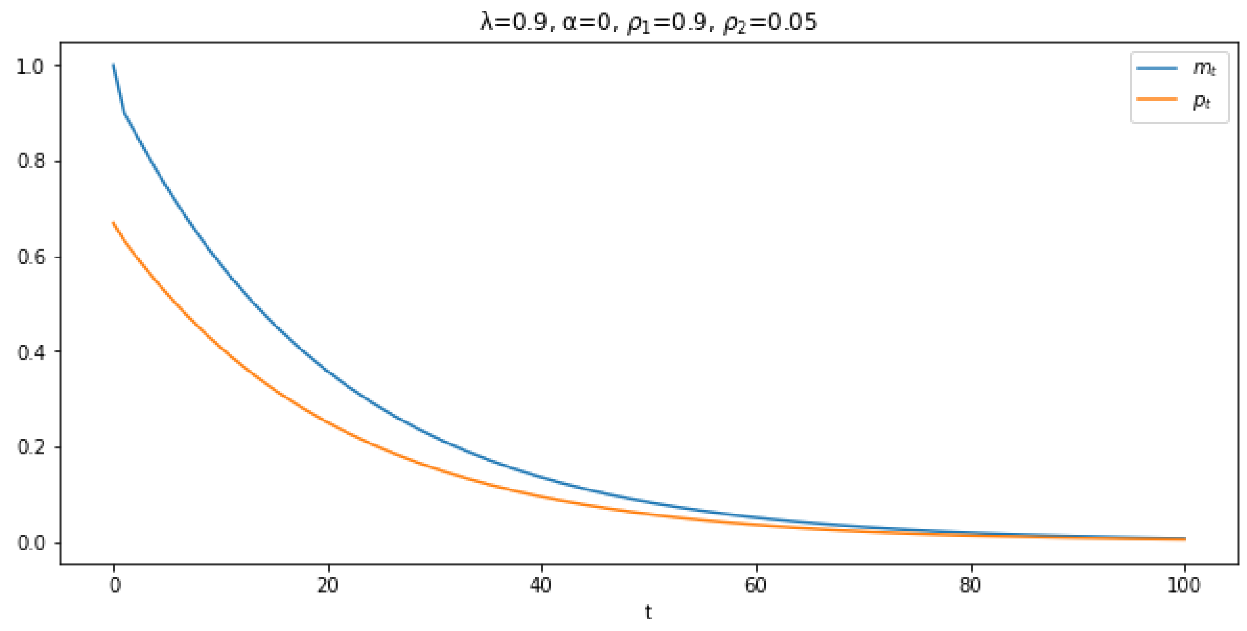
```
# set the initial state
x0 = np.array([1, 1, 0])

T = 100 # length of simulation

m_seq = np.empty(T+1)
p_seq = np.empty(T+1)

m_seq[0] = G @ x0
p_seq[0] = F @ x0
```

(continues on next page)

(continued from previous page)

```python

# simulate for T periods
x_old = x0
for t in range(T):

    x = A @ x_old

    m_seq[t+1] = G @ x
    p_seq[t+1] = F @ x

    x_old = x
```

```python
plt.figure()
plt.plot(range(T+1), m_seq, label='$m_t$')
plt.plot(range(T+1), p_seq, label='$p_t$')
plt.xlabel('t')
plt.title(f'λ={λ}, α={α}, $ρ_1$={ρ1}, $ρ_2$={ρ2}')
plt.legend()
plt.show()
```



In the above graph, why is the log of the price level always less than the log of the money supply?

Because

- according to equation (9), $p_t$ is a geometric weighted average of current and future values of $m_t$, and
- it happens that in this example future m's are always less than the current m

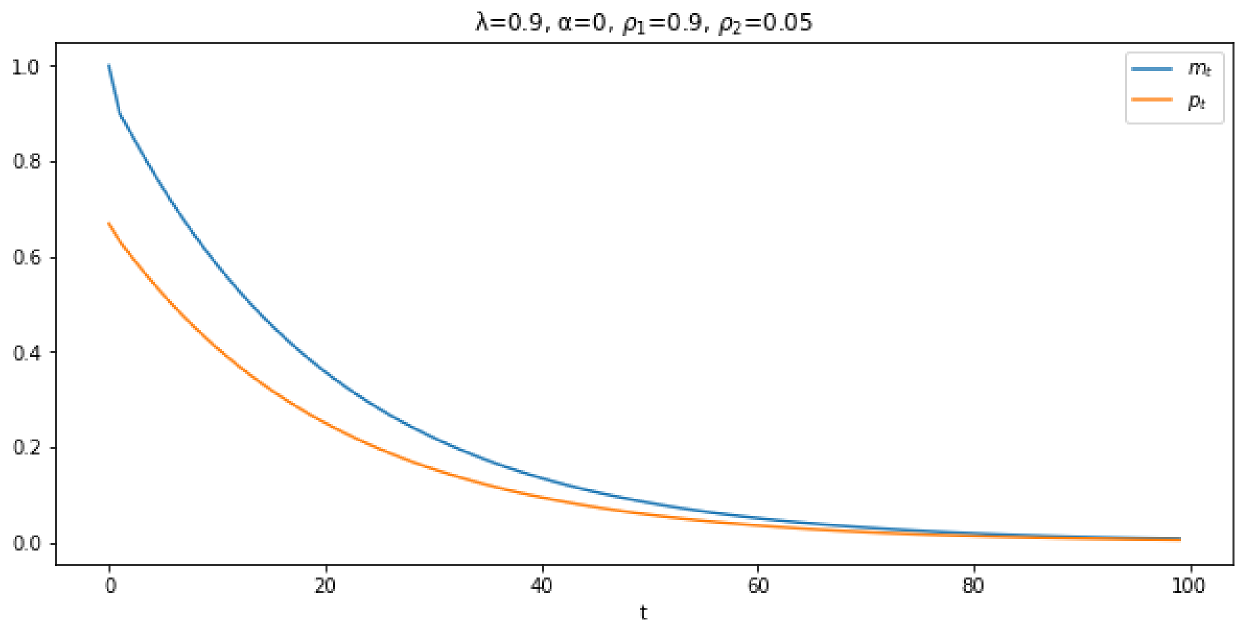
55.5 Alternative code

We could also have run the simulation using the quantecon **LinearStateSpace** code.

The following code block performs the calculation with that code.

```
# construct a LinearStateSpace instance

# stack G and F
G_ext = np.vstack([G, F])

C = np.zeros((A.shape[0], 1))

ss = qe.LinearStateSpace(A, C, G_ext, mu_0=x0)
```

```
T = 100

# simulate using LinearStateSpace
x, y = ss.simulate(ts_length=T)

# plot
plt.figure()
plt.plot(range(T), y[0,:], label='$m_t$')
plt.plot(range(T), y[1,:], label='$p_t$')
plt.xlabel('t')
plt.title(f'λ={λ}, α={α}, $ρ_1$={ρ1}, $ρ_2$={ρ2}')
plt.legend()
plt.show()
```



55.5.1 Special case

To simplify our presentation in ways that will let focus on an important idea, in the above second-order difference equation (14) that governs m_t, we now set $\alpha = 0$, $\rho_1 = \rho \in (-1, 1)$, and $\rho_2 = 0$ so that the law of motion for m_t becomes

$$m_{t+1} = \rho m_t \tag{15}$$

and the state x_t becomes

$$x_t = m_t.$$

Consequently, we can set $G = 1, A = \rho$ making our formula (13) for F become

$$F = (1 - \lambda)(1 - \lambda\rho)^{-1}.$$

so that the log the log price level satisfies

$$p_t = F m_t.$$

Please keep these formulas in mind as we investigate an alternative route to and interpretation of our formula for F.

55.6 Another perspective

Above, we imposed stability or non-explosiveness on the solution of the key difference equation (8) in Cagan's model by solving the unstable root of the characteristic polynomial forward.

To shed light on the mechanics involved in imposing stability on a solution of a potentially unstable system of linear difference equations and to prepare the way for generalizations of our model in which the money supply is allowed to feed back on the price level itself, we stack equations (8) and (15) to form the system

$$\begin{bmatrix} m_{t+1} \\ p_{t+1} \end{bmatrix} = \begin{bmatrix} \rho & 0 \\ -(1-\lambda)/\lambda & \lambda^{-1} \end{bmatrix} \begin{bmatrix} m_t \\ p_t \end{bmatrix} \tag{16}$$

or

$$y_{t+1} = H y_t, \quad t \geq 0 \tag{17}$$

where

$$H = \begin{bmatrix} \rho & 0 \\ -(1-\lambda)/\lambda & \lambda^{-1} \end{bmatrix}. \tag{18}$$

Transition matrix H has eigenvalues $\rho \in (0, 1)$ and $\lambda^{-1} > 1$.

Because an eigenvalue of H exceeds unity, if we iterate on equation (17) starting from an arbitrary initial vector $y_0 = \begin{bmatrix} m_0 \\ p_0 \end{bmatrix}$ with $m_0 > 0, p_0 > 0$, we discover that in general absolute values of both components of y_t diverge toward $+\infty$ as $t \to +\infty$.

To substantiate this claim, we can use the eigenvector matrix decomposition of H that is available to us because the eigenvalues of H are distinct

$$H = Q \Lambda Q^{-1}.$$

Here Λ is a diagonal matrix of eigenvalues of H and Q is a matrix whose columns are eigenvectors associated with the corresponding eigenvalues.

Note that

$$H^t = Q\Lambda^t Q^{-1}$$

so that

$$y_t = Q\Lambda^t Q^{-1} y_0$$

For almost all initial vectors y_0, the presence of the eigenvalue $\lambda^{-1} > 1$ causes both components of y_t to diverge in absolute value to $+\infty$.

To explore this outcome in more detail, we can use the following transformation

$$y_t^* = Q^{-1} y_t$$

that allows us to represent the dynamics in a way that isolates the source of the propensity of paths to diverge:

$$y_{t+1}^* = \Lambda^t y_t^*$$

Staring at this equation indicates that unless

$$y_0^* = \begin{bmatrix} y_{1,0}^* \\ 0 \end{bmatrix} \tag{19}$$

the path of y_t^* and therefore the paths of both components of $y_t = Qy_t^*$ will diverge in absolute value as $t \to +\infty$. (We say that the paths *explode*)

Equation (19) also leads us to conclude that there is a unique setting for the initial vector y_0 for which both components of y_t do not diverge.

The required setting of y_0 must evidently have the property that

$$Qy_0 = y_0^* = \begin{bmatrix} y_{1,0}^* \\ 0 \end{bmatrix}.$$

But note that since $y_0 = \begin{bmatrix} m_0 \\ p_0 \end{bmatrix}$ and m_0 is given to us an initial condition, p_0 has to do all the adjusting to satisfy this equation.

Sometimes this situation is described by saying that while m_0 is truly a **state** variable, p_0 is a **jump** variable that must adjust at $t = 0$ in order to satisfy the equation.

Thus, in a nutshell the unique value of the vector y_0 for which the paths of y_t do not diverge must have second component p_0 that verifies equality (19) by setting the second component of y_0^* equal to zero.

The component p_0 of the initial vector $y_0 = \begin{bmatrix} m_0 \\ p_0 \end{bmatrix}$ must evidently satisfy

$$Q^{\{2\}} y_0 = 0$$

where $Q^{\{2\}}$ denotes the second row of Q^{-1}, a restriction that is equivalent to

$$Q^{21} m_0 + Q^{22} p_0 = 0 \tag{20}$$

where Q^{ij} denotes the (i, j) component of Q^{-1}.

Solving this equation for p_0, we find

$$p_0 = -(Q^{22})^{-1} Q^{21} m_0. \tag{21}$$

This is the unique **stabilizing value** of p_0 expressed as a function of m_0.

55.6.1 Refining the formula

We can get an even more convenient formula for p_0 that is cast in terms of components of Q instead of components of Q^{-1}.

To get this formula, first note that because $(Q^{21}\ Q^{22})$ is the second row of the inverse of Q and because $Q^{-1}Q = I$, it follows that

$$\begin{bmatrix} Q^{21} & Q^{22} \end{bmatrix} \begin{bmatrix} Q_{11} \\ Q_{21} \end{bmatrix} = 0$$

which implies that

$$Q^{21}Q_{11} + Q^{22}Q_{21} = 0.$$

Therefore,

$$-(Q^{22})^{-1}Q^{21} = Q_{21}Q_{11}^{-1}.$$

So we can write

$$p_0 = Q_{21}Q_{11}^{-1}m_0. \tag{22}$$

It can be verified that this formula replicates itself over time in the sense that

$$p_t = Q_{21}Q_{11}^{-1}m_t. \tag{23}$$

To implement formula (23), we want to compute Q_1 the eigenvector of Q associated with the stable eigenvalue ρ of Q.

By hand it can be verified that the eigenvector associated with the stable eigenvalue ρ is proportional to

$$Q_1 = \begin{bmatrix} 1 - \lambda\rho \\ 1 - \lambda \end{bmatrix}.$$

Notice that if we set $A = \rho$ and $G = 1$ in our earlier formula for p_t we get

$$p_t = G(I - \lambda A)^{-1}m_t = (1 - \lambda)(1 - \lambda\rho)^{-1}m_t,$$

a formula that is equivalent with

$$p_t = Q_{21}Q_{11}^{-1}m_t,$$

where

$$Q_1 = \begin{bmatrix} Q_{11} \\ Q_{21} \end{bmatrix}.$$

55.6.2 Some remarks about feedback

We have expressed (16) in what superficially appears to be a form in which y_{t+1} feeds back on y_t, even though what we actually want to represent is that the component p_t feeds **forward** on p_{t+1}, and through it, on future m_{t+j}, $j = 0, 1, 2, \ldots$.

A tell-tale sign that we should look beyond its superficial "feedback" form is that $\lambda^{-1} > 1$ so that the matrix H in (16) is **unstable**

- it has one eigenvalue ρ that is less than one in modulus that does not imperil stability, but …
- it has a second eigenvalue λ^{-1} that exceeds one in modulus and that makes H an unstable matrix

We'll keep these observations in mind as we turn now to a case in which the log money supply actually does feed back on the log of the price level.

55.7 Log money supply feeds back on log price level

An arrangement of eigenvalues that split around unity, with one being below unity and another being greater than unity, sometimes prevails when there is *feedback* from the log price level to the log money supply.

Let the feedback rule be

$$m_{t+1} = \rho m_t + \delta p_t \tag{24}$$

where $\rho \in (0, 1)$ and where we shall now allow $\delta \neq 0$.

Warning: If things are to fit together as we wish to deliver a stable system for some initial value p_0 that we want to determine uniquely, δ cannot be too large.

The forward-looking equation (8) continues to describe equality between the demand and supply of money.

We assume that equations (8) and (24) govern $y_t \equiv \begin{bmatrix} m_t \\ p_t \end{bmatrix}$ for $t \geq 0$.

The transition matrix H in the law of motion

$$y_{t+1} = H y_t$$

now becomes

$$H = \begin{bmatrix} \rho & \delta \\ -(1-\lambda)/\lambda & \lambda^{-1} \end{bmatrix}.$$

We take m_0 as a given initial condition and as before seek an initial value p_0 that stabilizes the system in the sense that y_t converges as $t \to +\infty$.

Our approach is identical with the one followed above and is based on an eigenvalue decomposition in which, cross our fingers, one eigenvalue exceeds unity and the other is less than unity in absolute value.

When $\delta \neq 0$ as we now assume, the eigenvalues of H will no longer be $\rho \in (0, 1)$ and $\lambda^{-1} > 1$

We'll just calculate them and apply the same algorithm that we used above.

That algorithm remains valid so long as the eigenvalues split around unity as before.

Again we assume that m_0 is an initial condition, but that p_0 is not given but to be solved for.

Let's write and execute some Python code that will let us explore how outcomes depend on δ.

```python
def construct_H(ρ, λ, δ):
    "contruct matrix H given parameters."

    H = np.empty((2, 2))
    H[0, :] = ρ,δ
    H[1, :] = - (1 - λ) / λ, 1 / λ

    return H

def H_eigvals(ρ=.9, λ=.5, δ=0):
    "compute the eigenvalues of matrix H given parameters."

    # construct H matrix
    H = construct_H(ρ, λ, δ)

    # compute eigenvalues
    eigvals = np.linalg.eigvals(H)

    return eigvals
```

```
H_eigvals()
```

```
array([2. , 0.9])
```

Notice that a negative δ will not imperil the stability of the matrix H, even if it has a big absolute value.

```
# small negative δ
H_eigvals(δ=-0.05)
```

```
array([0.8562829, 2.0437171])
```

```
# large negative δ
H_eigvals(δ=-1.5)
```

```
array([0.10742784, 2.79257216])
```

A sufficiently small positive δ also causes no problem.

```
# sufficiently small positive δ
H_eigvals(δ=0.05)
```

```
array([0.94750622, 1.95249378])
```

But a large enough positive δ makes both eigenvalues of H strictly greater than unity in modulus.

For example,

```
H_eigvals(δ=0.2)
```

```
array([1.12984379, 1.77015621])
```

We want to study systems in which one eigenvalue exceeds unity in modulus while the other is less than unity in modulus, so we avoid values of δ that are too.

That is, we want to avoid too much positive feedback from p_t to m_{t+1}.

```
def magic_p0(m0, ρ=.9, λ=.5, δ=0):
    """
    Use the magic formula (8) to compute the level of p0
    that makes the system stable.
    """

    H = construct_H(ρ, λ, δ)
    eigvals, Q = np.linalg.eig(H)

    # find the index of the smaller eigenvalue
    ind = 0 if eigvals[0] < eigvals[1] else 1

    # verify that the eigenvalue is less than unity
    if eigvals[ind] > 1:

        print("both eigenvalues exceed unity in modulus")

        return None
```

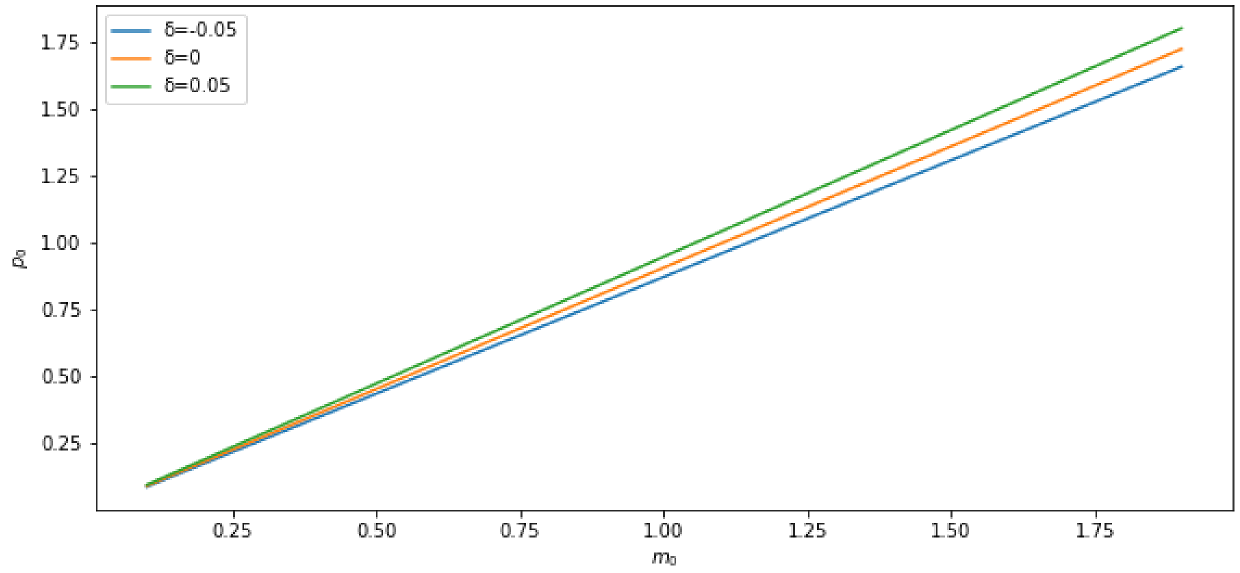
(continues on next page)

(continued from previous page)

```
    p0 = Q[1, ind] / Q[0, ind] * m0

    return p0
```

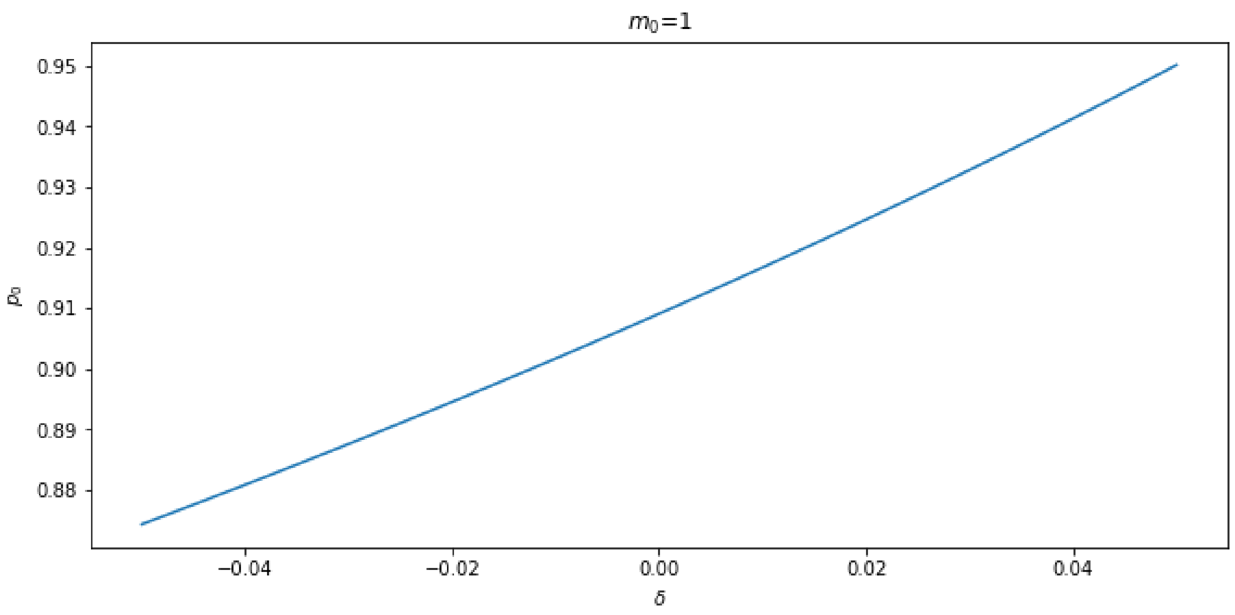
Let's plot how the solution p_0 changes as m_0 changes for different settings of δ.

```
m_range = np.arange(0.1, 2., 0.1)

for δ in [-0.05, 0, 0.05]:
    plt.plot(m_range, [magic_p0(m0, δ=δ) for m0 in m_range], label=f"δ={δ}")
plt.legend()

plt.xlabel("$m_0$")
plt.ylabel("$p_0$")
plt.show()
```



To look at things from a different angle, we can fix the initial value m_0 and see how p_0 changes as δ changes.

```
m0 = 1

δ_range = np.linspace(-0.05, 0.05, 100)
plt.plot(δ_range, [magic_p0(m0, δ=δ) for δ in δ_range])
plt.xlabel('$\delta$')
plt.ylabel('$p_0$')
plt.title(f'$m_0$={m0}')
plt.show()
```



Notice that when δ is large enough, both eigenvalues exceed unity in modulus, causing a stabilizing value of p_0 not to exist.

```
magic_p0(1, δ=0.2)
```

```
both eigenvalues exceed unity in modulus
```

55.8 Big P, little p interpretation

It is helpful to view our solutions of difference equations having feedback from the price level or inflation to money or the rate of money creation in terms of the Big K, little k idea discussed in *Rational Expectations Models*.

This will help us sort out what is taken as given by the decision makers who use the difference equation (9) to determine p_t as a function of their forecasts of future values of m_t.

Let's write the stabilizing solution that we have computed using the eigenvector decomposition of H as $P_t = F^* m_t$, where

$$F^* = Q_{21} Q_{11}^{-1}.$$

Then from $P_{t+1} = F^* m_{t+1}$ and $m_{t+1} = \rho m_t + \delta P_t$ we can deduce the recursion $P_{t+1} = F^* \rho m_t + F^* \delta P_t$ and create the stacked system

$$\begin{bmatrix} m_{t+1} \\ P_{t+1} \end{bmatrix} = \begin{bmatrix} \rho & \delta \\ F^* \rho & F^* \delta \end{bmatrix} \begin{bmatrix} m_t \\ P_t \end{bmatrix}$$

or

$$x_{t+1} = A x_t$$

where $x_t = \begin{bmatrix} m_t \\ P_t \end{bmatrix}$.

Apply formula (13) for F to deduce that

$$p_t = F \begin{bmatrix} m_t \\ P_t \end{bmatrix} = F \begin{bmatrix} m_t \\ F^* m_t \end{bmatrix}$$

which implies that

$$p_t = \begin{bmatrix} F_1 & F_2 \end{bmatrix} \begin{bmatrix} m_t \\ F^* m_t \end{bmatrix} = F_1 m_t + F_2 F^* m_t$$

so that we can anticipate that

$$F^* = F_1 + F_2 F^*$$

We shall verify this equality in the next block of Python code that implements the following computations.

1. For the system with $\delta \neq 0$ so that there is feedback, we compute the stabilizing solution for p_t in the form $p_t = F^* m_t$ where $F^* = Q_{21} Q_{11}^{-1}$ as above.
2. Recalling the system (11), (12), and (13) above, we define $x_t = \begin{bmatrix} m_t \\ P_t \end{bmatrix}$ and notice that it is Big P_t and not little p_t here. Then we form A and G as $A = \begin{bmatrix} \rho & \delta \\ F^* \rho & F^* \delta \end{bmatrix}$ and $G = \begin{bmatrix} 1 & 0 \end{bmatrix}$ and we compute $\begin{bmatrix} F_1 & F_2 \end{bmatrix} \equiv F$ from equation (13) above.
3. We compute $F_1 + F_2 F^*$ and compare it with F^* and check for the anticipated equality.

```
# set parameters
ρ = .9
λ = .5
δ = .05
```

```
# solve for F_star
H = construct_H(ρ, λ, δ)
eigvals, Q = np.linalg.eig(H)

ind = 0 if eigvals[0] < eigvals[1] else 1
F_star = Q[1, ind] / Q[0, ind]
F_star
```

```
0.9501243788791095
```

```
# solve for F_check
A = np.empty((2, 2))
A[0, :] = ρ, δ
A[1, :] = F_star * A[0, :]

G = np.array([1, 0])

F_check= (1 - λ) * G @ np.linalg.inv(np.eye(2) - λ * A)
F_check
```

```
array([0.92755597, 0.02375311])
```

Compare F^* with $F_1 + F_2 F^*$

```
F_check[0] + F_check[1] * F_star, F_star
```

```
(0.9501243788791096, 0.9501243788791095)
```

55.9 Fun with SymPy code

This section is a gift for readers who have made it this far.

It puts SymPy to work on our model.

Thus, we use Sympy to compute some key objects comprising the eigenvector decomposition of H.

We start by generating an H with nonzero δ.

```
λ, δ, ρ = symbols('λ, δ, ρ')
```

```
H1 = Matrix([[ρ,δ], [- (1 - λ) / λ, λ ** -1]])
```

```
H1
```

$$\begin{bmatrix} \rho & \delta \\ \frac{\lambda-1}{\lambda} & \frac{1}{\lambda} \end{bmatrix}$$

```
H1.eigenvals()
```

$$\left\{(\lambda\rho+1)/2\lambda - \sqrt{4\delta\lambda^2 - 4\delta\lambda + \lambda^2\rho^2 - 2\lambda\rho + 1}/2\lambda : 1,\ (\lambda\rho+1)/2\lambda + \sqrt{4\delta\lambda^2 - 4\delta\lambda + \lambda^2\rho^2 - 2\lambda\rho + 1}/2\lambda : 1\right\}$$

```
H1.eigenvects()
```

$$\left[\left(\frac{\lambda\rho+1}{2\lambda} - \frac{\sqrt{4\delta\lambda^2 - 4\delta\lambda + \lambda^2\rho^2 - 2\lambda\rho + 1}}{2\lambda},\ 1,\ \left[\begin{bmatrix} -\frac{2\delta\lambda}{\lambda\rho+\sqrt{4\delta\lambda^2-4\delta\lambda+\lambda^2\rho^2-2\lambda\rho+1}-1} \\ 1 \end{bmatrix}\right]\right),\ \left(\frac{\lambda\rho+1}{2\lambda} + \frac{\sqrt{4\delta\lambda^2 - 4\delta\lambda + \lambda^2\rho^2 - 2}}{2\lambda}\right.$$

Now let's compute H when δ is zero.

```
H2 = Matrix([[ρ,0], [- (1 - λ) / λ, λ ** -1]])
```

```
H2
```

$$\begin{bmatrix} \rho & 0 \\ \frac{\lambda-1}{\lambda} & \frac{1}{\lambda} \end{bmatrix}$$

```
H2.eigenvals()
```

$$\{1/\lambda : 1,\ \rho : 1\}$$

```
H2.eigenvects()
```

$$\left[\left(\frac{1}{\lambda},\ 1,\ \left[\begin{bmatrix} 0 \\ 1 \end{bmatrix}\right]\right),\ \left(\rho,\ 1,\ \left[\begin{bmatrix} \frac{\lambda\rho-1}{\lambda-1} \\ 1 \end{bmatrix}\right]\right)\right]$$

Below we do induce SymPy to do the following fun things for us analytically:

1. We compute the matrix Q whose first column is the eigenvector associated with ρ. and whose second column is the eigenvector associated with λ^{-1}.
2. We use SymPy to compute the inverse Q^{-1} of Q (both in symbols).
3. We use SymPy to compute $Q_{21}Q_{11}^{-1}$ (in symbols).
4. Where Q^{ij} denotes the (i, j) component of Q^{-1}, we use SymPy to compute $-(Q^{22})^{-1}Q^{21}$ (again in symbols)

```
# construct Q
vec = []
for i, (eigval, _, eigvec) in enumerate(H2.eigenvects()):

    vec.append(eigvec[0])

    if eigval == ρ:
        ind = i

Q = vec[ind].col_insert(1, vec[1-ind])
```

```
Q
```

$$\begin{bmatrix}\frac{\lambda\rho-1}{\lambda-1} & 0\\ 1 & 1\end{bmatrix}$$

Q^{-1}

```
Q_inv = Q ** (-1)
Q_inv
```

$$\begin{bmatrix}\frac{\lambda-1}{\lambda\rho-1} & 0\\ -\frac{\lambda-1}{\lambda\rho-1} & 1\end{bmatrix}$$

$Q_{21}Q_{11}^{-1}$

```
Q[1, 0] / Q[0, 0]
```

$$\frac{\lambda-1}{\lambda\rho-1}$$

$-(Q^{22})^{-1}Q^{21}$

```
- Q_inv[1, 0] / Q_inv[1, 1]
```

$$\frac{\lambda-1}{\lambda\rho-1}$$

CHAPTER

FIFTYSIX

MARKOV PERFECT EQUILIBRIUM

Contents

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

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

56.1 Overview

This lecture describes the concept of Markov perfect equilibrium.

Markov perfect equilibrium is a key notion for analyzing economic problems involving dynamic strategic interaction, and a cornerstone of applied game theory.

In this lecture, we teach Markov perfect equilibrium by example.

We will focus on settings with

- two players
- quadratic payoff functions
- linear transition rules for the state

Other references include chapter 7 of [LS18].

Let's start with some standard imports:

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

(continues on next page)

(continued from previous page)

```python
import numpy as np
import quantecon as qe
```

56.2 Background

Markov perfect equilibrium is a refinement of the concept of Nash equilibrium.

It is used to study settings where multiple decision-makers interact non-cooperatively over time, each pursuing its own objective.

The agents in the model face a common state vector, the time path of which is influenced by – and influences – their decisions.

In particular, the transition law for the state that confronts each agent is affected by decision rules of other agents.

Individual payoff maximization requires that each agent solve a dynamic programming problem that includes this transition law.

Markov perfect equilibrium prevails when no agent wishes to revise its policy, taking as given the policies of all other agents.

Well known examples include

- Choice of price, output, location or capacity for firms in an industry (e.g., [EP95], [Rya12], [DS10]).
- Rate of extraction from a shared natural resource, such as a fishery (e.g., [LM80], [VL11]).

Let's examine a model of the first type.

56.2.1 Example: A Duopoly Model

Two firms are the only producers of a good, the demand for which is governed by a linear inverse demand function

$$p = a_0 - a_1(q_1 + q_2) \tag{1}$$

Here $p = p_t$ is the price of the good, $q_i = q_{it}$ is the output of firm $i = 1, 2$ at time t and $a_0 > 0, a_1 > 0$.

In (1) and what follows,

- the time subscript is suppressed when possible to simplify notation
- $\hat{x}$ denotes a next period value of variable x

Each firm recognizes that its output affects total output and therefore the market price.

The one-period payoff function of firm i is price times quantity minus adjustment costs:

$$\pi_i = pq_i - \gamma(\hat{q}_i - q_i)^2, \quad \gamma > 0, \tag{2}$$

Substituting the inverse demand curve (1) into (2) lets us express the one-period payoff as

$$\pi_i(q_i, q_{-i}, \hat{q}_i) = a_0 q_i - a_1 q_i^2 - a_1 q_i q_{-i} - \gamma(\hat{q}_i - q_i)^2, \tag{3}$$

where q_{-i} denotes the output of the firm other than i.

The objective of the firm is to maximize $\sum_{t=0}^{\infty} \beta^t \pi_{it}$.

Firm i chooses a decision rule that sets next period quantity $\hat{q}_i$ as a function f_i of the current state (q_i, q_{-i}).

An essential aspect of a Markov perfect equilibrium is that each firm takes the decision rule of the other firm as known and given.

Given f_{-i}, the Bellman equation of firm i is

$$v_i(q_i, q_{-i}) = \max_{\hat{q}_i} \{\pi_i(q_i, q_{-i}, \hat{q}_i) + \beta v_i(\hat{q}_i, f_{-i}(q_{-i}, q_i))\} \tag{4}$$

Definition A *Markov perfect equilibrium* of the duopoly model is a pair of value functions (v_1, v_2) and a pair of policy functions (f_1, f_2) such that, for each $i \in \{1, 2\}$ and each possible state,

- The value function v_i satisfies Bellman equation (4).
- The maximizer on the right side of (4) equals $f_i(q_i, q_{-i})$.

The adjective "Markov" denotes that the equilibrium decision rules depend only on the current values of the state variables, not other parts of their histories.

"Perfect" means complete, in the sense that the equilibrium is constructed by backward induction and hence builds in optimizing behavior for each firm at all possible future states.

- These include many states that will not be reached when we iterate forward on the pair of equilibrium strategies f_i starting from a given initial state.

56.2.2 Computation

One strategy for computing a Markov perfect equilibrium is iterating to convergence on pairs of Bellman equations and decision rules.

In particular, let v_i^j, f_i^j be the value function and policy function for firm i at the j-th iteration.

Imagine constructing the iterates

$$v_i^{j+1}(q_i, q_{-i}) = \max_{\hat{q}_i} \left\{\pi_i(q_i, q_{-i}, \hat{q}_i) + \beta v_i^j(\hat{q}_i, f_{-i}(q_{-i}, q_i))\right\} \tag{5}$$

These iterations can be challenging to implement computationally.

However, they simplify for the case in which one-period payoff functions are quadratic and transition laws are linear — which takes us to our next topic.

56.3 Linear Markov Perfect Equilibria

As we saw in the duopoly example, the study of Markov perfect equilibria in games with two players leads us to an interrelated pair of Bellman equations.

In linear-quadratic dynamic games, these "stacked Bellman equations" become "stacked Riccati equations" with a tractable mathematical structure.

We'll lay out that structure in a general setup and then apply it to some simple problems.

56.3.1 Coupled Linear Regulator Problems

We consider a general linear-quadratic regulator game with two players.

For convenience, we'll start with a finite horizon formulation, where t_0 is the initial date and t_1 is the common terminal date.

Player i takes $\{u_{-it}\}$ as given and minimizes

$$\sum_{t=t_0}^{t_1-1} \beta^{t-t_0} \left\{ x_t' R_i x_t + u_{it}' Q_i u_{it} + u_{-it}' S_i u_{-it} + 2 x_t' W_i u_{it} + 2 u_{-it}' M_i u_{it} \right\} \tag{6}$$

while the state evolves according to

$$x_{t+1} = A x_t + B_1 u_{1t} + B_2 u_{2t} \tag{7}$$

Here

- x_t is an $n \times 1$ state vector and u_{it} is a $k_i \times 1$ vector of controls for player i
- R_i is $n \times n$
- S_i is $k_{-i} \times k_{-i}$
- Q_i is $k_i \times k_i$
- W_i is $n \times k_i$
- M_i is $k_{-i} \times k_i$
- A is $n \times n$
- B_i is $n \times k_i$

56.3.2 Computing Equilibrium

We formulate a linear Markov perfect equilibrium as follows.

Player i employs linear decision rules $u_{it} = -F_{it} x_t$, where F_{it} is a $k_i \times n$ matrix.

A Markov perfect equilibrium is a pair of sequences $\{F_{1t}, F_{2t}\}$ over $t = t_0, \ldots, t_1 - 1$ such that

- $\{F_{1t}\}$ solves player 1's problem, taking $\{F_{2t}\}$ as given, and
- $\{F_{2t}\}$ solves player 2's problem, taking $\{F_{1t}\}$ as given

If we take $u_{2t} = -F_{2t} x_t$ and substitute it into (6) and (7), then player 1's problem becomes minimization of

$$\sum_{t=t_0}^{t_1-1} \beta^{t-t_0} \left\{ x_t' \Pi_{1t} x_t + u_{1t}' Q_1 u_{1t} + 2 u_{1t}' \Gamma_{1t} x_t \right\} \tag{8}$$

subject to

$$x_{t+1} = \Lambda_{1t} x_t + B_1 u_{1t}, \tag{9}$$

where

- $\Lambda_{it} := A - B_{-i} F_{-it}$
- $\Pi_{it} := R_i + F_{-it}' S_i F_{-it}$
- $\Gamma_{it} := W_i' - M_i' F_{-it}$

This is an LQ dynamic programming problem that can be solved by working backwards.

Decision rules that solve this problem are

$$F_{1t} = (Q_1 + \beta B_1' P_{1t+1} B_1)^{-1} (\beta B_1' P_{1t+1} \Lambda_{1t} + \Gamma_{1t}) \tag{10}$$

where P_{1t} solves the matrix Riccati difference equation

$$P_{1t} = \Pi_{1t} - (\beta B_1' P_{1t+1} \Lambda_{1t} + \Gamma_{1t})' (Q_1 + \beta B_1' P_{1t+1} B_1)^{-1} (\beta B_1' P_{1t+1} \Lambda_{1t} + \Gamma_{1t}) + \beta \Lambda_{1t}' P_{1t+1} \Lambda_{1t} \tag{11}$$

Similarly, decision rules that solve player 2's problem are

$$F_{2t} = (Q_2 + \beta B_2' P_{2t+1} B_2)^{-1} (\beta B_2' P_{2t+1} \Lambda_{2t} + \Gamma_{2t}) \tag{12}$$

where P_{2t} solves

$$P_{2t} = \Pi_{2t} - (\beta B_2' P_{2t+1} \Lambda_{2t} + \Gamma_{2t})' (Q_2 + \beta B_2' P_{2t+1} B_2)^{-1} (\beta B_2' P_{2t+1} \Lambda_{2t} + \Gamma_{2t}) + \beta \Lambda_{2t}' P_{2t+1} \Lambda_{2t} \tag{13}$$

Here, in all cases $t = t_0, \ldots, t_1 - 1$ and the terminal conditions are $P_{it_1} = 0$.

The solution procedure is to use equations (10), (11), (12), and (13), and "work backwards" from time $t_1 - 1$.

Since we're working backward, P_{1t+1} and P_{2t+1} are taken as given at each stage.

Moreover, since

- some terms on the right-hand side of (10) contain F_{2t}
- some terms on the right-hand side of (12) contain F_{1t}

we need to solve these $k_1 + k_2$ equations simultaneously.

Key Insight

A key insight is that equations (10) and (12) are linear in F_{1t} and F_{2t}.

After these equations have been solved, we can take F_{it} and solve for P_{it} in (11) and (13).

Infinite Horizon

We often want to compute the solutions of such games for infinite horizons, in the hope that the decision rules F_{it} settle down to be time-invariant as $t_1 \to +\infty$.

In practice, we usually fix t_1 and compute the equilibrium of an infinite horizon game by driving $t_0 \to -\infty$.

This is the approach we adopt in the next section.

56.3.3 Implementation

We use the function nnash from QuantEcon.py that computes a Markov perfect equilibrium of the infinite horizon linear-quadratic dynamic game in the manner described above.

56.4 Application

Let's use these procedures to treat some applications, starting with the duopoly model.

56.4.1 A Duopoly Model

To map the duopoly model into coupled linear-quadratic dynamic programming problems, define the state and controls as

$$x_t := \begin{bmatrix} 1 \\ q_{1t} \\ q_{2t} \end{bmatrix} \quad \text{and} \quad u_{it} := q_{i,t+1} - q_{it}, \quad i = 1, 2$$

If we write

$$x_t' R_i x_t + u_{it}' Q_i u_{it}$$

where $Q_1 = Q_2 = \gamma$,

$$R_1 := \begin{bmatrix} 0 & -\frac{a_0}{2} & 0 \\ -\frac{a_0}{2} & a_1 & \frac{a_1}{2} \\ 0 & \frac{a_1}{2} & 0 \end{bmatrix} \quad \text{and} \quad R_2 := \begin{bmatrix} 0 & 0 & -\frac{a_0}{2} \\ 0 & 0 & \frac{a_1}{2} \\ -\frac{a_0}{2} & \frac{a_1}{2} & a_1 \end{bmatrix}$$

then we recover the one-period payoffs in expression (3).

The law of motion for the state x_t is $x_{t+1} = Ax_t + B_1 u_{1t} + B_2 u_{2t}$ where

$$A := \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad B_1 := \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad B_2 := \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

The optimal decision rule of firm i will take the form $u_{it} = -F_i x_t$, inducing the following closed-loop system for the evolution of x in the Markov perfect equilibrium:

$$x_{t+1} = (A - B_1 F_1 - B_1 F_2) x_t \tag{14}$$

56.4.2 Parameters and Solution

Consider the previously presented duopoly model with parameter values of:

- $a_0 = 10$
- $a_1 = 2$
- $\beta = 0.96$
- $\gamma = 12$

From these, we compute the infinite horizon MPE using the preceding code

```python
import numpy as np
import quantecon as qe

# Parameters
a0 = 10.0
a1 = 2.0
```

(continues on next page)

(continued from previous page)

```
β = 0.96
γ = 12.0

# In LQ form
A = np.eye(3)
B1 = np.array([[0.], [1.], [0.]])
B2 = np.array([[0.], [0.], [1.]])


R1 = [[      0.,     -a0 / 2,          0.],
      [-a0 / 2.,          a1,     a1 / 2.],
      [       0,     a1 / 2.,          0.]]

R2 = [[     0.,           0.,      -a0 / 2],
      [     0.,           0.,      a1 / 2.],
      [-a0 / 2,      a1 / 2.,           a1]]

Q1 = Q2 = γ
S1 = S2 = W1 = W2 = M1 = M2 = 0.0

# Solve using QE's nnash function
F1, F2, P1, P2 = qe.nnash(A, B1, B2, R1, R2, Q1,
                          Q2, S1, S2, W1, W2, M1,
                          M2, beta=β)

# Display policies
print("Computed policies for firm 1 and firm 2:\n")
print(f"F1 = {F1}")
print(f"F2 = {F2}")
print("\n")
```

```
Computed policies for firm 1 and firm 2:

F1 = [[-0.66846615  0.29512482  0.07584666]]
F2 = [[-0.66846615  0.07584666  0.29512482]]
```

Running the code produces the following output.

One way to see that F_i is indeed optimal for firm i taking F_2 as given is to use QuantEcon.py's LQ class.

In particular, let's take F2 as computed above, plug it into (8) and (9) to get firm 1's problem and solve it using LQ.

We hope that the resulting policy will agree with F1 as computed above

```
Λ1 = A - B2 @ F2
lq1 = qe.LQ(Q1, R1, Λ1, B1, beta=β)
P1_ih, F1_ih, d = lq1.stationary_values()
F1_ih
```

```
array([[-0.66846613,  0.29512482,  0.07584666]])
```

This is close enough for rock and roll, as they say in the trade.

Indeed, np.allclose agrees with our assessment

```
np.allclose(F1, F1_ih)
```

```
True
```

56.4.3 Dynamics

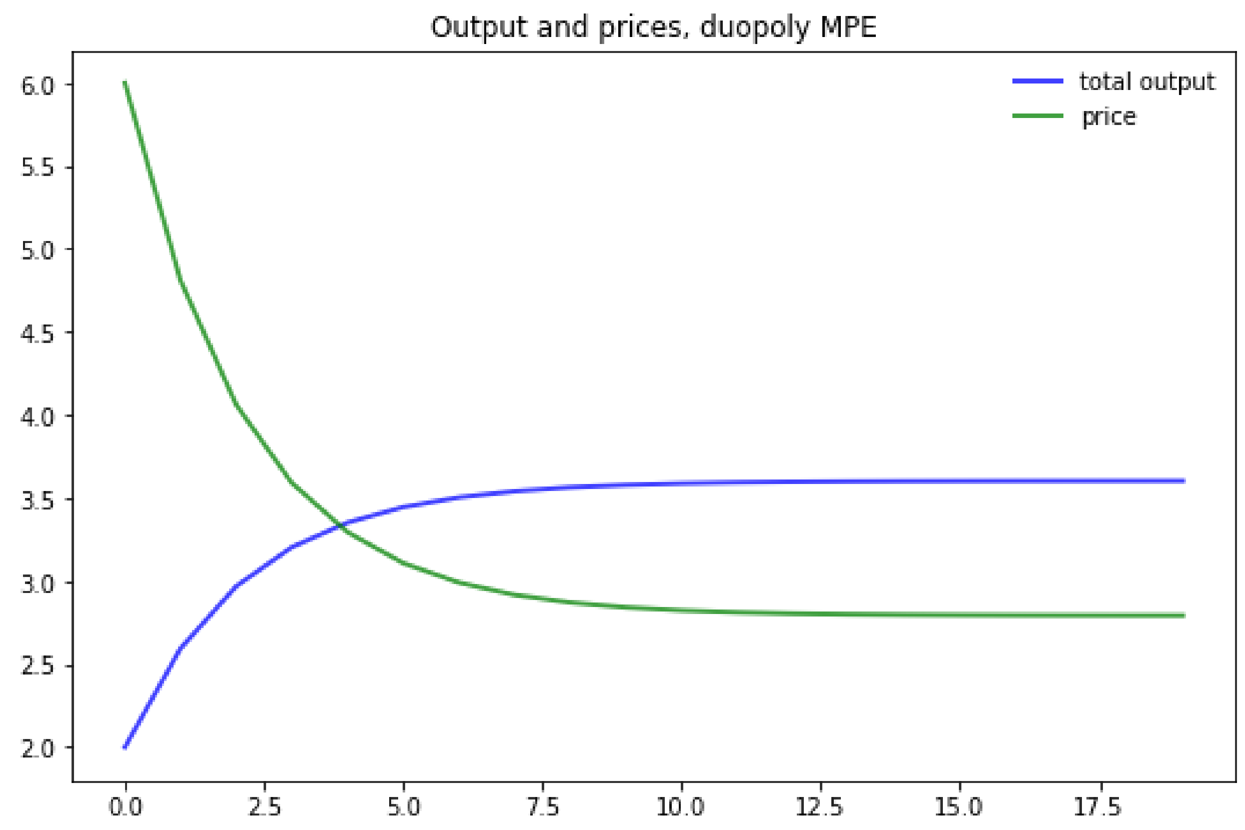
Let's now investigate the dynamics of price and output in this simple duopoly model under the MPE policies.

Given our optimal policies $F1$ and $F2$, the state evolves according to (14).

The following program

- imports $F1$ and $F2$ from the previous program along with all parameters.
- computes the evolution of x_t using (14).
- extracts and plots industry output $q_t = q_{1t} + q_{2t}$ and price $p_t = a_0 - a_1 q_t$.

```
AF = A - B1 @ F1 - B2 @ F2
n = 20
x = np.empty((3, n))
x[:, 0] = 1, 1, 1
for t in range(n-1):
    x[:, t+1] = AF @ x[:, t]
q1 = x[1, :]
q2 = x[2, :]
q = q1 + q2       # Total output, MPE
p = a0 - a1 * q   # Price, MPE

fig, ax = plt.subplots(figsize=(9, 5.8))
ax.plot(q, 'b-', lw=2, alpha=0.75, label='total output')
ax.plot(p, 'g-', lw=2, alpha=0.75, label='price')
ax.set_title('Output and prices, duopoly MPE')
ax.legend(frameon=False)
plt.show()
```



Note that the initial condition has been set to $q_{10} = q_{20} = 1.0$.

To gain some perspective we can compare this to what happens in the monopoly case.

The first panel in the next figure compares output of the monopolist and industry output under the MPE, as a function of time.

The second panel shows analogous curves for price.

Here parameters are the same as above for both the MPE and monopoly solutions.

The monopolist initial condition is $q_0 = 2.0$ to mimic the industry initial condition $q_{10} = q_{20} = 1.0$ in the MPE case.

As expected, output is higher and prices are lower under duopoly than monopoly.

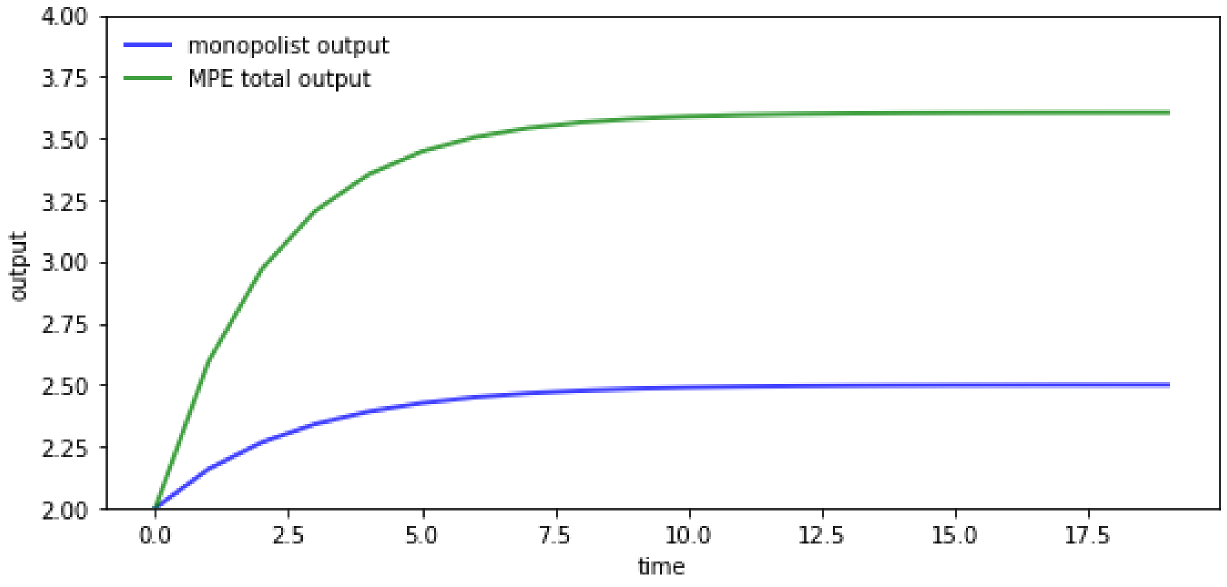
56.5 Exercises

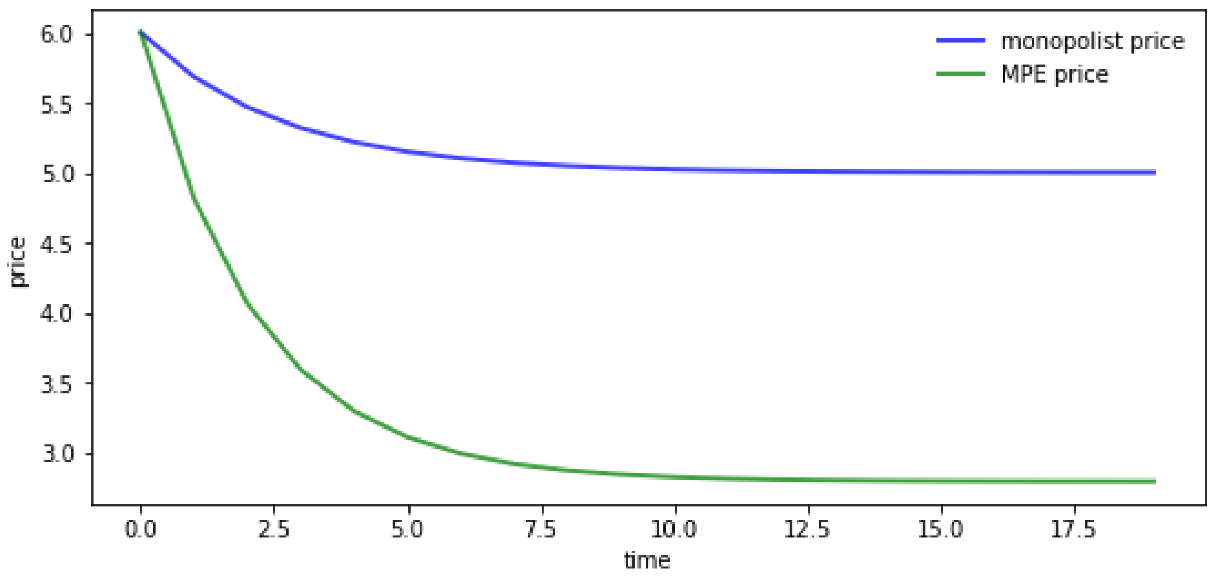
56.5.1 Exercise 1

Replicate the *pair of figures* showing the comparison of output and prices for the monopolist and duopoly under MPE.

Parameters are as in duopoly_mpe.py and you can use that code to compute MPE policies under duopoly.

The optimal policy in the monopolist case can be computed using QuantEcon.py's LQ class.


monopolist output
MPE total output
output
time
4.00
3.75
3.50
3.25
3.00
2.75
2.50
2.25
2.00
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5


monopolist price
MPE price
price
time
6.0
5.5
5.0
4.5
4.0
3.5
3.0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5

56.5.2 Exercise 2

In this exercise, we consider a slightly more sophisticated duopoly problem.

It takes the form of infinite horizon linear-quadratic game proposed by Judd [Jud90].

Two firms set prices and quantities of two goods interrelated through their demand curves.

Relevant variables are defined as follows:

- I_{it} = inventories of firm i at beginning of t
- q_{it} = production of firm i during period t
- p_{it} = price charged by firm i during period t
- S_{it} = sales made by firm i during period t
- E_{it} = costs of production of firm i during period t
- C_{it} = costs of carrying inventories for firm i during t

The firms' cost functions are

- $C_{it} = c_{i1} + c_{i2} I_{it} + 0.5 c_{i3} I_{it}^2$
- $E_{it} = e_{i1} + e_{i2} q_{it} + 0.5 e_{i3} q_{it}^2$ where e_{ij}, c_{ij} are positive scalars

Inventories obey the laws of motion

$$I_{i,t+1} = (1 - \delta) I_{it} + q_{it} - S_{it}$$

Demand is governed by the linear schedule

$$S_t = D p_{it} + b$$

where

- $S_t = \begin{bmatrix} S_{1t} & S_{2t} \end{bmatrix}'$
- D is a 2×2 negative definite matrix and
- b is a vector of constants

Firm i maximizes the undiscounted sum

$$\lim_{T \to \infty} \frac{1}{T} \sum_{t=0}^{T} \left(p_{it} S_{it} - E_{it} - C_{it} \right)$$

We can convert this to a linear-quadratic problem by taking

$$u_{it} = \begin{bmatrix} p_{it} \\ q_{it} \end{bmatrix} \quad \text{and} \quad x_t = \begin{bmatrix} I_{1t} \\ I_{2t} \\ 1 \end{bmatrix}$$

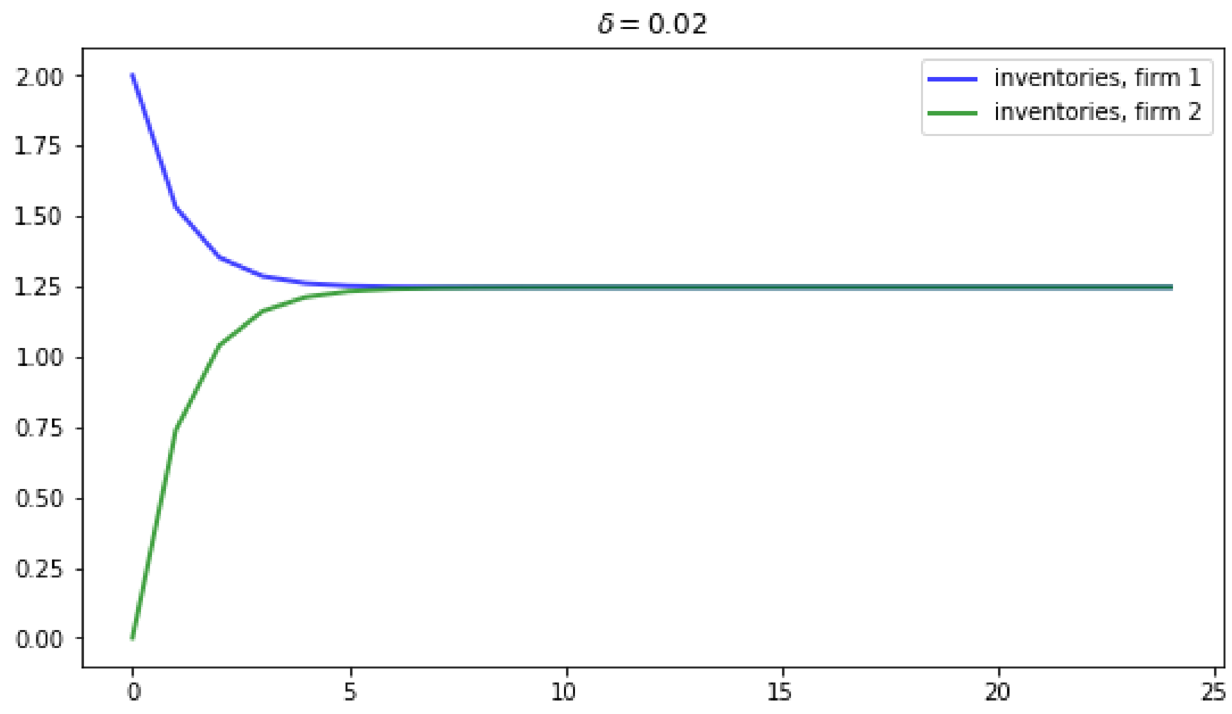
Decision rules for price and quantity take the form $u_{it} = -F_i x_t$.

The Markov perfect equilibrium of Judd's model can be computed by filling in the matrices appropriately.

The exercise is to calculate these matrices and compute the following figures.

The first figure shows the dynamics of inventories for each firm when the parameters are

```
δ = 0.02
D = np.array([[-1, 0.5], [0.5, -1]])
b = np.array([25, 25])
c1 = c2 = np.array([1, -2, 1])
e1 = e2 = np.array([10, 10, 3])
```



Inventories trend to a common steady state.

If we increase the depreciation rate to $\delta = 0.05$, then we expect steady state inventories to fall.

This is indeed the case, as the next figure shows

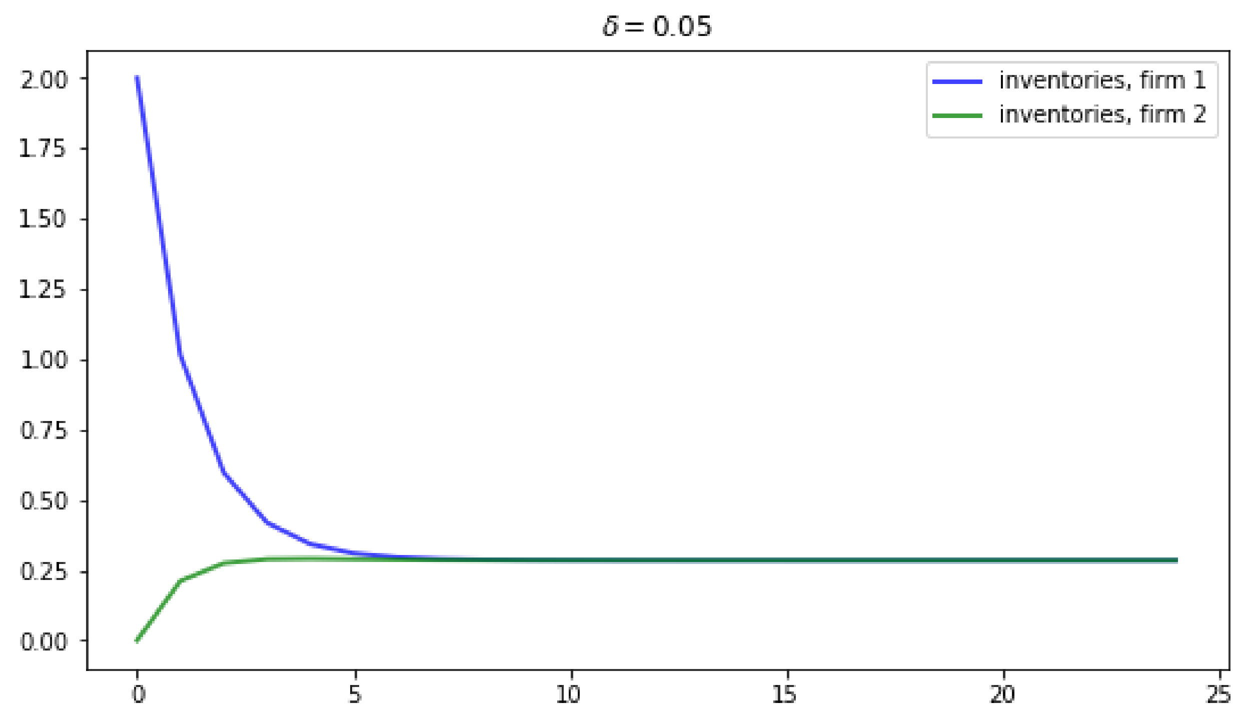
56.6 Solutions

56.6.1 Exercise 1

First, let's compute the duopoly MPE under the stated parameters

```
# == Parameters == #
a0 = 10.0
a1 = 2.0
β = 0.96
γ = 12.0

# == In LQ form == #
A  = np.eye(3)
B1 = np.array([[0.], [1.], [0.]])
B2 = np.array([[0.], [0.], [1.]])
R1 = [[      0.,      -a0/2,          0.],
```

(continues on next page)



(continued from previous page)

```
      [-a0 / 2.,          a1,      a1 / 2.],
      [       0,     a1 / 2.,           0.]]

R2 = [[      0.,            0.,      -a0 / 2],
      [      0.,            0.,      a1 / 2.],
      [-a0 / 2,      a1 / 2.,           a1]]

Q1 = Q2 = γ
S1 = S2 = W1 = W2 = M1 = M2 = 0.0

# == Solve using QE's nnash function == #
F1, F2, P1, P2 = qe.nnash(A, B1, B2, R1, R2, Q1,
                          Q2, S1, S2, W1, W2, M1,
                          M2, beta=β)
```

Now we evaluate the time path of industry output and prices given initial condition $q_{10} = q_{20} = 1$.

```
AF = A - B1 @ F1 - B2 @ F2
n = 20
x = np.empty((3, n))
x[:, 0] = 1, 1, 1
for t in range(n-1):
    x[:, t+1] = AF @ x[:, t]
q1 = x[1, :]
q2 = x[2, :]
q = q1 + q2       # Total output, MPE
p = a0 - a1 * q   # Price, MPE
```

Next, let's have a look at the monopoly solution.

For the state and control, we take

$$x_t = q_t - \bar{q} \quad \text{and} \quad u_t = q_{t+1} - q_t$$

To convert to an LQ problem we set

$$R = a_1 \quad \text{and} \quad Q = \gamma$$

in the payoff function $x_t' R x_t + u_t' Q u_t$ and

$$A = B = 1$$

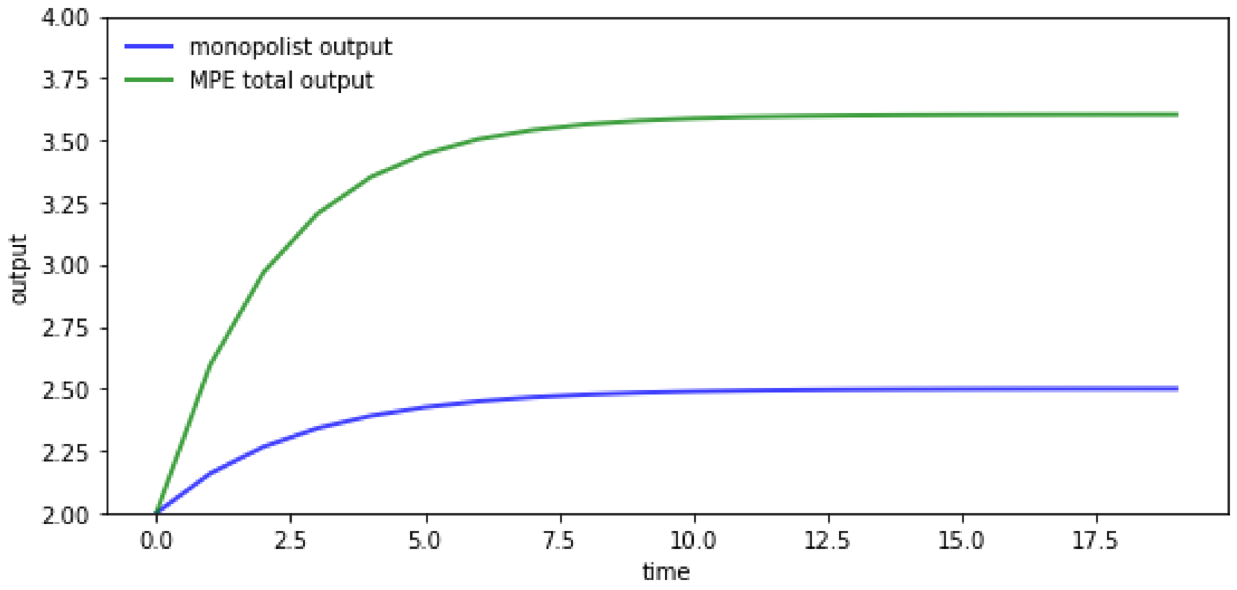
in the law of motion $x_{t+1} = A x_t + B u_t$.

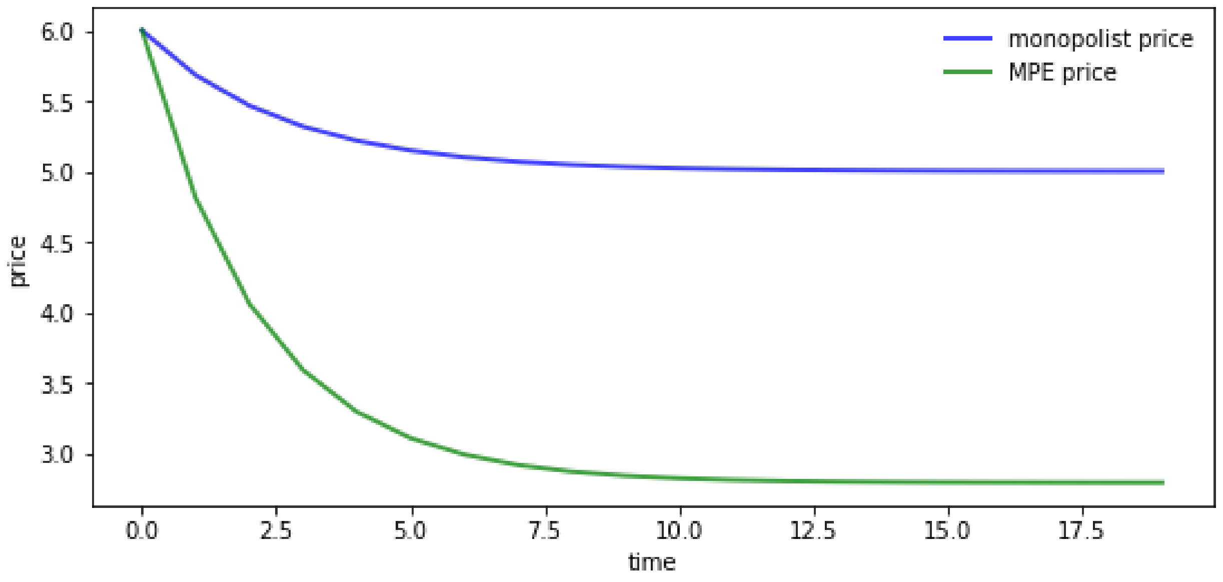
We solve for the optimal policy $u_t = -F x_t$ and track the resulting dynamics of $\{q_t\}$, starting at $q_0 = 2.0$.

```
R = a1
Q = γ
A = B = 1
lq_alt = qe.LQ(Q, R, A, B, beta=β)
P, F, d = lq_alt.stationary_values()
q_bar = a0 / (2.0 * a1)
qm = np.empty(n)
qm[0] = 2
x0 = qm[0] - q_bar
x = x0
for i in range(1, n):
    x = A * x - B * F * x
    qm[i] = float(x) + q_bar
pm = a0 - a1 * qm
```

Let's have a look at the different time paths

```
fig, axes = plt.subplots(2, 1, figsize=(9, 9))

ax = axes[0]
ax.plot(qm, 'b-', lw=2, alpha=0.75, label='monopolist output')
ax.plot(q, 'g-', lw=2, alpha=0.75, label='MPE total output')
ax.set(ylabel="output", xlabel="time", ylim=(2, 4))
ax.legend(loc='upper left', frameon=0)

ax = axes[1]
ax.plot(pm, 'b-', lw=2, alpha=0.75, label='monopolist price')
ax.plot(p, 'g-', lw=2, alpha=0.75, label='MPE price')
ax.set(ylabel="price", xlabel="time")
ax.legend(loc='upper right', frameon=0)
plt.show()
```





56.6.2 Exercise 2

We treat the case $\delta = 0.02$

```
δ = 0.02
D = np.array([[-1, 0.5], [0.5, -1]])
b = np.array([25, 25])
c1 = c2 = np.array([1, -2, 1])
e1 = e2 = np.array([10, 10, 3])

δ_1 = 1 - δ
```

Recalling that the control and state are

$$u_{it} = \begin{bmatrix} p_{it} \\ q_{it} \end{bmatrix} \quad \text{and} \quad x_t = \begin{bmatrix} I_{1t} \\ I_{2t} \\ 1 \end{bmatrix}$$

we set up the matrices as follows:

```
# ==  Create matrices needed to compute the Nash feedback equilibrium == #

A = np.array([[δ_1,       0,    -δ_1 * b[0]],
              [   0,    δ_1,    -δ_1 * b[1]],
              [   0,      0,              1]])

B1 = δ_1 * np.array([[1, -D[0, 0]],
                     [0, -D[1, 0]],
                     [0,       0]])
B2 = δ_1 * np.array([[0, -D[0, 1]],
                     [1, -D[1, 1]],
                     [0,       0]])

R1 = -np.array([[0.5 * c1[2],     0,    0.5 * c1[1]],
                [          0,     0,              0],
                [0.5 * c1[1],     0,          c1[0]]])
R2 = -np.array([[0,               0,              0],
                [0,     0.5 * c2[2],    0.5 * c2[1]],
                [0,     0.5 * c2[1],          c2[0]]])

Q1 = np.array([[-0.5 * e1[2], 0], [0, D[0, 0]]])
Q2 = np.array([[-0.5 * e2[2], 0], [0, D[1, 1]]])

S1 = np.zeros((2, 2))
S2 = np.copy(S1)

W1 = np.array([[           0,             0],
               [           0,             0],
               [-0.5 * e1[1],    b[0] / 2.]])
W2 = np.array([[           0,             0],
               [           0,             0],
               [-0.5 * e2[1],    b[1] / 2.]])

M1 = np.array([[0, 0], [0, D[0, 1] / 2.]])
M2 = np.copy(M1)
```

We can now compute the equilibrium using `qe.nnash`

```
F1, F2, P1, P2 = qe.nnash(A, B1, B2, R1,
                          R2, Q1, Q2, S1,
                          S2, W1, W2, M1, M2)

print("\nFirm 1's feedback rule:\n")
print(F1)

print("\nFirm 2's feedback rule:\n")
print(F2)
```

```
Firm 1's feedback rule:

```

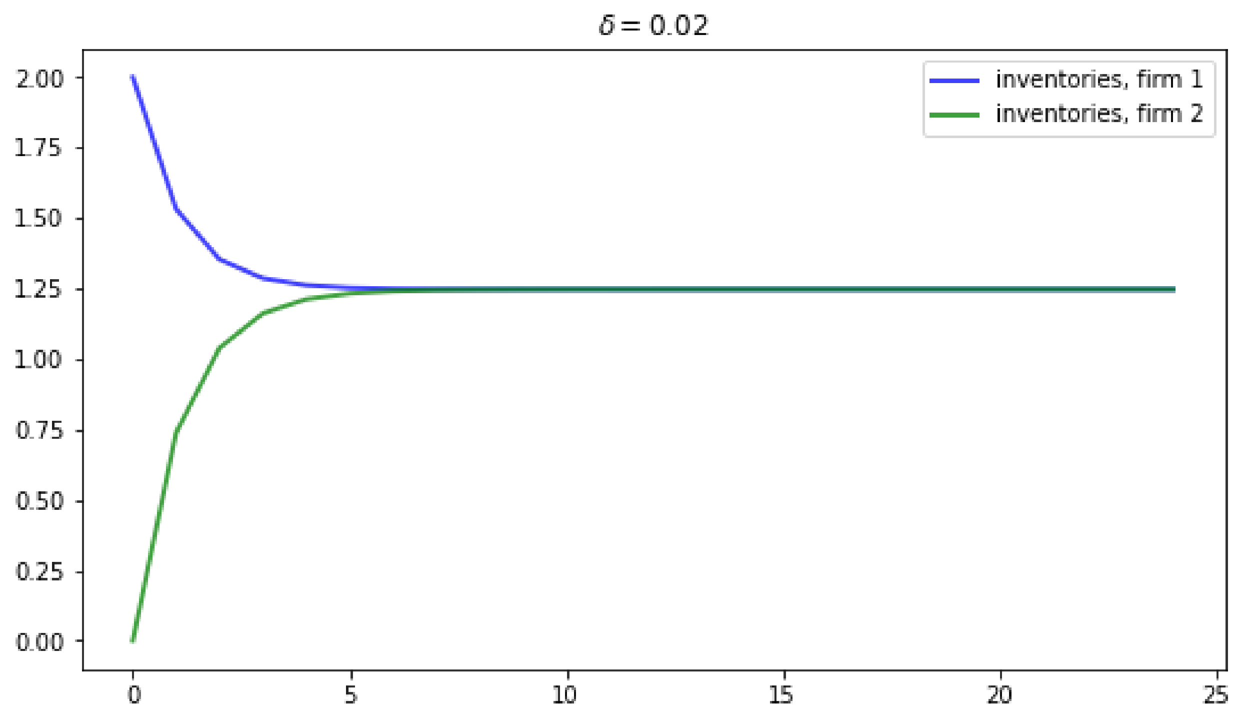
(continues on next page)

(continued from previous page)

```
[[ 2.43666582e-01  2.72360627e-02 -6.82788293e+00]
 [ 3.92370734e-01  1.39696451e-01 -3.77341073e+01]]

Firm 2's feedback rule:

[[ 2.72360627e-02  2.43666582e-01 -6.82788293e+00]
 [ 1.39696451e-01  3.92370734e-01 -3.77341073e+01]]
```

Now let's look at the dynamics of inventories, and reproduce the graph corresponding to $\delta = 0.02$

```
AF = A - B1 @ F1 - B2 @ F2
n = 25
x = np.empty((3, n))
x[:, 0] = 2, 0, 1
for t in range(n-1):
    x[:, t+1] = AF @ x[:, t]
I1 = x[0, :]
I2 = x[1, :]
fig, ax = plt.subplots(figsize=(9, 5))
ax.plot(I1, 'b-', lw=2, alpha=0.75, label='inventories, firm 1')
ax.plot(I2, 'g-', lw=2, alpha=0.75, label='inventories, firm 2')
ax.set_title(rf'$\delta = {δ}$')
ax.legend()
plt.show()
```



CHAPTER

FIFTYSEVEN

UNCERTAINTY TRAPS

Contents

57.1 Overview

In this lecture, we study a simplified version of an uncertainty traps model of Fajgelbaum, Schaal and Taschereau-Dumouchel [FSTD15].

The model features self-reinforcing uncertainty that has big impacts on economic activity.

In the model,

- Fundamentals vary stochastically and are not fully observable.
- At any moment there are both active and inactive entrepreneurs; only active entrepreneurs produce.
- Agents – active and inactive entrepreneurs – have beliefs about the fundamentals expressed as probability distributions.
- Greater uncertainty means greater dispersions of these distributions.
- Entrepreneurs are risk-averse and hence less inclined to be active when uncertainty is high.
- The output of active entrepreneurs is observable, supplying a noisy signal that helps everyone inside the model infer fundamentals.
- Entrepreneurs update their beliefs about fundamentals using Bayes' Law, implemented via *Kalman filtering*.

Uncertainty traps emerge because:

- High uncertainty discourages entrepreneurs from becoming active.

- A low level of participation – i.e., a smaller number of active entrepreneurs – diminishes the flow of information about fundamentals.
- Less information translates to higher uncertainty, further discouraging entrepreneurs from choosing to be active, and so on.

Uncertainty traps stem from a positive externality: high aggregate economic activity levels generates valuable information.

Let's start with some standard imports:

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

57.2 The Model

The original model described in [FSTD15] has many interesting moving parts.

Here we examine a simplified version that nonetheless captures many of the key ideas.

57.2.1 Fundamentals

The evolution of the fundamental process $\{\theta_t\}$ is given by

$$\theta_{t+1} = \rho\theta_t + \sigma_\theta w_{t+1}$$

where

- $\sigma_\theta > 0$ and $0 < \rho < 1$
- $\{w_t\}$ is IID and standard normal

The random variable θ_t is not observable at any time.

57.2.2 Output

There is a total $\bar{M}$ of risk-averse entrepreneurs.

Output of the m-th entrepreneur, conditional on being active in the market at time t, is equal to

$$x_m = \theta + \epsilon_m \quad \text{where} \quad \epsilon_m \sim N\left(0, \gamma_x^{-1}\right) \tag{1}$$

Here the time subscript has been dropped to simplify notation.

The inverse of the shock variance, γ_x, is called the shock's **precision**.

The higher is the precision, the more informative x_m is about the fundamental.

Output shocks are independent across time and firms.

57.2.3 Information and Beliefs

All entrepreneurs start with identical beliefs about θ_0.

Signals are publicly observable and hence all agents have identical beliefs always.

Dropping time subscripts, beliefs for current θ are represented by the normal distribution $N(\mu, \gamma^{-1})$.

Here γ is the precision of beliefs; its inverse is the degree of uncertainty.

These parameters are updated by Kalman filtering.

Let

- $\mathbb{M} \subset \{1, \ldots, \bar{M}\}$ denote the set of currently active firms.
- $M := |\mathbb{M}|$ denote the number of currently active firms.
- X be the average output $\frac{1}{M} \sum_{m \in \mathbb{M}} x_m$ of the active firms.

With this notation and primes for next period values, we can write the updating of the mean and precision via

$$\mu' = \rho \frac{\gamma \mu + M \gamma_x X}{\gamma + M \gamma_x} \tag{2}$$

$$\gamma' = \left(\frac{\rho^2}{\gamma + M \gamma_x} + \sigma_\theta^2 \right)^{-1} \tag{3}$$

These are standard Kalman filtering results applied to the current setting.

Exercise 1 provides more details on how (2) and (3) are derived and then asks you to fill in remaining steps.

The next figure plots the law of motion for the precision in (3) as a 45 degree diagram, with one curve for each $M \in \{0, \ldots, 6\}$.

The other parameter values are $\rho = 0.99, \gamma_x = 0.5, \sigma_\theta = 0.5$

Points where the curves hit the 45 degree lines are long-run steady states for precision for different values of M.

Thus, if one of these values for M remains fixed, a corresponding steady state is the equilibrium level of precision

- high values of M correspond to greater information about the fundamental, and hence more precision in steady state
- low values of M correspond to less information and more uncertainty in steady state

In practice, as we'll see, the number of active firms fluctuates stochastically.

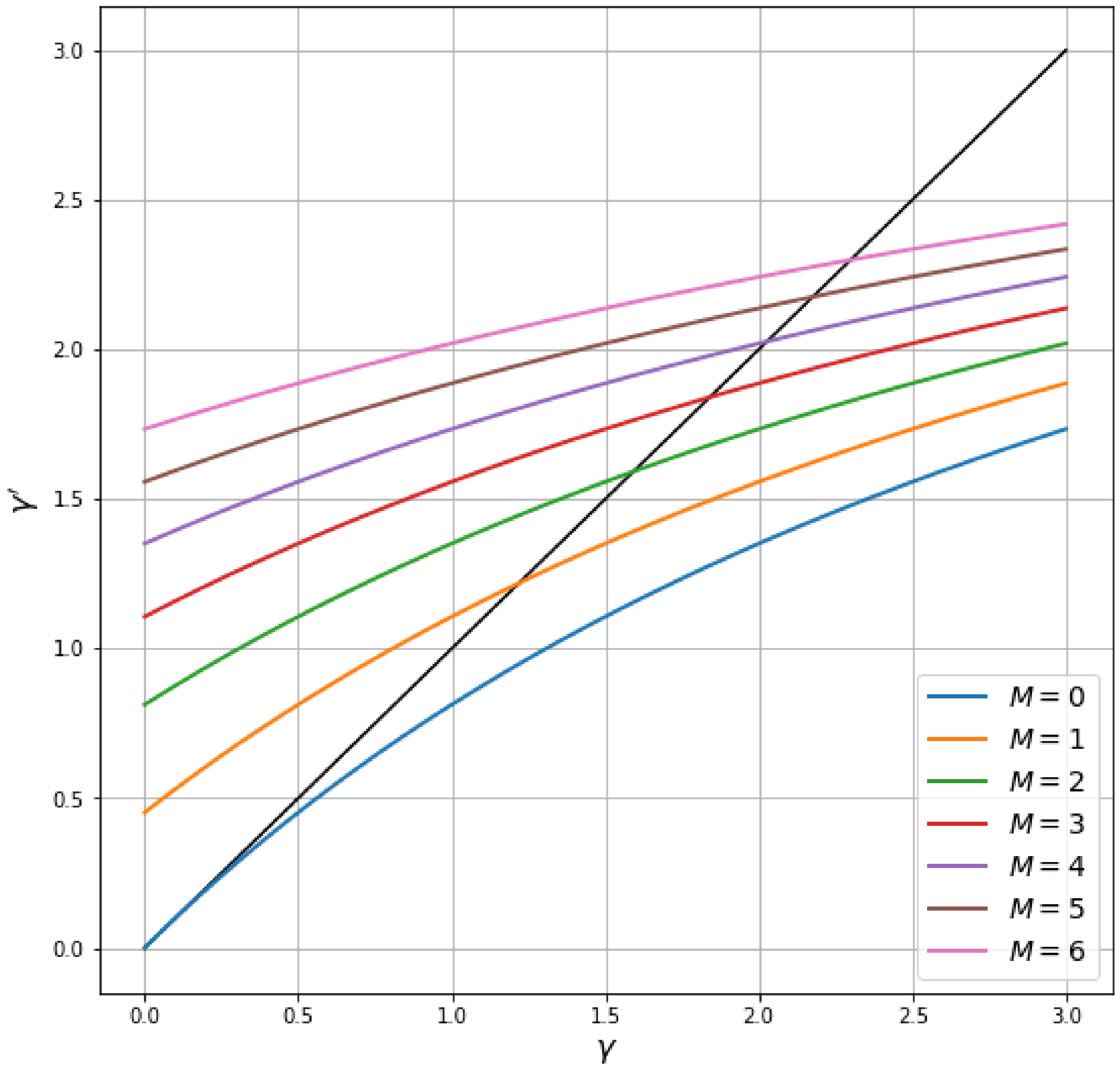
57.2.4 Participation

Omitting time subscripts once more, entrepreneurs enter the market in the current period if

$$\mathbb{E}[u(x_m - F_m)] > c \tag{4}$$

Here

- the mathematical expectation of x_m is based on (1) and beliefs $N(\mu, \gamma^{-1})$ for θ
- F_m is a stochastic but pre-visible fixed cost, independent across time and firms
- c is a constant reflecting opportunity costs


3.0
2.5
2.0
1.5
1.0
0.5
0.0
γ'
0.0
0.5
1.0
1.5
2.0
2.5
3.0
γ
$M = 0$
$M = 1$
$M = 2$
$M = 3$
$M = 4$
$M = 5$
$M = 6$

The statement that F_m is pre-visible means that it is realized at the start of the period and treated as a constant in (4).

The utility function has the constant absolute risk aversion form

$$u(x) = \frac{1}{a}(1 - \exp(-ax)) \tag{5}$$

where a is a positive parameter.

Combining (4) and (5), entrepreneur m participates in the market (or is said to be active) when

$$\frac{1}{a}\{1 - \mathbb{E}[\exp(-a(\theta + \epsilon_m - F_m))]\} > c$$

Using standard formulas for expectations of lognormal random variables, this is equivalent to the condition

$$\psi(\mu, \gamma, F_m) := \frac{1}{a}\left(1 - \exp\left(-a\mu + aF_m + \frac{a^2\left(\frac{1}{\gamma} + \frac{1}{\gamma_x}\right)}{2}\right)\right) - c > 0 \tag{6}$$

57.3 Implementation

We want to simulate this economy.

As a first step, let's put together a class that bundles

- the parameters, the current value of θ and the current values of the two belief parameters μ and γ
- methods to update θ, μ and γ, as well as to determine the number of active firms and their outputs

The updating methods follow the laws of motion for θ, μ and γ given above.

The method to evaluate the number of active firms generates $F_1, \dots, F_{\bar{M}}$ and tests condition (6) for each firm.

The **init** method encodes as default values the parameters we'll use in the simulations below

```
class UncertaintyTrapEcon:

    def __init__(self,
                a=1.5,             # Risk aversion
                γ_x=0.5,           # Production shock precision
                ρ=0.99,            # Correlation coefficient for θ
                σ_θ=0.5,           # Standard dev of θ shock
                num_firms=100,     # Number of firms
                σ_F=1.5,           # Standard dev of fixed costs
                c=-420,            # External opportunity cost
                μ_init=0,          # Initial value for μ
                γ_init=4,          # Initial value for γ
                θ_init=0):         # Initial value for θ

        # == Record values == #
        self.a, self.γ_x, self.ρ, self.σ_θ = a, γ_x, ρ, σ_θ
        self.num_firms, self.σ_F, self.c, = num_firms, σ_F, c
        self.σ_x = np.sqrt(1/γ_x)

        # == Initialize states == #
        self.γ, self.μ, self.θ = γ_init, μ_init, θ_init

    def ψ(self, F):
        temp1 = -self.a * (self.μ - F)
```

(continues on next page)

(continued from previous page)

```
        temp2 = self.a**2 * (1/self.γ + 1/self.γ_x) / 2
        return (1 / self.a) * (1 - np.exp(temp1 + temp2)) - self.c

    def update_beliefs(self, X, M):
        """
        Update beliefs (μ, γ) based on aggregates X and M.
        """
        # Simplify names
        γ_x, ρ, σ_θ = self.γ_x, self.ρ, self.σ_θ
        # Update μ
        temp1 = ρ * (self.γ * self.μ + M * γ_x * X)
        temp2 = self.γ + M * γ_x
        self.μ = temp1 / temp2
        # Update γ
        self.γ = 1 / (ρ**2 / (self.γ + M * γ_x) + σ_θ**2)

    def update_θ(self, w):
        """
        Update the fundamental state θ given shock w.
        """
        self.θ = self.ρ * self.θ + self.σ_θ * w

    def gen_aggregates(self):
        """
        Generate aggregates based on current beliefs (μ, γ). This
        is a simulation step that depends on the draws for F.
        """
        F_vals = self.σ_F * np.random.randn(self.num_firms)
        M = np.sum(self.ψ(F_vals) > 0)  # Counts number of active firms
        if M > 0:
            x_vals = self.θ + self.σ_x * np.random.randn(M)
            X = x_vals.mean()
        else:
            X = 0
        return X, M
```

In the results below we use this code to simulate time series for the major variables.

57.4 Results

Let's look first at the dynamics of μ, which the agents use to track θ

We see that μ tracks θ well when there are sufficient firms in the market.

However, there are times when μ tracks θ poorly due to insufficient information.

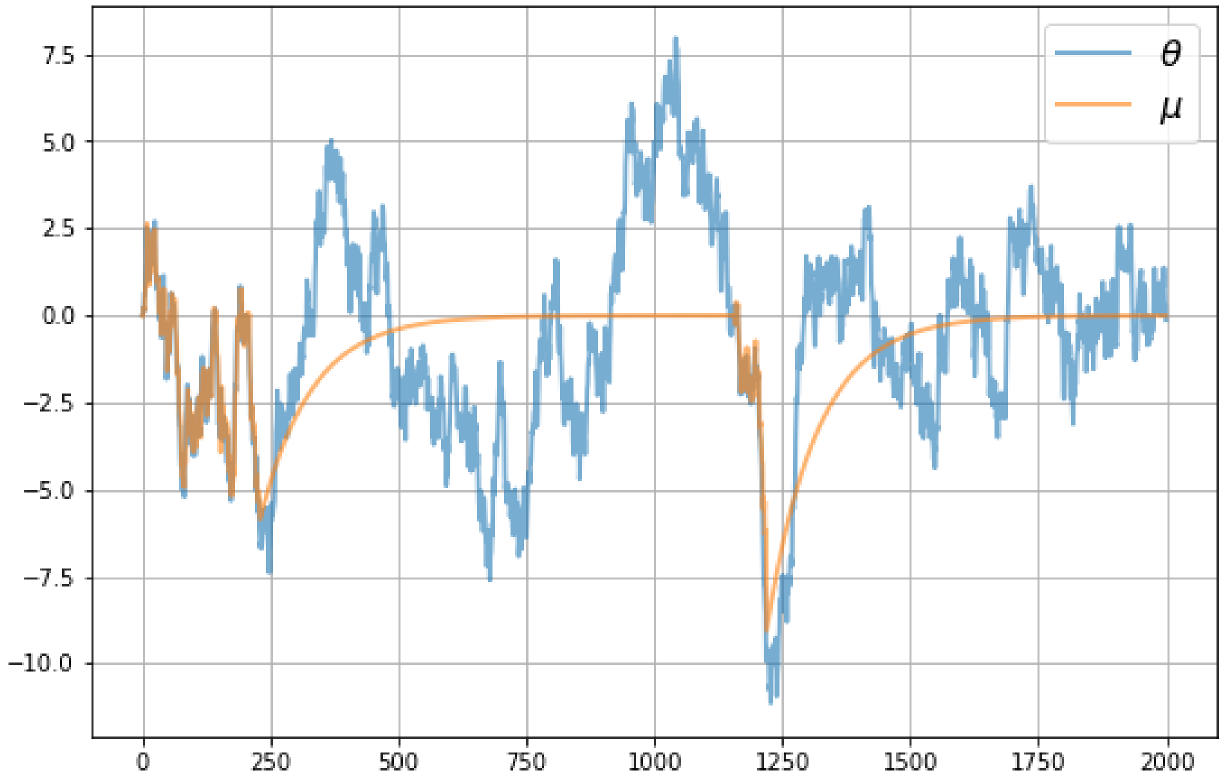
These are episodes where the uncertainty traps take hold.

During these episodes

- precision is low and uncertainty is high
- few firms are in the market

To get a clearer idea of the dynamics, let's look at all the main time series at once, for a given set of shocks

Notice how the traps only take hold after a sequence of bad draws for the fundamental.



Thus, the model gives us a *propagation mechanism* that maps bad random draws into long downturns in economic activity.

57.5 Exercises

57.5.1 Exercise 1

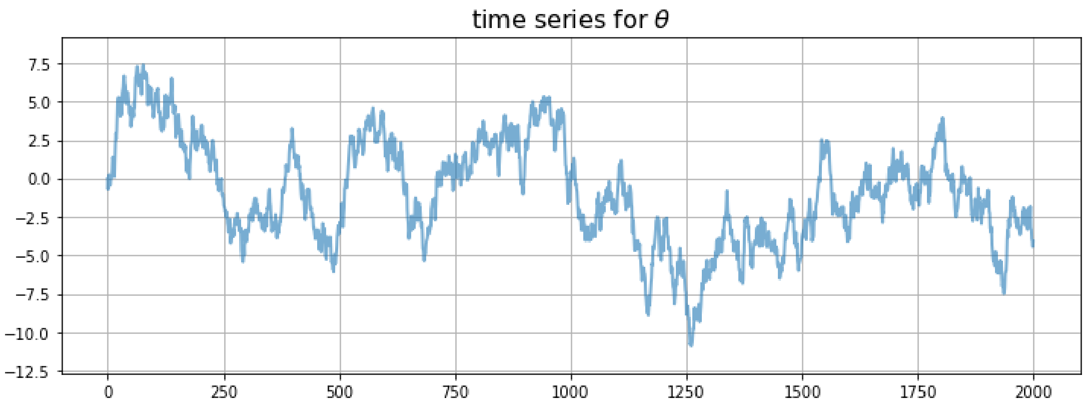
Fill in the details behind (2) and (3) based on the following standard result (see, e.g., p. 24 of [YS05]).

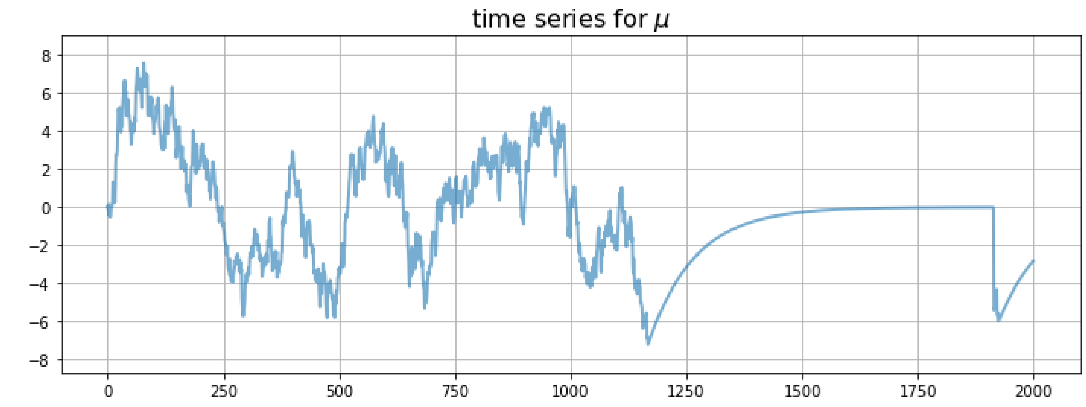
Fact Let $\mathbf{x} = (x_1, \ldots, x_M)$ be a vector of IID draws from common distribution $N(\theta, 1/\gamma_x)$ and let $\bar{x}$ be the sample mean. If γ_x is known and the prior for θ is $N(\mu, 1/\gamma)$, then the posterior distribution of θ given $\mathbf{x}$ is

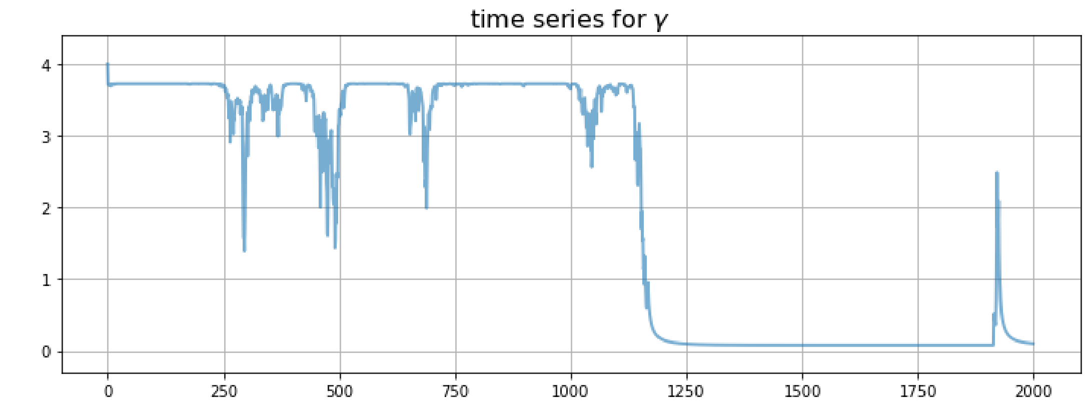
$$\pi(\theta \,|\, \mathbf{x}) = N(\mu_0, 1/\gamma_0)$$

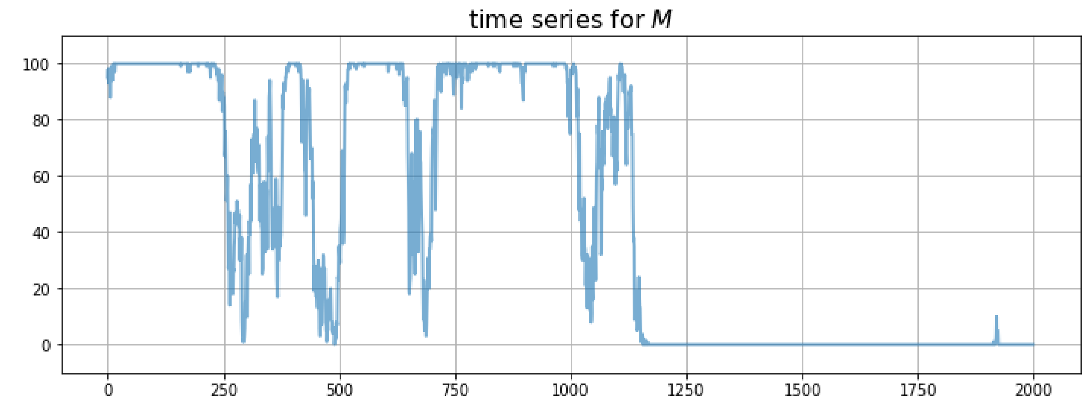
where

$$\mu_0 = \frac{\mu\gamma + M\bar{x}\gamma_x}{\gamma + M\gamma_x} \quad \text{and} \quad \gamma_0 = \gamma + M\gamma_x$$


time series for θ


time series for μ


time series for γ


time series for M

57.5.2 Exercise 2

Modulo randomness, replicate the simulation figures shown above.

- Use the parameter values listed as defaults in the **init** method of the UncertaintyTrapEcon class.

57.6 Solutions

57.6.1 Exercise 1

This exercise asked you to validate the laws of motion for γ and μ given in the lecture, based on the stated result about Bayesian updating in a scalar Gaussian setting. The stated result tells us that after observing average output X of the M firms, our posterior beliefs will be

$$N(\mu_0, 1/\gamma_0)$$

where

$$\mu_0 = \frac{\mu\gamma + MX\gamma_x}{\gamma + M\gamma_x} \quad \text{and} \quad \gamma_0 = \gamma + M\gamma_x$$

If we take a random variable θ with this distribution and then evaluate the distribution of $\rho\theta + \sigma_\theta w$ where w is independent and standard normal, we get the expressions for μ' and γ' given in the lecture.

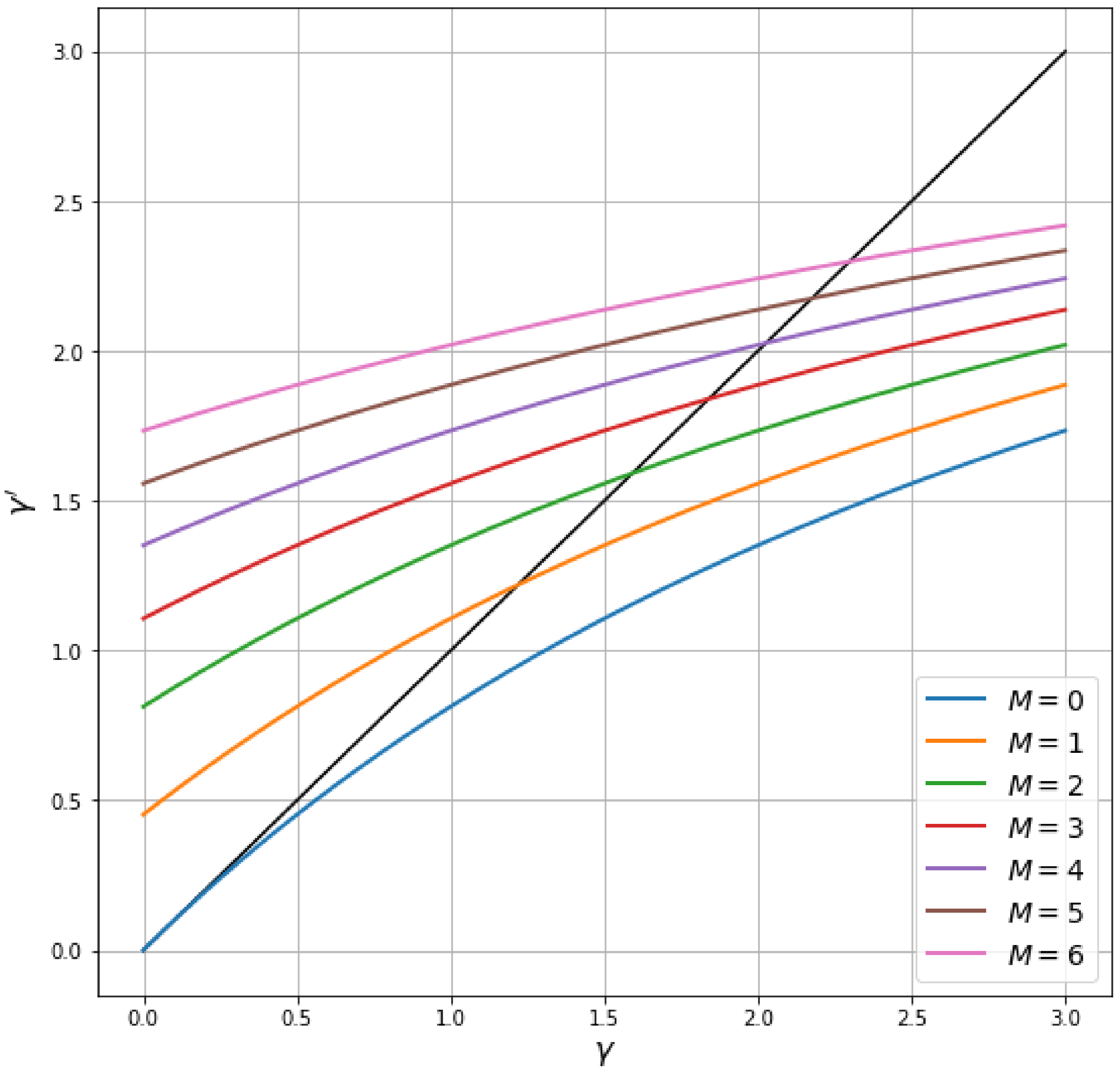
57.6.2 Exercise 2

First, let's replicate the plot that illustrates the law of motion for precision, which is

$$\gamma_{t+1} = \left(\frac{\rho^2}{\gamma_t + M\gamma_x} + \sigma_\theta^2\right)^{-1}$$

Here M is the number of active firms. The next figure plots γ_{t+1} against γ_t on a 45 degree diagram for different values of M

```
econ = UncertaintyTrapEcon()
ρ, σ_θ, γ_x = econ.ρ, econ.σ_θ, econ.γ_x    # Simplify names
γ = np.linspace(1e-10, 3, 200)              # γ grid
fig, ax = plt.subplots(figsize=(9, 9))
ax.plot(γ, γ, 'k-')                         # 45 degree line

for M in range(7):
    γ_next = 1 / (ρ**2 / (γ + M * γ_x) + σ_θ**2)
    label_string = f"$M = {M}$"
    ax.plot(γ, γ_next, lw=2, label=label_string)
ax.legend(loc='lower right', fontsize=14)
ax.set_xlabel(r'$\gamma$', fontsize=16)
ax.set_ylabel(r"$\gamma'$", fontsize=16)
ax.grid()
plt.show()
```



The points where the curves hit the 45 degree lines are the long-run steady states corresponding to each M, if that value of M was to remain fixed. As the number of firms falls, so does the long-run steady state of precision.

Next let's generate time series for beliefs and the aggregates – that is, the number of active firms and average output

```
sim_length=2000

μ_vec = np.empty(sim_length)
θ_vec = np.empty(sim_length)
γ_vec = np.empty(sim_length)
X_vec = np.empty(sim_length)
M_vec = np.empty(sim_length)

μ_vec[0] = econ.μ
γ_vec[0] = econ.γ
θ_vec[0] = 0
```

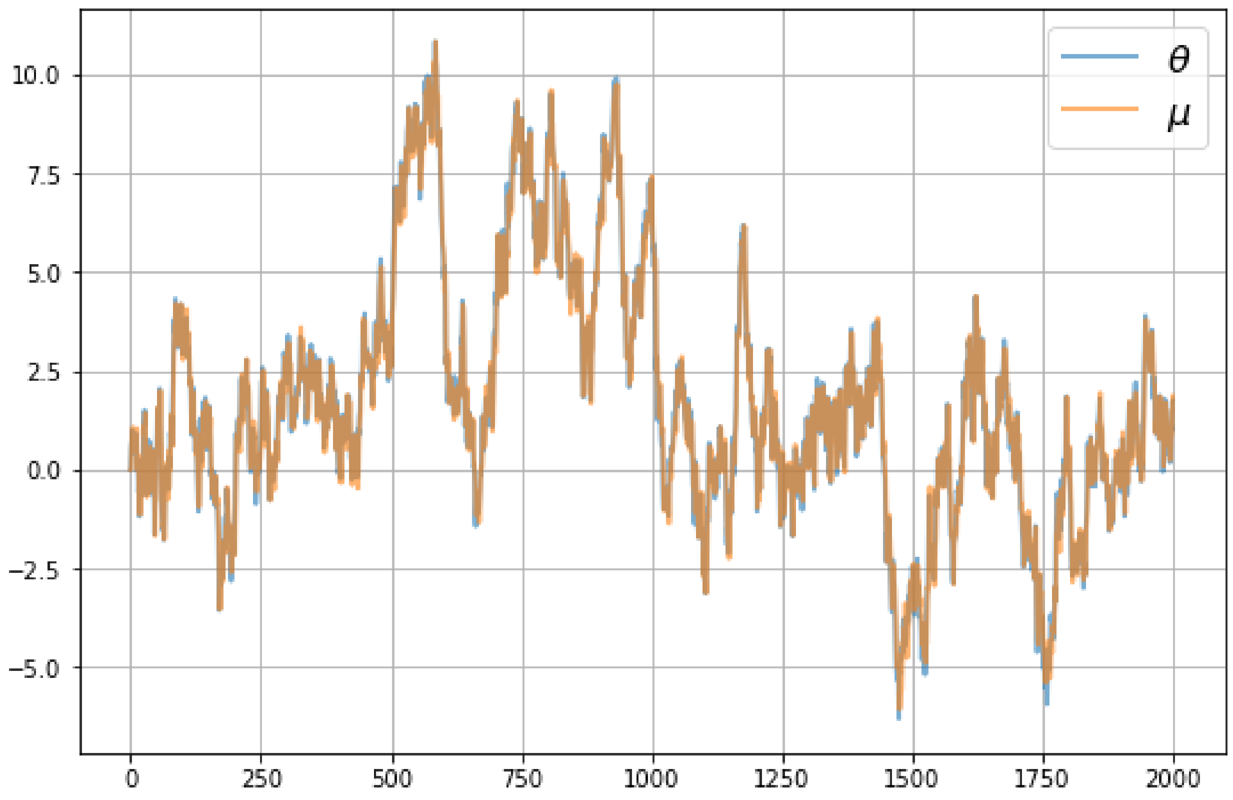
(continues on next page)

(continued from previous page)

```
w_shocks = np.random.randn(sim_length)

for t in range(sim_length-1):
    X, M = econ.gen_aggregates()
    X_vec[t] = X
    M_vec[t] = M

    econ.update_beliefs(X, M)
    econ.update_θ(w_shocks[t])

    μ_vec[t+1] = econ.μ
    γ_vec[t+1] = econ.γ
    θ_vec[t+1] = econ.θ

# Record final values of aggregates
X, M = econ.gen_aggregates()
X_vec[-1] = X
M_vec[-1] = M
```

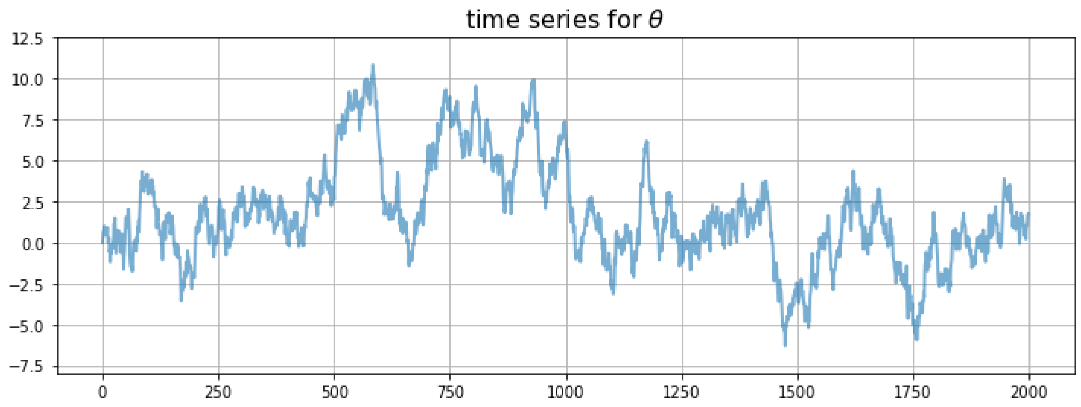
First, let's see how well $\mu$ tracks $\theta$ in these simulations

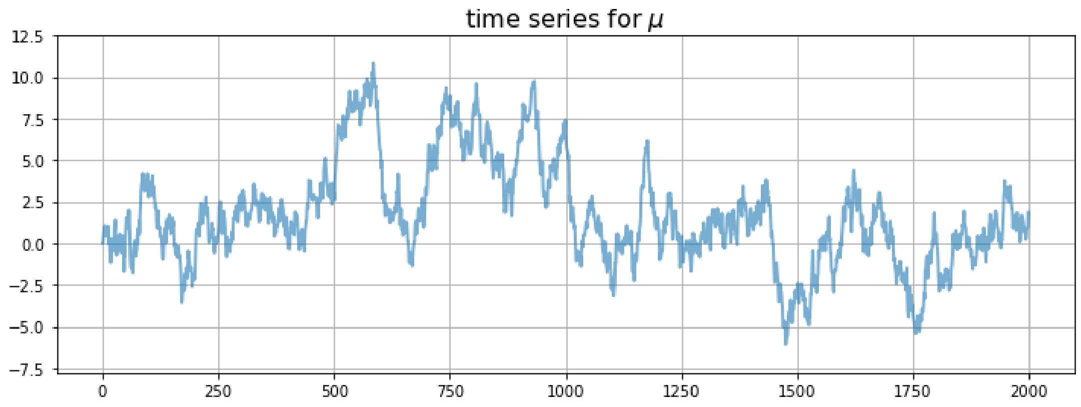
```
fig, ax = plt.subplots(figsize=(9, 6))
ax.plot(range(sim_length), θ_vec, alpha=0.6, lw=2, label=r"$\theta$")
ax.plot(range(sim_length), μ_vec, alpha=0.6, lw=2, label=r"$\mu$")
ax.legend(fontsize=16)
ax.grid()
plt.show()
```

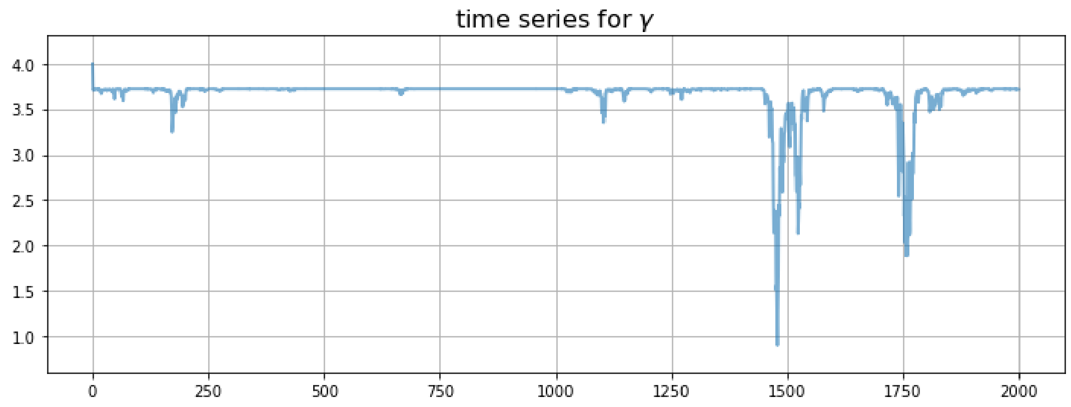


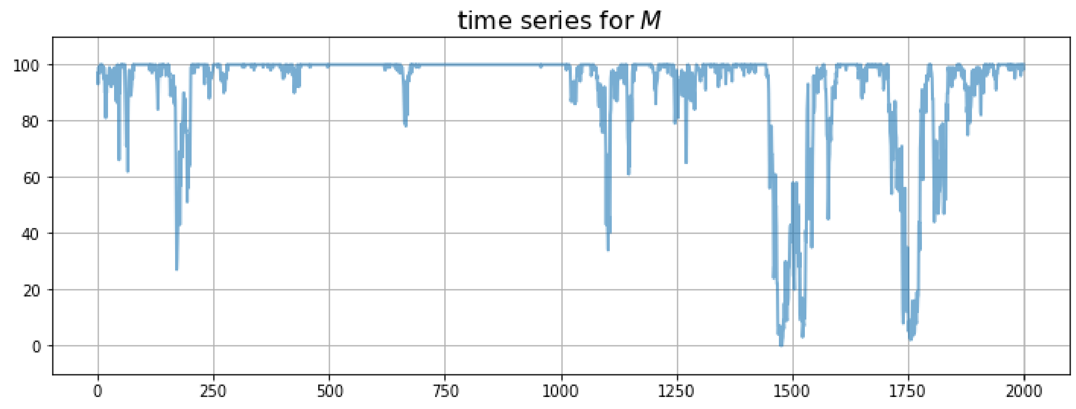
Now let's plot the whole thing together

```
fig, axes = plt.subplots(4, 1, figsize=(12, 20))
# Add some spacing
fig.subplots_adjust(hspace=0.3)

series = (θ_vec, μ_vec, γ_vec, M_vec)
names = r'$\theta$', r'$\mu$', r'$\gamma$', r'$M$'

for ax, vals, name in zip(axes, series, names):
    # Determine suitable y limits
    s_max, s_min = max(vals), min(vals)
    s_range = s_max - s_min
    y_max = s_max + s_range * 0.1
    y_min = s_min - s_range * 0.1
    ax.set_ylim(y_min, y_max)
    # Plot series
    ax.plot(range(sim_length), vals, alpha=0.6, lw=2)
    ax.set_title(f"time series for {name}", fontsize=16)
    ax.grid()

plt.show()
```


time series for θ
12.5
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
0
250
500
750
1000
1250
1500
1750
2000


time series for μ
12.5
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
0
250
500
750
1000
1250
1500
1750
2000


time series for γ
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0
250
500
750
1000
1250
1500
1750
2000


time series for M
100
80
60
40
20
0
0
250
500
750
1000
1250
1500
1750
2000

If you run the code above you'll get different plots, of course.

Try experimenting with different parameters to see the effects on the time series.

(It would also be interesting to experiment with non-Gaussian distributions for the shocks, but this is a big exercise since it takes us outside the world of the standard Kalman filter)

CHAPTER

FIFTYEIGHT

THE AIYAGARI MODEL

Contents

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

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

58.1 Overview

In this lecture, we describe the structure of a class of models that build on work by Truman Bewley [Bew77].

We begin by discussing an example of a Bewley model due to Rao Aiyagari.

The model features

- Heterogeneous agents
- A single exogenous vehicle for borrowing and lending
- Limits on amounts individual agents may borrow

The Aiyagari model has been used to investigate many topics, including

- precautionary savings and the effect of liquidity constraints [Aiy94]
- risk sharing and asset pricing [HL96]
- the shape of the wealth distribution [BBZ15]
- etc., etc., etc.

Let's start with some imports:

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

(continues on next page)

(continued from previous page)

```
import numpy as np
import quantecon as qe
from quantecon.markov import DiscreteDP
from numba import jit
```

58.1.1 References

The primary reference for this lecture is [Aiy94].

A textbook treatment is available in chapter 18 of [LS18].

A continuous time version of the model by SeHyoun Ahn and Benjamin Moll can be found here.

58.2 The Economy

58.2.1 Households

Infinitely lived households / consumers face idiosyncratic income shocks.

A unit interval of *ex-ante* identical households face a common borrowing constraint.

The savings problem faced by a typical household is

$$\max \mathbb{E} \sum_{t=0}^{\infty} \beta^t u(c_t)$$

subject to

$$a_{t+1} + c_t \leq w z_t + (1+r) a_t \quad c_t \geq 0, \quad \text{and} \quad a_t \geq -B$$

where

- c_t is current consumption
- a_t is assets
- z_t is an exogenous component of labor income capturing stochastic unemployment risk, etc.
- w is a wage rate
- r is a net interest rate
- B is the maximum amount that the agent is allowed to borrow

The exogenous process $\{z_t\}$ follows a finite state Markov chain with given stochastic matrix P.

The wage and interest rate are fixed over time.

In this simple version of the model, households supply labor inelastically because they do not value leisure.

58.3 Firms

Firms produce output by hiring capital and labor.

Firms act competitively and face constant returns to scale.

Since returns to scale are constant the number of firms does not matter.

Hence we can consider a single (but nonetheless competitive) representative firm.

The firm's output is

$$Y_t = A K_t^{\alpha} N^{1-\alpha}$$

where

- A and α are parameters with $A > 0$ and $\alpha \in (0, 1)$
- K_t is aggregate capital
- N is total labor supply (which is constant in this simple version of the model)

The firm's problem is

$$max_{K,N} \left\{ A K_t^{\alpha} N^{1-\alpha} - (r + \delta) K - w N \right\}$$

The parameter δ is the depreciation rate.

From the first-order condition with respect to capital, the firm's inverse demand for capital is

$$r = A\alpha \left(\frac{N}{K} \right)^{1-\alpha} - \delta \tag{1}$$

Using this expression and the firm's first-order condition for labor, we can pin down the equilibrium wage rate as a function of r as

$$w(r) = A(1-\alpha)(A\alpha/(r+\delta))^{\alpha/(1-\alpha)} \tag{2}$$

58.3.1 Equilibrium

We construct a *stationary rational expectations equilibrium* (SREE).

In such an equilibrium

- prices induce behavior that generates aggregate quantities consistent with the prices
- aggregate quantities and prices are constant over time

In more detail, an SREE lists a set of prices, savings and production policies such that

- households want to choose the specified savings policies taking the prices as given
- firms maximize profits taking the same prices as given
- the resulting aggregate quantities are consistent with the prices; in particular, the demand for capital equals the supply
- aggregate quantities (defined as cross-sectional averages) are constant

In practice, once parameter values are set, we can check for an SREE by the following steps

1. pick a proposed quantity K for aggregate capital

2. determine corresponding prices, with interest rate r determined by (1) and a wage rate $w(r)$ as given in (2)
3. determine the common optimal savings policy of the households given these prices
4. compute aggregate capital as the mean of steady state capital given this savings policy

If this final quantity agrees with K then we have a SREE.

58.4 Code

Let's look at how we might compute such an equilibrium in practice.

To solve the household's dynamic programming problem we'll use the DiscreteDP class from QuantEcon.py.

Our first task is the least exciting one: write code that maps parameters for a household problem into the `R` and `Q` matrices needed to generate an instance of `DiscreteDP`.

Below is a piece of boilerplate code that does just this.

In reading the code, the following information will be helpful

- `R` needs to be a matrix where `R[s, a]` is the reward at state `s` under action `a`.
- `Q` needs to be a three-dimensional array where `Q[s, a, s']` is the probability of transitioning to state `s'` when the current state is `s` and the current action is `a`.

(A more detailed discussion of `DiscreteDP` is available in the Discrete State Dynamic Programming lecture in the Advanced Quantitative Economics with Python lecture series.)

Here we take the state to be $s_t := (a_t, z_t)$, where a_t is assets and z_t is the shock.

The action is the choice of next period asset level a_{t+1}.

We use Numba to speed up the loops so we can update the matrices efficiently when the parameters change.

The class also includes a default set of parameters that we'll adopt unless otherwise specified.

```python
class Household:
    """
    This class takes the parameters that define a household asset accumulation
    problem and computes the corresponding reward and transition matrices R
    and Q required to generate an instance of DiscreteDP, and thereby solve
    for the optimal policy.

    Comments on indexing: We need to enumerate the state space S as a sequence
    S = {0, ..., n}.  To this end, (a_i, z_i) index pairs are mapped to s_i
    indices according to the rule

        s_i = a_i * z_size + z_i

    To invert this map, use

        a_i = s_i // z_size  (integer division)
        z_i = s_i % z_size

    """


    def __init__(self,
                r=0.01,                      # Interest rate
```

(continues on next page)

(continued from previous page)

```
                w=1.0,                          # Wages
                β=0.96,                         # Discount factor
                a_min=1e-10,
                Π=[[0.9, 0.1], [0.1, 0.9]],     # Markov chain
                z_vals=[0.1, 1.0],              # Exogenous states
                a_max=18,
                a_size=200):

        # Store values, set up grids over a and z
        self.r, self.w, self.β = r, w, β
        self.a_min, self.a_max, self.a_size = a_min, a_max, a_size

        self.Π = np.asarray(Π)
        self.z_vals = np.asarray(z_vals)
        self.z_size = len(z_vals)

        self.a_vals = np.linspace(a_min, a_max, a_size)
        self.n = a_size * self.z_size

        # Build the array Q
        self.Q = np.zeros((self.n, a_size, self.n))
        self.build_Q()

        # Build the array R
        self.R = np.empty((self.n, a_size))
        self.build_R()

    def set_prices(self, r, w):
        """
        Use this method to reset prices. Calling the method will trigger a
        re-build of R.
        """
        self.r, self.w = r, w
        self.build_R()

    def build_Q(self):
        populate_Q(self.Q, self.a_size, self.z_size, self.Π)

    def build_R(self):
        self.R.fill(-np.inf)
        populate_R(self.R,
                self.a_size,
                self.z_size,
                self.a_vals,
                self.z_vals,
                self.r,
                self.w)


# Do the hard work using JIT-ed functions

@jit(nopython=True)
def populate_R(R, a_size, z_size, a_vals, z_vals, r, w):
    n = a_size * z_size
    for s_i in range(n):
        a_i = s_i // z_size
        z_i = s_i % z_size
```

(continues on next page)

(continued from previous page)

```
        a = a_vals[a_i]
        z = z_vals[z_i]
        for new_a_i in range(a_size):
            a_new = a_vals[new_a_i]
            c = w * z + (1 + r) * a - a_new
            if c > 0:
                R[s_i, new_a_i] = np.log(c)  # Utility

@jit(nopython=True)
def populate_Q(Q, a_size, z_size, Π):
    n = a_size * z_size
    for s_i in range(n):
        z_i = s_i % z_size
        for a_i in range(a_size):
            for next_z_i in range(z_size):
                Q[s_i, a_i, a_i*z_size + next_z_i] = Π[z_i, next_z_i]


@jit(nopython=True)
def asset_marginal(s_probs, a_size, z_size):
    a_probs = np.zeros(a_size)
    for a_i in range(a_size):
        for z_i in range(z_size):
            a_probs[a_i] += s_probs[a_i*z_size + z_i]
    return a_probs
```

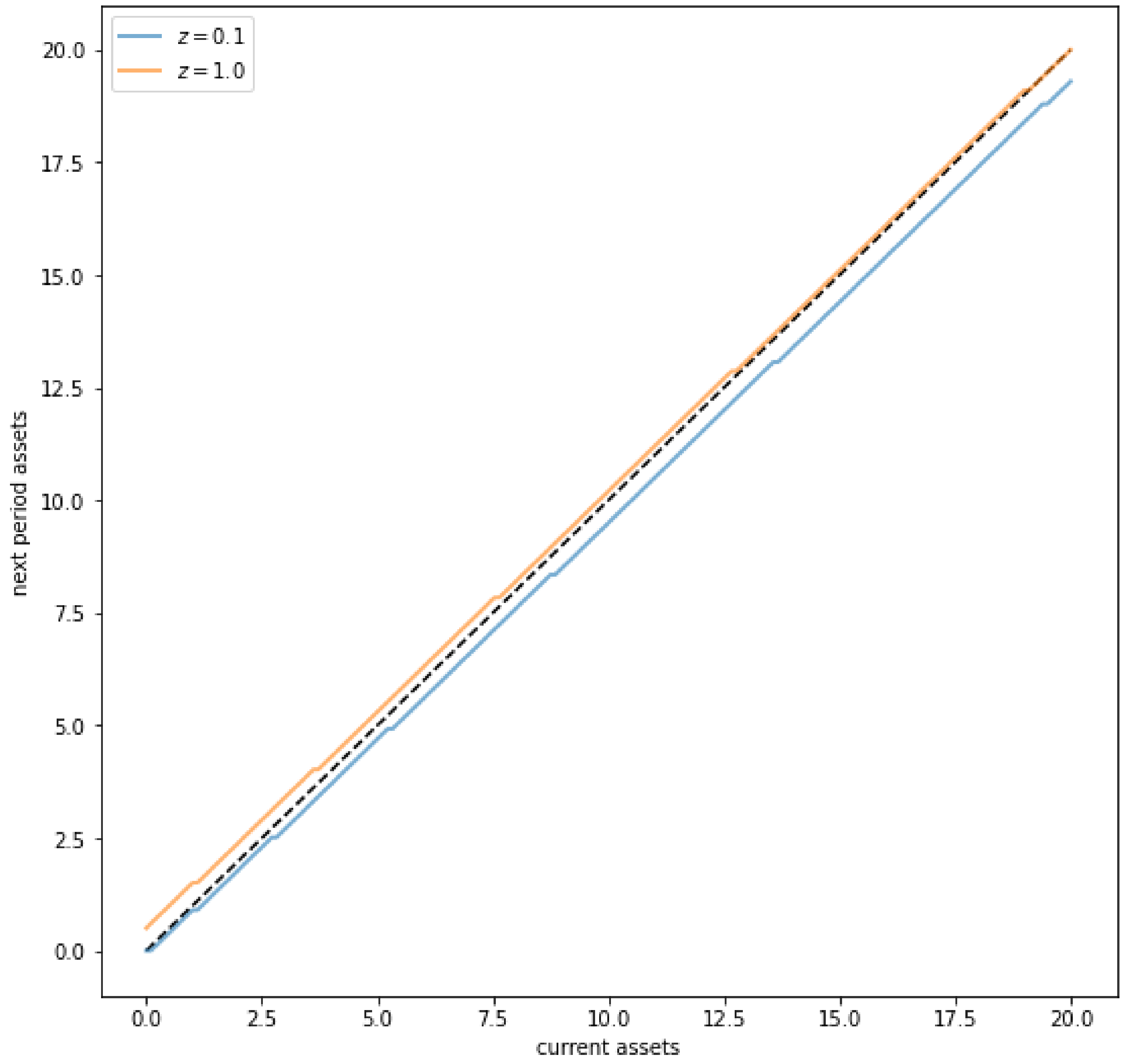
As a first example of what we can do, let's compute and plot an optimal accumulation policy at fixed prices.

```
# Example prices
r = 0.03
w = 0.956

# Create an instance of Household
am = Household(a_max=20, r=r, w=w)

# Use the instance to build a discrete dynamic program
am_ddp = DiscreteDP(am.R, am.Q, am.β)

# Solve using policy function iteration
results = am_ddp.solve(method='policy_iteration')

# Simplify names
z_size, a_size = am.z_size, am.a_size
z_vals, a_vals = am.z_vals, am.a_vals
n = a_size * z_size

# Get all optimal actions across the set of a indices with z fixed in each row
a_star = np.empty((z_size, a_size))
for s_i in range(n):
    a_i = s_i // z_size
    z_i = s_i % z_size
    a_star[z_i, a_i] = a_vals[results.sigma[s_i]]

fig, ax = plt.subplots(figsize=(9, 9))
ax.plot(a_vals, a_vals, 'k--')  # 45 degrees
for i in range(z_size):
    lb = f'$z = {z_vals[i]:.2}$'
```

(continues on next page)

(continued from previous page)

```
    ax.plot(a_vals, a_star[i, :], lw=2, alpha=0.6, label=lb)
    ax.set_xlabel('current assets')
    ax.set_ylabel('next period assets')
ax.legend(loc='upper left')

plt.show()
```



The plot shows asset accumulation policies at different values of the exogenous state.

Now we want to calculate the equilibrium.

Let's do this visually as a first pass.

The following code draws aggregate supply and demand curves.

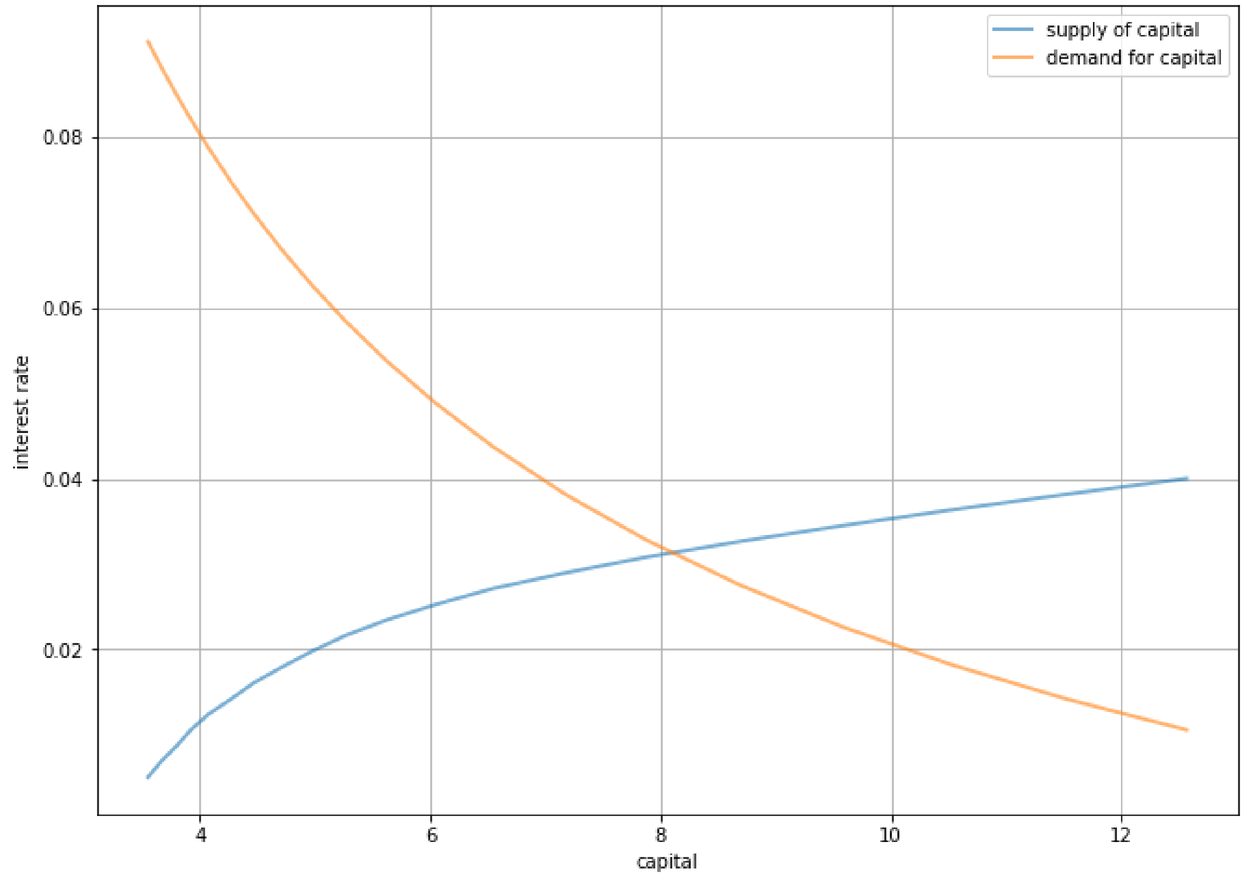
The intersection gives equilibrium interest rates and capital.

```
A = 1.0
N = 1.0
α = 0.33
β = 0.96
δ = 0.05


def r_to_w(r):
    """
    Equilibrium wages associated with a given interest rate r.
    """
    return A * (1 - α) * (A * α / (r + δ))**(α / (1 - α))

def rd(K):
    """
    Inverse demand curve for capital.  The interest rate associated with a
    given demand for capital K.
    """
    return A * α * (N / K)**(1 - α) - δ


def prices_to_capital_stock(am, r):
    """
    Map prices to the induced level of capital stock.

    Parameters:
    ----------

    am : Household
        An instance of an aiyagari_household.Household
    r : float
        The interest rate
    """
    w = r_to_w(r)
    am.set_prices(r, w)
    aiyagari_ddp = DiscreteDP(am.R, am.Q, β)
    # Compute the optimal policy
    results = aiyagari_ddp.solve(method='policy_iteration')
    # Compute the stationary distribution
    stationary_probs = results.mc.stationary_distributions[0]
    # Extract the marginal distribution for assets
    asset_probs = asset_marginal(stationary_probs, am.a_size, am.z_size)
    # Return K
    return np.sum(asset_probs * am.a_vals)


# Create an instance of Household
am = Household(a_max=20)

# Use the instance to build a discrete dynamic program
am_ddp = DiscreteDP(am.R, am.Q, am.β)

# Create a grid of r values at which to compute demand and supply of capital
num_points = 20
r_vals = np.linspace(0.005, 0.04, num_points)

# Compute supply of capital
```

(continues on next page)

(continued from previous page)

```
k_vals = np.empty(num_points)
for i, r in enumerate(r_vals):
    k_vals[i] = prices_to_capital_stock(am, r)

# Plot against demand for capital by firms
fig, ax = plt.subplots(figsize=(11, 8))
ax.plot(k_vals, r_vals, lw=2, alpha=0.6, label='supply of capital')
ax.plot(k_vals, rd(k_vals), lw=2, alpha=0.6, label='demand for capital')
ax.grid()
ax.set_xlabel('capital')
ax.set_ylabel('interest rate')
ax.legend(loc='upper right')

plt.show()
```


supply of capital
demand for capital
interest rate
capital