Equilibrium Models

Thomas J. Sargent and John Stachurski

May 03, 2024

CONTENTS

This website presents a set of lectures on equilibrium economic models.

Part I

Multiple Agent Models

CHAPTER

ONE

UNCERTAINTY TRAPS

Contents

1.1 Overview

In this lecture, we study a simplified version of an uncertainty traps model of Fajgelbaum, Schaal and Taschereau-Dumouchel [Fajgelbaum *et al.*, 2015].

The model features self-reinforcing uncertainty that has big impacts on economic activity.

In the model,

- Fundamentals vary stochastically and are not fully observable.
- At any moment there are both active and inactive entrepreneurs; only active entrepreneurs produce.
- Agents – active and inactive entrepreneurs – have beliefs about the fundamentals expressed as probability distributions.
- Greater uncertainty means greater dispersions of these distributions.
- Entrepreneurs are risk-averse and hence less inclined to be active when uncertainty is high.
- The output of active entrepreneurs is observable, supplying a noisy signal that helps everyone inside the model infer fundamentals.
- Entrepreneurs update their beliefs about fundamentals using Bayes' Law, implemented via Kalman filtering.

Uncertainty traps emerge because:

- High uncertainty discourages entrepreneurs from becoming active.
- A low level of participation – i.e., a smaller number of active entrepreneurs – diminishes the flow of information about fundamentals.

- Less information translates to higher uncertainty, further discouraging entrepreneurs from choosing to be active, and so on.

Uncertainty traps stem from a positive externality: high aggregate economic activity levels generates valuable information.

Let's start with some standard imports:

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

1.2 The Model

The original model described in [Fajgelbaum *et al.*, 2015] has many interesting moving parts.

Here we examine a simplified version that nonetheless captures many of the key ideas.

1.2.1 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.

1.2.2 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

$$x_m = \theta + \epsilon_m \quad \text{where} \quad \epsilon_m \sim N\left(0, \gamma_x^{-1}\right) \tag{1.1}$$

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.

1.2.3 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

$$\mu' = \rho \frac{\gamma \mu + M \gamma_x X}{\gamma + M \gamma_x} \tag{1.2}$$

$$\gamma' = \left(\frac{\rho^2}{\gamma + M \gamma_x} + \sigma_\theta^2 \right)^{-1} \tag{1.3}$$

These are standard Kalman filtering results applied to the current setting.

Exercise 1 provides more details on how (1.2) and (1.3) are derived and then asks you to fill in remaining steps.

The next figure plots the law of motion for the precision in (1.3) 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.

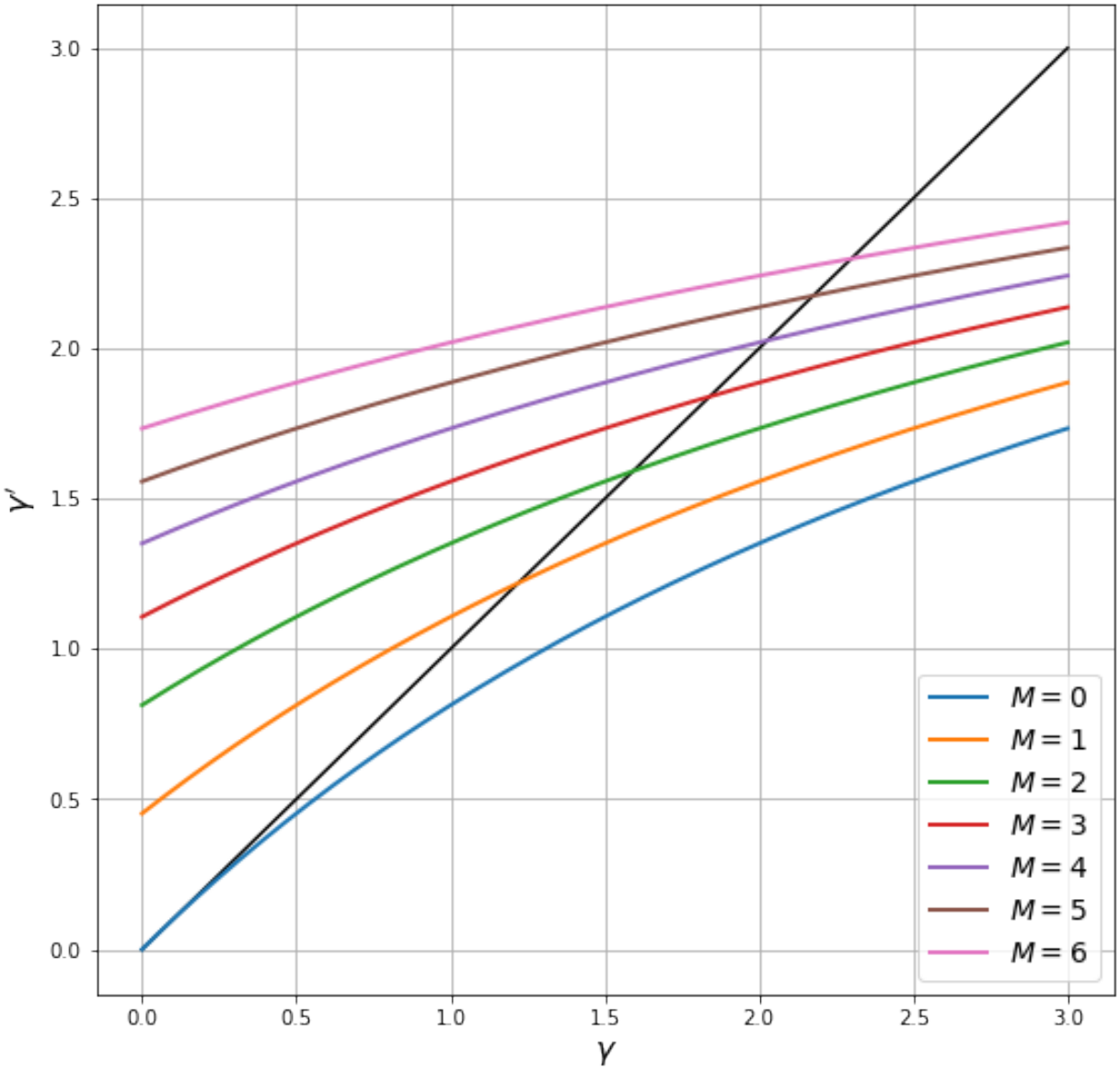
1.2.4 Participation

Omitting time subscripts once more, entrepreneurs enter the market in the current period if

$$\mathbb{E}[u(x_m - F_m)] > c \tag{1.4}$$

Here

- the mathematical expectation of x_m is based on (1.1) 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
Y'
0.0
0.5
1.0
1.5
2.0
2.5
3.0
Y
$M = 0$
$M = 1$
$M = 2$
$M = 3$
$M = 4$
$M = 5$
$M = 6$

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 (1.4).

The utility function has the constant absolute risk aversion form

$$u(x) = \frac{1}{a}\left(1 - \exp(-ax)\right) \tag{1.5}$$

where a is a positive parameter.

Combining (1.4) and (1.5), entrepreneur m participates in the market (or is said to be active) when

$$\frac{1}{a}\left\{1 - \mathbb{E}[\exp\left(-a(\theta + \epsilon_m - F_m)\right)]\right\} > c$$

Using standard formulas for expectations of lognormal random variables, this is equivalent to the condition

$$\psi(\mu, \gamma, F_m) := \frac{1}{a}\left(1 - \exp\left(-a\mu + aF_m + \frac{a^2\left(\frac{1}{\gamma} + \frac{1}{\gamma_x}\right)}{2}\right)\right) - c > 0 \tag{1.6}$$

1.3 Implementation

We want to simulate this economy.

As a first step, let's put together a class that bundles

- the parameters, the current value of θ and the current values of the two belief parameters μ and γ
- methods to update θ, μ and γ, as well as to determine the number of active firms and their outputs

The updating methods follow the laws of motion for θ, μ and γ given above.

The method to evaluate the number of active firms generates $F_1, \ldots, F_{\bar{M}}$ and tests condition (1.6) 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.

1.4 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.

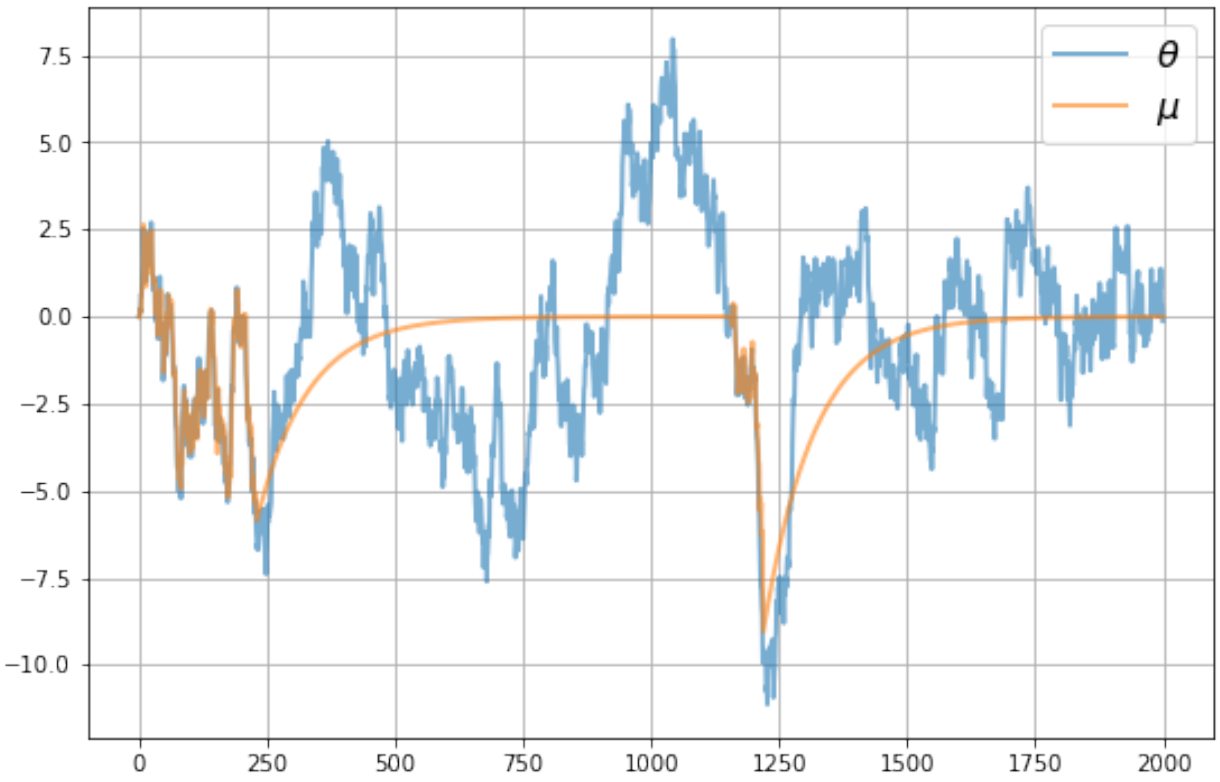
These are episodes where the uncertainty traps take hold.

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.

1.5 Exercises

Exercise 1.5.1

Fill in the details behind (1.2) and (1.3) based on the following standard result (see, e.g., p. 24 of [Young and Smith, 2005]).

Fact Let $\mathbf{x} = (x_1, \ldots, x_M)$ be a vector of IID draws from common distribution $N(\theta, 1/\gamma_x)$ and let $\bar{x}$ be the sample mean. If γ_x is known and the prior for θ is $N(\mu, 1/\gamma)$, then the posterior distribution of θ given $\mathbf{x}$ is

$$\pi(\theta \,|\, \mathbf{x}) = N(\mu_0, 1/\gamma_0)$$

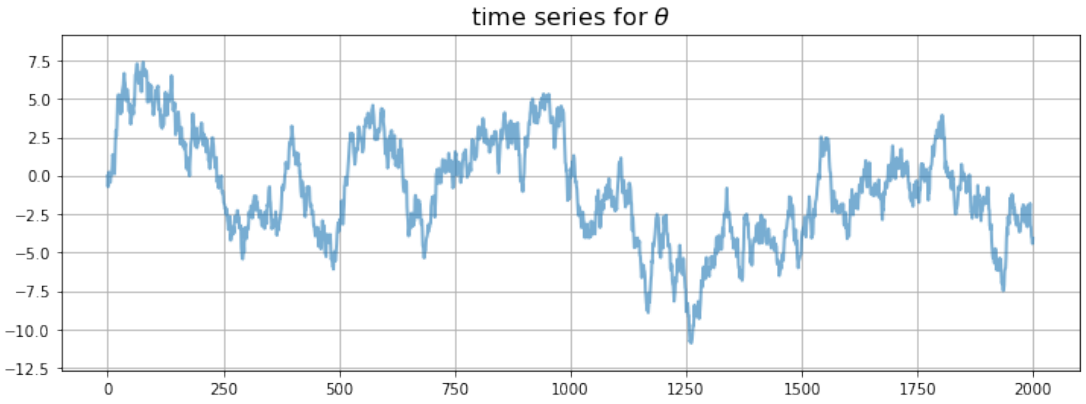
where

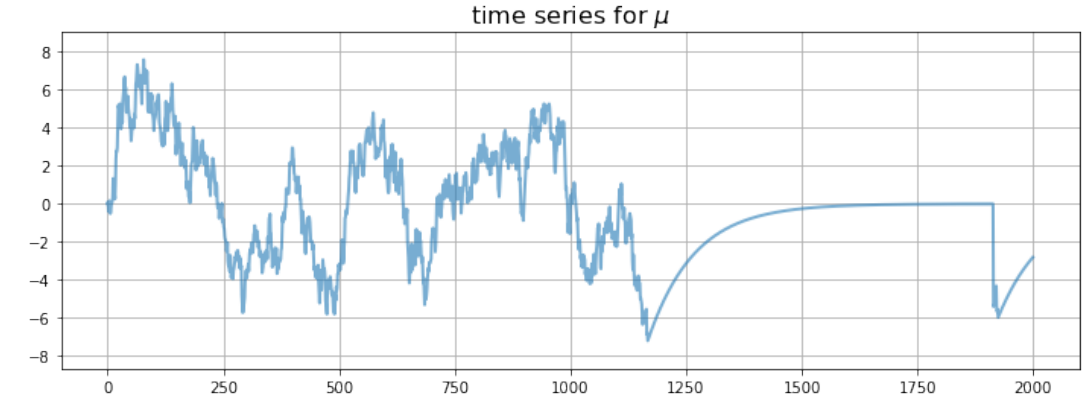
$$\mu_0 = \frac{\mu\gamma + M\bar{x}\gamma_x}{\gamma + M\gamma_x} \quad \text{and} \quad \gamma_0 = \gamma + M\gamma_x$$

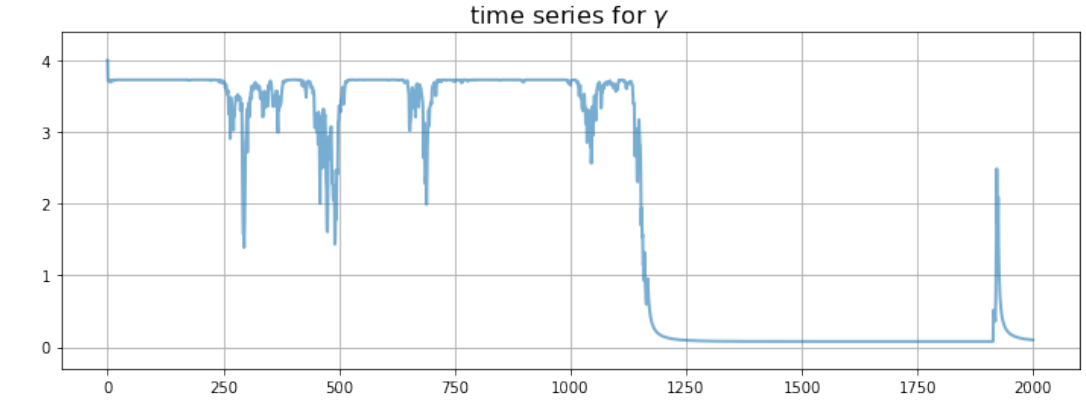
Solution to Exercise 1.5.1

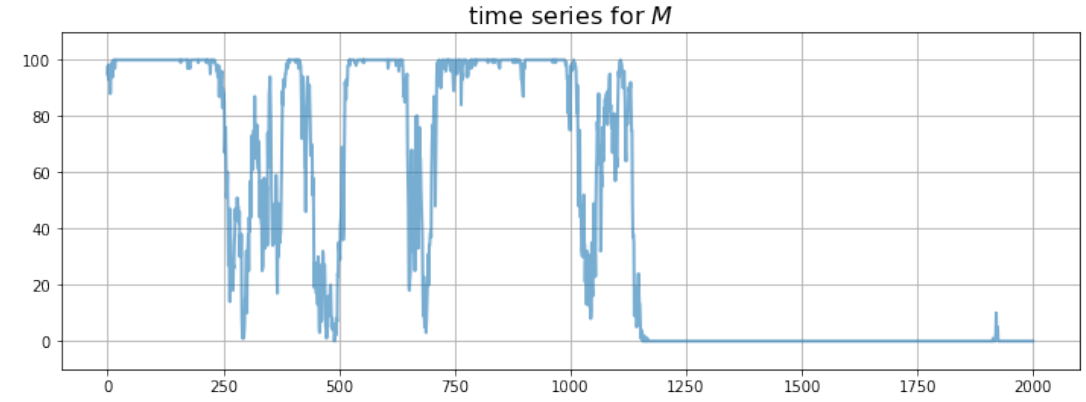
This exercise asked you to validate the laws of motion for γ and μ given in the lecture, based on the stated result about Bayesian updating in a scalar Gaussian setting. The stated result tells us that after observing average output X of the M firms, our posterior beliefs will be

$$N(\mu_0, 1/\gamma_0)$$


time series for θ
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−12.5
0
250
500
750
1000
1250
1500
1750
2000


time series for μ
8
6
4
2
0
−2
−4
−6
−8
0
250
500
750
1000
1250
1500
1750
2000


time series for γ
4
3
2
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

where

$$\mu_0 = \frac{\mu\gamma + MX\gamma_x}{\gamma + M\gamma_x} \quad \text{and} \quad \gamma_0 = \gamma + M\gamma_x$$

If we take a random variable θ with this distribution and then evaluate the distribution of $\rho\theta + \sigma_\theta w$ where w is independent and standard normal, we get the expressions for μ' and γ' given in the lecture.

Exercise 1.5.2

Modulo randomness, replicate the simulation figures shown above.

- Use the parameter values listed as defaults in the **init** method of the UncertaintyTrapEcon class.

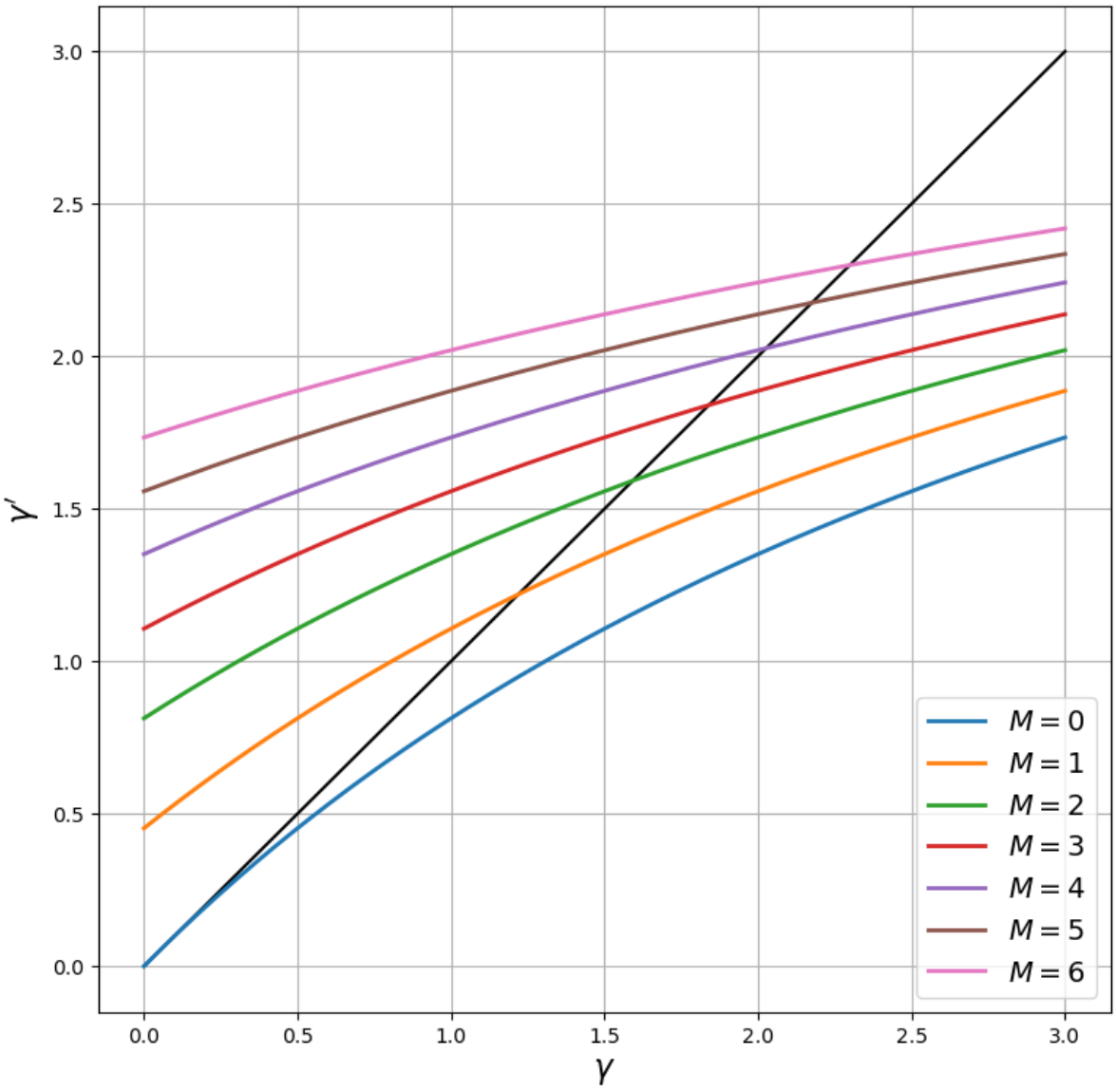
Solution to Exercise 1.5.2

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
```

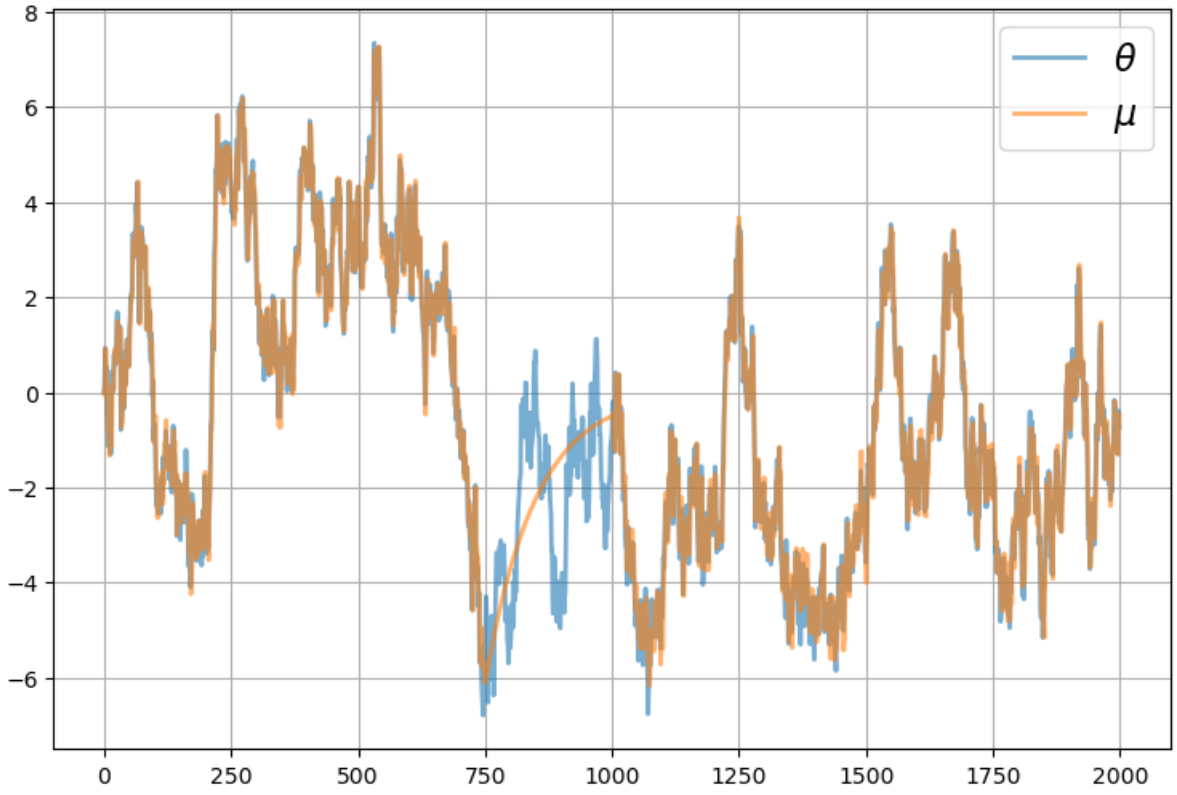
(continues on next page)

(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
    y_vec[t+1] = econ.y
    θ_vec[t+1] = econ.θ

# Record final values of aggregates
X, M = econ.gen_aggregates()
X_vec[-1] = X
M_vec[-1] = M
```

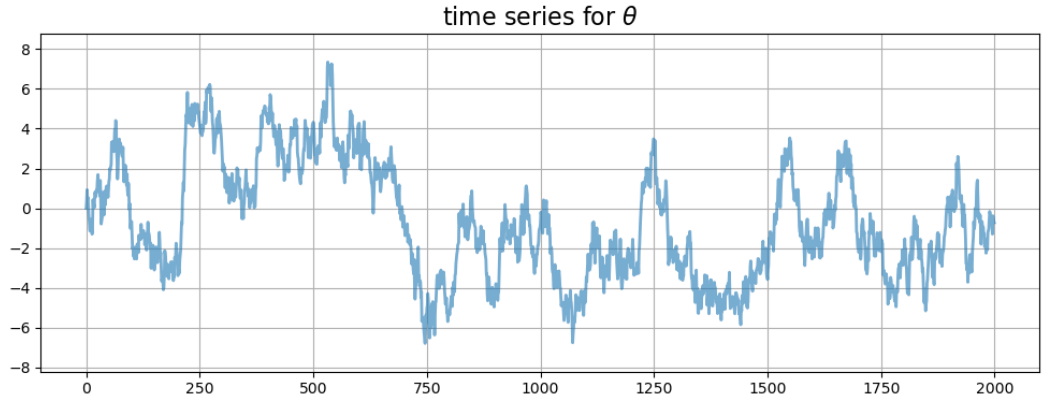
First, let's see how well μ tracks θ in these simulations

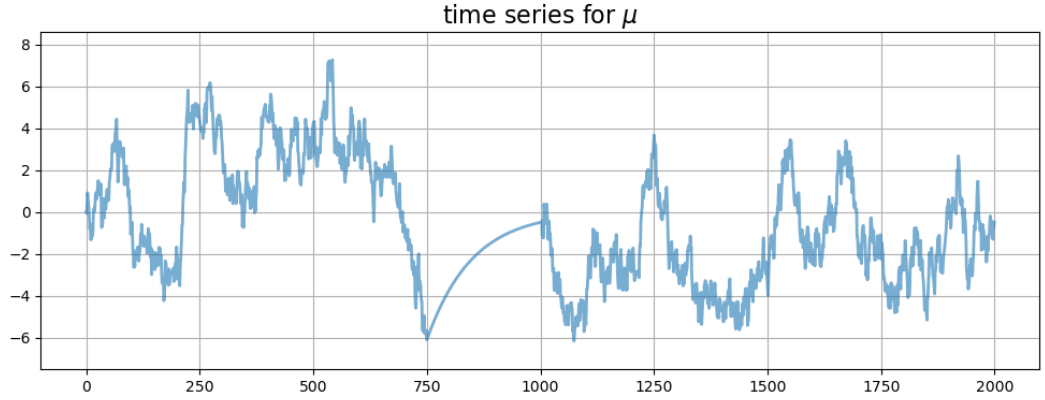
```
fig, ax = plt.subplots(figsize=(9, 6))
ax.plot(range(sim_length), θ_vec, alpha=0.6, lw=2, label=r"$\theta$")
ax.plot(range(sim_length), μ_vec, alpha=0.6, lw=2, label=r"$\mu$")
ax.legend(fontsize=16)
ax.grid()
plt.show()
```

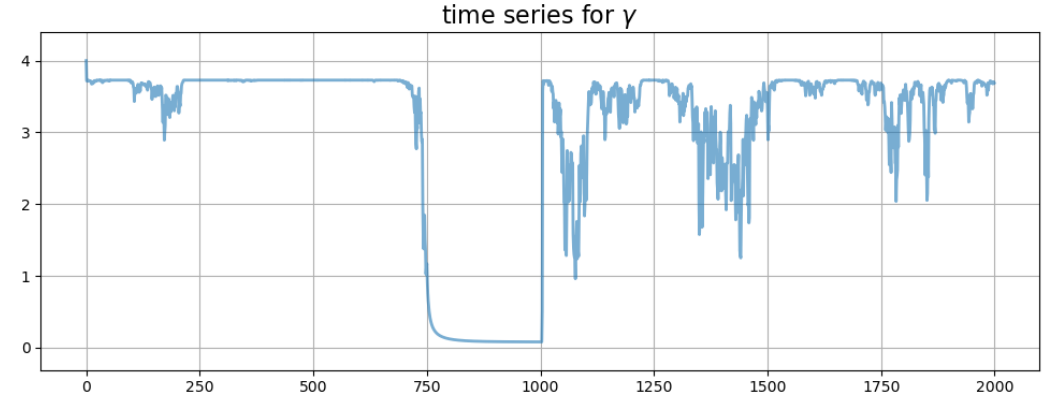


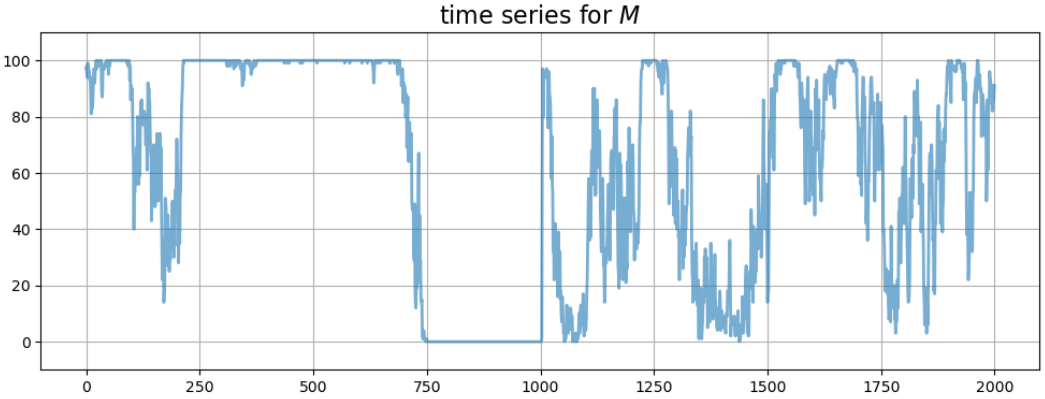
Now let's plot the whole thing together

```
fig, axes = plt.subplots(4, 1, figsize=(12, 20))
# Add some spacing
fig.subplots_adjust(hspace=0.3)

series = (θ_vec, μ_vec, γ_vec, M_vec)
names = r'$\theta$', r'$\mu$', r'$\gamma$', r'$M$'

for ax, vals, name in zip(axes, series, names):
    # Determine suitable y limits
    s_max, s_min = max(vals), min(vals)
    s_range = s_max - s_min
    y_max = s_max + s_range * 0.1
    y_min = s_min - s_range * 0.1
    ax.set_ylim(y_min, y_max)
    # Plot series
    ax.plot(range(sim_length), vals, alpha=0.6, lw=2)
    ax.set_title(f"time series for {name}", fontsize=16)
    ax.grid()

plt.show()
```


time series for θ
8
6
4
2
0
−2
−4
−6
−8
0
250
500
750
1000
1250
1500
1750
2000


time series for μ
8
6
4
2
0
−2
−4
−6
0
250
500
750
1000
1250
1500
1750
2000


time series for γ
4
3
2
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

TWO

THE AIYAGARI MODEL

Contents

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

```
!pip install quantecon
```

2.1 Overview

In this lecture, we describe the structure of a class of models that build on work by Truman Bewley [Bewley, 1977].

We begin by discussing an example of a Bewley model due to Rao Aiyagari [Aiyagari, 1994].

The model features

- Heterogeneous agents
- A single exogenous vehicle for borrowing and lending
- Limits on amounts individual agents may borrow

The Aiyagari model has been used to investigate many topics, including

- precautionary savings and the effect of liquidity constraints [Aiyagari, 1994]
- risk sharing and asset pricing [Heaton and Lucas, 1996]
- the shape of the wealth distribution [Benhabib *et al.*, 2015]
- etc., etc., etc.

Let's start with some imports:

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from quantecon.markov import DiscreteDP
from numba import jit
```

2.1.1 References

The primary reference for this lecture is [Aiyagari, 1994].

A textbook treatment is available in chapter 18 of [Ljungqvist and Sargent, 2018].

A continuous time version of the model by SeHyoun Ahn and Benjamin Moll can be found here.

2.2 The Economy

2.2.1 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.

2.3 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

$$Y_t = AK_t^\alpha N^{1-\alpha}$$

where

- A and α are parameters with $A > 0$ and $\alpha \in (0, 1)$
- K_t is aggregate capital
- N is total labor supply (which is constant in this simple version of the model)

The firm's problem is

$$max_{K,N} \left\{ AK_t^\alpha N^{1-\alpha} - (r + \delta)K - wN \right\}$$

The parameter δ is the depreciation rate.

From the first-order condition with respect to capital, the firm's inverse demand for capital is

$$r = A\alpha \left(\frac{N}{K} \right)^{1-\alpha} - \delta \tag{2.1}$$

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

$$w(r) = A(1 - \alpha)(A\alpha/(r + \delta))^{\alpha/(1-\alpha)} \tag{2.2}$$

2.3.1 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 (2.1) and a wage rate $w(r)$ as given in (2.2)
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.

2.4 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.

```
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
```

(continues on next page)

(continued from previous page)

```
        z_i = s_i % z_size
        a = a_vals[a_i]
        z = z_vals[z_i]
        for new_a_i in range(a_size):
            a_new = a_vals[new_a_i]
            c = w * z + (1 + r) * a - a_new
            if c > 0:
                R[s_i, new_a_i] = np.log(c)  # Utility

@jit(nopython=True)
def populate_Q(Q, a_size, z_size, Π):
    n = a_size * z_size
    for s_i in range(n):
        z_i = s_i % z_size
        for a_i in range(a_size):
            for next_z_i in range(z_size):
                Q[s_i, a_i, a_i*z_size + next_z_i] = Π[z_i, next_z_i]


@jit(nopython=True)
def asset_marginal(s_probs, a_size, z_size):
    a_probs = np.zeros(a_size)
    for a_i in range(a_size):
        for z_i in range(z_size):
            a_probs[a_i] += s_probs[a_i*z_size + z_i]
    return a_probs
```

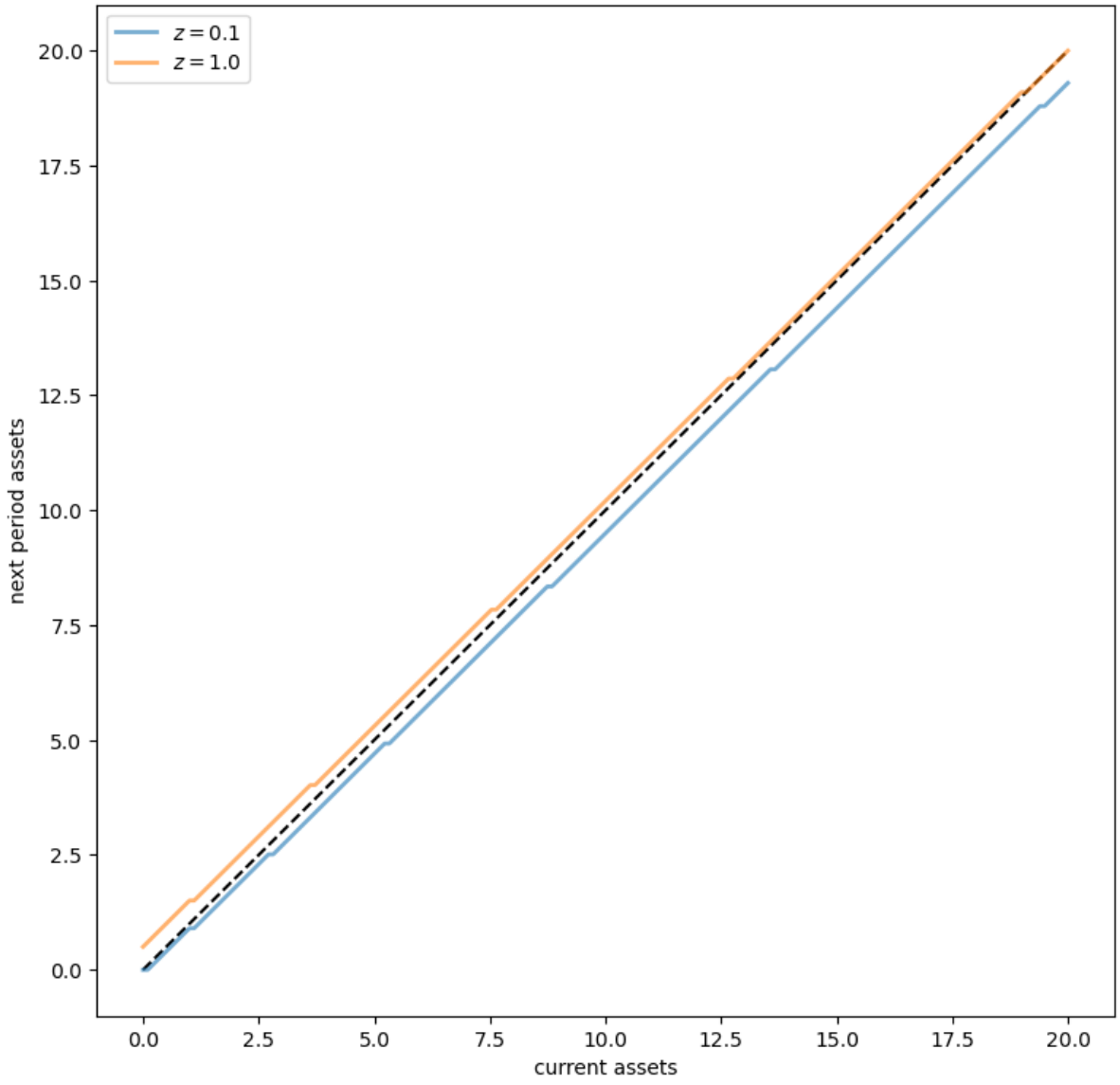
As a first example of what we can do, let's compute and plot an optimal accumulation policy at fixed prices.

```
# 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
```

(continues on next page)

(continued from previous page)

```
for i in range(z_size):
    lb = f'$z = {z_vals[i]:.2}$'
    ax.plot(a_vals, a_star[i, :], lw=2, alpha=0.6, label=lb)
    ax.set_xlabel('current assets')
    ax.set_ylabel('next period assets')
ax.legend(loc='upper left')

plt.show()
```



The plot shows asset accumulation policies at different values of the exogenous state.

Now we want to calculate the equilibrium.

Let's do this visually as a first pass.

The following code draws aggregate supply and demand curves.

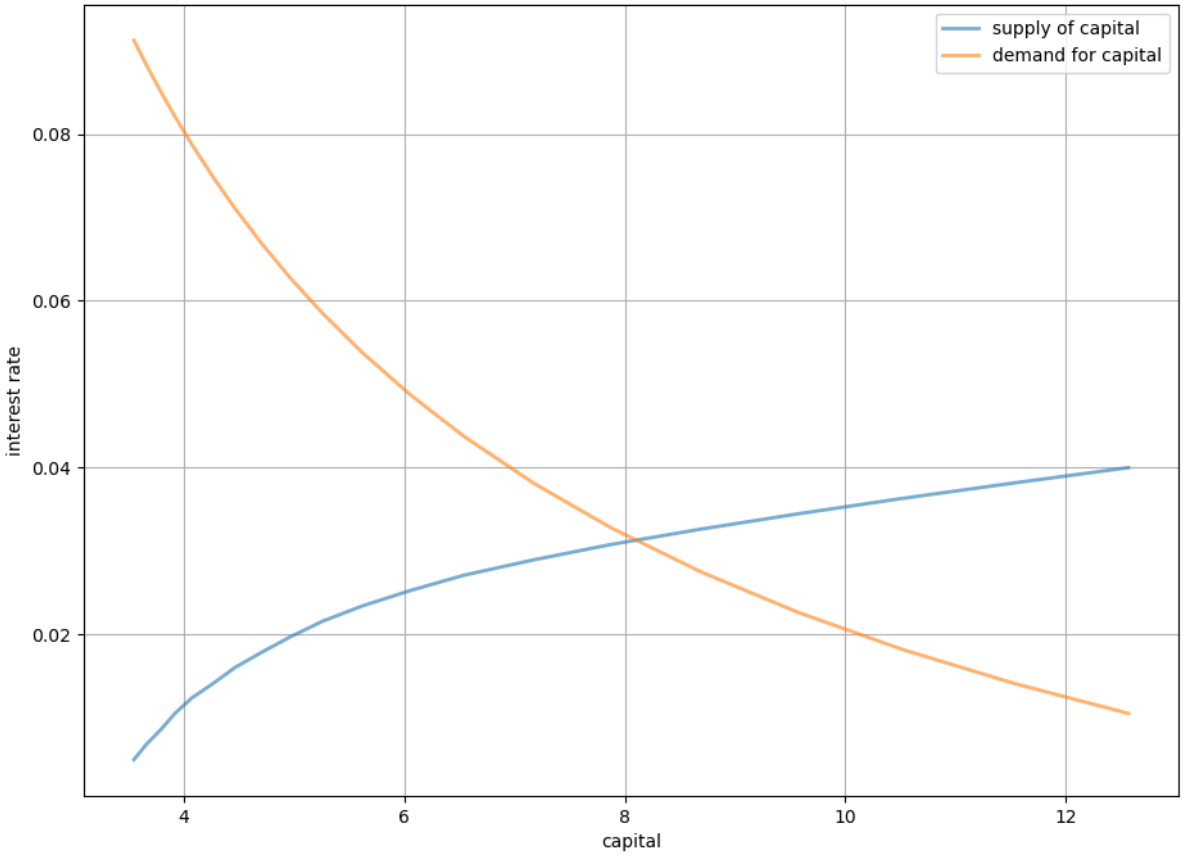
The intersection gives equilibrium interest rates and capital.

```
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()
```



CHAPTER

THREE

DEFAULT RISK AND INCOME FLUCTUATIONS

Contents

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

```
!pip install --upgrade quantecon
```

3.1 Overview

This lecture computes versions of Arellano's [Arellano, 2008] model of sovereign default.

The model describes interactions among default risk, output, and an equilibrium interest rate that includes a premium for endogenous default risk.

The decision maker is a government of a small open economy that borrows from risk-neutral foreign creditors.

The foreign lenders must be compensated for default risk.

The government borrows and lends abroad in order to smooth the consumption of its citizens.

The government repays its debt only if it wants to, but declining to pay has adverse consequences.

The interest rate on government debt adjusts in response to the state-dependent default probability chosen by government.

The model yields outcomes that help interpret sovereign default experiences, including

- countercyclical interest rates on sovereign debt
- countercyclical trade balances
- high volatility of consumption relative to output

Notably, long recessions caused by bad draws in the income process increase the government's incentive to default.

This can lead to

- spikes in interest rates
- temporary losses of access to international credit markets
- large drops in output, consumption, and welfare
- large capital outflows during recessions

Such dynamics are consistent with experiences of many countries.

Let's start with some imports:

```
import matplotlib.pyplot as plt
import numpy as np
import quantecon as qe

from numba import njit, prange
%matplotlib inline
```

3.2 Structure

In this section we describe the main features of the model.

3.2.1 Output, Consumption and Debt

A small open economy is endowed with an exogenous stochastically fluctuating potential output stream $\{y_t\}$.

Potential output is realized only in periods in which the government honors its sovereign debt.

The output good can be traded or consumed.

The sequence $\{y_t\}$ is described by a Markov process with stochastic density kernel $p(y, y')$.

Households within the country are identical and rank stochastic consumption streams according to

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t u(c_t) \tag{3.1}$$

Here

- $0 < \beta < 1$ is a time discount factor
- u is an increasing and strictly concave utility function

Consumption sequences enjoyed by households are affected by the government's decision to borrow or lend internationally.

The government is benevolent in the sense that its aim is to maximize (3.1).

The government is the only domestic actor with access to foreign credit.

Because household are averse to consumption fluctuations, the government will try to smooth consumption by borrowing from (and lending to) foreign creditors.

3.2.2 Asset Markets

The only credit instrument available to the government is a one-period bond traded in international credit markets.

The bond market has the following features

- The bond matures in one period and is not state contingent.
- A purchase of a bond with face value B' is a claim to B' units of the consumption good next period.
- To purchase B' next period costs qB' now, or, what is equivalent.
- For selling $-B'$ units of next period goods the seller earns $-qB'$ of today's goods.
  - If $B' < 0$, then $-qB'$ units of the good are received in the current period, for a promise to repay $-B'$ units next period.
  - There is an equilibrium price function $q(B', y)$ that makes q depend on both B' and y.

Earnings on the government portfolio are distributed (or, if negative, taxed) lump sum to households.

When the government is not excluded from financial markets, the one-period national budget constraint is

$$c = y + B - q(B', y)B' \tag{3.2}$$

Here and below, a prime denotes a next period value or a claim maturing next period.

To rule out Ponzi schemes, we also require that $B \geq -Z$ in every period.

- Z is chosen to be sufficiently large that the constraint never binds in equilibrium.

3.2.3 Financial Markets

Foreign creditors

- are risk neutral
- know the domestic output stochastic process $\{y_t\}$ and observe $y_t, y_{t-1}, \ldots$, at time t
- can borrow or lend without limit in an international credit market at a constant international interest rate r
- receive full payment if the government chooses to pay
- receive zero if the government defaults on its one-period debt due

When a government is expected to default next period with probability δ, the expected value of a promise to pay one unit of consumption next period is $1 - \delta$.

Therefore, the discounted expected value of a promise to pay B next period is

$$q = \frac{1 - \delta}{1 + r} \tag{3.3}$$

Next we turn to how the government in effect chooses the default probability δ.

3.2.4 Government's Decisions

At each point in time t, the government chooses between

1. defaulting
2. meeting its current obligations and purchasing or selling an optimal quantity of one-period sovereign debt

Defaulting means declining to repay all of its current obligations.

If the government defaults in the current period, then consumption equals current output.

But a sovereign default has two consequences:

1. Output immediately falls from y to $h(y)$, where $0 \leq h(y) \leq y$.
    - It returns to y only after the country regains access to international credit markets.
2. The country loses access to foreign credit markets.

3.2.5 Reentering International Credit Market

While in a state of default, the economy regains access to foreign credit in each subsequent period with probability θ.

3.3 Equilibrium

Informally, an equilibrium is a sequence of interest rates on its sovereign debt, a stochastic sequence of government default decisions and an implied flow of household consumption such that

1. Consumption and assets satisfy the national budget constraint.
2. The government maximizes household utility taking into account
    - the resource constraint
    - the effect of its choices on the price of bonds
    - consequences of defaulting now for future net output and future borrowing and lending opportunities
3. The interest rate on the government's debt includes a risk-premium sufficient to make foreign creditors expect on average to earn the constant risk-free international interest rate.

To express these ideas more precisely, consider first the choices of the government, which

1. enters a period with initial assets B, or what is the same thing, initial debt to be repaid now of $-B$
2. observes current output y, and
3. chooses either
    1. to default, or
    2. to pay $-B$ and set next period's debt due to $-B'$

In a recursive formulation,

- state variables for the government comprise the pair (B, y)
- $v(B, y)$ is the optimum value of the government's problem when at the beginning of a period it faces the choice of whether to honor or default
- $v_c(B, y)$ is the value of choosing to pay obligations falling due
- $v_d(y)$ is the value of choosing to default

$v_d(y)$ does not depend on B because, when access to credit is eventually regained, net foreign assets equal 0.

Expressed recursively, the value of defaulting is

$$v_d(y) = u(h(y)) + \beta \int \{\theta v(0, y') + (1-\theta) v_d(y')\} \, p(y, y') dy'$$

The value of paying is

$$v_c(B, y) = \max_{B' \geq -Z} \left\{ u(y - q(B', y)B' + B) + \beta \int v(B', y') p(y, y') dy' \right\}$$

The three value functions are linked by

$$v(B, y) = \max\{v_c(B, y), v_d(y)\}$$

The government chooses to default when

$$v_c(B, y) < v_d(y)$$

and hence given B' the probability of default next period is

$$\delta(B', y) := \int \mathbb{1}\{v_c(B', y') < v_d(y')\} p(y, y') dy' \tag{3.4}$$

Given zero profits for foreign creditors in equilibrium, we can combine (3.3) and (3.4) to pin down the bond price function:

$$q(B', y) = \frac{1 - \delta(B', y)}{1 + r} \tag{3.5}$$

3.3.1 Definition of Equilibrium

An *equilibrium* is

- a pricing function $q(B', y)$,
- a triple of value functions $(v_c(B, y), v_d(y), v(B, y))$,
- a decision rule telling the government when to default and when to pay as a function of the state (B, y), and
- an asset accumulation rule that, conditional on choosing not to default, maps (B, y) into B'

such that

- The three Bellman equations for $(v_c(B, y), v_d(y), v(B, y))$ are satisfied
- Given the price function $q(B', y)$, the default decision rule and the asset accumulation decision rule attain the optimal value function $v(B, y)$, and
- The price function $q(B', y)$ satisfies equation (3.5)

3.4 Computation

Let's now compute an equilibrium of Arellano's model.

The equilibrium objects are the value function $v(B, y)$, the associated default decision rule, and the pricing function $q(B', y)$.

We'll use our code to replicate Arellano's results.

After that we'll perform some additional simulations.

We use a slightly modified version of the algorithm recommended by Arellano.

- The appendix to [Arellano, 2008] recommends value function iteration until convergence, updating the price, and then repeating.
- Instead, we update the bond price at every value function iteration step.

The second approach is faster and the two different procedures deliver very similar results.

Here is a more detailed description of our algorithm:

1. Guess a pair of non-default and default value functions v_c and v_d.
2. Using these functions, calculate the value function v, the corresponding default probabilities and the price function q.
3. At each pair (B, y),
    1. update the value of defaulting $v_d(y)$.
    2. update the value of remaining $v_c(B, y)$.
4. Check for convergence. If converged, stop – if not, go to step 2.

We use simple discretization on a grid of asset holdings and income levels.

The output process is discretized using a quadrature method due to Tauchen.

As we have in other places, we accelerate our code using Numba.

We define a class that will store parameters, grids and transition probabilities.

```
class Arellano_Economy:
    " Stores data and creates primitives for the Arellano economy. "

    def __init__(self,
            B_grid_size= 251,    # Grid size for bonds
            B_grid_min=-0.45,    # Smallest B value
            B_grid_max=0.45,     # Largest B value
            y_grid_size=51,      # Grid size for income
            β=0.953,             # Time discount parameter
            γ=2.0,               # Utility parameter
            r=0.017,             # Lending rate
            ρ=0.945,             # Persistence in the income process
            η=0.025,             # Standard deviation of the income process
            θ=0.282,             # Prob of re-entering financial markets
            def_y_param=0.969): # Parameter governing income in default

        # Save parameters
        self.β, self.γ, self.r, = β, γ, r
        self.ρ, self.η, self.θ = ρ, η, θ

        self.y_grid_size = y_grid_size
        self.B_grid_size = B_grid_size
        self.B_grid = np.linspace(B_grid_min, B_grid_max, B_grid_size)
        mc = qe.markov.tauchen(y_grid_size, ρ, η, 0, 3)
        self.y_grid, self.P = np.exp(mc.state_values), mc.P

        # The index at which B_grid is (close to) zero
        self.B0_idx = np.searchsorted(self.B_grid, 1e-10)

        # Output recieved while in default, with same shape as y_grid
        self.def_y = np.minimum(def_y_param * np.mean(self.y_grid), self.y_grid)
```

(continues on next page)

(continued from previous page)

```
    def params(self):
        return self.β, self.γ, self.r, self.ρ, self.η, self.θ

    def arrays(self):
        return self.P, self.y_grid, self.B_grid, self.def_y, self.B0_idx
```

Notice how the class returns the data it stores as simple numerical values and arrays via the methods `params` and `arrays`.

We will use this data in the Numba-jitted functions defined below.

Jitted functions prefer simple arguments, since type inference is easier.

Here is the utility function.

```
@njit
def u(c, γ):
    return c**(1-γ)/(1-γ)
```

Here is a function to compute the bond price at each state, given v_c and v_d.

```
@njit
def compute_q(v_c, v_d, q, params, arrays):
    """
    Compute the bond price function q(b, y) at each (b, y) pair.

    This function writes to the array q that is passed in as an argument.
    """

    # Unpack
    β, γ, r, ρ, η, θ = params
    P, y_grid, B_grid, def_y, B0_idx = arrays

    for B_idx in range(len(B_grid)):
        for y_idx in range(len(y_grid)):
            # Compute default probability and corresponding bond price
            delta = P[y_idx, v_c[B_idx, :] < v_d].sum()
            q[B_idx, y_idx] = (1 - delta ) / (1 + r)
```

Next we introduce Bellman operators that updated v_d and v_c.

```
@njit
def T_d(y_idx, v_c, v_d, params, arrays):
    """
    The RHS of the Bellman equation when income is at index y_idx and
    the country has chosen to default.  Returns an update of v_d.
    """
    # Unpack
    β, γ, r, ρ, η, θ = params
    P, y_grid, B_grid, def_y, B0_idx = arrays

    current_utility = u(def_y[y_idx], γ)
    v = np.maximum(v_c[B0_idx, :], v_d)
    cont_value = np.sum((θ * v + (1 - θ) * v_d) * P[y_idx, :])

    return current_utility + β * cont_value
```

(continues on next page)

(continued from previous page)

```
@njit
def T_c(B_idx, y_idx, v_c, v_d, q, params, arrays):
    """
    The RHS of the Bellman equation when the country is not in a
    defaulted state on their debt.  Returns a value that corresponds to
    v_c[B_idx, y_idx], as well as the optimal level of bond sales B'.
    """
    # Unpack
    β, γ, r, ρ, η, θ = params
    P, y_grid, B_grid, def_y, B0_idx = arrays
    B = B_grid[B_idx]
    y = y_grid[y_idx]

    # Compute the RHS of Bellman equation
    current_max = -1e10
    # Step through choices of next period B'
    for Bp_idx, Bp in enumerate(B_grid):
        c = y + B - q[Bp_idx, y_idx] * Bp
        if c > 0:
            v = np.maximum(v_c[Bp_idx, :], v_d)
            val = u(c, γ) + β * np.sum(v * P[y_idx, :])
            if val > current_max:
                current_max = val
                Bp_star_idx = Bp_idx
    return current_max, Bp_star_idx
```

Here is a fast function that calls these operators in the right sequence.

```
@njit(parallel=True)
def update_values_and_prices(v_c, v_d,      # Current guess of value functions
                             B_star, q,     # Arrays to be written to
                             params, arrays):

    # Unpack
    β, γ, r, ρ, η, θ = params
    P, y_grid, B_grid, def_y, B0_idx = arrays
    y_grid_size = len(y_grid)
    B_grid_size = len(B_grid)

    # Compute bond prices and write them to q
    compute_q(v_c, v_d, q, params, arrays)

    # Allocate memory
    new_v_c = np.empty_like(v_c)
    new_v_d = np.empty_like(v_d)

    # Calculate and return new guesses for v_c and v_d
    for y_idx in prange(y_grid_size):
        new_v_d[y_idx] = T_d(y_idx, v_c, v_d, params, arrays)
        for B_idx in range(B_grid_size):
            new_v_c[B_idx, y_idx], Bp_idx = T_c(B_idx, y_idx,
                                                v_c, v_d, q, params, arrays)
            B_star[B_idx, y_idx] = Bp_idx
```

(continues on next page)

(continued from previous page)

```
    return new_v_c, new_v_d
```

We can now write a function that will use the `Arellano_Economy` class and the functions defined above to compute the solution to our model.

We do not need to JIT compile this function since it only consists of outer loops (and JIT compiling makes almost zero difference).

In fact, one of the jobs of this function is to take an instance of `Arellano_Economy`, which is hard for the JIT compiler to handle, and strip it down to more basic objects, which are then passed out to jitted functions.

```
def solve(model, tol=1e-8, max_iter=10_000):
    """
    Given an instance of Arellano_Economy, this function computes the optimal
    policy and value functions.
    """
    # Unpack
    params = model.params()
    arrays = model.arrays()
    y_grid_size, B_grid_size = model.y_grid_size, model.B_grid_size

    # Initial conditions for v_c and v_d
    v_c = np.zeros((B_grid_size, y_grid_size))
    v_d = np.zeros(y_grid_size)

    # Allocate memory
    q = np.empty_like(v_c)
    B_star = np.empty_like(v_c, dtype=int)

    current_iter = 0
    dist = np.inf
    while (current_iter < max_iter) and (dist > tol):

        if current_iter % 100 == 0:
            print(f"Entering iteration {current_iter}.")

        new_v_c, new_v_d = update_values_and_prices(v_c, v_d, B_star, q, params,␣
↪arrays)
        # Check tolerance and update
        dist = np.max(np.abs(new_v_c - v_c)) + np.max(np.abs(new_v_d - v_d))
        v_c = new_v_c
        v_d = new_v_d
        current_iter += 1

    print(f"Terminating at iteration {current_iter}.")
    return v_c, v_d, q, B_star
```

Finally, we write a function that will allow us to simulate the economy once we have the policy functions

```
def simulate(model, T, v_c, v_d, q, B_star, y_idx=None, B_idx=None):
    """
    Simulates the Arellano 2008 model of sovereign debt

    Here `model` is an instance of `Arellano_Economy` and `T` is the length of
    the simulation.  Endogenous objects `v_c`, `v_d`, `q` and `B_star` are
    assumed to come from a solution to `model`.
```

(continues on next page)

(continued from previous page)

```
    """
    # Unpack elements of the model
    B0_idx = model.B0_idx
    y_grid = model.y_grid
    B_grid, y_grid, P = model.B_grid, model.y_grid, model.P

    # Set initial conditions to middle of grids
    if y_idx == None:
        y_idx = np.searchsorted(y_grid, y_grid.mean())
    if B_idx == None:
        B_idx = B0_idx
    in_default = False

    # Create Markov chain and simulate income process
    mc = qe.MarkovChain(P, y_grid)
    y_sim_indices = mc.simulate_indices(T+1, init=y_idx)

    # Allocate memory for outputs
    y_sim = np.empty(T)
    y_a_sim = np.empty(T)
    B_sim = np.empty(T)
    q_sim = np.empty(T)
    d_sim = np.empty(T, dtype=int)

    # Perform simulation
    t = 0
    while t < T:

        # Store the value of y_t and B_t
        y_sim[t] = y_grid[y_idx]
        B_sim[t] = B_grid[B_idx]

        # if in default:
        if v_c[B_idx, y_idx] < v_d[y_idx] or in_default:
            y_a_sim[t] = model.def_y[y_idx]
            d_sim[t] = 1
            Bp_idx = B0_idx
            # Re-enter financial markets next period with prob θ
            in_default = False if np.random.rand() < model.θ else True
        else:
            y_a_sim[t] = y_sim[t]
            d_sim[t] = 0
            Bp_idx = B_star[B_idx, y_idx]

        q_sim[t] = q[Bp_idx, y_idx]

        # Update time and indices
        t += 1
        y_idx = y_sim_indices[t]
        B_idx = Bp_idx

    return y_sim, y_a_sim, B_sim, q_sim, d_sim
```

3.5 Results

Let's start by trying to replicate the results obtained in [Arellano, 2008].

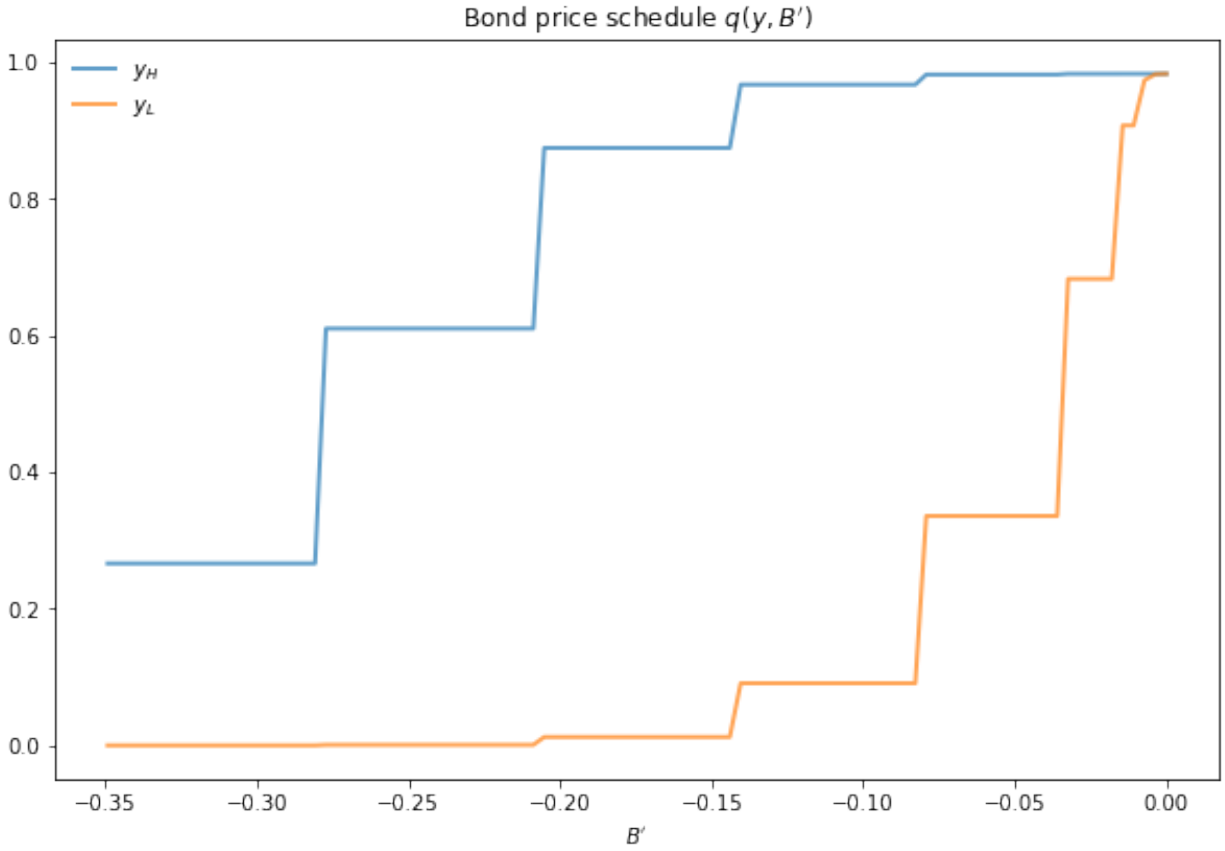
In what follows, all results are computed using Arellano's parameter values.

The values can be seen in the `__init__` method of the `Arellano_Economy` shown above.

For example, `r=0.017` matches the average quarterly rate on a 5 year US treasury over the period 1983–2001.

Details on how to compute the figures are reported as solutions to the exercises.

The first figure shows the bond price schedule and replicates Figure 3 of Arellano, where y_L and Y_H are particular below average and above average values of output y.



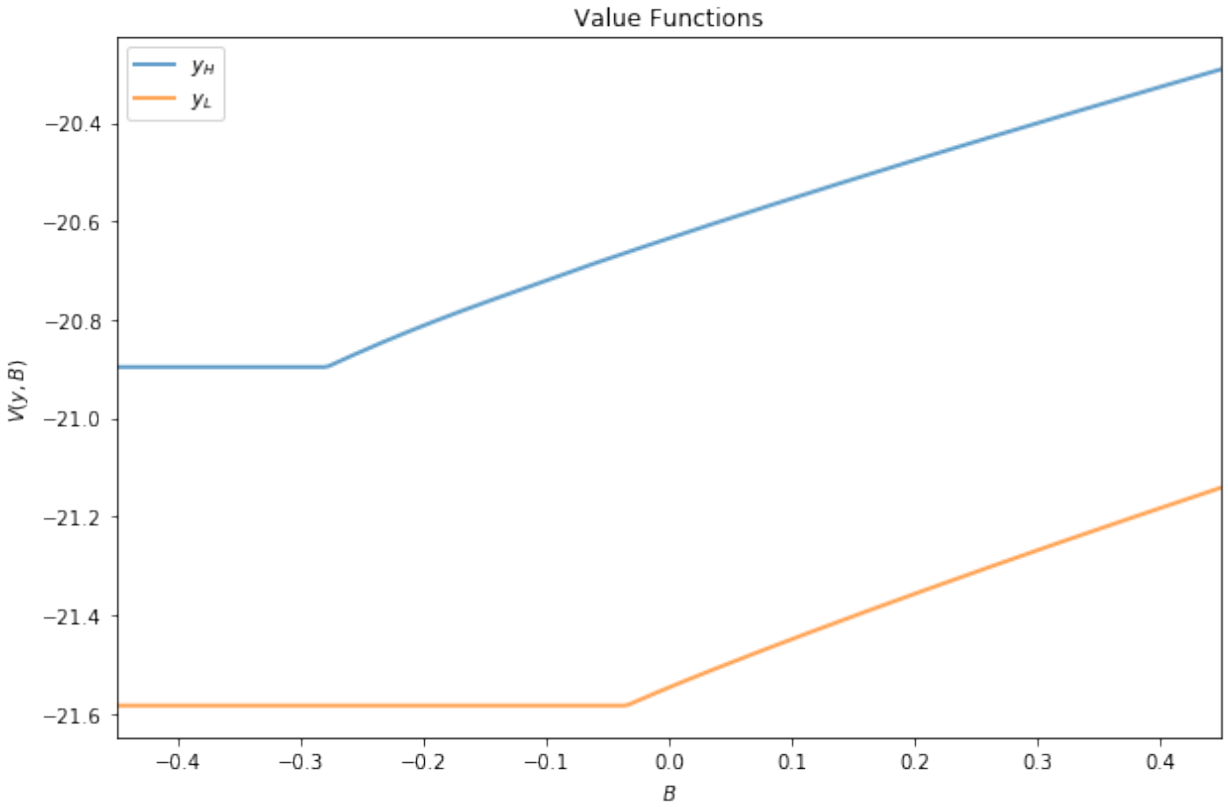
- y_L is 5% below the mean of the y grid values
- y_H is 5% above the mean of the y grid values

The grid used to compute this figure was relatively fine (`y_grid_size, B_grid_size = 51, 251`), which explains the minor differences between this and Arrelano's figure.

The figure shows that

- Higher levels of debt (larger $-B'$) induce larger discounts on the face value, which correspond to higher interest rates.
- Lower income also causes more discounting, as foreign creditors anticipate greater likelihood of default.

The next figure plots value functions and replicates the right hand panel of Figure 4 of [Arellano, 2008].



We can use the results of the computation to study the default probability $\delta(B', y)$ defined in (3.4).

The next plot shows these default probabilities over (B', y) as a heat map.

As anticipated, the probability that the government chooses to default in the following period increases with indebtedness and falls with income.

Next let's run a time series simulation of $\{y_t\}$, $\{B_t\}$ and $q(B_{t+1}, y_t)$.

The grey vertical bars correspond to periods when the economy is excluded from financial markets because of a past default.

One notable feature of the simulated data is the nonlinear response of interest rates.

Periods of relative stability are followed by sharp spikes in the discount rate on government debt.

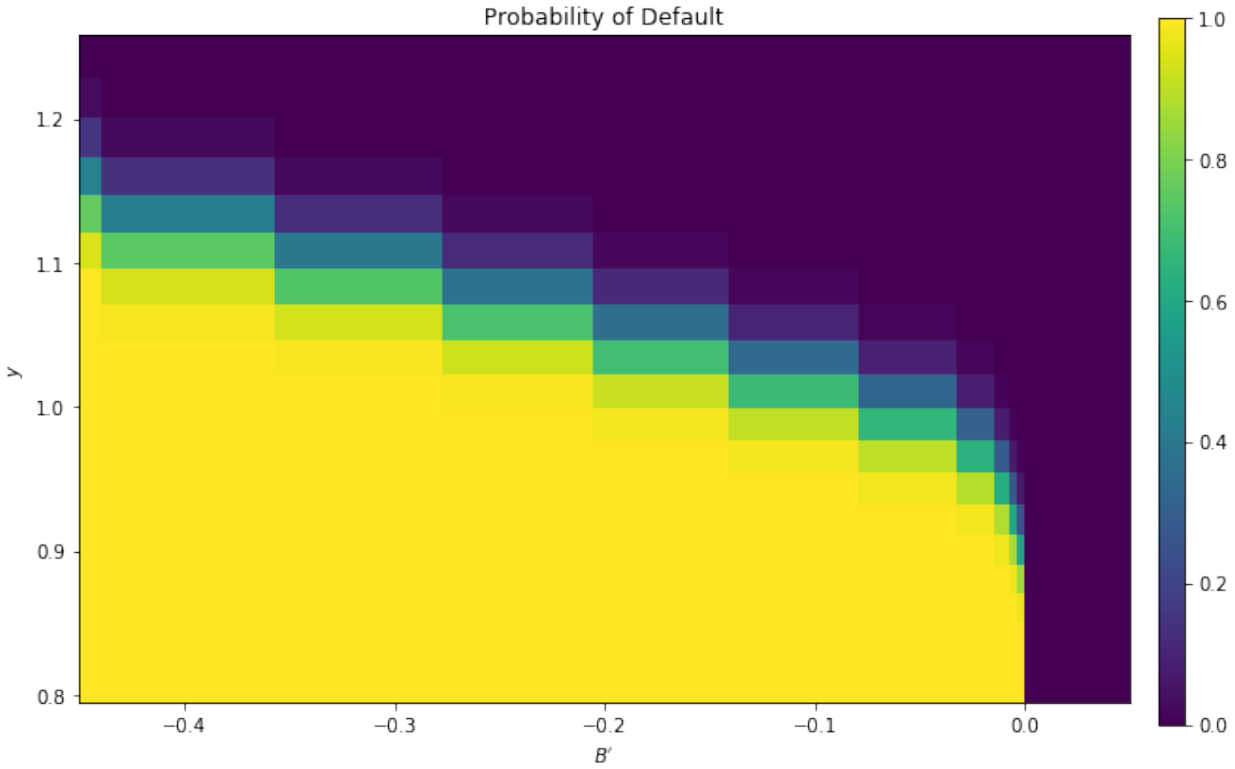
3.6 Exercises

Exercise 3.6.1

To the extent that you can, replicate the figures shown above

- Use the parameter values listed as defaults in `Arellano_Economy`.
- The time series will of course vary depending on the shock draws.

Solution to Exercise 3.6.1



Compute the value function, policy and equilibrium prices

```
ae = Arellano_Economy()
```

```
v_c, v_d, q, B_star = solve(ae)
```

```
Entering iteration 0.
```

```
Entering iteration 100.
```

```
Entering iteration 200.
```

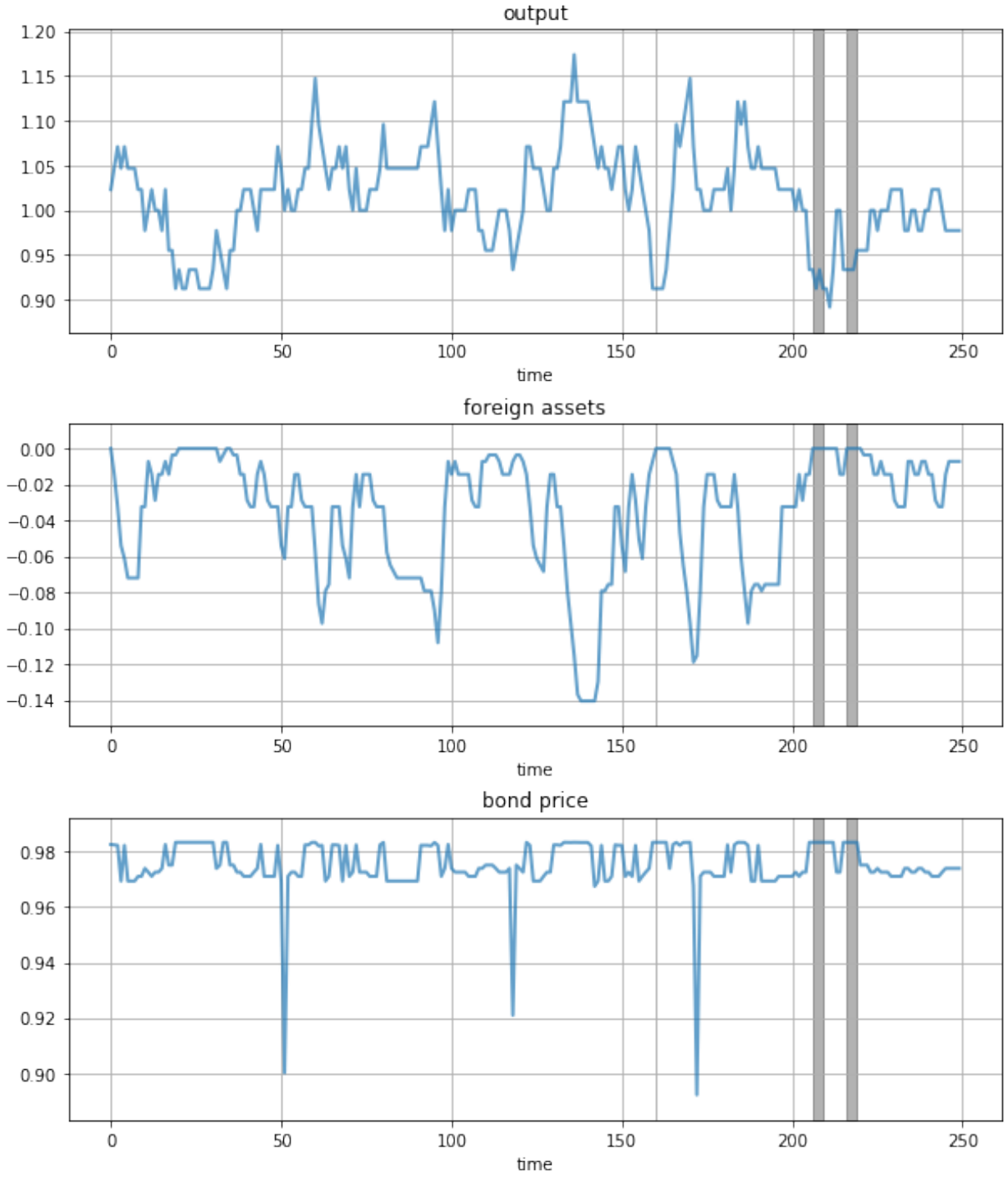
```
Entering iteration 300.
```

```
Terminating at iteration 399.
```

Compute the bond price schedule as seen in figure 3 of Arellano (2008)

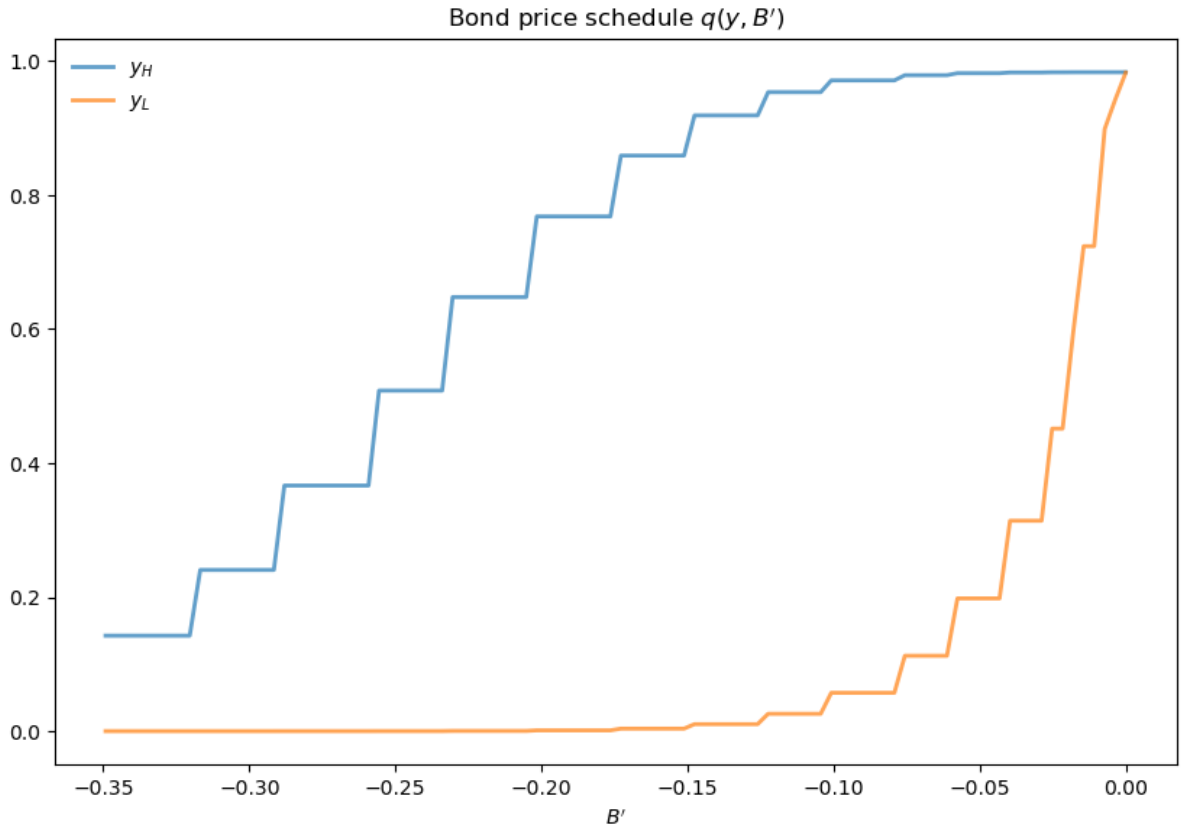
```
# Unpack some useful names
B_grid, y_grid, P = ae.B_grid, ae.y_grid, ae.P
B_grid_size, y_grid_size = len(B_grid), len(y_grid)
r = ae.r

# Create "Y High" and "Y Low" values as 5% devs from mean
```

(continues on next page)


output
1.20
1.15
1.10
1.05
1.00
0.95
0.90
0
50
100
150
200
250
time
foreign assets
0.00
−0.02
−0.04
−0.06
−0.08
−0.10
−0.12
−0.14
0
50
100
150
200
250
time
bond price
0.98
0.96
0.94
0.92
0.90
0
50
100
150
200
250
time

(continued from previous page)

```
high, low = np.mean(y_grid) * 1.05, np.mean(y_grid) * .95
iy_high, iy_low = (np.searchsorted(y_grid, x) for x in (high, low))

fig, ax = plt.subplots(figsize=(10, 6.5))
ax.set_title("Bond price schedule $q(y, B')$")

# Extract a suitable plot grid
x = []
q_low = []
q_high = []
for i, B in enumerate(B_grid):
    if -0.35 <= B <= 0:  # To match fig 3 of Arellano
        x.append(B)
        q_low.append(q[i, iy_low])
        q_high.append(q[i, iy_high])
ax.plot(x, q_high, label="$y_H$", lw=2, alpha=0.7)
ax.plot(x, q_low, label="$y_L$", lw=2, alpha=0.7)
ax.set_xlabel("$B'$")
ax.legend(loc='upper left', frameon=False)
plt.show()
```



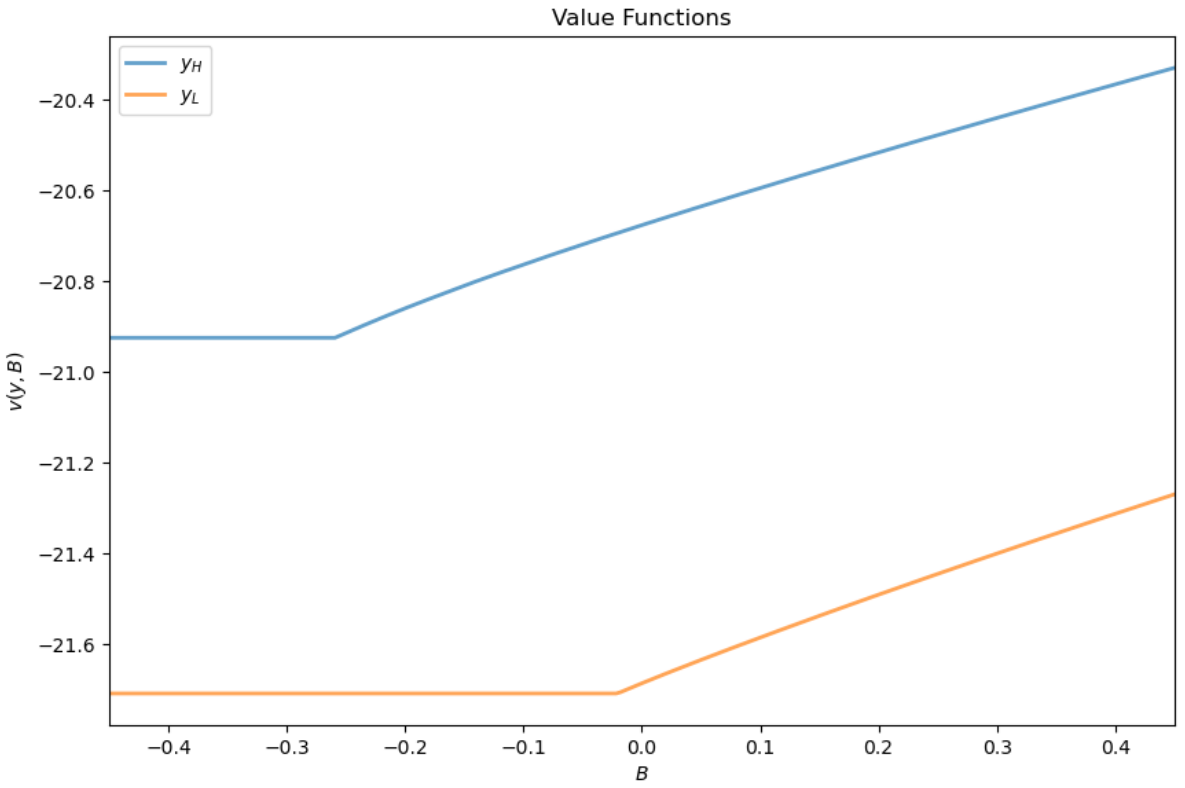
Draw a plot of the value functions

```
v = np.maximum(v_c, np.reshape(v_d, (1, y_grid_size)))

fig, ax = plt.subplots(figsize=(10, 6.5))
```

(continues on next page)

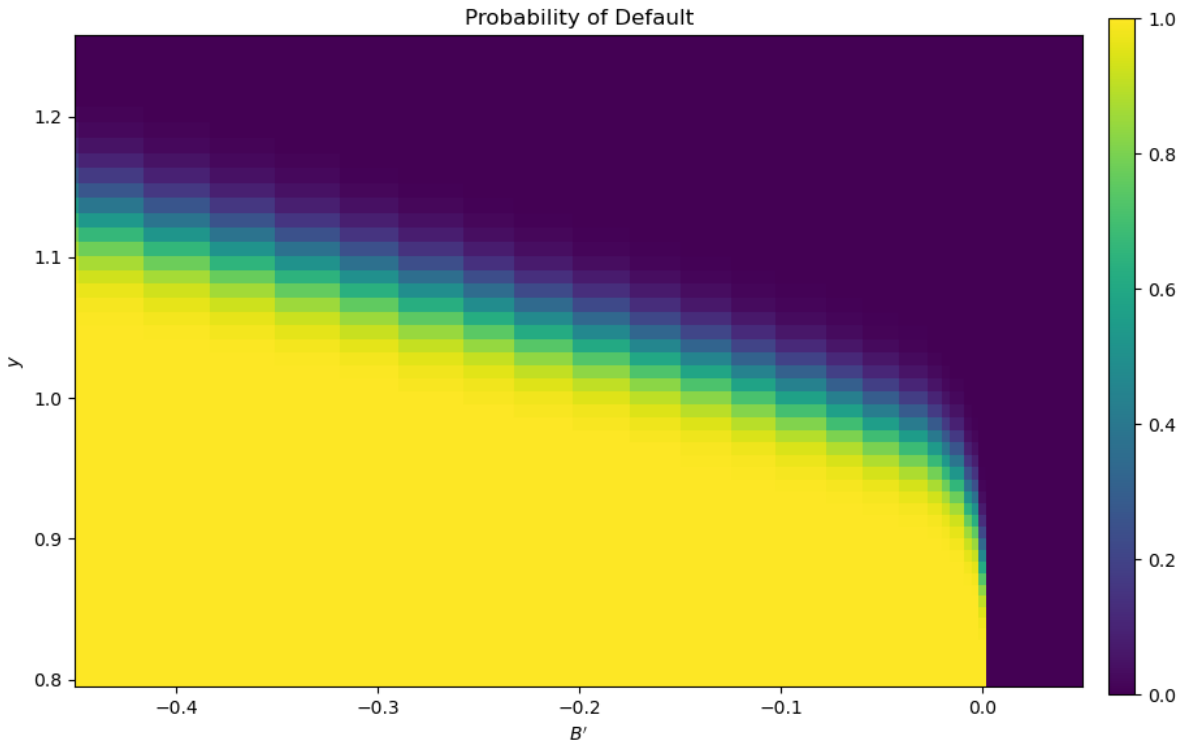
(continued from previous page)

```
ax.set_title("Value Functions")
ax.plot(B_grid, v[:, iy_high], label="$y_H$", lw=2, alpha=0.7)
ax.plot(B_grid, v[:, iy_low], label="$y_L$", lw=2, alpha=0.7)
ax.legend(loc='upper left')
ax.set(xlabel="$B$", ylabel="$v(y, B)$")
ax.set_xlim(min(B_grid), max(B_grid))
plt.show()
```



Draw a heat map for default probability

```
xx, yy = B_grid, y_grid
zz = np.empty_like(v_c)

for B_idx in range(B_grid_size):
    for y_idx in range(y_grid_size):
        zz[B_idx, y_idx] = P[y_idx, v_c[B_idx, :] < v_d].sum()

# Create figure
fig, ax = plt.subplots(figsize=(10, 6.5))
hm = ax.pcolormesh(xx, yy, zz.T)
cax = fig.add_axes([.92, .1, .02, .8])
fig.colorbar(hm, cax=cax)
ax.axis([xx.min(), 0.05, yy.min(), yy.max()])
ax.set(xlabel="$B'$", ylabel="$y$", title="Probability of Default")
plt.show()
```



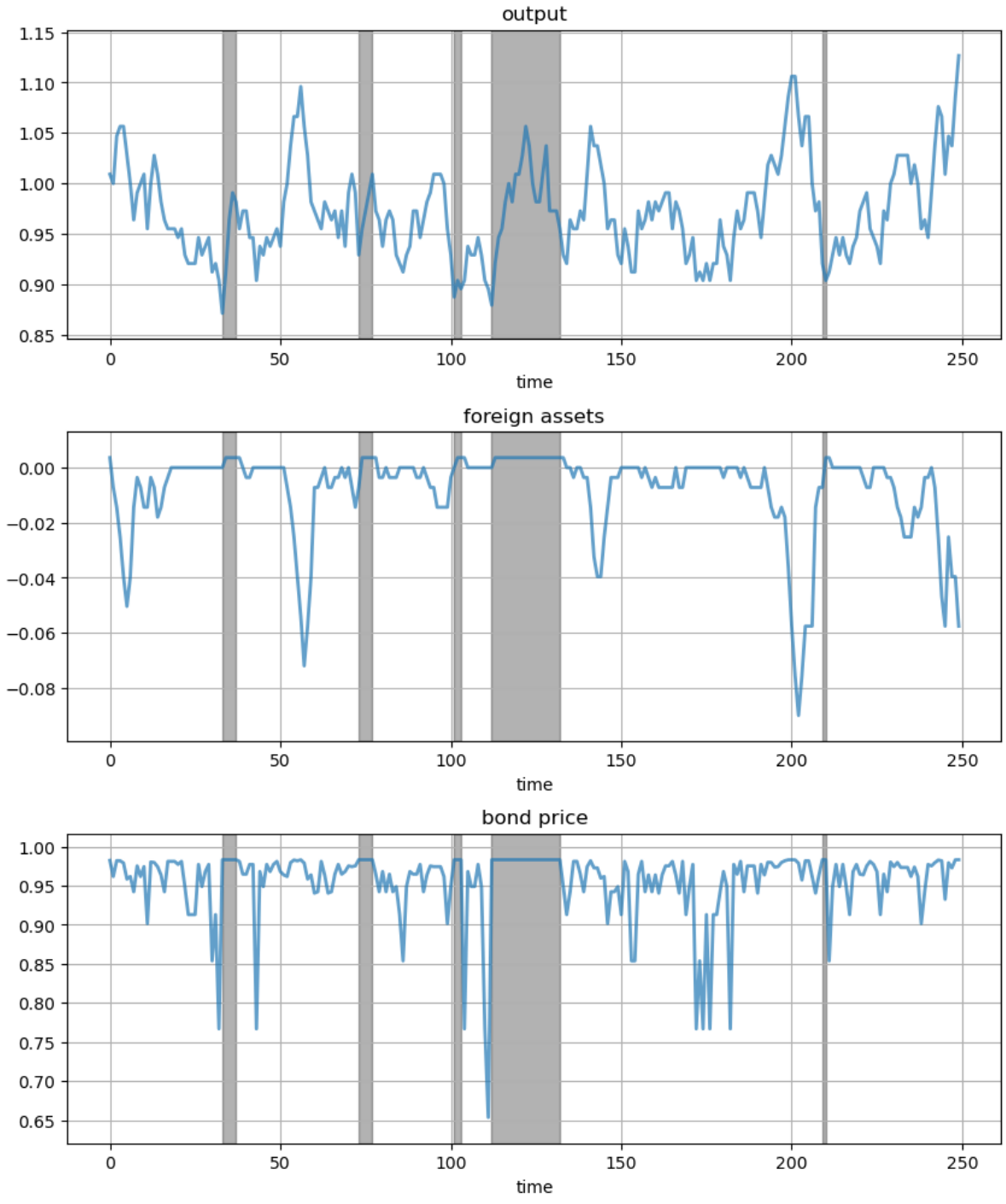
Plot a time series of major variables simulated from the model

```
T = 250
np.random.seed(42)
y_sim, y_a_sim, B_sim, q_sim, d_sim = simulate(ae, T, v_c, v_d, q, B_star)

# Pick up default start and end dates
start_end_pairs = []
i = 0
while i < len(d_sim):
    if d_sim[i] == 0:
        i += 1
    else:
        # If we get to here we're in default
        start_default = i
        while i < len(d_sim) and d_sim[i] == 1:
            i += 1
        end_default = i - 1
        start_end_pairs.append((start_default, end_default))

plot_series = (y_sim, B_sim, q_sim)
titles = 'output', 'foreign assets', 'bond price'

fig, axes = plt.subplots(len(plot_series), 1, figsize=(10, 12))
fig.subplots_adjust(hspace=0.3)

for ax, series, title in zip(axes, plot_series, titles):
    # Determine suitable y limits
    s_max, s_min = max(series), min(series)
    s_range = s_max - s_min
    y_max = s_max + s_range * 0.1
```

(continues on next page)

(continued from previous page)

```
    y_min = s_min - s_range * 0.1
    ax.set_ylim(y_min, y_max)
    for pair in start_end_pairs:
        ax.fill_between(pair, (y_min, y_min), (y_max, y_max),
                        color='k', alpha=0.3)
    ax.grid()
    ax.plot(range(T), series, lw=2, alpha=0.7)
    ax.set(title=title, xlabel="time")

plt.show()
```


output
1.15
1.10
1.05
1.00
0.95
0.90
0.85
0
50
100
150
200
250
time
foreign assets
0.00
−0.02
−0.04
−0.06
−0.08
0
50
100
150
200
250
time
bond price
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0.65
0
50
100
150
200
250
time

CHAPTER

FOUR

GLOBALIZATION AND CYCLES

Contents

4.1 Overview

In this lecture, we review the paper Globalization and Synchronization of Innovation Cycles by Kiminori Matsuyama, Laura Gardini and Iryna Sushko.

This model helps us understand several interesting stylized facts about the world economy.

One of these is synchronized business cycles across different countries.

Most existing models that generate synchronized business cycles do so by assumption, since they tie output in each country to a common shock.

They also fail to explain certain features of the data, such as the fact that the degree of synchronization tends to increase with trade ties.

By contrast, in the model we consider in this lecture, synchronization is both endogenous and increasing with the extent of trade integration.

In particular, as trade costs fall and international competition increases, innovation incentives become aligned and countries synchronize their innovation cycles.

Let's start with some imports:

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from numba import jit
from ipywidgets import interact
```

4.1.1 Background

The model builds on work by Judd [Judd, 1985], Deneckner and Judd [Deneckere and Judd, 1992] and Helpman and Krugman [Helpman and Krugman, 1985] by developing a two-country model with trade and innovation.

On the technical side, the paper introduces the concept of coupled oscillators to economic modeling.

As we will see, coupled oscillators arise endogenously within the model.

Below we review the model and replicate some of the results on synchronization of innovation across countries.

4.2 Key Ideas

It is helpful to begin with an overview of the mechanism.

4.2.1 Innovation Cycles

As discussed above, two countries produce and trade with each other.

In each country, firms innovate, producing new varieties of goods and, in doing so, receiving temporary monopoly power.

Imitators follow and, after one period of monopoly, what had previously been new varieties now enter competitive production.

Firms have incentives to innovate and produce new goods when the mass of varieties of goods currently in production is relatively low.

In addition, there are strategic complementarities in the timing of innovation.

Firms have incentives to innovate in the same period, so as to avoid competing with substitutes that are competitively produced.

This leads to temporal clustering in innovations in each country.

After a burst of innovation, the mass of goods currently in production increases.

However, goods also become obsolete, so that not all survive from period to period.

This mechanism generates a cycle, where the mass of varieties increases through simultaneous innovation and then falls through obsolescence.

4.2.2 Synchronization

In the absence of trade, the timing of innovation cycles in each country is decoupled.

This will be the case when trade costs are prohibitively high.

If trade costs fall, then goods produced in each country penetrate each other's markets.

As illustrated below, this leads to synchronization of business cycles across the two countries.

4.3 Model

Let's write down the model more formally.

(The treatment is relatively terse since full details can be found in the original paper)

Time is discrete with $t = 0, 1, ...$

There are two countries indexed by j or k.

In each country, a representative household inelastically supplies L_j units of labor at wage rate $w_{j,t}$.

Without loss of generality, it is assumed that $L_1 \geq L_2$.

Households consume a single nontradeable final good which is produced competitively.

Its production involves combining two types of tradeable intermediate inputs via

$$Y_{k,t} = C_{k,t} = \left(\frac{X^o_{k,t}}{1-\alpha}\right)^{1-\alpha} \left(\frac{X_{k,t}}{\alpha}\right)^{\alpha}$$

Here $X^o_{k,t}$ is a homogeneous input which can be produced from labor using a linear, one-for-one technology.

It is freely tradeable, competitively supplied, and homogeneous across countries.

By choosing the price of this good as numeraire and assuming both countries find it optimal to always produce the homogeneous good, we can set $w_{1,t} = w_{2,t} = 1$.

The good $X_{k,t}$ is a composite, built from many differentiated goods via

$$X_{k,t}^{1-\frac{1}{\sigma}} = \int_{\Omega_t} \left[x_{k,t}(\nu)\right]^{1-\frac{1}{\sigma}} d\nu$$

Here $x_{k,t}(\nu)$ is the total amount of a differentiated good $\nu \in \Omega_t$ that is produced.

The parameter $\sigma > 1$ is the direct partial elasticity of substitution between a pair of varieties and Ω_t is the set of varieties available in period t.

We can split the varieties into those which are supplied competitively and those supplied monopolistically; that is, $\Omega_t = \Omega_t^c + \Omega_t^m$.

4.3.1 Prices

Demand for differentiated inputs is

$$x_{k,t}(\nu) = \left(\frac{p_{k,t}(\nu)}{P_{k,t}}\right)^{-\sigma} \frac{\alpha L_k}{P_{k,t}}$$

Here

- $p_{k,t}(\nu)$ is the price of the variety ν and
- $P_{k,t}$ is the price index for differentiated inputs in k, defined by

$$\left[P_{k,t}\right]^{1-\sigma} = \int_{\Omega_t} \left[p_{k,t}(\nu)\right]^{1-\sigma} d\nu$$

The price of a variety also depends on the origin, j, and destination, k, of the goods because shipping varieties between countries incurs an iceberg trade cost $\tau_{j,k}$.

Thus the effective price in country k of a variety ν produced in country j becomes $p_{k,t}(\nu) = \tau_{j,k}\, p_{j,t}(\nu)$.

Using these expressions, we can derive the total demand for each variety, which is

$$D_{j,t}(\nu) = \sum_k \tau_{j,k} x_{k,t}(\nu) = \alpha A_{j,t} (p_{j,t}(\nu))^{-\sigma}$$

where

$$A_{j,t} := \sum_k \frac{\rho_{j,k} L_k}{(P_{k,t})^{1-\sigma}} \quad \text{and} \quad \rho_{j,k} = (\tau_{j,k})^{1-\sigma} \leq 1$$

It is assumed that $\tau_{1,1} = \tau_{2,2} = 1$ and $\tau_{1,2} = \tau_{2,1} = \tau$ for some $\tau > 1$, so that

$$\rho_{1,2} = \rho_{2,1} = \rho := \tau^{1-\sigma} < 1$$

The value $\rho \in [0, 1)$ is a proxy for the degree of globalization.

Producing one unit of each differentiated variety requires ψ units of labor, so the marginal cost is equal to ψ for $\nu \in \Omega_{j,t}$.

Additionally, all competitive varieties will have the same price (because of equal marginal cost), which means that, for all $\nu \in \Omega^c$,

$$p_{j,t}(\nu) = p^c_{j,t} := \psi \quad \text{and} \quad D_{j,t} = y^c_{j,t} := \alpha A_{j,t} (p^c_{j,t})^{-\sigma}$$

Monopolists will have the same marked-up price, so, for all $\nu \in \Omega^m$,

$$p_{j,t}(\nu) = p^m_{j,t} := \frac{\psi}{1 - \frac{1}{\sigma}} \quad \text{and} \quad D_{j,t} = y^m_{j,t} := \alpha A_{j,t} (p^m_{j,t})^{-\sigma}$$

Define

$$\theta := \frac{p^c_{j,t}}{p^m_{j,t}} \frac{y^c_{j,t}}{y^m_{j,t}} = \left(1 - \frac{1}{\sigma}\right)^{1-\sigma}$$

Using the preceding definitions and some algebra, the price indices can now be rewritten as

$$\left(\frac{P_{k,t}}{\psi}\right)^{1-\sigma} = M_{k,t} + \rho M_{j,t} \quad \text{where} \quad M_{j,t} := N^c_{j,t} + \frac{N^m_{j,t}}{\theta}$$

The symbols $N^c_{j,t}$ and $N^m_{j,t}$ will denote the measures of Ω^c and Ω^m respectively.

4.3.2 New Varieties

To introduce a new variety, a firm must hire f units of labor per variety in each country.

Monopolist profits must be less than or equal to zero in expectation, so

$$N^m_{j,t} \geq 0, \quad \pi^m_{j,t} := (p^m_{j,t} - \psi) y^m_{j,t} - f \leq 0 \quad \text{and} \quad \pi^m_{j,t} N^m_{j,t} = 0$$

With further manipulations, this becomes

$$N^m_{j,t} = \theta(M_{j,t} - N^c_{j,t}) \geq 0, \quad \frac{1}{\sigma}\left[\frac{\alpha L_j}{\theta(M_{j,t} + \rho M_{k,t})} + \frac{\alpha L_k}{\theta(M_{j,t} + M_{k,t}/\rho)}\right] \leq f$$

4.3.3 Law of Motion

With δ as the exogenous probability of a variety becoming obsolete, the dynamic equation for the measure of firms becomes

$$N^c_{j,t+1} = \delta(N^c_{j,t} + N^m_{j,t}) = \delta(N^c_{j,t} + \theta(M_{j,t} - N^c_{j,t}))$$

We will work with a normalized measure of varieties

$$n_{j,t} := \frac{\theta\sigma f N^c_{j,t}}{\alpha(L_1 + L_2)}, \quad i_{j,t} := \frac{\theta\sigma f N^m_{j,t}}{\alpha(L_1 + L_2)}, \quad m_{j,t} := \frac{\theta\sigma f M_{j,t}}{\alpha(L_1 + L_2)} = n_{j,t} + \frac{i_{j,t}}{\theta}$$

We also use $s_j := \frac{L_j}{L_1+L_2}$ to be the share of labor employed in country j.

We can use these definitions and the preceding expressions to obtain a law of motion for $n_t := (n_{1,t}, n_{2,t})$.

In particular, given an initial condition, $n_0 = (n_{1,0}, n_{2,0}) \in \mathbb{R}^2_+$, the equilibrium trajectory, $\{n_t\}_{t=0}^\infty = \{(n_{1,t}, n_{2,t})\}_{t=0}^\infty$, is obtained by iterating on $n_{t+1} = F(n_t)$ where $F : \mathbb{R}^2_+ \to \mathbb{R}^2_+$ is given by

$$F(n_t) = \begin{cases} (\delta(\theta s_1(\rho) + (1-\theta)n_{1,t}), \delta(\theta s_2(\rho) + (1-\theta)n_{2,t})) & \text{for } n_t \in D_{LL} \\ (\delta n_{1,t}, \delta n_{2,t}) & \text{for } n_t \in D_{HH} \\ (\delta n_{1,t}, \delta(\theta h_2(n_{1,t}) + (1-\theta)n_{2,t})) & \text{for } n_t \in D_{HL} \\ (\delta(\theta h_1(n_{2,t}) + (1-\theta)n_{1,t}, \delta n_{2,t})) & \text{for } n_t \in D_{LH} \end{cases}$$

Here

$$\begin{aligned} D_{LL} &:= \{(n_1, n_2) \in \mathbb{R}^2_+ | n_j \leq s_j(\rho)\} \\ D_{HH} &:= \{(n_1, n_2) \in \mathbb{R}^2_+ | n_j \geq h_j(n_k)\} \\ D_{HL} &:= \{(n_1, n_2) \in \mathbb{R}^2_+ | n_1 \geq s_1(\rho) \text{ and } n_2 \leq h_2(n_1)\} \\ D_{LH} &:= \{(n_1, n_2) \in \mathbb{R}^2_+ | n_1 \leq h_1(n_2) \text{ and } n_2 \geq s_2(\rho)\} \end{aligned}$$

while

$$s_1(\rho) = 1 - s_2(\rho) = \min\left\{\frac{s_1 - \rho s_2}{1-\rho}, 1\right\}$$

and $h_j(n_k)$ is defined implicitly by the equation

$$1 = \frac{s_j}{h_j(n_k) + \rho n_k} + \frac{s_k}{h_j(n_k) + n_k/\rho}$$

Rewriting the equation above gives us a quadratic equation in terms of $h_j(n_k)$.

Since we know $h_j(n_k) > 0$ then we can just solve the quadratic equation and return the positive root.

This gives us

$$h_j(n_k)^2 + \left((\rho + \frac{1}{\rho})n_k - s_j - s_k\right) h_j(n_k) + (n_k^2 - \frac{s_j n_k}{\rho} - s_k n_k \rho) = 0$$

4.4 Simulation

Let's try simulating some of these trajectories.

We will focus in particular on whether or not innovation cycles synchronize across the two countries.

As we will see, this depends on initial conditions.

For some parameterizations, synchronization will occur for "most" initial conditions, while for others synchronization will be rare.

The computational burden of testing synchronization across many initial conditions is not trivial.

In order to make our code fast, we will use just in time compiled functions that will get called and handled by our class.

These are the `@jit` statements that you see below (review this lecture if you don't recall how to use JIT compilation).

Here's the main body of code

```
@jit(nopython=True)
def _hj(j, nk, s1, s2, θ, δ, ρ):
    """
    If we expand the implicit function for h_j(n_k) then we find that
    it is quadratic. We know that h_j(n_k) > 0 so we can get its
    value by using the quadratic form
    """
    # Find out who's h we are evaluating
    if j == 1:
        sj = s1
        sk = s2
    else:
        sj = s2
        sk = s1

    # Coefficients on the quadratic a x^2 + b x + c = 0
    a = 1.0
    b = ((ρ + 1 / ρ) * nk - sj - sk)
    c = (nk * nk - (sj * nk) / ρ - sk * ρ * nk)

    # Positive solution of quadratic form
    root = (-b + np.sqrt(b * b - 4 * a * c)) / (2 * a)

    return root

@jit(nopython=True)
def DLL(n1, n2, s1_ρ, s2_ρ, s1, s2, θ, δ, ρ):
    "Determine whether (n1, n2) is in the set DLL"
    return (n1 <= s1_ρ) and (n2 <= s2_ρ)

@jit(nopython=True)
def DHH(n1, n2, s1_ρ, s2_ρ, s1, s2, θ, δ, ρ):
    "Determine whether (n1, n2) is in the set DHH"
    return (n1 >= _hj(1, n2, s1, s2, θ, δ, ρ)) and \
           (n2 >= _hj(2, n1, s1, s2, θ, δ, ρ))

@jit(nopython=True)
def DHL(n1, n2, s1_ρ, s2_ρ, s1, s2, θ, δ, ρ):
    "Determine whether (n1, n2) is in the set DHL"
    return (n1 >= s1_ρ) and (n2 <= _hj(2, n1, s1, s2, θ, δ, ρ))

@jit(nopython=True)
def DLH(n1, n2, s1_ρ, s2_ρ, s1, s2, θ, δ, ρ):
    "Determine whether (n1, n2) is in the set DLH"
    return (n1 <= _hj(1, n2, s1, s2, θ, δ, ρ)) and (n2 >= s2_ρ)

@jit(nopython=True)
def one_step(n1, n2, s1_ρ, s2_ρ, s1, s2, θ, δ, ρ):
```

(continues on next page)

(continued from previous page)

```
    """
    Takes a current value for (n_{1, t}, n_{2, t}) and returns the
    values (n_{1, t+1}, n_{2, t+1}) according to the law of motion.
    """
    # Depending on where we are, evaluate the right branch
    if DLL(n1, n2, s1_ρ, s2_ρ, s1, s2, θ, δ, ρ):
        n1_tp1 = δ * (θ * s1_ρ + (1 - θ) * n1)
        n2_tp1 = δ * (θ * s2_ρ + (1 - θ) * n2)
    elif DHH(n1, n2, s1_ρ, s2_ρ, s1, s2, θ, δ, ρ):
        n1_tp1 = δ * n1
        n2_tp1 = δ * n2
    elif DHL(n1, n2, s1_ρ, s2_ρ, s1, s2, θ, δ, ρ):
        n1_tp1 = δ * n1
        n2_tp1 = δ * (θ * _hj(2, n1, s1, s2, θ, δ, ρ) + (1 - θ) * n2)
    elif DLH(n1, n2, s1_ρ, s2_ρ, s1, s2, θ, δ, ρ):
        n1_tp1 = δ * (θ * _hj(1, n2, s1, s2, θ, δ, ρ) + (1 - θ) * n1)
        n2_tp1 = δ * n2

    return n1_tp1, n2_tp1

@jit(nopython=True)
def n_generator(n1_0, n2_0, s1_ρ, s2_ρ, s1, s2, θ, δ, ρ):
    """
    Given an initial condition, continues to yield new values of
    n1 and n2
    """
    n1_t, n2_t = n1_0, n2_0
    while True:
        n1_tp1, n2_tp1 = one_step(n1_t, n2_t, s1_ρ, s2_ρ, s1, s2, θ, δ, ρ)
        yield (n1_tp1, n2_tp1)
        n1_t, n2_t = n1_tp1, n2_tp1

@jit(nopython=True)
def _pers_till_sync(n1_0, n2_0, s1_ρ, s2_ρ, s1, s2, θ, δ, ρ, maxiter, npers):
    """
    Takes initial values and iterates forward to see whether
    the histories eventually end up in sync.

    If countries are symmetric then as soon as the two countries have the
    same measure of firms then they will be synchronized -- However, if
    they are not symmetric then it is possible they have the same measure
    of firms but are not yet synchronized. To address this, we check whether
    firms stay synchronized for `npers` periods with Euclidean norm

    Parameters
    ----------
    n1_0 : scalar(Float)
        Initial normalized measure of firms in country one
    n2_0 : scalar(Float)
        Initial normalized measure of firms in country two
    maxiter : scalar(Int)
        Maximum number of periods to simulate
    npers : scalar(Int)
        Number of periods we would like the countries to have the
        same measure for
```

(continues on next page)

(continued from previous page)

```
    Returns
    -------
    synchronized : scalar(Bool)
        Did the two economies end up synchronized
    pers_2_sync : scalar(Int)
        The number of periods required until they synchronized
    """
    # Initialize the status of synchronization
    synchronized = False
    pers_2_sync = maxiter
    iters = 0

    # Initialize generator
    n_gen = n_generator(n1_0, n2_0, s1_ρ, s2_ρ, s1, s2, θ, δ, ρ)

    # Will use a counter to determine how many times in a row
    # the firm measures are the same
    nsync = 0

    while (not synchronized) and (iters < maxiter):
        # Increment the number of iterations and get next values
        iters += 1
        n1_t, n2_t = next(n_gen)

        # Check whether same in this period
        if abs(n1_t - n2_t) < 1e-8:
            nsync += 1
        # If not, then reset the nsync counter
        else:
            nsync = 0

        # If we have been in sync for npers then stop and countries
        # became synchronized nsync periods ago
        if nsync > npers:
            synchronized = True
            pers_2_sync = iters - nsync

    return synchronized, pers_2_sync

@jit(nopython=True)
def _create_attraction_basis(s1_ρ, s2_ρ, s1, s2, θ, δ, ρ,
        maxiter, npers, npts):
    # Create unit range with npts
    synchronized, pers_2_sync = False, 0
    unit_range = np.linspace(0.0, 1.0, npts)

    # Allocate space to store time to sync
    time_2_sync = np.empty((npts, npts))
    # Iterate over initial conditions
    for (i, n1_0) in enumerate(unit_range):
        for (j, n2_0) in enumerate(unit_range):
            synchronized, pers_2_sync = _pers_till_sync(n1_0, n2_0, s1_ρ,
                                                        s2_ρ, s1, s2, θ, δ,
                                                        ρ, maxiter, npers)
            time_2_sync[i, j] = pers_2_sync
```

(continues on next page)

(continued from previous page)

```
    return time_2_sync


# == Now we define a class for the model == #

class MSGSync:
    """
    The paper "Globalization and Synchronization of Innovation Cycles" presents
    a two-country model with endogenous innovation cycles. Combines elements
    from Deneckere Judd (1985) and Helpman Krugman (1985) to allow for a
    model with trade that has firms who can introduce new varieties into
    the economy.

    We focus on being able to determine whether the two countries eventually
    synchronize their innovation cycles. To do this, we only need a few
    of the many parameters. In particular, we need the parameters listed
    below

    Parameters
    ----------
    s1 : scalar(Float)
        Amount of total labor in country 1 relative to total worldwide labor
    θ : scalar(Float)
        A measure of how much more of the competitive variety is used in
        production of final goods
    δ : scalar(Float)
        Percentage of firms that are not exogenously destroyed every period
    ρ : scalar(Float)
        Measure of how expensive it is to trade between countries
    """
    def __init__(self, s1=0.5, θ=2.5, δ=0.7, ρ=0.2):
        # Store model parameters
        self.s1, self.θ, self.δ, self.ρ = s1, θ, δ, ρ

        # Store other cutoffs and parameters we use
        self.s2 = 1 - s1
        self.s1_ρ = self._calc_s1_ρ()
        self.s2_ρ = 1 - self.s1_ρ

    def _unpack_params(self):
        return self.s1, self.s2, self.θ, self.δ, self.ρ

    def _calc_s1_ρ(self):
        # Unpack params
        s1, s2, θ, δ, ρ = self._unpack_params()

        # s_1(ρ) = min(val, 1)
        val = (s1 - ρ * s2) / (1 - ρ)
        return min(val, 1)

    def simulate_n(self, n1_0, n2_0, T):
        """
        Simulates the values of (n1, n2) for T periods

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

(continues on next page)

(continued from previous page)

```
        n1_0 : scalar(Float)
            Initial normalized measure of firms in country one
        n2_0 : scalar(Float)
            Initial normalized measure of firms in country two
        T : scalar(Int)
            Number of periods to simulate

        Returns
        -------
        n1 : Array(Float64, ndim=1)
            A history of normalized measures of firms in country one
        n2 : Array(Float64, ndim=1)
            A history of normalized measures of firms in country two
        """
        # Unpack parameters
        s1, s2, θ, δ, ρ = self._unpack_params()
        s1_ρ, s2_ρ = self.s1_ρ, self.s2_ρ

        # Allocate space
        n1 = np.empty(T)
        n2 = np.empty(T)

        # Create the generator
        n1[0], n2[0] = n1_0, n2_0
        n_gen = n_generator(n1_0, n2_0, s1_ρ, s2_ρ, s1, s2, θ, δ, ρ)

        # Simulate for T periods
        for t in range(1, T):
            # Get next values
            n1_tp1, n2_tp1 = next(n_gen)

            # Store in arrays
            n1[t] = n1_tp1
            n2[t] = n2_tp1

        return n1, n2

    def pers_till_sync(self, n1_0, n2_0, maxiter=500, npers=3):
        """
        Takes initial values and iterates forward to see whether
        the histories eventually end up in sync.

        If countries are symmetric then as soon as the two countries have the
        same measure of firms then they will be synchronized -- However, if
        they are not symmetric then it is possible they have the same measure
        of firms but are not yet synchronized. To address this, we check whether
        firms stay synchronized for `npers` periods with Euclidean norm

        Parameters
        ----------
        n1_0 : scalar(Float)
            Initial normalized measure of firms in country one
        n2_0 : scalar(Float)
            Initial normalized measure of firms in country two
        maxiter : scalar(Int)
            Maximum number of periods to simulate
```

(continues on next page)

(continued from previous page)

```
        npers : scalar(Int)
            Number of periods we would like the countries to have the
            same measure for

        Returns
        -------
        synchronized : scalar(Bool)
            Did the two economies end up synchronized
        pers_2_sync : scalar(Int)
            The number of periods required until they synchronized
        """
        # Unpack parameters
        s1, s2, θ, δ, ρ = self._unpack_params()
        s1_ρ, s2_ρ = self.s1_ρ, self.s2_ρ

        return _pers_till_sync(n1_0, n2_0, s1_ρ, s2_ρ,
                               s1, s2, θ, δ, ρ, maxiter, npers)

    def create_attraction_basis(self, maxiter=250, npers=3, npts=50):
        """
        Creates an attraction basis for values of n on [0, 1] X [0, 1]
        with npts in each dimension
        """
        # Unpack parameters
        s1, s2, θ, δ, ρ = self._unpack_params()
        s1_ρ, s2_ρ = self.s1_ρ, self.s2_ρ

        ab = _create_attraction_basis(s1_ρ, s2_ρ, s1, s2, θ, δ,
                                      ρ, maxiter, npers, npts)

        return ab
```

4.4.1 Time Series of Firm Measures

We write a short function below that exploits the preceding code and plots two time series.

Each time series gives the dynamics for the two countries.

The time series share parameters but differ in their initial condition.

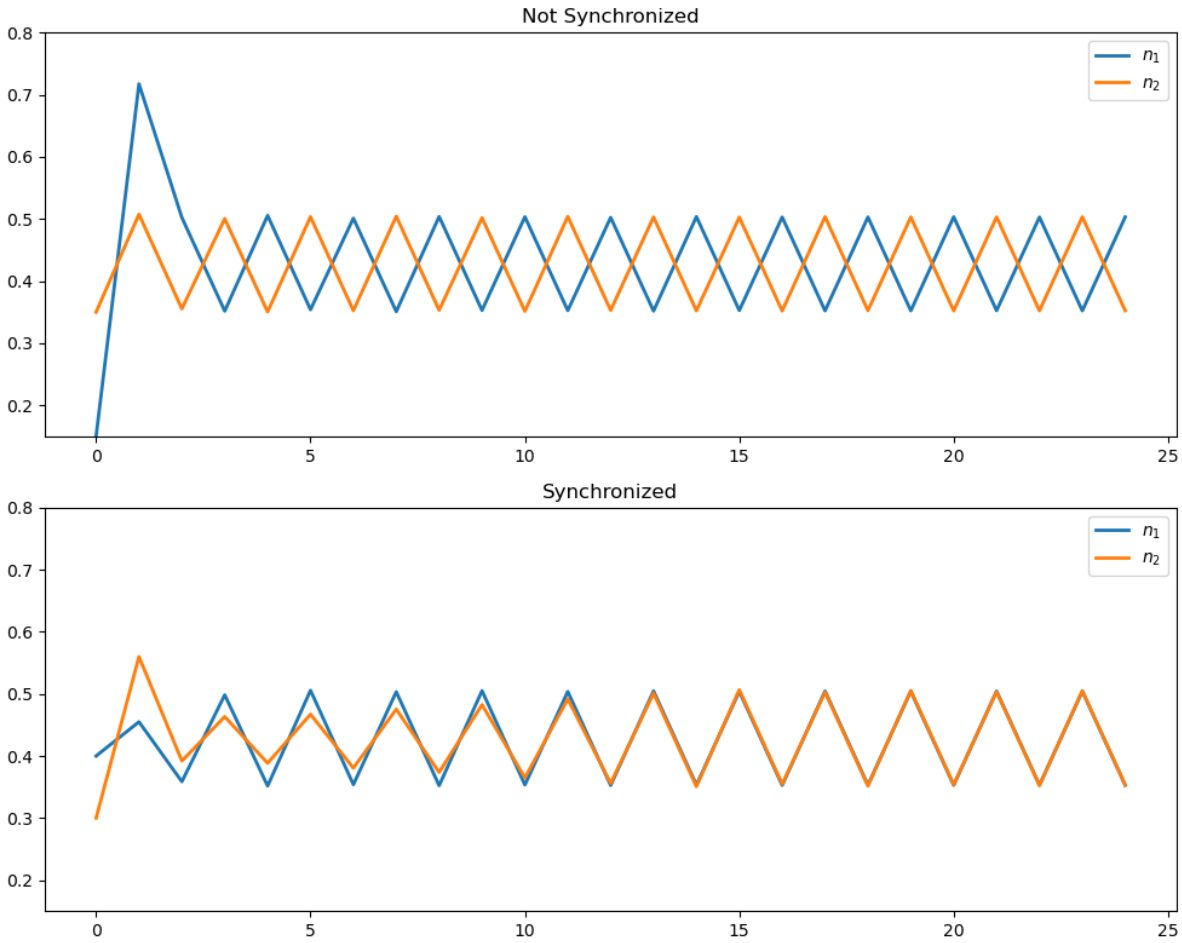
Here's the function

```
def plot_timeseries(n1_0, n2_0, s1=0.5, θ=2.5,
        δ=0.7, ρ=0.2, ax=None, title=''):
    """
    Plot a single time series with initial conditions
    """
    if ax is None:
        fig, ax = plt.subplots()

    # Create the MSG Model and simulate with initial conditions
    model = MSGSync(s1, θ, δ, ρ)
    n1, n2 = model.simulate_n(n1_0, n2_0, 25)

    ax.plot(np.arange(25), n1, label="$n_1$", lw=2)
```

(continues on next page)

(continued from previous page)

```
    ax.plot(np.arange(25), n2, label="$n_2$", lw=2)

    ax.legend()
    ax.set(title=title, ylim=(0.15, 0.8))

    return ax


# Create figure
fig, ax = plt.subplots(2, 1, figsize=(10, 8))

plot_timeseries(0.15, 0.35, ax=ax[0], title='Not Synchronized')
plot_timeseries(0.4, 0.3, ax=ax[1], title='Synchronized')

fig.tight_layout()

plt.show()
```



In the first case, innovation in the two countries does not synchronize.

In the second case, different initial conditions are chosen, and the cycles become synchronized.

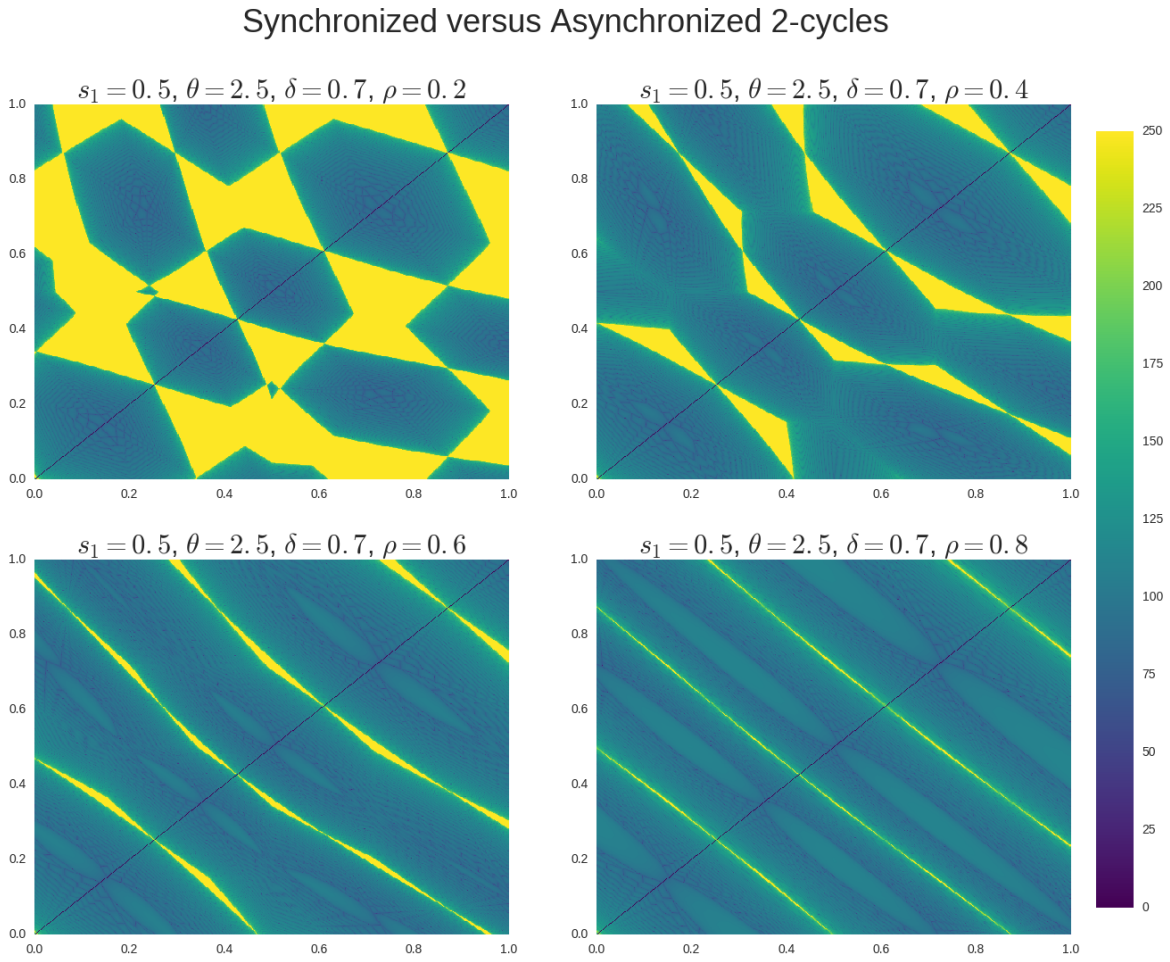
4.4.2 Basin of Attraction

Next, let's study the initial conditions that lead to synchronized cycles more systematically.

We generate time series from a large collection of different initial conditions and mark those conditions with different colors according to whether synchronization occurs or not.

The next display shows exactly this for four different parameterizations (one for each subfigure).

Dark colors indicate synchronization, while light colors indicate failure to synchronize.

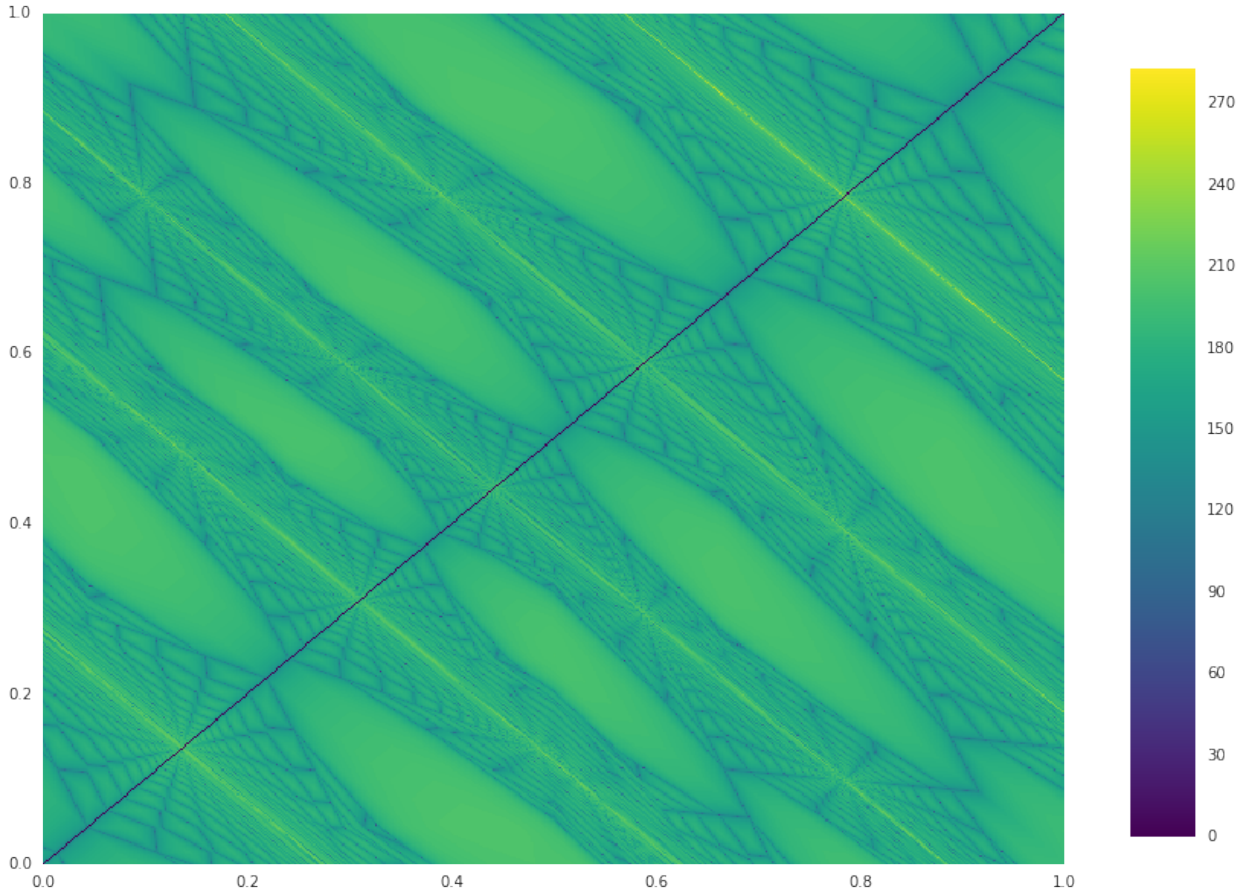


As you can see, larger values of ρ translate to more synchronization.

You are asked to replicate this figure in the exercises.

In the solution to the exercises, you'll also find a figure with sliders, allowing you to experiment with different parameters.

Here's one snapshot from the interactive figure


1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
270
240
210
180
150
120
90
60
30
0

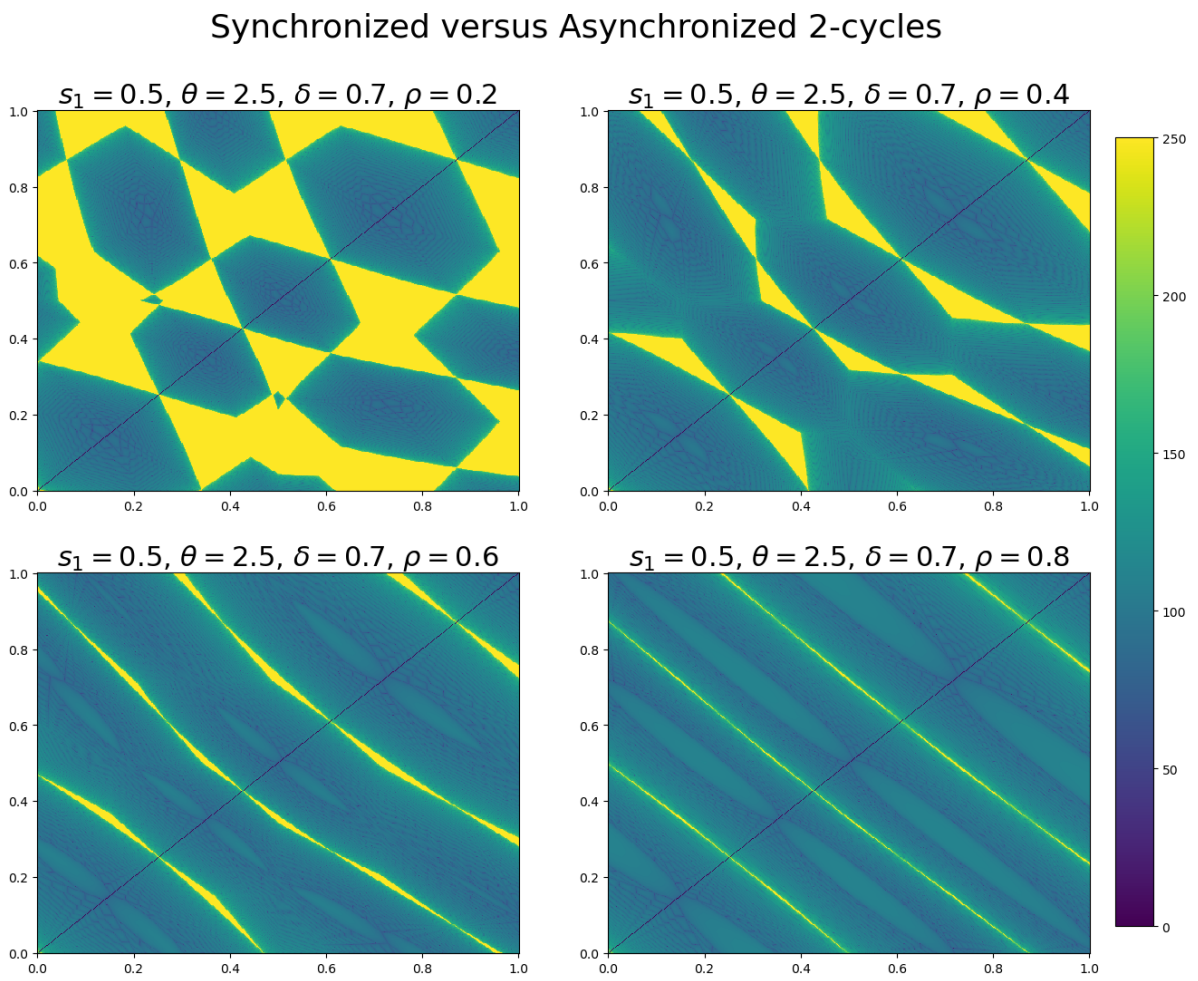
4.5 Exercises

Exercise 4.5.1

Replicate the figure *shown above* by coloring initial conditions according to whether or not synchronization occurs from those conditions.

Solution to Exercise 4.5.1

```
def plot_attraction_basis(s1=0.5, θ=2.5, δ=0.7, ρ=0.2, npts=250, ax=None):
    if ax is None:
        fig, ax = plt.subplots()

    # Create attraction basis
    unitrange = np.linspace(0, 1, npts)
    model = MSGSync(s1, θ, δ, ρ)
    ab = model.create_attraction_basis(npts=npts)
    cf = ax.pcolormesh(unitrange, unitrange, ab, cmap="viridis")

    return ab, cf


fig = plt.figure(figsize=(14, 12))

# Left - Bottom - Width - Height
ax0 = fig.add_axes((0.05, 0.475, 0.38, 0.35), label="axes0")
ax1 = fig.add_axes((0.5, 0.475, 0.38, 0.35), label="axes1")
ax2 = fig.add_axes((0.05, 0.05, 0.38, 0.35), label="axes2")
ax3 = fig.add_axes((0.5, 0.05, 0.38, 0.35), label="axes3")

params = [[0.5, 2.5, 0.7, 0.2],
          [0.5, 2.5, 0.7, 0.4],
          [0.5, 2.5, 0.7, 0.6],
          [0.5, 2.5, 0.7, 0.8]]

ab0, cf0 = plot_attraction_basis(*params[0], npts=500, ax=ax0)
ab1, cf1 = plot_attraction_basis(*params[1], npts=500, ax=ax1)
ab2, cf2 = plot_attraction_basis(*params[2], npts=500, ax=ax2)
ab3, cf3 = plot_attraction_basis(*params[3], npts=500, ax=ax3)

cbar_ax = fig.add_axes([0.9, 0.075, 0.03, 0.725])
plt.colorbar(cf0, cax=cbar_ax)

ax0.set_title(r"$s_1=0.5$, $\theta=2.5$, $\delta=0.7$, $\rho=0.2$",
              fontsize=22)
ax1.set_title(r"$s_1=0.5$, $\theta=2.5$, $\delta=0.7$, $\rho=0.4$",
              fontsize=22)
ax2.set_title(r"$s_1=0.5$, $\theta=2.5$, $\delta=0.7$, $\rho=0.6$",
              fontsize=22)
ax3.set_title(r"$s_1=0.5$, $\theta=2.5$, $\delta=0.7$, $\rho=0.8$",
              fontsize=22)

fig.suptitle("Synchronized versus Asynchronized 2-cycles",
             x=0.475, y=0.915, size=26)
plt.show()
```



Additionally, instead of just seeing 4 plots at once, we might want to manually be able to change ρ and see how it affects the plot in real-time. Below we use an interactive plot to do this.

Note, interactive plotting requires the ipywidgets module to be installed and enabled.

```
def interact_attraction_basis(ρ=0.2, maxiter=250, npts=250):
    # Create the figure and axis that we will plot on
    fig, ax = plt.subplots(figsize=(12, 10))

    # Create model and attraction basis
    s1, θ, δ = 0.5, 2.5, 0.75
    model = MSGSync(s1, θ, δ, ρ)
    ab = model.create_attraction_basis(maxiter=maxiter, npts=npts)

    # Color map with colormesh
    unitrange = np.linspace(0, 1, npts)
    cf = ax.pcolormesh(unitrange, unitrange, ab, cmap="viridis")
    cbar_ax = fig.add_axes([0.95, 0.15, 0.05, 0.7])
    plt.colorbar(cf, cax=cbar_ax)
    plt.show()
    return None
```

```
fig = interact(interact_attraction_basis,
```

(continues on next page)

(continued from previous page)

```
                ρ=(0.0, 1.0, 0.05),
                maxiter=(50, 5000, 50),
                npts=(25, 750, 25))
```

```
interactive(children=(FloatSlider(value=0.2, description='ρ', max=1.0, step=0.05),␣
↪IntSlider(value=250, descri…
```

CHAPTER

FIVE

COASE'S THEORY OF THE FIRM

Contents

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

```
!pip install interpolation
```

5.1 Overview

In 1937, Ronald Coase wrote a brilliant essay on the nature of the firm [Coase, 1937].

Coase was writing at a time when the Soviet Union was rising to become a significant industrial power.

At the same time, many free-market economies were afflicted by a severe and painful depression.

This contrast led to an intensive debate on the relative merits of decentralized, price-based allocation versus top-down planning.

In the midst of this debate, Coase made an important observation: even in free-market economies, a great deal of top-down planning does in fact take place.

This is because *firms* form an integral part of free-market economies and, within firms, allocation is by planning.

In other words, free-market economies blend both planning (within firms) and decentralized production coordinated by prices.

The question Coase asked is this: if prices and free markets are so efficient, then why do firms even exist?

Couldn't the associated within-firm planning be done more efficiently by the market?

We'll use the following imports:

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from scipy.optimize import fminbound
from interpolation import interp
```

5.1.1 Why Firms Exist

On top of asking a deep and fascinating question, Coase also supplied an illuminating answer: firms exist because of transaction costs.

Here's one example of a transaction cost:

Suppose agent A is considering setting up a small business and needs a web developer to construct and help run an online store.

She can use the labor of agent B, a web developer, by writing up a freelance contract for these tasks and agreeing on a suitable price.

But contracts like this can be time-consuming and difficult to verify

- How will agent A be able to specify exactly what she wants, to the finest detail, when she herself isn't sure how the business will evolve?
- And what if she isn't familiar with web technology? How can she specify all the relevant details?
- And, if things go badly, will failure to comply with the contract be verifiable in court?

In this situation, perhaps it will be easier to *employ* agent B under a simple labor contract.

The cost of this contract is far smaller because such contracts are simpler and more standard.

The basic agreement in a labor contract is: B will do what A asks him to do for the term of the contract, in return for a given salary.

Making this agreement is much easier than trying to map every task out in advance in a contract that will hold up in a court of law.

So agent A decides to hire agent B and a firm of nontrivial size appears, due to transaction costs.

5.1.2 A Trade-Off

Actually, we haven't yet come to the heart of Coase's investigation.

The issue of why firms exist is a binary question: should firms have positive size or zero size?

A better and more general question is: **what determines the size of firms**?

The answer Coase came up with was that "a firm will tend to expand until the costs of organizing an extra transaction within the firm become equal to the costs of carrying out the same transaction by means of an exchange on the open market…" ([Coase, 1937], p. 395).

But what are these internal and external costs?

In short, Coase envisaged a trade-off between

- transaction costs, which add to the expense of operating *between* firms, and
- diminishing returns to management, which adds to the expense of operating *within* firms

We discussed an example of transaction costs above (contracts).

The other cost, diminishing returns to management, is a catch-all for the idea that big operations are increasingly costly to manage.

For example, you could think of management as a pyramid, so hiring more workers to implement more tasks requires expansion of the pyramid, and hence labor costs grow at a rate more than proportional to the range of tasks.

Diminishing returns to management makes in-house production expensive, favoring small firms.

5.1.3 Summary

Here's a summary of our discussion:

- Firms grow because transaction costs encourage them to take some operations in house.
- But as they get large, in-house operations become costly due to diminishing returns to management.
- The size of firms is determined by balancing these effects, thereby equalizing the marginal costs of each form of operation.

5.1.4 A Quantitative Interpretation

Coases ideas were expressed verbally, without any mathematics.

In fact, his essay is a wonderful example of how far you can get with clear thinking and plain English.

However, plain English is not good for quantitative analysis, so let's bring some mathematical and computation tools to bear.

In doing so we'll add a bit more structure than Coase did, but this price will be worth paying.

Our exposition is based on [Kikuchi *et al.*, 2018].

5.2 The Model

The model we study involves production of a single unit of a final good.

Production requires a linearly ordered chain, requiring sequential completion of a large number of processing stages.

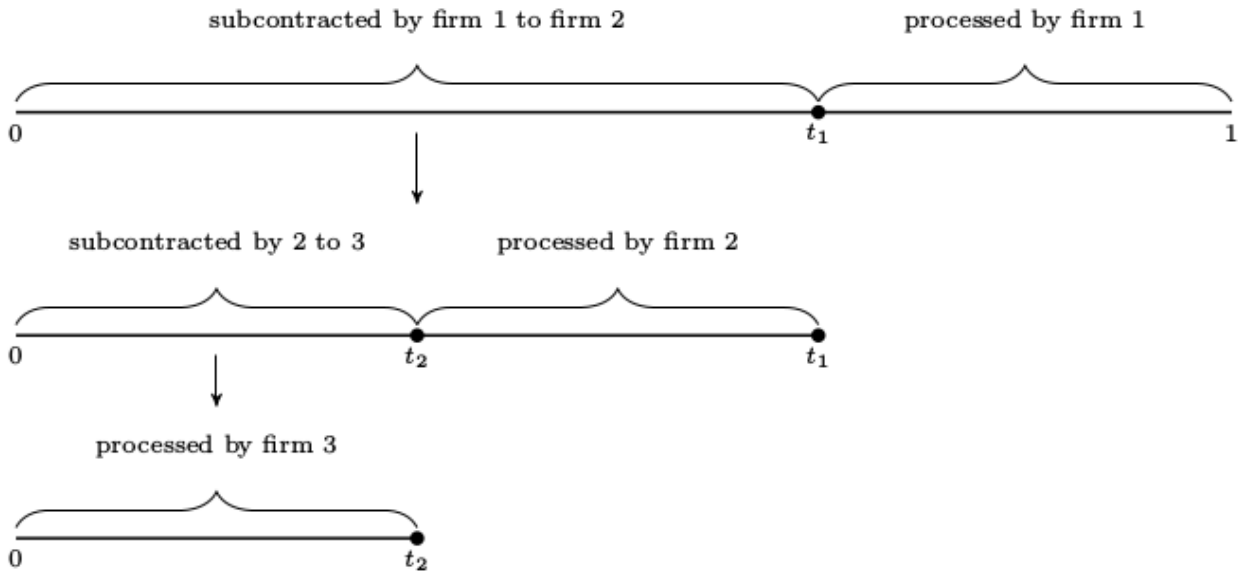
The stages are indexed by $t \in [0, 1]$, with $t = 0$ indicating that no tasks have been undertaken and $t = 1$ indicating that the good is complete.

5.2.1 Subcontracting

The subcontracting scheme by which tasks are allocated across firms is illustrated in the figure below

In this example,

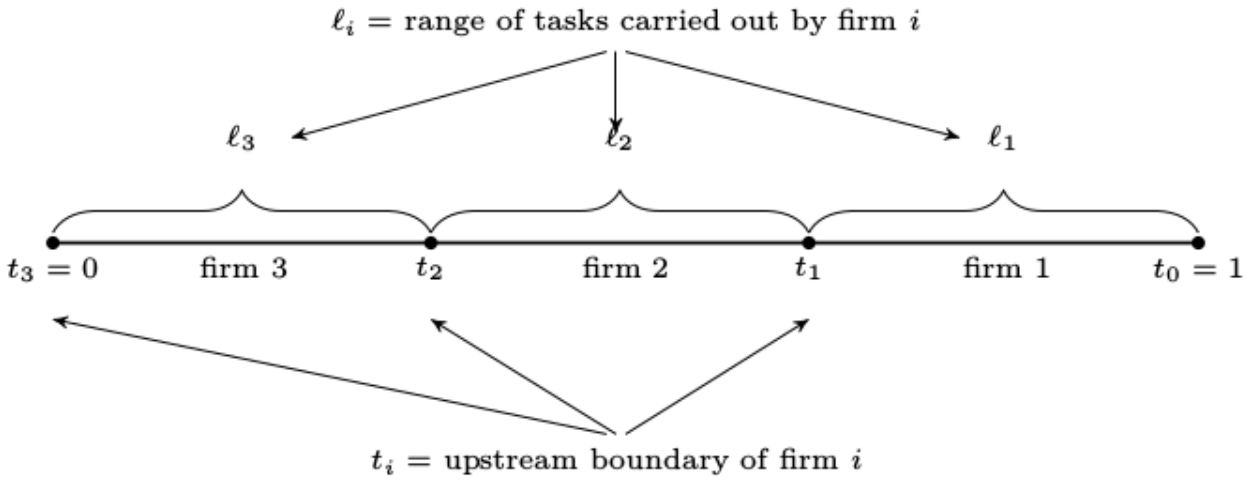
- Firm 1 receives a contract to sell one unit of the completed good to a final buyer.
- Firm 1 then forms a contract with firm 2 to purchase the partially completed good at stage t_1, with the intention of implementing the remaining $1 - t_1$ tasks in-house (i.e., processing from stage t_1 to stage 1).
- Firm 2 repeats this procedure, forming a contract with firm 3 to purchase the good at stage t_2.
- Firm 3 decides to complete the chain, selecting $t_3 = 0$.



At this point, production unfolds in the opposite direction (i.e., from upstream to downstream).

- Firm 3 completes processing stages from $t_3 = 0$ up to t_2 and transfers the good to firm 2.
- Firm 2 then processes from t_2 up to t_1 and transfers the good to firm 1,
- Firm 1 processes from t_1 to 1 and delivers the completed good to the final buyer.

The length of the interval of stages (range of tasks) carried out by firm i is denoted by ℓ_i.



Each firm chooses only its *upstream* boundary, treating its downstream boundary as given.

The benefit of this formulation is that it implies a recursive structure for the decision problem for each firm.

In choosing how many processing stages to subcontract, each successive firm faces essentially the same decision problem as the firm above it in the chain, with the only difference being that the decision space is a subinterval of the decision space for the firm above.

We will exploit this recursive structure in our study of equilibrium.

5.2.2 Costs

Recall that we are considering a trade-off between two types of costs.

Let's discuss these costs and how we represent them mathematically.

Diminishing returns to management means rising costs per task when a firm expands the range of productive activities coordinated by its managers.

We represent these ideas by taking the cost of carrying out ℓ tasks in-house to be $c(\ell)$, where c is increasing and strictly convex.

Thus, the average cost per task rises with the range of tasks performed in-house.

We also assume that c is continuously differentiable, with $c(0) = 0$ and $c'(0) > 0$.

Transaction costs are represented as a wedge between the buyer's and seller's prices.

It matters little for us whether the transaction cost is borne by the buyer or the seller.

Here we assume that the cost is borne only by the buyer.

In particular, when two firms agree to a trade at face value v, the buyer's total outlay is δv, where $\delta > 1$.

The seller receives only v, and the difference is paid to agents outside the model.

5.3 Equilibrium

We assume that all firms are *ex-ante* identical and act as price takers.

As price takers, they face a price function p, which is a map from $[0, 1]$ to $\mathbb{R}_+$, with $p(t)$ interpreted as the price of the good at processing stage t.

There is a countable infinity of firms indexed by i and no barriers to entry.

The cost of supplying the initial input (the good processed up to stage zero) is set to zero for simplicity.

Free entry and the infinite fringe of competitors rule out positive profits for incumbents, since any incumbent could be replaced by a member of the competitive fringe filling the same role in the production chain.

Profits are never negative in equilibrium because firms can freely exit.

5.3.1 Informal Definition of Equilibrium

An equilibrium in this setting is an allocation of firms and a price function such that

1. all active firms in the chain make zero profits, including suppliers of raw materials
2. no firm in the production chain has an incentive to deviate, and
3. no inactive firms can enter and extract positive profits

5.3.2 Formal Definition of Equilibrium

Let's make this definition more formal.

(You might like to skip this section on first reading)

An **allocation** of firms is a nonnegative sequence $\{\ell_i\}_{i\in\mathbb{N}}$ such that $\ell_i = 0$ for all sufficiently large i.

Recalling the figures above,

- ℓ_i represents the range of tasks implemented by the i-th firm

As a labeling convention, we assume that firms enter in order, with firm 1 being the furthest downstream.

An allocation $\{\ell_i\}$ is called **feasible** if $\sum_{i\geq 1} \ell_i = 1$.

In a feasible allocation, the entire production process is completed by finitely many firms.

Given a feasible allocation, $\{\ell_i\}$, let $\{t_i\}$ represent the corresponding transaction stages, defined by

$$t_0 = s \quad \text{and} \quad t_i = t_{i-1} - \ell_i \tag{5.1}$$

In particular, t_{i-1} is the downstream boundary of firm i and t_i is its upstream boundary.

As transaction costs are incurred only by the buyer, its profits are

$$\pi_i = p(t_{i-1}) - c(\ell_i) - \delta p(t_i) \tag{5.2}$$

Given a price function p and a feasible allocation $\{\ell_i\}$, let

- $\{t_i\}$ be the corresponding firm boundaries.
- $\{\pi_i\}$ be corresponding profits, as defined in (5.2).

This price-allocation pair is called an **equilibrium** for the production chain if

1. $p(0) = 0$,
2. $\pi_i = 0$ for all i, and
3. $p(s) - c(s-t) - \delta p(t) \leq 0$ for any pair s, t with $0 \leq s \leq t \leq 1$.

The rationale behind these conditions was given in our informal definition of equilibrium above.

5.4 Existence, Uniqueness and Computation of Equilibria

We have defined an equilibrium but does one exist? Is it unique? And, if so, how can we compute it?

5.4.1 A Fixed Point Method

To address these questions, we introduce the operator T mapping a nonnegative function p on $[0, 1]$ to Tp via

$$Tp(s) = \min_{t\leq s} \{c(s-t) + \delta p(t)\} \quad \text{for all} \quad s \in [0, 1]. \tag{5.3}$$

Here and below, the restriction $0 \leq t$ in the minimum is understood.

The operator T is similar to a Bellman operator.

Under this analogy, p corresponds to a value function and δ to a discount factor.

But $\delta > 1$, so T is not a contraction in any obvious metric, and in fact, $T^n p$ diverges for many choices of p.

Nevertheless, there exists a domain on which T is well-behaved: the set of convex increasing continuous functions $p\colon [0,1] \to \mathbb{R}$ such that $c'(0)s \leq p(s) \leq c(s)$ for all $0 \leq s \leq 1$.

We denote this set of functions by $\mathcal{P}$.

In [Kikuchi *et al.*, 2018] it is shown that the following statements are true:

1. T maps $\mathcal{P}$ into itself.
2. T has a unique fixed point in $\mathcal{P}$, denoted below by p^*.
3. For all $p \in \mathcal{P}$ we have $T^k p \to p^*$ uniformly as $k \to \infty$.

Now consider the choice function

$$t^*(s) := \text{ the solution to } \min_{t \leq s}\{c(s-t) + \delta p^*(t)\} \tag{5.4}$$

By definition, $t^*(s)$ is the cost-minimizing upstream boundary for a firm that is contracted to deliver the good at stage s and faces the price function p^*.

Since p^* lies in $\mathcal{P}$ and since c is strictly convex, it follows that the right-hand side of (5.4) is continuous and strictly convex in t.

Hence the minimizer $t^*(s)$ exists and is uniquely defined.

We can use t^* to construct an equilibrium allocation as follows:

Recall that firm 1 sells the completed good at stage $s = 1$, its optimal upstream boundary is $t^*(1)$.

Hence firm 2's optimal upstream boundary is $t^*(t^*(1))$.

Continuing in this way produces the sequence $\{t_i^*\}$ defined by

$$t_0^* = 1 \quad \text{and} \quad t_i^* = t^*(t_{i-1}) \tag{5.5}$$

The sequence ends when a firm chooses to complete all remaining tasks.

We label this firm (and hence the number of firms in the chain) as

$$n^* := \inf\{i \in \mathbb{N} : t_i^* = 0\} \tag{5.6}$$

The task allocation corresponding to (5.5) is given by $\ell_i^* := t_{i-1}^* - t_i^*$ for all i.

In [Kikuchi *et al.*, 2018] it is shown that

1. The value n^* in (5.6) is well-defined and finite,
2. the allocation $\{\ell_i^*\}$ is feasible, and
3. the price function p^* and this allocation together forms an equilibrium for the production chain.

While the proofs are too long to repeat here, much of the insight can be obtained by observing that, as a fixed point of T, the equilibrium price function must satisfy

$$p^*(s) = \min_{t \leq s}\{c(s-t) + \delta p^*(t)\} \quad \text{for all} \quad s \in [0,1] \tag{5.7}$$

From this equation, it is clear that so profits are zero for all incumbent firms.

5.4.2 Marginal Conditions

We can develop some additional insights on the behavior of firms by examining marginal conditions associated with the equilibrium.

As a first step, let $\ell^*(s) := s - t^*(s)$.

This is the cost-minimizing range of in-house tasks for a firm with downstream boundary s.

In [Kikuchi *et al.*, 2018] it is shown that t^* and ℓ^* are increasing and continuous, while p^* is continuously differentiable at all $s \in (0, 1)$ with

$$(p^*)'(s) = c'(\ell^*(s)) \tag{5.8}$$

Equation (5.8) follows from $p^*(s) = \min_{t \leq s} \{c(s-t) + \delta p^*(t)\}$ and the envelope theorem for derivatives.

A related equation is the first order condition for $p^*(s) = \min_{t \leq s} \{c(s-t) + \delta p^*(t)\}$, the minimization problem for a firm with upstream boundary s, which is

$$\delta (p^*)'(t^*(s)) = c'(s - t^*(s)) \tag{5.9}$$

This condition matches the marginal condition expressed verbally by Coase that we stated above:

> "A firm will tend to expand until the costs of organizing an extra transaction within the firm become equal to the costs of carrying out the same transaction by means of an exchange on the open market…"

Combining (5.8) and (5.9) and evaluating at $s = t_i$, we see that active firms that are adjacent satisfy

$$\delta \, c'(\ell^*_{i+1}) = c'(\ell^*_i) \tag{5.10}$$

In other words, the marginal in-house cost per task at a given firm is equal to that of its upstream partner multiplied by gross transaction cost.

This expression can be thought of as a **Coase–Euler equation,** which determines inter-firm efficiency by indicating how two costly forms of coordination (markets and management) are jointly minimized in equilibrium.

5.5 Implementation

For most specifications of primitives, there is no closed-form solution for the equilibrium as far as we are aware.

However, we know that we can compute the equilibrium corresponding to a given transaction cost parameter δ and a cost function c by applying the results stated above.

In particular, we can

1. fix initial condition $p \in \mathcal{P}$,
2. iterate with T until $T^n p$ has converged to p^*, and
3. recover firm choices via the choice function (5.3)

At each iterate, we will use continuous piecewise linear interpolation of functions.

To begin, here's a class to store primitives and a grid:

```
class ProductionChain:

    def __init__(self,
                 n=1000,
```

(continues on next page)

(continued from previous page)

```
            delta=1.05,
            c=lambda t: np.exp(10 * t) - 1):

        self.n, self.delta, self.c = n, delta, c
        self.grid = np.linspace(1e-04, 1, n)
```

Now let's implement and iterate with T until convergence.

Recalling that our initial condition must lie in $\mathcal{P}$, we set $p_0 = c$

```
def compute_prices(pc, tol=1e-5, max_iter=5000):
    """
    Compute prices by iterating with T

        * pc is an instance of ProductionChain
        * The initial condition is p = c

    """
    delta, c, n, grid = pc.delta, pc.c, pc.n, pc.grid
    p = c(grid)  # Initial condition is c(s), as an array
    new_p = np.empty_like(p)
    error = tol + 1
    i = 0

    while error > tol and i < max_iter:
        for j, s in enumerate(grid):
            Tp = lambda t: delta * interp(grid, p, t) + c(s - t)
            new_p[j] = Tp(fminbound(Tp, 0, s))
        error = np.max(np.abs(p - new_p))
        p = new_p
        i = i + 1

    if i < max_iter:
        print(f"Iteration converged in {i} steps")
    else:
        print(f"Warning: iteration hit upper bound {max_iter}")

    p_func = lambda x: interp(grid, p, x)
    return p_func
```

The next function computes optimal choice of upstream boundary and range of task implemented for a firm face price function p_function and with downstream boundary s.

```
def optimal_choices(pc, p_function, s):
    """
    Takes p_func as the true function, minimizes on [0,s]

    Returns optimal upstream boundary t_star and optimal size of
    firm ell_star

    In fact, the algorithm minimizes on [-1,s] and then takes the
    max of the minimizer and zero. This results in better results
    close to zero

    """
    delta, c = pc.delta, pc.c
```

(continues on next page)

(continued from previous page)

```
    f = lambda t: delta * p_function(t) + c(s - t)
    t_star = max(fminbound(f, -1, s), 0)
    ell_star = s - t_star
    return t_star, ell_star
```

The allocation of firms can be computed by recursively stepping through firms' choices of their respective upstream boundary, treating the previous firm's upstream boundary as their own downstream boundary.

In doing so, we start with firm 1, who has downstream boundary $s = 1$.

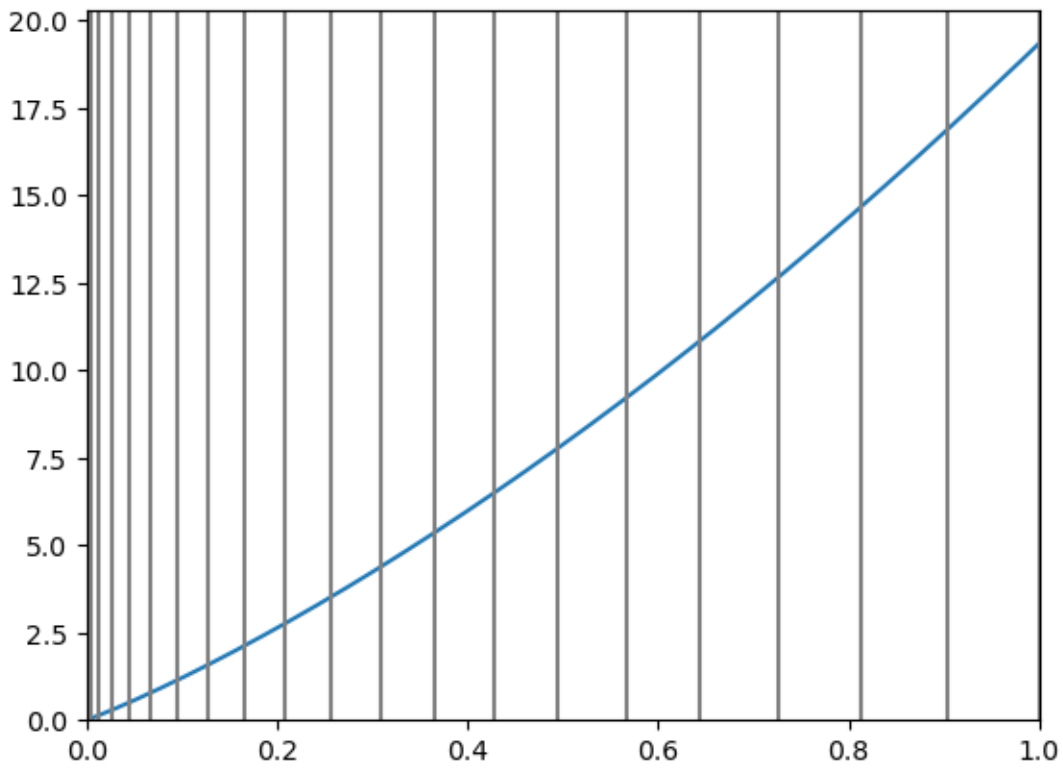
```
def compute_stages(pc, p_function):
    s = 1.0
    transaction_stages = [s]
    while s > 0:
        s, ell = optimal_choices(pc, p_function, s)
        transaction_stages.append(s)
    return np.array(transaction_stages)
```

Let's try this at the default parameters.

The next figure shows the equilibrium price function, as well as the boundaries of firms as vertical lines

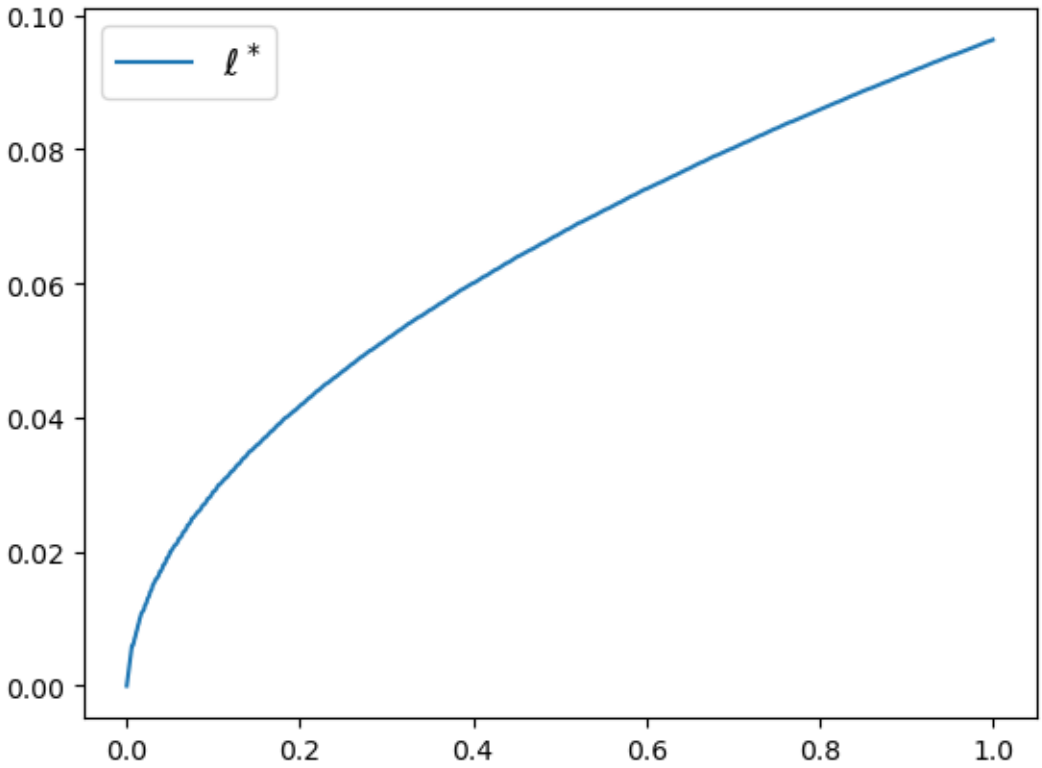
```
pc = ProductionChain()
p_star = compute_prices(pc)

transaction_stages = compute_stages(pc, p_star)

fig, ax = plt.subplots()

ax.plot(pc.grid, p_star(pc.grid))
ax.set_xlim(0.0, 1.0)
ax.set_ylim(0.0)
for s in transaction_stages:
    ax.axvline(x=s, c="0.5")
plt.show()
```

```
Iteration converged in 2 steps
```



Here's the function ℓ^*, which shows how large a firm with downstream boundary s chooses to be

```
ell_star = np.empty(pc.n)
for i, s in enumerate(pc.grid):
    t, e = optimal_choices(pc, p_star, s)
    ell_star[i] = e

fig, ax = plt.subplots()
ax.plot(pc.grid, ell_star, label="$\ell^*$")
ax.legend(fontsize=14)
plt.show()
```



Note that downstream firms choose to be larger, a point we return to below.

5.6 Exercises

Exercise 5.6.1

The number of firms is endogenously determined by the primitives.

What do you think will happen in terms of the number of firms as δ increases? Why?

Check your intuition by computing the number of firms at delta in (1.01, 1.05, 1.1).

Solution to Exercise 5.6.1

Here is one solution

```
for delta in (1.01, 1.05, 1.1):

    pc = ProductionChain(delta=delta)
    p_star = compute_prices(pc)
    transaction_stages = compute_stages(pc, p_star)
    num_firms = len(transaction_stages)
    print(f"When delta={delta} there are {num_firms} firms")
```

```
Iteration converged in 2 steps
When delta=1.01 there are 64 firms
```

```
Iteration converged in 2 steps
When delta=1.05 there are 41 firms
```

```
Iteration converged in 2 steps
When delta=1.1 there are 35 firms
```

Exercise 5.6.2

The **value added** of firm i is $v_i := p^*(t_{i-1}) - p^*(t_i)$.

One of the interesting predictions of the model is that value added is increasing with downstreamness, as are several other measures of firm size.

Can you give any intution?

Try to verify this phenomenon (value added increasing with downstreamness) using the code above.

Solution to Exercise 5.6.2

Firm size increases with downstreamness because p^*, the equilibrium price function, is increasing and strictly convex.

This means that, for a given producer, the marginal cost of the input purchased from the producer just upstream from itself in the chain increases as we go further downstream.

Hence downstream firms choose to do more in house than upstream firms — and are therefore larger.

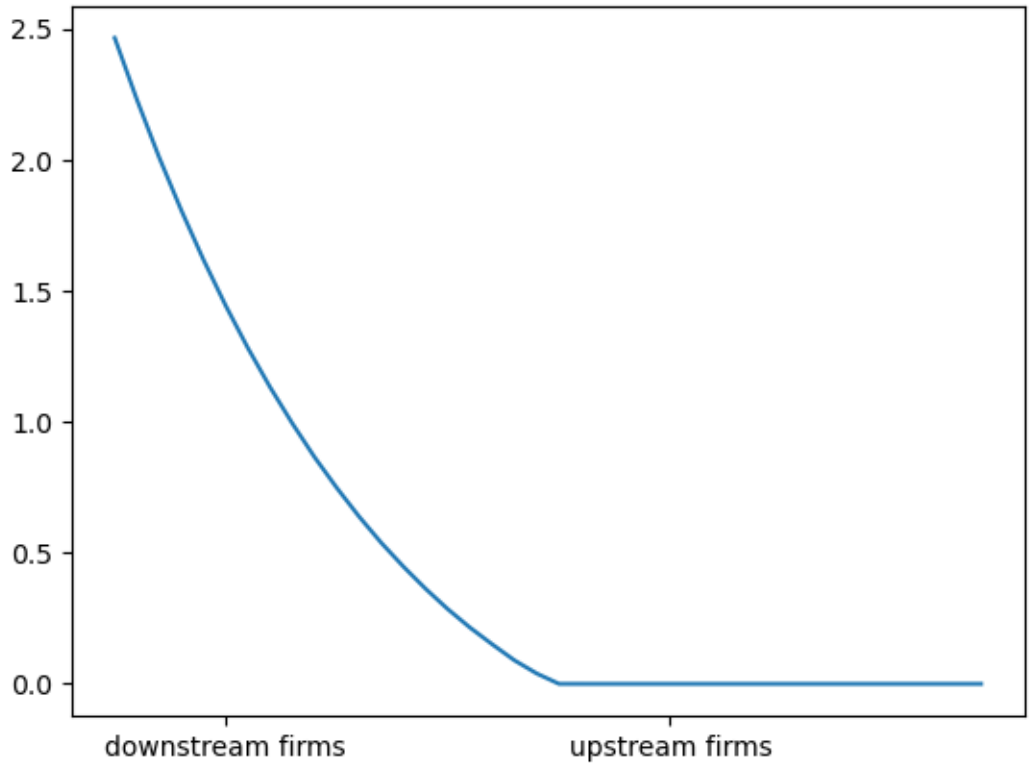
The equilibrium price function is strictly convex due to both transaction costs and diminishing returns to management.

One way to put this is that firms are prevented from completely mitigating the costs associated with diminishing returns to management — which induce convexity — by transaction costs. This is because transaction costs force firms to have nontrivial size.

Here's one way to compute and graph value added across firms

```
pc = ProductionChain()
p_star = compute_prices(pc)
stages = compute_stages(pc, p_star)

va = []

for i in range(len(stages) - 1):
    va.append(p_star(stages[i]) - p_star(stages[i+1]))

fig, ax = plt.subplots()
ax.plot(va, label="value added by firm")
ax.set_xticks((5, 25))
ax.set_xticklabels(("downstream firms", "upstream firms"))
plt.show()
```

```
Iteration converged in 2 steps
```


2.5
2.0
1.5
1.0
0.5
0.0
downstream firms
upstream firms

Part II

Auctions & Other Applications

CHAPTER

SIX

FIRST-PRICE AND SECOND-PRICE AUCTIONS

This lecture is designed to set the stage for a subsequent lecture about Multiple Good Allocation Mechanisms

In that lecture, a planner or auctioneer simultaneously allocates several goods to set of people.

In the present lecture, a single good is allocated to one person within a set of people.

Here we'll learn about and simulate two classic auctions :

- a First-Price Sealed-Bid Auction (FPSB)
- a Second-Price Sealed-Bid Auction (SPSB) created by William Vickrey [Vickrey, 1961]

We'll also learn about and apply a

- Revenue Equivalent Theorem

We recommend watching this video about second price auctions by Anders Munk-Nielsen:

https://youtu.be/qwWk_Bqtue8

and

https://youtu.be/eYTGQCGpmXI

Anders Munk-Nielsen put his code on GitHub.

Much of our Python code below is based on his.

6.1 First-Price Sealed-Bid Auction (FPSB)

Protocols:

- A single good is auctioned.
- Prospective buyers simultaneously submit sealed bids.
- Each bidder knows only his/her own bid.
- The good is allocated to the person who submits the highest bid.
- The winning bidder pays price she has bid.

Detailed Setting:

There are $n > 2$ prospective buyers named $i = 1, 2, \dots, n$.

Buyer i attaches value v_i to the good being sold.

Buyer i wants to maximize the expected value of her **surplus** defined as $v_i - p$, where p is the price that she pays, conditional on her winning the auction.

Evidently,

- If i bids exactly v_i, she pays what she thinks it is worth and gathers no surplus value.
- Buyer i will never want to bid more than v_i.
- If buyer i bids $b < v_i$ and wins the auction, then she gathers surplus value $b - v_i > 0$.
- If buyer i bids $b < v_i$ and someone else bids more than b, buyer i loses the auction and gets no surplus value.
- To proceed, buyer i wants to know the probability that she wins the auction as a function of her bid v_i
  - this requires that she know a probability distribution of bids v_j made by prospective buyers $j \neq i$
- Given her idea about that probability distribution, buyer i wants to set a bid that maximizes the mathematical expectation of her surplus value.

Bids are sealed, so no bidder knows bids submitted by other prospective buyers.

This means that bidders are in effect participating in a game in which players do not know **payoffs** of other players.

This is a **Bayesian game**, a Nash equilibrium of which is called a **Bayesian Nash equilibrium**.

To complete the specification of the situation, we'll assume that prospective buyers' valuations are independently and identically distributed according to a probability distribution that is known by all bidders.

Bidder optimally chooses to bid less than v_i.

6.1.1 Characterization of FPSB Auction

A FPSB auction has a unique symmetric Bayesian Nash Equilibrium.

The optimal bid of buyer i is

$$\mathbf{E}[y_i | y_i < v_i] \tag{6.1}$$

where v_i is the valuation of bidder i and y_i is the maximum valuation of all other bidders:

$$y_i = \max_{j \neq i} v_j \tag{6.2}$$

A proof for this assertion is available at the Wikepedia page about Vickrey auctions

6.2 Second-Price Sealed-Bid Auction (SPSB)

Protocols: In a second-price sealed-bid (SPSB) auction, the winner pays the second-highest bid.

6.3 Characterization of SPSB Auction

In a SPSB auction bidders optimally choose to bid their values.

Formally, a dominant strategy profile in a SPSB auction with a single, indivisible item has each bidder bidding its value.

A proof is provided at the Wikepedia page about Vickrey auctions

6.4 Uniform Distribution of Private Values

We assume valuation v_i of bidder i is distributed $v_i \overset{\text{i.i.d.}}{\sim} U(0,1)$.

Under this assumption, we can analytically compute probability distributions of prices bid in both FPSB and SPSB.

We'll simulate outcomes and, by using a law of large numbers, verify that the simulated outcomes agree with analytical ones.

We can use our simulation to illustrate a **Revenue Equivalence Theorem** that asserts that on average first-price and second-price sealed bid auctions provide a seller the same revenue.

To read about the revenue equivalence theorem, see this Wikepedia page

6.5 Setup

There are n bidders.

Each bidder knows that there are $n-1$ other bidders.

6.6 First price sealed bid auction

An optimal bid for bidder i in a **FPSB** is described by equations (6.1) and (6.2).

When bids are i.i.d. draws from a uniform distribution, the CDF of y_i is

$$\begin{aligned}\tilde{F}_{n-1}(y) = \mathbf{P}(y_i \leq y) = \mathbf{P}(\max_{j \neq i} v_j \leq y) \\ = \prod_{j \neq i} \mathbf{P}(v_j \leq y) \\ = y^{n-1}\end{aligned}$$

and the PDF of y_i is $\tilde{f}_{n-1}(y) = (n-1)y^{n-2}$.

Then bidder i's optimal bid in a **FPSB** auction is:

$$\begin{aligned}\mathbf{E}(y_i | y_i < v_i) &= \frac{\int_0^{v_i} y_i \tilde{f}_{n-1}(y_i) dy_i}{\int_0^{v_i} \tilde{f}_{n-1}(y_i) dy_i} \\ &= \frac{\int_0^{v_i} (n-1) y_i^{n-1} dy_i}{\int_0^{v_i} (n-1) y_i^{n-2} dy_i} \\ &= \frac{n-1}{n} y_i \Big|_0^{v_i} \\ &= \frac{n-1}{n} v_i\end{aligned}$$

6.7 Second Price Sealed Bid Auction

In a **SPSB**, it is optimal for bidder i to bid v_i.

6.8 Python Code

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import scipy.stats as stats
import scipy.interpolate as interp

# for plots
plt.rcParams.update({"text.usetex": True, 'font.size': 14})
colors = plt. rcParams['axes.prop_cycle'].by_key()['color']

# ensure the notebook generate the same randomess
np.random.seed(1337)
```

We repeat an auction with 5 bidders for 100,000 times.

The valuations of each bidder is distributed $U(0, 1)$.

```
N = 5
R = 100_000

v = np.random.uniform(0,1,(N,R))

# BNE in first-price sealed bid

b_star = lambda vi,N :((N-1)/N) * vi
b = b_star(v,N)
```

We compute and sort bid price distributions that emerge under both FPSB and SPSB.

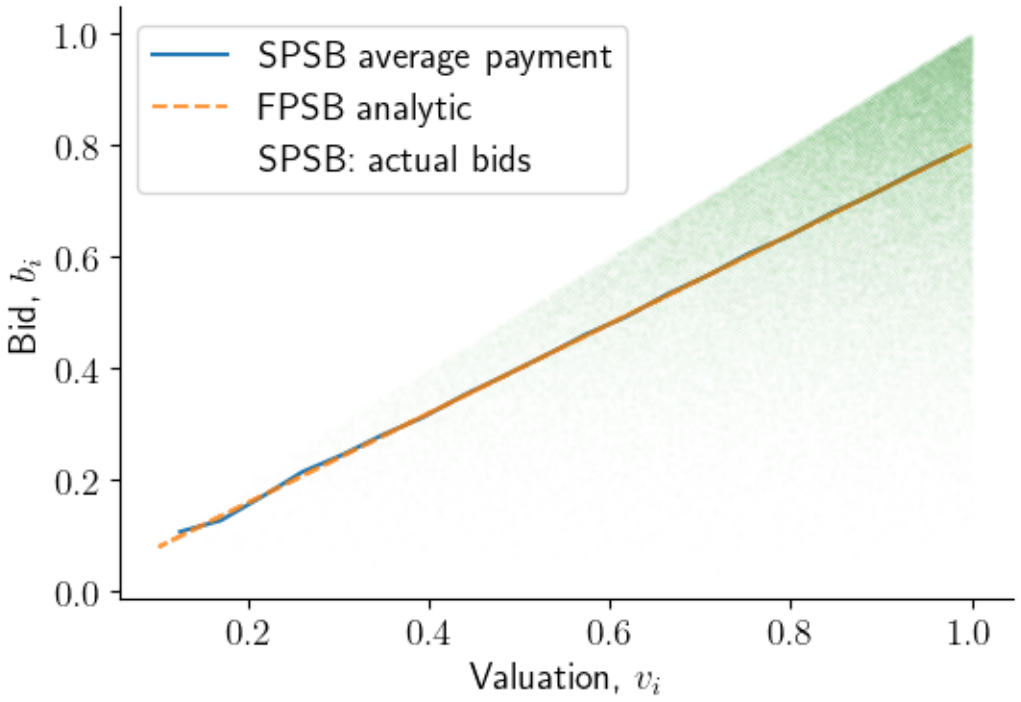
```
idx = np.argsort(v, axis=0)  # Biders' values are sorted in ascending order in each␣
↪auction.
# We record the order because we want to apply it to bid price and their id.

v = np.take_along_axis(v, idx, axis=0)  # same as np.sort(v, axis=0), except now we␣
↪retain the idx
b = np.take_along_axis(b, idx, axis=0)

ii = np.repeat(np.arange(1,N+1)[:,None], R, axis=1)  # the id for the bidders is␣
↪created.
ii = np.take_along_axis(ii, idx, axis=0)  # the id is sorted according to bid price␣
↪as well.

winning_player = ii[-1,:] # In FPSB and SPSB, winners are those with highest values.

winner_pays_fpsb = b[-1,:]  # highest bid
winner_pays_spsb = v[-2,:]  # 2nd-highest valuation
```

Let's now plot the *winning* bids $b_{(n)}$ (i.e. the payment) against valuations, $v_{(n)}$ for both FPSB and SPSB.

Note that

- FPSB: There is a unique bid corresponding to each valuation
- SPSB: Because it equals the valuation of a second-highest bidder, what a winner pays varies even holding fixed the winner's valuation. So here there is a frequency distribution of payments for each valuation.

```
# We intend to compute average payments of different groups of bidders

binned = stats.binned_statistic(v[-1,:], v[-2,:], statistic='mean', bins=20)
xx = binned.bin_edges
xx = [(xx[ii]+xx[ii+1])/2 for ii in range(len(xx)-1)]
yy = binned.statistic

fig, ax = plt.subplots(figsize=(6, 4))

ax.plot(xx, yy, label='SPSB average payment')
ax.plot(v[-1,:], b[-1,:], '--', alpha = 0.8, label = 'FPSB analytic')
ax.plot(v[-1,:], v[-2,:], 'o', alpha = 0.05, markersize = 0.1, label = 'SPSB: actual␣
↪bids')

ax.legend(loc='best')
ax.set_xlabel('Valuation, $v_i$')
ax.set_ylabel('Bid, $b_i$')
sns.despine()
```



6.9 Revenue Equivalence Theorem

We now compare FPSB and a SPSB auctions from the point of view of the revenues that a seller can expect to acquire.

Expected Revenue FPSB:

The winner with valuation y pays $\frac{n-1}{n} * y$, where n is the number of bidders.

Above we computed that the CDF is $F_n(y) = y^n$ and the PDF is $f_n = ny^{n-1}$.

Consequently, expected revenue is

$$\mathbf{R} = \int_0^1 \frac{n-1}{n} v_i \times n v_i^{n-1} dv_i = \frac{n-1}{n+1}$$

Expected Revenue SPSB:

The expected revenue equals n $\times$ expected payment of a bidder.

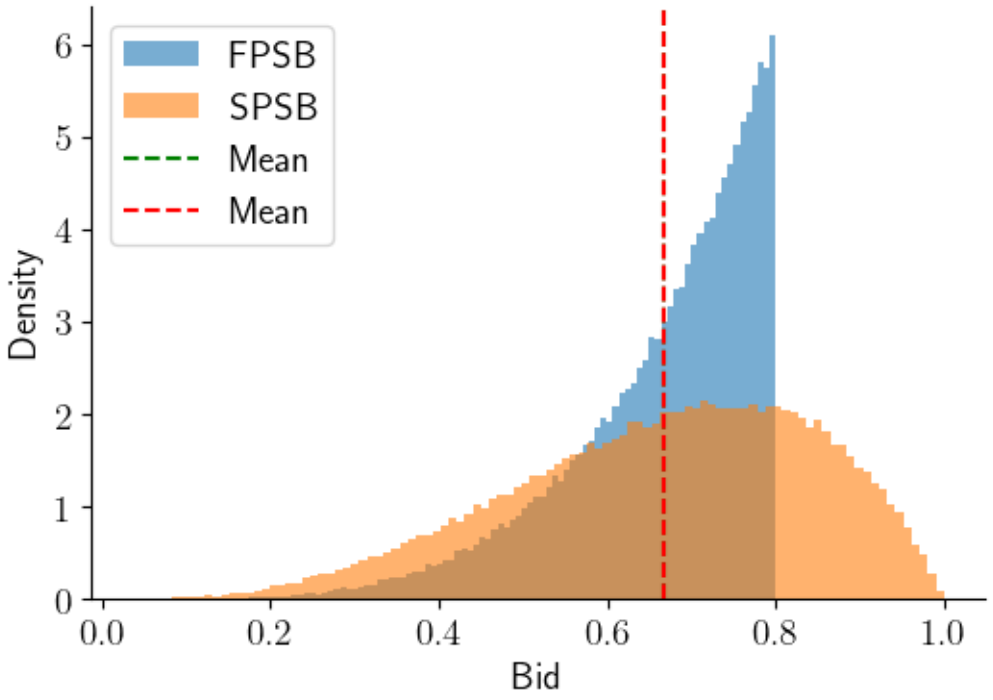
Computing this we get

$$\begin{aligned}
\mathbf{TR} &= n\mathbf{E}_{\mathbf{v_i}}\left[\mathbf{E}_{\mathbf{y_i}}[y_i|y_i < v_i]\mathbf{P}(y_i < v_i) + 0 \times \mathbf{P}(y_i > v_i)\right] \\
&= n\mathbf{E}_{\mathbf{v_i}}\left[\mathbf{E}_{\mathbf{y_i}}[y_i|y_i < v_i]\tilde{F}_{n-1}(v_i)\right] \\
&= n\mathbf{E}_{\mathbf{v_i}}[\frac{n-1}{n} \times v_i \times v_i^{n-1}] \\
&= (n-1)\mathbf{E}_{\mathbf{v_i}}[v_i^n] \\
&= \frac{n-1}{n+1}
\end{aligned}$$

Thus, while probability distributions of winning bids typically differ across the two types of auction, we deduce that expected payments are identical in FPSB and SPSB.

```
fig, ax = plt.subplots(figsize=(6, 4))

for payment,label in zip([winner_pays_fpsb, winner_pays_spsb], ['FPSB', 'SPSB']):
    print('The average payment of %s: %.4f. Std.: %.4f. Median: %.4f'% (label,
 ↪payment.mean(),payment.std(),np.median(payment)))
    ax.hist(payment, density=True, alpha=0.6, label=label, bins=100)

ax.axvline(winner_pays_fpsb.mean(), ls='--', c='g', label='Mean')
ax.axvline(winner_pays_spsb.mean(), ls='--', c='r', label='Mean')

ax.legend(loc='best')
ax.set_xlabel('Bid')
ax.set_ylabel('Density')
sns.despine()
```

```
The average payment of FPSB: 0.6665. Std.: 0.1129. Median: 0.6967
The average payment of SPSB: 0.6667. Std.: 0.1782. Median: 0.6862
```



Summary of FPSB and SPSB results with uniform distribution on $[0, 1]$

Auction: Sealed-Bid	First-Price	Second-Price
Winner	Agent with highest bid	Agent with highest bid
Winner pays	Winner's bid	Second-highest bid
Loser pays	0	0
Dominant strategy	No dominant strategy	Bidding truthfully is dominant strategy
Bayesian Nash equilibrium	Bidder i bids $\frac{n-1}{n}v_i$	Bidder i truthfully bids v_i
Auctioneer's revenue	$\frac{n-1}{n+1}$	$\frac{n-1}{n+1}$

Detour: Computing a Bayesian Nash Equibrium for FPSB

The Revenue Equivalence Theorem lets us an optimal bidding strategy for a FPSB auction from outcomes of a SPSB auction.

Let $b(v_i)$ be the optimal bid in a FPSB auction.

The revenue equivalence theorem tells us that a bidder agent with value v_i on average receives the same **payment** in the two types of auction.

Consequently,

$$b(v_i)\mathbf{P}(y_i < v_i) + 0 * \mathbf{P}(y_i \geq v_i) = \mathbf{E}_{y_i}[y_i | y_i < v_i]\mathbf{P}(y_i < v_i) + 0 * \mathbf{P}(y_i \geq v_i)$$

It follows that an optimal bidding strategy in a FPSB auction is $b(v_i) = \mathbf{E}_{y_i}[y_i | y_i < v_i]$.

6.10 Calculation of Bid Price in FPSB

In equations (6.1) and (6.1), we displayed formulas for optimal bids in a symmetric Bayesian Nash Equilibrium of a FPSB auction.

$$\mathbf{E}[y_i|y_i < v_i]$$

where

- $v_i =$ value of bidder i
- $y_i =$: maximum value of all bidders except i, i.e., $y_i = \max_{j \neq i} v_j$

Above, we computed an optimal bid price in a FPSB auction analytically for a case in which private values are uniformly distributed.

For most probability distributions of private values, analytical solutions aren't easy to compute.

Instead, we can compute bid prices in FPSB auctions numerically as functions of the distribution of private values.

```
def evaluate_largest(v_hat, array, order=1):
    """
    A method to estimate the largest (or certain-order largest) value of the other␣
↪biders,
    conditional on player 1 wins the auction.

    Parameters:
    ----------
    v_hat : float, the value of player 1. The biggest value in the auction that␣
↪player 1 wins.

    array: 2 dimensional array of bidders' values in shape of (N,R),
           where N: number of players, R: number of auctions

    order: int. The order of largest number among bidders who lose.
                e.g. the order for largest number beside winner is 1.
                     the order for second-largest number beside winner is 2.

    """
    N,R = array.shape
    array_residual=array[1:,:].copy()  # drop the first row because we assume first␣
↪row is the winner's bid

    index=(array_residual<v_hat).all(axis=0)
    array_conditional=array_residual[:,index].copy()

    array_conditional=np.sort(array_conditional, axis=0)
    return array_conditional[-order,:].mean()
```

We can check the accuracy of our `evaluate_largest` method by comparing it with an analytical solution.

We find that despite small discrepancy, the evaluate_largest method functions well.

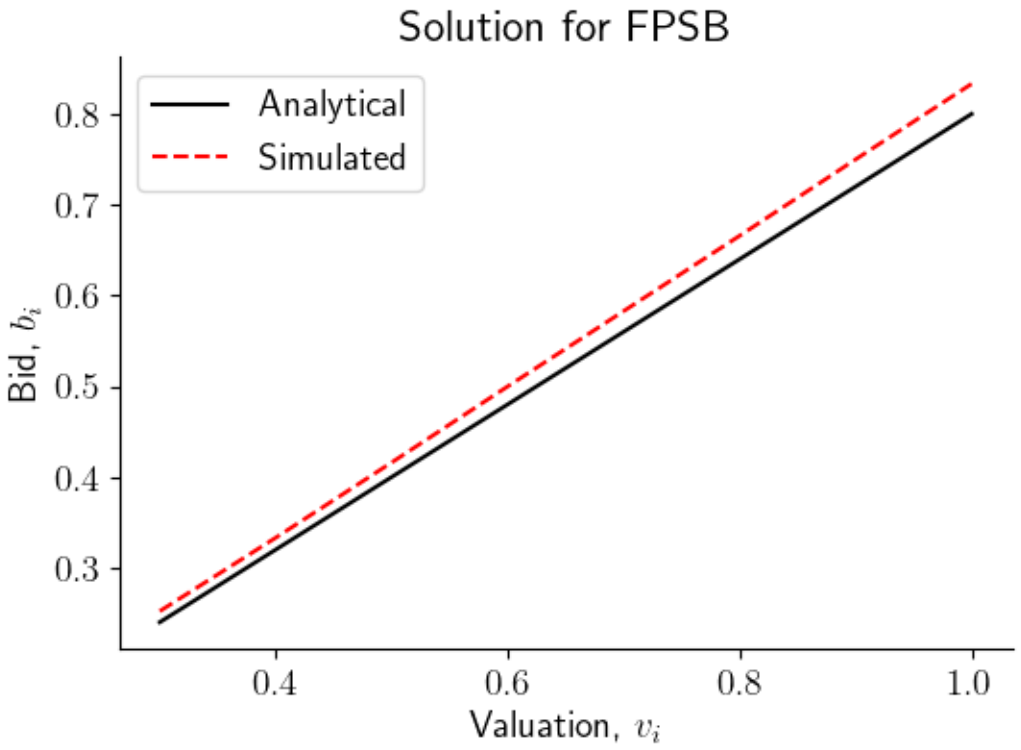
Furthermore, if we take a very large number of auctions, say 1 million, the discrepancy disappears.

```
v_grid = np.linspace(0.3,1,8)
bid_analytical = b_star(v_grid,N)
bid_simulated = [evaluate_largest(ii, v) for ii in v_grid]
```

(continues on next page)

(continued from previous page)

```
fig, ax = plt.subplots(figsize=(6, 4))

ax.plot(v_grid, bid_analytical, '-', color='k', label='Analytical')
ax.plot(v_grid, bid_simulated, '--', color='r', label='Simulated')

ax.legend(loc='best')
ax.set_xlabel('Valuation, $v_i$')
ax.set_ylabel('Bid, $b_i$')
ax.set_title('Solution for FPSB')
sns.despine()
```

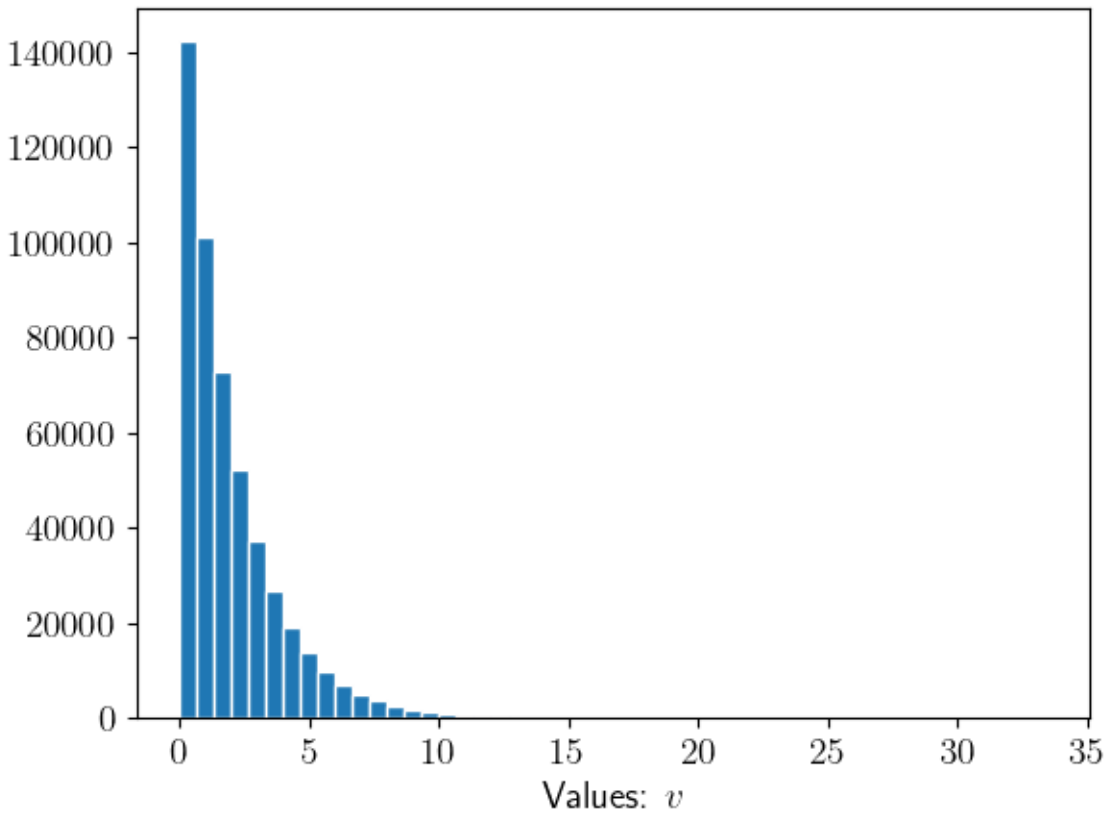


6.11 χ^2 Distribution

Let's try an example in which the distribution of private values is a χ^2 distribution.

We'll start by taking a look at a χ^2 distribution with the help of the following Python code:

```
np.random.seed(1337)
v = np.random.chisquare(df=2, size=(N*R,))

plt.hist(v, bins=50, edgecolor='w')
plt.xlabel('Values: $v$')
plt.show()
```



Now we'll get Python to construct a bid price function

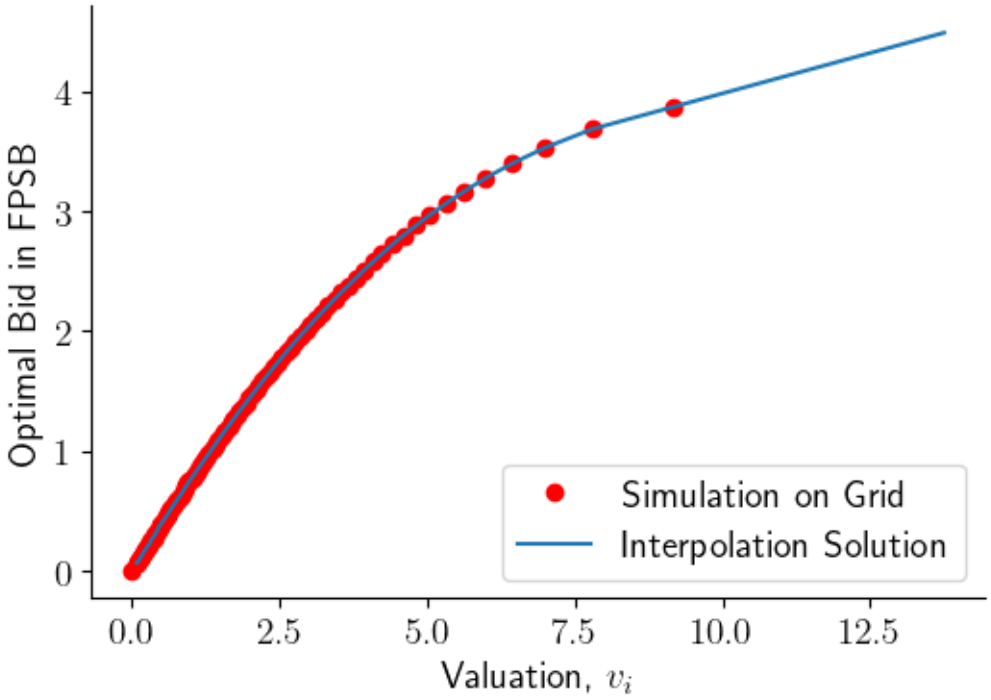
```
np.random.seed(1337)
v = np.random.chisquare(df=2, size=(N,R))


# we compute the quantile of v as our grid
pct_quantile = np.linspace(0, 100, 101)[1:-1]
v_grid = np.percentile(v.flatten(), q=pct_quantile)

EV=[evaluate_largest(ii, v) for ii in v_grid]
# nan values are returned for some low quantiles due to lack of observations
```

```
/tmp/ipykernel_6757/521884726.py:25: RuntimeWarning: Mean of empty slice.
  return array_conditional[-order,:].mean()
/home/runner/miniconda3/envs/quantecon/lib/python3.11/site-packages/numpy/core/_
↪methods.py:129: RuntimeWarning: invalid value encountered in scalar divide
  ret = ret.dtype.type(ret / rcount)
```

```
# we insert 0 into our grid and bid price function as a complement
EV=np.insert(EV,0,0)
v_grid=np.insert(v_grid,0,0)

b_star_num = interp.interp1d(v_grid, EV, fill_value="extrapolate")
```

We check our bid price function by computing and visualizing the result.

```
pct_quantile_fine = np.linspace(0, 100, 1001)[1:-1]
v_grid_fine = np.percentile(v.flatten(), q=pct_quantile_fine)

fig, ax = plt.subplots(figsize=(6, 4))

ax.plot(v_grid, EV, 'or', label='Simulation on Grid')
ax.plot(v_grid_fine, b_star_num(v_grid_fine) , '-', label='Interpolation Solution')

ax.legend(loc='best')
ax.set_xlabel('Valuation, $v_i$')
ax.set_ylabel('Optimal Bid in FPSB')
sns.despine()
```

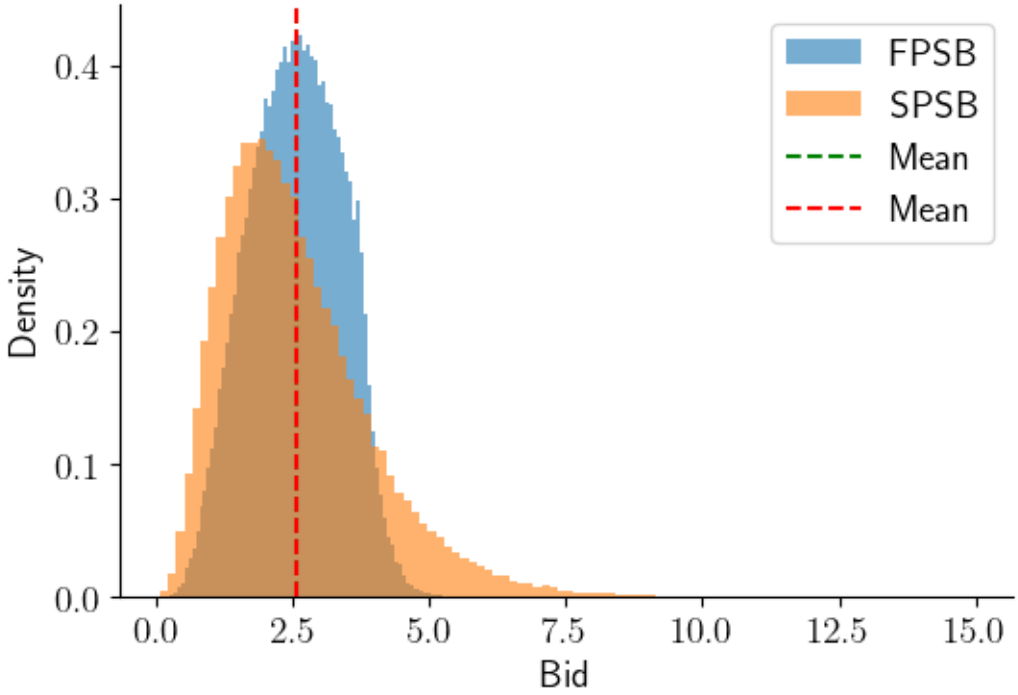


Now we can use Python to compute the probability distribution of the price paid by the winning bidder

```
b=b_star_num(v)

idx = np.argsort(v, axis=0)
v = np.take_along_axis(v, idx, axis=0)  # same as np.sort(v, axis=0), except now we␣
 ↪retain the idx
b = np.take_along_axis(b, idx, axis=0)

ii = np.repeat(np.arange(1,N+1)[:,None], R, axis=1)
ii = np.take_along_axis(ii, idx, axis=0)

winning_player = ii[-1,:]

winner_pays_fpsb = b[-1,:]  # highest bid
winner_pays_spsb = v[-2,:]  # 2nd-highest valuation
```

```
fig, ax = plt.subplots(figsize=(6, 4))

for payment,label in zip([winner_pays_fpsb, winner_pays_spsb], ['FPSB', 'SPSB']):
    print('The average payment of %s: %.4f. Std.: %.4f. Median: %.4f'% (label,
 ↪payment.mean(),payment.std(),np.median(payment)))
    ax.hist(payment, density=True, alpha=0.6, label=label, bins=100)

ax.axvline(winner_pays_fpsb.mean(), ls='--', c='g', label='Mean')
ax.axvline(winner_pays_spsb.mean(), ls='--', c='r', label='Mean')

ax.legend(loc='best')
ax.set_xlabel('Bid')
ax.set_ylabel('Density')
sns.despine()
```

```
The average payment of FPSB: 2.5693. Std.: 0.8383. Median: 2.5829
The average payment of SPSB: 2.5661. Std.: 1.3580. Median: 2.3180
```



6.12 5 Code Summary

We assemble the functions that we have used into a Python class

```
class bid_price_solution:

    def __init__(self, array):
        """
        A class that can plot the value distribution of bidders,
        compute the optimal bid price for bidders in FPSB
```

(continues on next page)

(continued from previous page)

```
        and plot the distribution of winner's payment in both FPSB and SPSB

        Parameters:
        ----------

        array: 2 dimensional array of bidders' values in shape of (N,R),
               where N: number of players, R: number of auctions

        """
        self.value_mat=array.copy()

        return None

    def plot_value_distribution(self):
        plt.hist(self.value_mat.flatten(), bins=50, edgecolor='w')
        plt.xlabel('Values: $v$')
        plt.show()

        return None

    def evaluate_largest(self, v_hat, order=1):
        N,R = self.value_mat.shape
        array_residual = self.value_mat[1:,:].copy()
        # drop the first row because we assume first row is the winner's bid

        index=(array_residual<v_hat).all(axis=0)
        array_conditional=array_residual[:,index].copy()

        array_conditional=np.sort(array_conditional, axis=0)

        return array_conditional[-order,:].mean()

    def compute_optimal_bid_FPSB(self):
        # we compute the quantile of v as our grid
        pct_quantile = np.linspace(0, 100, 101)[1:-1]
        v_grid = np.percentile(self.value_mat.flatten(), q=pct_quantile)

        EV=[self.evaluate_largest(ii) for ii in v_grid]
        # nan values are returned for some low quantiles due to lack of observations

        # we insert 0 into our grid and bid price function as a complement
        EV=np.insert(EV,0,0)
        v_grid=np.insert(v_grid,0,0)

        self.b_star_num = interp.interp1d(v_grid, EV, fill_value="extrapolate")

        pct_quantile_fine = np.linspace(0, 100, 1001)[1:-1]
        v_grid_fine = np.percentile(self.value_mat.flatten(), q=pct_quantile_fine)

        fig, ax = plt.subplots(figsize=(6, 4))

        ax.plot(v_grid, EV, 'or', label='Simulation on Grid')
        ax.plot(v_grid_fine, self.b_star_num(v_grid_fine) , '-', label='Interpolation␣
↪Solution')

        ax.legend(loc='best')
```

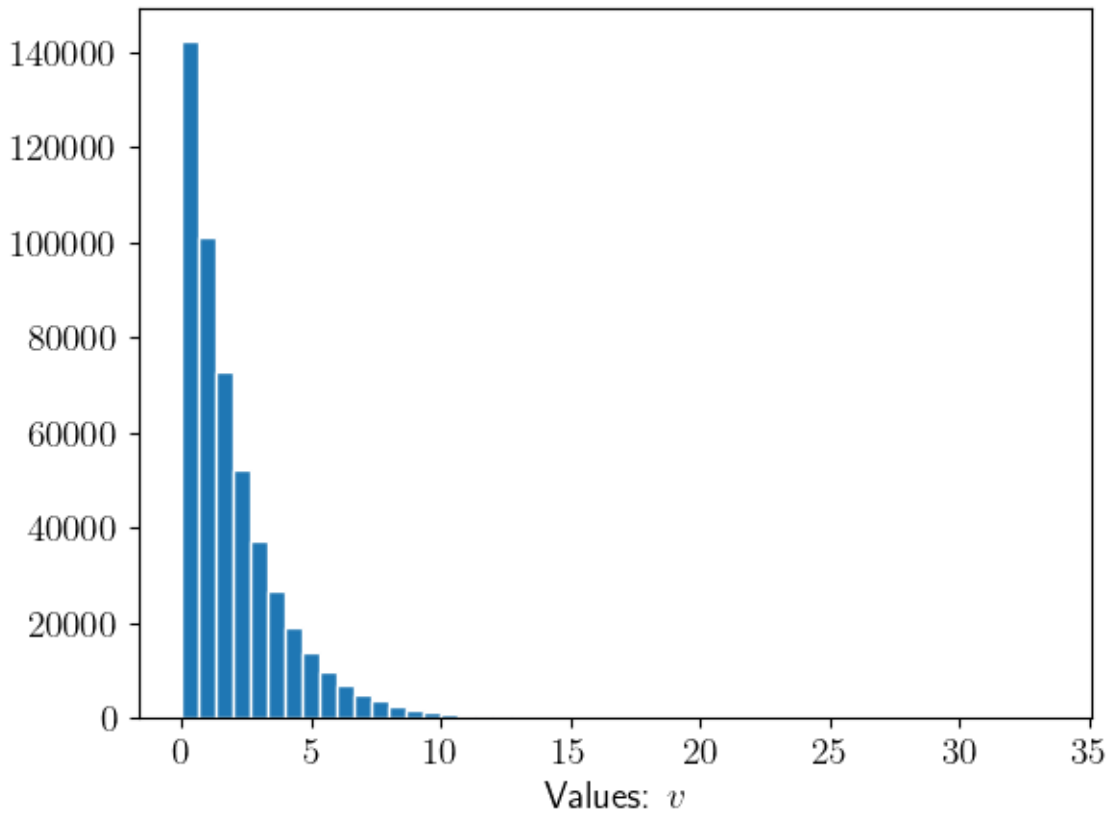
(continues on next page)

(continued from previous page)

```
        ax.set_xlabel('Valuation, $v_i$')
        ax.set_ylabel('Optimal Bid in FPSB')
        sns.despine()

        return None

    def plot_winner_payment_distribution(self):
        self.b = self.b_star_num(self.value_mat)

        idx = np.argsort(self.value_mat, axis=0)
        self.v = np.take_along_axis(self.value_mat, idx, axis=0)  # same as np.sort(v,
↪ axis=0), except now we retain the idx
        self.b = np.take_along_axis(self.b, idx, axis=0)

        self.ii = np.repeat(np.arange(1,N+1)[:,None], R, axis=1)
        self.ii = np.take_along_axis(self.ii, idx, axis=0)

        winning_player = self.ii[-1,:]

        winner_pays_fpsb = self.b[-1,:]  # highest bid
        winner_pays_spsb = self.v[-2,:]  # 2nd-highest valuation

        fig, ax = plt.subplots(figsize=(6, 4))

        for payment,label in zip([winner_pays_fpsb, winner_pays_spsb], ['FPSB', 'SPSB
↪']):
            print('The average payment of %s: %.4f. Std.: %.4f. Median: %.4f'%␣
↪(label,payment.mean(),payment.std(),np.median(payment)))
            ax.hist(payment, density=True, alpha=0.6, label=label, bins=100)

        ax.axvline(winner_pays_fpsb.mean(), ls='--', c='g', label='Mean')
        ax.axvline(winner_pays_spsb.mean(), ls='--', c='r', label='Mean')

        ax.legend(loc='best')
        ax.set_xlabel('Bid')
        ax.set_ylabel('Density')
        sns.despine()

        return None
```

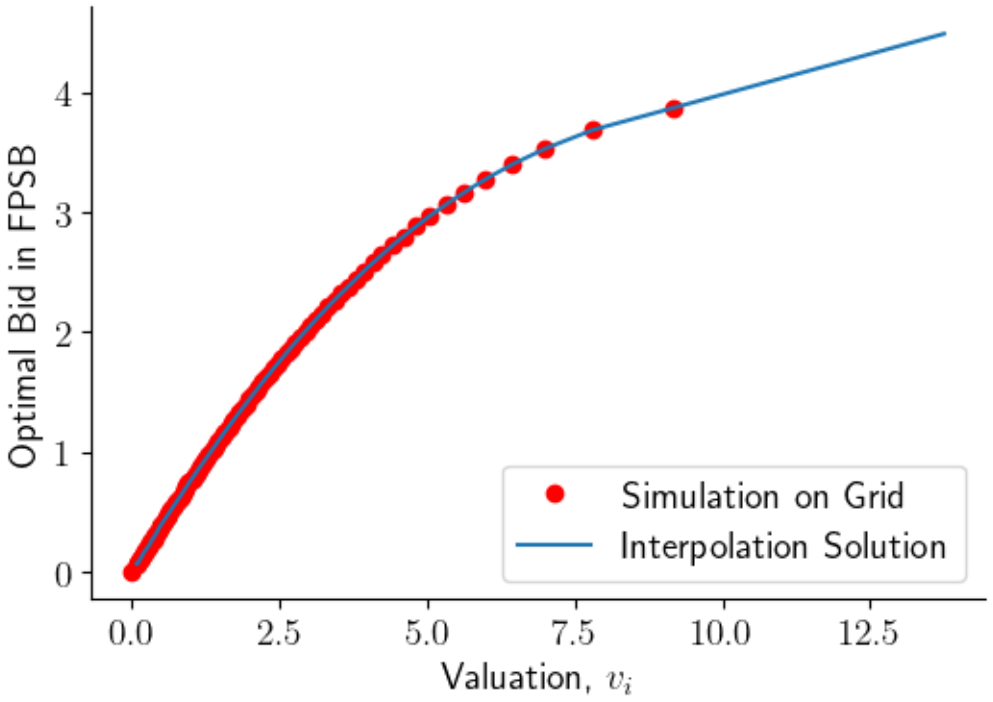
```
np.random.seed(1337)
v = np.random.chisquare(df=2, size=(N,R))

chi_squ_case = bid_price_solution(v)
```

```
chi_squ_case.plot_value_distribution()
```



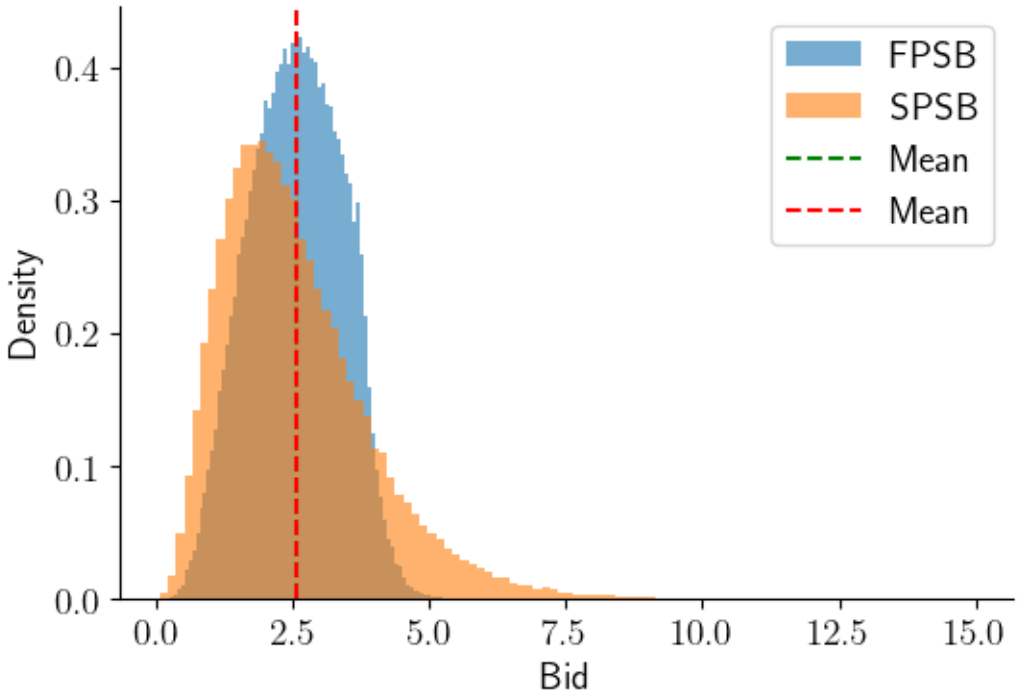

```
chi_squ_case.compute_optimal_bid_FPSB()
```

```
/tmp/ipykernel_6757/919518230.py:37: RuntimeWarning: Mean of empty slice.
  return array_conditional[-order,:].mean()
/home/runner/miniconda3/envs/quantecon/lib/python3.11/site-packages/numpy/core/_
↪methods.py:129: RuntimeWarning: invalid value encountered in scalar divide
  ret = ret.dtype.type(ret / rcount)
```




```
chi_squ_case.plot_winner_payment_distribution()
```

```
The average payment of FPSB: 2.5693. Std.: 0.8383. Median: 2.5829
The average payment of SPSB: 2.5661. Std.: 1.3580. Median: 2.3180
```



6.13 References

1. Wikipedia for FPSB: https://en.wikipedia.org/wiki/First-price_sealed-bid_auction
2. Wikipedia for SPSB: https://en.wikipedia.org/wiki/Vickrey_auction
3. Chandra Chekuri's lecture note for algorithmic game theory: http://chekuri.cs.illinois.edu/teaching/spring2008/Lectures/scribed/Notes20.pdf
4. Tim Salmon. ECO 4400 Supplemental Handout: All About Auctions: https://s2.smu.edu/tsalmon/auctions.pdf
5. Auction Theory- Revenue Equivalence Theorem: https://michaellevet.wordpress.com/2015/07/06/auction-theory-revenue-equivalence-theorem/
6. Order Statistics: https://online.stat.psu.edu/stat415/book/export/html/834

CHAPTER

SEVEN

MULTIPLE GOOD ALLOCATION MECHANISMS

```
!pip install prettytable
```

7.1 Overview

This lecture describes two mechanisms for allocating n private goods ("houses") to m people ("buyers").

We assume that $m > n$ so that there are more potential buyers than there are houses.

Prospective buyers regard the houses as **substitutes**.

Buyer j attaches value v_{ij} to house i.

These values are **private**

- v_{ij} is known only to person j unless person j chooses to tell someone.

We require that a mechanism allocate **at most** one house to one prospective buyer.

We describe two distinct mechanisms

- A multiple rounds, ascending bid auction
- A special case of a Groves-Clarke [Groves, 1973], [Clarke, 1971] mechanism with a benevolent social planner

---

Note: In 1994, the multiple rounds, ascending bid auction was actually used by Stanford University to sell leases to 9 lots on the Stanford campus to eligible faculty members.

We begin with overviews of the two mechanisms.

7.2 Ascending Bids Auction for Multiple Goods

An auction is administered by an **auctioneer**

The auctioneer has an $n \times 1$ vector r of reservation prices on the n houses.

The auctioneer sells house i only if the final price bid for it exceeds r_i

The auctioneer allocates all n houses **simultaneously**

The auctioneer does not know bidders' private values v_{ij}

There are multiple **rounds**

- during each round, active participants can submit bids on any of the n houses
- each bidder can bid on only one house during one round
- a person who was high bidder on a particular house in one round is understood to submit that same bid for the same house in the next round
- between rounds, a bidder who was not a high bidder can change the house on which he/she chooses to bid
- the auction ends when the price of no house changes from one round to the next
- all n houses are allocated after the final round
- house i is retained by the auctioneer if not prospective buyer offers more that r_i for the house

In this auction, person j never tells anyone else his/her private values v_{ij}

7.3 A Benevolent Planner

This mechanism is designed so that all prospective buyers voluntarily choose to reveal their private values to a **social planner** who uses them to construct a socially optimal allocation.

Among all feasible allocations, a **socially optimal allocation** maximizes the sum of private values across all prospective buyers.

The planner tells everyone in advance how he/she will allocate houses based on the matrix of values that prospective buyers report.

The mechanism provide every prospective buyer an incentive to reveal his vector of private values to the planner.

After the planner receives everyone's vector of private values, the planner deploys a **sequential** algorithm to determine an **allocation** of houses and a set of **fees** that he charges awardees for the negative **externality** that their presence impose on other prospective buyers.

7.4 Equivalence of Allocations

Remarkably, these two mechanisms can produce virtually identical allocations.

We construct Python code for both mechanism.

We also work out some examples by hand or almost by hand.

Next, let's dive down into the details.

7.5 Ascending Bid Auction

7.5.1 Basic Setting

We start with a more detailed description of the setting.

- A seller owns n houses that he wants to sell for the maximum possible amounts to a set of m prospective eligible buyers.
- The seller wants to sell at most one house to each potential buyer.
- There are m potential eligible buyers, identified by $j = [1, 2, \ldots, m]$

  - Each potential buyer is permitted to buy at most one house.
  - Buyer j would be willing to pay at most v_{ij} for house i.
  - Buyer j knows $v_{ij}, i = 1, \ldots, n$, but no one else does.
  - If buyer j pays p_i for house i, he enjoys surplus value $v_{ij} - p_i$.
  - Each buyer j wants to choose the i that maximizes his/her surplus value $v_{ij} - p_i$.
  - The seller wants to maximize $\sum_i p_i$.

The seller conducts a **simultaneous, multiple goods, ascending bid auction**.

Outcomes of the auction are

- An $n \times 1$ vector p of sales prices $p = [p_1, \ldots, p_n]$ for the n houses.
- An $n \times m$ matrix Q of 0's and 1's, where $Q_{ij} = 1$ if and only if person j bought house i.
- An $n \times m$ matrix S of surplus values consisting of all zeros unless person j bought house i, in which case $S_{ij} = v_{ij} - p_i$

We describe rules for the auction it terms of **pseudo code**.

The pseudo code will provide a road map for writing Python code to implement the auction.

7.6 Pseudocode

Here is a quick sketch of a possible simple structure for our Python code

Inputs:

- n, m.
- an $n \times m$ non-negative matrix v of private values
- an $n \times 1$ vector r of seller-specified reservation prices
- the seller will not accept a price less than r_i for house i
- we are free to think of these reservation prices as private values of a fictitious $m + 1$ th buyer who does not actually participate in the auction
- initial bids can be thought of starting at r
- a scalar ϵ of seller-specified minimum price-bid increments

For each round of the auction, new bids on a house must be at least the prevailing highest bid so far **plus** ϵ

Auction Protocols

- the auction consists of a finite number of **rounds**
- in each round, a prospective buyer can bid on one and only one house
- after each round, a house is temporarily awarded to the person who made the highest bid for that house
  - temporarily winning bids on each house are announced
  - this sets the stage to move on to the next round
- a new round is held
  - bids for temporary winners from the previous round are again attached to the houses on which they bid; temporary winners of the last round leave their bids from the previous round unchanged

  - all other active prospective buyers must submit a new bid on some house
  - new bids on a house must be at least equal to the prevailing temporary price that won the last round **plus** ϵ
  - if a person does not submit a new bid and was also not a temporary winner from the previous round, that person must drop out of the auction permanently
  - for each house, the highest bid, whether it is a new bid or was the temporary winner from the previous round, is announced, with the person who made that new (temporarily) winning bid being (temporarily) awarded the house to start the next round
- rounds continue until no price on **any** house changes from the previous round
- houses are sold to the winning bidders from the final round at the prices that they bid

Outputs:

- an $n \times 1$ vector p of sales prices
- an $n \times m$ matrix S of surplus values consisting of all zeros unless person j bought house i, in which case $S_{ij} = v_{ij} - p_i$
- an $n \times (m+1)$ matrix Q of 0's and 1's that tells which buyer bought which house. (The last column accounts for unsold houses.)

Proposed buyer strategy:

In this pseudo code and the actual Python code below, we'll assume that all buyers choose to use the following strategy

- The strategy is optimal for each buyer

Each buyer $j = 1, \ldots, m$ uses the same strategy.

The strategy has the form:

- Let $\check{p}^t$ be the $n \times 1$ vector of prevailing highest-bid prices at the beginning of round t
- Let $\epsilon > 0$ be the minimum bid increment specified by the seller
- For each prospective buyer j, compute the index of the best house to bid on during round t, namely $\hat{i}_t = \text{argmax}_i\{[v_{ij} - \check{p}_i^t - \epsilon]\}$
- If $\max_i\{[v_{ij} - \check{p}_i^t - \epsilon]\} \leq 0$, person j permanently drops out of the auction at round t
- If $v_{\hat{i}_t,j} - \check{p}_i^t - \epsilon > 0$, person j bids $\check{p}_i^t + \epsilon$ on house j

Resolving ambiguities: The protocols we have described so far leave open two possible sources of ambiguity.

(1) **The optimal bid choice for buyers in each round.** It is possible that a buyer has the same surplus value for multiple houses. The argmax function in Python always returns the first argmax element. We instead prefer to randomize among such winner. For that reason, we write our own argmax function below.

(2) **Seller's choice of winner if same price bid cast by several buyers.** To resolve this ambiguity, we use the np.random.choice function below.

Given the randomness in outcomes, it is possible that different allocations of houses could emerge from the same inputs.

However, this will happen only when the bid price increment ϵ is nonnegligible.

```
import numpy as np
import prettytable as pt

np.random.seed(100)
```

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

7.7 An Example

Before building a Python class, let's step by step solve things almost "by hand" to grasp how the auction proceeds.

A step-by-step procedure also helps reduce bugs, especially when the value matrix is peculiar (e.g. the differences between values are negligible, a column containing identical values or multiple buyers have the same valuation etc.).

Fortunately, our auction behaves well and robustly with various peculiar matrices.

We provide some examples later in this lecture.

```
v = np.array([[8, 5, 9, 4],
              [4, 11, 7, 4],
              [9, 7, 6, 4]])
n, m = v.shape
r = np.array([2, 1, 0])
ϵ = 1
p = r.copy()
buyer_list = np.arange(m)
house_list = np.arange(n)
```

```
v
```

```
array([[ 8,  5,  9,  4],
       [ 4, 11,  7,  4],
       [ 9,  7,  6,  4]])
```

Remember that column indexes j indicate buyers and row indexes i indicate houses.

The above value matrix v is peculiar in the sense that Buyer 3 (indexed from 0) puts the same value 4 on every house being sold.

Maybe buyer 3 is a bureaucrat who purchases these house simply by following instructions from his superior.

```
r
```

```
array([2, 1, 0])
```

```
def find_argmax_with_randomness(v):
    """
    We build our own verion of argmax function such that the argmax index will be␣
↪returned randomly
    when there are multiple maximum values. This function is similiar to np.argmax(v,
↪axis=0)

    Parameters:
    ----------
    v: 2 dimensional np.array

    """

    n, m = v.shape
    index_array = np.arange(n)
    result=[]
```

(continues on next page)

(continued from previous page)

```
    for ii in range(m):
        max_value = v[:,ii].max()
        result.append(np.random.choice(index_array[v[:,ii] == max_value]))

    return np.array(result)
```

```
def present_dict(dt):
    """
    A function that present the information in table.

    Parameters:
    ----------
    dt: dictionary.

    """

    ymtb = pt.PrettyTable()
    ymtb.field_names = ['House Number', *dt.keys()]
    ymtb.add_row(['Buyer', *dt.values()])
    print(ymtb)
```

Check Kick Off Condition

```
def check_kick_off_condition(v, r, ϵ):
    """
    A function that checks whether the auction could be initiated given the␣
 ↪reservation price and value matrix.
    To avoid the situation that the reservation prices are so high that no one would␣
 ↪even bid in the first round.

    Parameters:
    ----------
    v : value matrix of the shape (n,m).

    r: the reservation price

    ϵ: the minimun price increment in each round

    """

    # we convert the price vector to a matrix in the same shape as value matrix to␣
 ↪facilitate subtraction
    p_start = (ϵ+r)[:,None] @ np.ones(m)[None,:]

    surplus_value = v - p_start
    buyer_decision = (surplus_value > 0).any(axis = 0)
    return buyer_decision.any()
```

```
check_kick_off_condition(v, r, ϵ)
```

```
True
```

7.7.1 round 1

submit bid

```
def submit_initial_bid(p_initial, ϵ, v):
    """
    A function that describes the bid information in the first round.

    Parameters:
    ----------
    p_initial: the price (or the reservation prices) at the beginning of auction.

    v: the value matrix

    ϵ: the minimun price increment in each round

    Returns:
    ----------
    p: price array after this round of bidding

    bid_info: a dictionary that contains bidding information (house number as keys␣
 ↪and buyer as values).

    """

    p = p_initial.copy()
    p_start_mat = (ϵ + p)[:,None] @ np.ones(m)[None,:]
    surplus_value = v - p_start_mat

    # we only care about active buyers who have positve surplus values
    active_buyer_diagnosis = (surplus_value > 0).any(axis = 0)
    active_buyer_list = buyer_list[active_buyer_diagnosis]
    active_buyer_surplus_value = surplus_value[:,active_buyer_diagnosis]
    active_buyer_choice = find_argmax_with_randomness(active_buyer_surplus_value)
    # choice means the favourite houses given the current price and ϵ

    # we only retain the unique house index because prices increase once at one round
    house_bid =  list(set(active_buyer_choice))
    p[house_bid] += ϵ

    bid_info = {}
    for house_num in house_bid:
        bid_info[house_num] = active_buyer_list[active_buyer_choice == house_num]

    return p, bid_info
```

```
p, bid_info = submit_initial_bid(p, ϵ, v)
```

```
p
```

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

```
present_dict(bid_info)
```

```
+--------------+-----+-----+-------+
| House Number |  0  |  1  |   2   |
+--------------+-----+-----+-------+
|    Buyer     | [2] | [1] | [0 3] |
+--------------+-----+-----+-------+
```

check terminal condition

Notice that two buyers bid for house 2 (indexed from 0).

Because the auction protocol does not specify a selection rule in this case, we simply select a winner **randomly**.

This is reasonable because the seller can't distinguish these buyers and doesn't know the valuation of each buyer.

It is both convenient and practical for him to just pick a winner randomly.

There is a 50% probability that Buyer 3 is chosen as the winner for house 2, although he values it less than buyer 0.

In this case, buyer 0 has to bid one more time with a higher price, which crowds out Buyer 3.

Therefore, final price could be 3 or 4, depending on the winner in the last round.

```
def check_terminal_condition(bid_info, p, v):
    """
    A function that checks whether the auction ends.

    Recall that the auction ends when either losers have non-positive surplus values␣
↪for each house
    or there is no loser (every buyer gets a house).

    Parameters:
    ----------
    bid_info: a dictionary that contains bidding information of house numbers (as␣
↪keys) and buyers (as values).

    p: np.array. price array of houses

    v: value matrix

    Returns:
    ----------
    allocation: a dictionary that descirbe how the houses bid are assigned.

    winner_list: a list of winners

    loser_list: a list of losers

    """

    # there may be several buyers bidding one house, we choose a winner randomly
    winner_list=[np.random.choice(bid_info[ii]) for ii in bid_info.keys()]

    allocation = {house_num:winner for house_num,winner in zip(bid_info.keys(),winner_
↪list)}

    loser_set = set(buyer_list).difference(set(winner_list))
    loser_list = list(loser_set)
    loser_num = len(loser_list)
```

(continues on next page)

(continued from previous page)

```
    if loser_num == 0:
        print('The auction ends because every buyer gets one house.')
        return allocation,winner_list,loser_list

    p_mat = (ϵ + p)[:,None] @ np.ones(loser_num)[None,:]
    loser_surplus_value = v[:,loser_list] - p_mat
    loser_decision = (loser_surplus_value > 0).any(axis = 0)

    print(~(loser_decision.any()))

    return allocation,winner_list,loser_list
```

```
allocation,winner_list,loser_list = check_terminal_condition(bid_info, p, v)
```

```
False
```

```
present_dict(allocation)
```

```
+---------------+---+---+---+
| House Number | 0 | 1 | 2 |
+---------------+---+---+---+
|    Buyer     | 2 | 1 | 0 |
+---------------+---+---+---+
```

```
winner_list
```

```
[2, 1, 0]
```

```
loser_list
```

```
[3]
```

7.7.2 round 2

From the second round on, the auction proceeds differently from the first round.

Now only active losers (those who have positive surplus values) have an incentive to submit bids to displace temporary winners from the previous round.

```
def submit_bid(loser_list, p, ϵ, v, bid_info):
    """
    A function that executes the bid operation after the first round.
    After the first round, only active losers would cast a new bid with price as old␣
↪price + increment.
    By such bid, winners of last round are replaced by the active losers.

    Parameters:
    ----------
```

(continues on next page)

(continued from previous page)

```
    loser_list: a list that includes the indexes of losers

    p: np.array. price array of houses

    ϵ: minimum increment of bid price

    v: value matrix

    bid_info: a dictionary that contains bidding information of house numbers (as␣
↪keys) and buyers (as values).

    Returns:
    ----------
    p_end: a price array after this round of bidding

    bid_info: a dictionary that contains updated bidding information.

    """

    p_end=p.copy()

    loser_num = len(loser_list)
    p_mat = (ϵ + p_end)[:,None] @ np.ones(loser_num)[None,:]
    loser_surplus_value = v[:,loser_list] - p_mat
    loser_decision = (loser_surplus_value > 0).any(axis = 0)

    active_loser_list = np.array(loser_list)[loser_decision]
    active_loser_surplus_value = loser_surplus_value[:,loser_decision]
    active_loser_choice = find_argmax_with_randomness(active_loser_surplus_value)

    # we retain the unique house index and increasing the corresponding bid price
    house_bid = list(set(active_loser_choice))
    p_end[house_bid] += ϵ

    # we record the bidding information from active losers
    bid_info_active_loser = {}
    for house_num in house_bid:
        bid_info_active_loser[house_num] = active_loser_list[active_loser_choice ==␣
↪house_num]

    # we update the bidding information according to the bidding from actice losers
    for house_num in bid_info_active_loser.keys():
        bid_info[house_num] = bid_info_active_loser[house_num]

    return p_end,bid_info
```

```
p,bid_info = submit_bid(loser_list, p, ϵ, v, bid_info)
```

```
p
```

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

```
present_dict(bid_info)
```

```
+--------------+-----+-----+-----+
| House Number |  0  |  1  |  2  |
+--------------+-----+-----+-----+
|    Buyer     | [2] | [1] | [3] |
+--------------+-----+-----+-----+
```

```
allocation,winner_list,loser_list = check_terminal_condition(bid_info, p, v)
```

```
False
```

```
present_dict(allocation)
```

```
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Buyer     | 2 | 1 | 3 |
+--------------+---+---+---+
```

7.7.3 round 3

```
p,bid_info = submit_bid(loser_list, p, ϵ, v, bid_info)
```

```
p
```

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

```
present_dict(bid_info)
```

```
+--------------+-----+-----+-----+
| House Number |  0  |  1  |  2  |
+--------------+-----+-----+-----+
|    Buyer     | [2] | [1] | [0] |
+--------------+-----+-----+-----+
```

```
allocation,winner_list,loser_list = check_terminal_condition(bid_info, p, v)
```

```
False
```

```
present_dict(allocation)
```

```
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Buyer     | 2 | 1 | 0 |
+--------------+---+---+---+
```

7.7.4 round 4

```
p,bid_info = submit_bid(loser_list, p, ϵ, v, bid_info)
```

```
p
```

```
array([3, 3, 3])
```

```
present_dict(bid_info)
```

```
+--------------+-----+-----+-----+
| House Number |  0  |  1  |  2  |
+--------------+-----+-----+-----+
|    Buyer     | [2] | [3] | [0] |
+--------------+-----+-----+-----+
```

Notice that Buyer 3 now switches to bid for house 1 having recongized that house 2 is no longer his best option.

```
allocation,winner_list,loser_list = check_terminal_condition(bid_info, p, v)
```

```
False
```

```
present_dict(allocation)
```

```
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Buyer     | 2 | 3 | 0 |
+--------------+---+---+---+
```

7.7.5 round 5

```
p,bid_info = submit_bid(loser_list, p, ϵ, v, bid_info)
```

```
p
```

```
array([3, 4, 3])
```

```
present_dict(bid_info)
```

```
+--------------+-----+-----+-----+
| House Number |  0  |  1  |  2  |
+--------------+-----+-----+-----+
|    Buyer     | [2] | [1] | [0] |
+--------------+-----+-----+-----+
```

Now Buyer 1 bids for house 1 again with price at 4, which crowds out Buyer 3, marking the end of the auction.

```
allocation,winner_list,loser_list = check_terminal_condition(bid_info, p, v)
```

```
True
```

```
present_dict(allocation)
```

```
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Buyer     | 2 | 1 | 0 |
+--------------+---+---+---+
```

```
# as for the houses unsold

house_unsold_list = list(set(house_list).difference(set(allocation.keys())))
house_unsold_list
```

```
[]
```

```
total_revenue = p[list(allocation.keys())].sum()
total_revenue
```

```
10
```

7.8 A Python Class

Above we simulated an ascending bid auction step by step.

When defining functions, we repeatedly computed some intermediate objects because our Python function loses track of variables once the function is executed.

That of course led to redundancy in our code

It is much more efficient to collect all of the aforementioned code into a class that records information about all rounds.

```
class ascending_bid_auction:

    def __init__(self, v, r, ϵ):
        """
        A class that simulates an ascending bid auction for houses.

        Given buyers' value matrix, sellers' reservation prices and minimum increment␣
↪of bid prices,
        this class can execute an ascending bid auction and present information round␣
↪by round until the end.

        Parameters:
        ----------
```

(continues on next page)

(continued from previous page)

```
        v: 2 dimensional value matrix

        r: np.array of reservation prices

        ϵ: minimum increment of bid price

        """

        self.v = v.copy()
        self.n,self.m = self.v.shape
        self.r = r
        self.ϵ = ϵ
        self.p = r.copy()
        self.buyer_list = np.arange(self.m)
        self.house_list = np.arange(self.n)
        self.bid_info_history = []
        self.allocation_history = []
        self.winner_history = []
        self.loser_history = []


    def find_argmax_with_randomness(self, v):
        n,m = v.shape
        index_array = np.arange(n)
        result=[]

        for ii in range(m):
            max_value = v[:,ii].max()
            result.append(np.random.choice(index_array[v[:,ii] == max_value]))

        return np.array(result)


    def check_kick_off_condition(self):
        # we convert the price vector to a matrix in the same shape as value matrix␣
↪to facilitate subtraction
        p_start = (self.ϵ + self.r)[:,None] @ np.ones(self.m)[None,:]
        self.surplus_value = self.v - p_start
        buyer_decision = (self.surplus_value > 0).any(axis = 0)
        return buyer_decision.any()


    def submit_initial_bid(self):
        # we intend to find the optimal choice of each buyer
        p_start_mat = (self.ϵ + self.p)[:,None] @ np.ones(self.m)[None,:]
        self.surplus_value = self.v - p_start_mat

        # we only care about active buyers who have positve surplus values
        active_buyer_diagnosis = (self.surplus_value > 0).any(axis = 0)
        active_buyer_list = self.buyer_list[active_buyer_diagnosis]
        active_buyer_surplus_value = self.surplus_value[:,active_buyer_diagnosis]
        active_buyer_choice = self.find_argmax_with_randomness(active_buyer_surplus_
↪value)

        # we only retain the unique house index because prices increase once at one␣
↪round
```

(continues on next page)

(continued from previous page)

```
        house_bid =  list(set(active_buyer_choice))
        self.p[house_bid] += self.ϵ

        bid_info = {}
        for house_num in house_bid:
            bid_info[house_num] = active_buyer_list[active_buyer_choice == house_num]
        self.bid_info_history.append(bid_info)

        print('The bid information is')
        ymtb = pt.PrettyTable()
        ymtb.field_names = ['House Number', *bid_info.keys()]
        ymtb.add_row(['Buyer', *bid_info.values()])
        print(ymtb)

        print('The bid prices for houses are')
        ymtb = pt.PrettyTable()
        ymtb.field_names = ['House Number', *self.house_list]
        ymtb.add_row(['Price', *self.p])
        print(ymtb)

        self.winner_list=[np.random.choice(bid_info[ii]) for ii in bid_info.keys()]
        self.winner_history.append(self.winner_list)

        self.allocation = {house_num:[winner] for house_num,winner in zip(bid_info.
↪keys(),self.winner_list)}
        self.allocation_history.append(self.allocation)

        loser_set = set(self.buyer_list).difference(set(self.winner_list))
        self.loser_list = list(loser_set)
        self.loser_history.append(self.loser_list)

        print('The winners are')
        print(self.winner_list)

        print('The losers are')
        print(self.loser_list)
        print('\n')


    def check_terminal_condition(self):
        loser_num = len(self.loser_list)

        if loser_num == 0:
            print('The auction ends because every buyer gets one house.')
            print('\n')
            return True

        p_mat = (self.ϵ + self.p)[:,None] @ np.ones(loser_num)[None,:]
        self.loser_surplus_value = self.v[:,self.loser_list] - p_mat
        self.loser_decision = (self.loser_surplus_value > 0).any(axis = 0)

        return ~(self.loser_decision.any())


    def submit_bid(self):
        bid_info = self.allocation_history[-1].copy()  # we only record the bid info␣
↪of winner
```

(continues on next page)

(continued from previous page)

```
        loser_num = len(self.loser_list)
        p_mat = (self.ϵ + self.p)[:,None] @ np.ones(loser_num)[None,:]
        self.loser_surplus_value = self.v[:,self.loser_list] - p_mat
        self.loser_decision = (self.loser_surplus_value > 0).any(axis = 0)

        active_loser_list = np.array(self.loser_list)[self.loser_decision]
        active_loser_surplus_value = self.loser_surplus_value[:,self.loser_decision]
        active_loser_choice = self.find_argmax_with_randomness(active_loser_surplus_
↪value)

        # we retain the unique house index and increasing the corresponding bid price
        house_bid = list(set(active_loser_choice))
        self.p[house_bid] += self.ϵ

        # we record the bidding information from active losers
        bid_info_active_loser = {}
        for house_num in house_bid:
            bid_info_active_loser[house_num] = active_loser_list[active_loser_choice␣
↪== house_num]

        # we update the bidding information according to the bidding from actice␣
↪losers
        for house_num in bid_info_active_loser.keys():
            bid_info[house_num] = bid_info_active_loser[house_num]
        self.bid_info_history.append(bid_info)

        print('The bid information is')
        ymtb = pt.PrettyTable()
        ymtb.field_names = ['House Number', *bid_info.keys()]
        ymtb.add_row(['Buyer', *bid_info.values()])
        print(ymtb)

        print('The bid prices for houses are')
        ymtb = pt.PrettyTable()
        ymtb.field_names = ['House Number', *self.house_list]
        ymtb.add_row(['Price', *self.p])
        print(ymtb)

        self.winner_list=[np.random.choice(bid_info[ii]) for ii in bid_info.keys()]
        self.winner_history.append(self.winner_list)

        self.allocation = {house_num:[winner] for house_num,winner in zip(bid_info.
↪keys(),self.winner_list)}
        self.allocation_history.append(self.allocation)

        loser_set = set(self.buyer_list).difference(set(self.winner_list))
        self.loser_list = list(loser_set)
        self.loser_history.append(self.loser_list)

        print('The winners are')
        print(self.winner_list)

        print('The losers are')
        print(self.loser_list)
        print('\n')
```

(continues on next page)

(continued from previous page)

```
    def start_auction(self):
        print('The Ascending Bid Auction for Houses')
        print('\n')

        print('Basic Information: %d houses, %d buyers'%(self.n, self.m))

        print('The valuation matrix is as follows')
        ymtb = pt.PrettyTable()
        ymtb.field_names = ['Buyer Number', *(np.arange(self.m))]
        for ii in range(self.n):
            ymtb.add_row(['House %d'%(ii), *self.v[ii,:]])
        print(ymtb)

        print('The reservation prices for houses are')
        ymtb = pt.PrettyTable()
        ymtb.field_names = ['House Number', *self.house_list]
        ymtb.add_row(['Price', *self.r])
        print(ymtb)
        print('The minimum increment of bid price is %.2f' % self.ϵ)
        print('\n')

        ctr = 1
        if self.check_kick_off_condition():
            print('Auction starts successfully')
            print('\n')
            print('Round %d'% ctr)

            self.submit_initial_bid()

            while True:
                if self.check_terminal_condition():
                    print('Auction ends')
                    print('\n')

                    print('The final result is as follows')
                    print('\n')
                    print('The allocation plan is')
                    ymtb = pt.PrettyTable()
                    ymtb.field_names = ['House Number', *self.allocation.keys()]
                    ymtb.add_row(['Buyer', *self.allocation.values()])
                    print(ymtb)

                    print('The bid prices for houses are')
                    ymtb = pt.PrettyTable()
                    ymtb.field_names = ['House Number', *self.house_list]
                    ymtb.add_row(['Price', *self.p])
                    print(ymtb)

                    print('The winners are')
                    print(self.winner_list)

                    print('The losers are')
                    print(self.loser_list)
```

(continues on next page)

(continued from previous page)

```
                    self.house_unsold_list = list(set(self.house_list).
↪difference(set(self.allocation.keys())))
                    print('The houses unsold are')
                    print(self.house_unsold_list)

                    self.total_revenue = self.p[list(self.allocation.keys())].sum()
                    print('The total revenue is %.2f' % self.total_revenue)

                    break

                ctr += 1
                print('Round %d'% ctr)
                self.submit_bid()

            # we compute the surplus matrix S and the quantity matrix X as required␣
↪in 1.1
            self.S = np.zeros((self.n, self.m))
            for ii,jj in zip(self.allocation.keys(),self.allocation.values()):
                self.S[ii,jj] = self.v[ii,jj] - self.p[ii]

            self.Q = np.zeros((self.n, self.m + 1))  # the last column records the␣
↪houses unsold
            for ii,jj in zip(self.allocation.keys(),self.allocation.values()):
                self.Q[ii,jj] = 1
            for ii in self.house_unsold_list:
                self.Q[ii,-1] = 1

            # we sort the allocation result by the house number
            house_sold_list = list(self.allocation.keys())
            house_sold_list.sort()

            dict_temp = {}
            for ii in house_sold_list:
                dict_temp[ii] = self.allocation[ii]
            self.allocation = dict_temp

        else:
            print('The auction can not start because of high reservation prices')
```

Let's use our class to conduct the auction described in one of the above examples.

```
v = np.array([[8,5,9,4],[4,11,7,4],[9,7,6,4]])
r = np.array([2,1,0])
ϵ = 1

auction_1 = ascending_bid_auction(v, r, ϵ)

auction_1.start_auction()
```

```
The Ascending Bid Auction for Houses


Basic Information: 3 houses, 4 buyers
The valuation matrix is as follows
+--------------+---+----+---+---+
```

(continues on next page)

(continued from previous page)

```
| Buyer Number | 0 | 1  | 2 | 3 |
+--------------+---+----+---+---+
|   House 0    | 8 | 5  | 9 | 4 |
|   House 1    | 4 | 11 | 7 | 4 |
|   House 2    | 9 | 7  | 6 | 4 |
+--------------+---+----+---+---+
The reservation prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 2 | 1 | 0 |
+--------------+---+---+---+
The minimum increment of bid price is 1.00


Auction starts successfully


Round 1
The bid information is
+--------------+-----+-----+-------+
| House Number |  0  |  1  |   2   |
+--------------+-----+-----+-------+
|    Buyer     | [2] | [1] | [0 3] |
+--------------+-----+-----+-------+
The bid prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 3 | 2 | 1 |
+--------------+---+---+---+
The winners are
[2, 1, 0]
The losers are
[3]


Round 2
The bid information is
+--------------+-----+-----+-----+
| House Number |  0  |  1  |  2  |
+--------------+-----+-----+-----+
|    Buyer     | [2] | [1] | [3] |
+--------------+-----+-----+-----+
The bid prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 3 | 2 | 2 |
+--------------+---+---+---+
The winners are
[2, 1, 3]
The losers are
[0]
```

(continues on next page)

(continued from previous page)

```
Round 3
The bid information is
+--------------+-----+-----+-----+
| House Number |  0  |  1  |  2  |
+--------------+-----+-----+-----+
|    Buyer     | [2] | [1] | [0] |
+--------------+-----+-----+-----+
The bid prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 3 | 2 | 3 |
+--------------+---+---+---+
The winners are
[2, 1, 0]
The losers are
[3]


Round 4
The bid information is
+--------------+-----+-----+-----+
| House Number |  0  |  1  |  2  |
+--------------+-----+-----+-----+
|    Buyer     | [2] | [3] | [0] |
+--------------+-----+-----+-----+
The bid prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 3 | 3 | 3 |
+--------------+---+---+---+
The winners are
[2, 3, 0]
The losers are
[1]


Round 5
The bid information is
+--------------+-----+-----+-----+
| House Number |  0  |  1  |  2  |
+--------------+-----+-----+-----+
|    Buyer     | [2] | [1] | [0] |
+--------------+-----+-----+-----+
The bid prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 3 | 4 | 3 |
+--------------+---+---+---+
The winners are
[2, 1, 0]
The losers are
[3]
```

(continues on next page)

(continued from previous page)

```
Auction ends


The final result is as follows


The allocation plan is
+--------------+-----+-----+-----+
| House Number |  0  |  1  |  2  |
+--------------+-----+-----+-----+
|    Buyer     | [2] | [1] | [0] |
+--------------+-----+-----+-----+
The bid prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 3 | 4 | 3 |
+--------------+---+---+---+
The winners are
[2, 1, 0]
The losers are
[3]
The houses unsold are
[]
The total revenue is 10.00
```

```
# the surplus matrix S

auction_1.S
```

```
array([[0., 0., 6., 0.],
       [0., 7., 0., 0.],
       [6., 0., 0., 0.]])
```

```
# the quantity matrix X

auction_1.Q
```

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

7.9 Robustness Checks

Let's do some stress testing of our code by applying it to auctions characterized by different matrices of private values.

1. number of houses = number of buyers

```
v2 = np.array([[8,5,9],[4,11,7],[9,7,6]])

auction_2 = ascending_bid_auction(v2, r, ϵ)

auction_2.start_auction()
```

```
The Ascending Bid Auction for Houses


Basic Information: 3 houses, 3 buyers
The valuation matrix is as follows
+--------------+---+----+---+
| Buyer Number | 0 | 1  | 2 |
+--------------+---+----+---+
|   House 0    | 8 | 5  | 9 |
|   House 1    | 4 | 11 | 7 |
|   House 2    | 9 | 7  | 6 |
+--------------+---+----+---+
The reservation prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 2 | 1 | 0 |
+--------------+---+---+---+
The minimum increment of bid price is 1.00


Auction starts successfully


Round 1
The bid information is
+--------------+-----+-----+-----+
| House Number |  0  |  1  |  2  |
+--------------+-----+-----+-----+
|    Buyer     | [2] | [1] | [0] |
+--------------+-----+-----+-----+
The bid prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 3 | 2 | 1 |
+--------------+---+---+---+
The winners are
[2, 1, 0]
The losers are
[]


The auction ends because every buyer gets one house.
```

(continues on next page)

(continued from previous page)

```
Auction ends


The final result is as follows


The allocation plan is
+--------------+-----+-----+-----+
| House Number |  0  |  1  |  2  |
+--------------+-----+-----+-----+
|    Buyer     | [2] | [1] | [0] |
+--------------+-----+-----+-----+
The bid prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 3 | 2 | 1 |
+--------------+---+---+---+
The winners are
[2, 1, 0]
The losers are
[]
The houses unsold are
[]
The total revenue is 6.00
```

2. multilple excess buyers

```
v3 = np.array([[8,5,9,4,3],[4,11,7,4,6],[9,7,6,4,2]])

auction_3 = ascending_bid_auction(v3, r, ϵ)

auction_3.start_auction()
```

```
The Ascending Bid Auction for Houses


Basic Information: 3 houses, 5 buyers
The valuation matrix is as follows
+--------------+---+----+---+---+---+
| Buyer Number | 0 | 1  | 2 | 3 | 4 |
+--------------+---+----+---+---+---+
|   House 0    | 8 | 5  | 9 | 4 | 3 |
|   House 1    | 4 | 11 | 7 | 4 | 6 |
|   House 2    | 9 | 7  | 6 | 4 | 2 |
+--------------+---+----+---+---+---+
The reservation prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 2 | 1 | 0 |
+--------------+---+---+---+
The minimum increment of bid price is 1.00
```

(continues on next page)

(continued from previous page)

```
Auction starts successfully


Round 1
The bid information is
+--------------+-----+-------+-------+
| House Number |  0  |   1   |   2   |
+--------------+-----+-------+-------+
|    Buyer     | [2] | [1 4] | [0 3] |
+--------------+-----+-------+-------+
The bid prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 3 | 2 | 1 |
+--------------+---+---+---+
The winners are
[2, 4, 3]
The losers are
[0, 1]


Round 2
The bid information is
+--------------+-----+-----+-----+
| House Number |  0  |  1  |  2  |
+--------------+-----+-----+-----+
|    Buyer     | [2] | [1] | [0] |
+--------------+-----+-----+-----+
The bid prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 3 | 3 | 2 |
+--------------+---+---+---+
The winners are
[2, 1, 0]
The losers are
[3, 4]


Round 3
The bid information is
+--------------+-----+-----+-----+
| House Number |  0  |  1  |  2  |
+--------------+-----+-----+-----+
|    Buyer     | [2] | [4] | [3] |
+--------------+-----+-----+-----+
The bid prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 3 | 4 | 3 |
+--------------+---+---+---+
```

(continues on next page)

(continued from previous page)

```
The winners are
[2, 4, 3]
The losers are
[0, 1]


Round 4
The bid information is
+--------------+-----+-----+-----+
| House Number |  0  |  1  |  2  |
+--------------+-----+-----+-----+
|    Buyer     | [2] | [1] | [0] |
+--------------+-----+-----+-----+
The bid prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 3 | 5 | 4 |
+--------------+---+---+---+
The winners are
[2, 1, 0]
The losers are
[3, 4]


Auction ends


The final result is as follows


The allocation plan is
+--------------+-----+-----+-----+
| House Number |  0  |  1  |  2  |
+--------------+-----+-----+-----+
|    Buyer     | [2] | [1] | [0] |
+--------------+-----+-----+-----+
The bid prices for houses are
+--------------+---+---+---+
| House Number | 0 | 1 | 2 |
+--------------+---+---+---+
|    Price     | 3 | 5 | 4 |
+--------------+---+---+---+
The winners are
[2, 1, 0]
The losers are
[3, 4]
The houses unsold are
[]
The total revenue is 12.00
```

3. more houses than buyers

```
v4 = np.array([[8,5,4],[4,11,7],[9,7,9],[6,4,5],[2,2,2]])
r2 = np.array([2,1,0,1,1])
```

(continues on next page)

(continued from previous page)

```
auction_4 = ascending_bid_auction(v4, r2, ϵ)

auction_4.start_auction()
```

```
The Ascending Bid Auction for Houses


Basic Information: 5 houses, 3 buyers
The valuation matrix is as follows
+--------------+---+----+---+
| Buyer Number | 0 | 1  | 2 |
+--------------+---+----+---+
|   House 0    | 8 | 5  | 4 |
|   House 1    | 4 | 11 | 7 |
|   House 2    | 9 | 7  | 9 |
|   House 3    | 6 | 4  | 5 |
|   House 4    | 2 | 2  | 2 |
+--------------+---+----+---+
The reservation prices for houses are
+--------------+---+---+---+---+---+
| House Number | 0 | 1 | 2 | 3 | 4 |
+--------------+---+---+---+---+---+
|    Price     | 2 | 1 | 0 | 1 | 1 |
+--------------+---+---+---+---+---+
The minimum increment of bid price is 1.00


Auction starts successfully


Round 1
The bid information is
+--------------+-----+-------+
| House Number |  1  |   2   |
+--------------+-----+-------+
|    Buyer     | [1] | [0 2] |
+--------------+-----+-------+
The bid prices for houses are
+--------------+---+---+---+---+---+
| House Number | 0 | 1 | 2 | 3 | 4 |
+--------------+---+---+---+---+---+
|    Price     | 2 | 2 | 1 | 1 | 1 |
+--------------+---+---+---+---+---+
The winners are
[1, 2]
The losers are
[0]


Round 2
The bid information is
+--------------+-----+-----+
| House Number |  1  |  2  |
+--------------+-----+-----+
|    Buyer     | [1] | [0] |
```

(continues on next page)

(continued from previous page)

```
+--------------+-----+-----+
The bid prices for houses are
+--------------+---+---+---+---+---+
| House Number | 0 | 1 | 2 | 3 | 4 |
+--------------+---+---+---+---+---+
|    Price     | 2 | 2 | 2 | 1 | 1 |
+--------------+---+---+---+---+---+
The winners are
[1, 0]
The losers are
[2]


Round 3
The bid information is
+--------------+-----+-----+
| House Number |  1  |  2  |
+--------------+-----+-----+
|    Buyer     | [1] | [2] |
+--------------+-----+-----+
The bid prices for houses are
+--------------+---+---+---+---+---+
| House Number | 0 | 1 | 2 | 3 | 4 |
+--------------+---+---+---+---+---+
|    Price     | 2 | 2 | 3 | 1 | 1 |
+--------------+---+---+---+---+---+
The winners are
[1, 2]
The losers are
[0]


Round 4
The bid information is
+--------------+-----+-----+
| House Number |  1  |  2  |
+--------------+-----+-----+
|    Buyer     | [1] | [0] |
+--------------+-----+-----+
The bid prices for houses are
+--------------+---+---+---+---+---+
| House Number | 0 | 1 | 2 | 3 | 4 |
+--------------+---+---+---+---+---+
|    Price     | 2 | 2 | 4 | 1 | 1 |
+--------------+---+---+---+---+---+
The winners are
[1, 0]
The losers are
[2]


Round 5
The bid information is
+--------------+-----+-----+
| House Number |  1  |  2  |
+--------------+-----+-----+
```

(continues on next page)

(continued from previous page)

```
|    Buyer     | [2] | [0] |
+--------------+-----+-----+
The bid prices for houses are
+--------------+---+---+---+---+---+
| House Number | 0 | 1 | 2 | 3 | 4 |
+--------------+---+---+---+---+---+
|    Price     | 2 | 3 | 4 | 1 | 1 |
+--------------+---+---+---+---+---+
The winners are
[2, 0]
The losers are
[1]


Round 6
The bid information is
+--------------+-----+-----+
| House Number |  1  |  2  |
+--------------+-----+-----+
|    Buyer     | [1] | [0] |
+--------------+-----+-----+
The bid prices for houses are
+--------------+---+---+---+---+---+
| House Number | 0 | 1 | 2 | 3 | 4 |
+--------------+---+---+---+---+---+
|    Price     | 2 | 4 | 4 | 1 | 1 |
+--------------+---+---+---+---+---+
The winners are
[1, 0]
The losers are
[2]


Round 7
The bid information is
+--------------+-----+-----+
| House Number |  1  |  2  |
+--------------+-----+-----+
|    Buyer     | [1] | [2] |
+--------------+-----+-----+
The bid prices for houses are
+--------------+---+---+---+---+---+
| House Number | 0 | 1 | 2 | 3 | 4 |
+--------------+---+---+---+---+---+
|    Price     | 2 | 4 | 5 | 1 | 1 |
+--------------+---+---+---+---+---+
The winners are
[1, 2]
The losers are
[0]


Round 8
The bid information is
+--------------+-----+-----+-----+
| House Number |  1  |  2  |  0  |
```

(continues on next page)

(continued from previous page)

```
+--------------+-----+-----+-----+
|    Buyer     | [1] | [2] | [0] |
+--------------+-----+-----+-----+
The bid prices for houses are
+--------------+---+---+---+---+---+
| House Number | 0 | 1 | 2 | 3 | 4 |
+--------------+---+---+---+---+---+
|    Price     | 3 | 4 | 5 | 1 | 1 |
+--------------+---+---+---+---+---+
The winners are
[1, 2, 0]
The losers are
[]


The auction ends because every buyer gets one house.


Auction ends


The final result is as follows


The allocation plan is
+--------------+-----+-----+-----+
| House Number |  1  |  2  |  0  |
+--------------+-----+-----+-----+
|    Buyer     | [1] | [2] | [0] |
+--------------+-----+-----+-----+
The bid prices for houses are
+--------------+---+---+---+---+---+
| House Number | 0 | 1 | 2 | 3 | 4 |
+--------------+---+---+---+---+---+
|    Price     | 3 | 4 | 5 | 1 | 1 |
+--------------+---+---+---+---+---+
The winners are
[1, 2, 0]
The losers are
[]
The houses unsold are
[3, 4]
The total revenue is 12.00
```

4. some houses have extremely high reservation prices

```
v5 = np.array([[8,5,4],[4,11,7],[9,7,9],[6,4,5],[2,2,2]])
r3 = np.array([10,1,0,1,1])

auction_5 = ascending_bid_auction(v5, r3, ϵ)

auction_5.start_auction()
```

```
The Ascending Bid Auction for Houses
```

(continues on next page)

(continued from previous page)

```
Basic Information: 5 houses, 3 buyers
The valuation matrix is as follows
+--------------+---+----+---+
| Buyer Number | 0 | 1  | 2 |
+--------------+---+----+---+
|   House 0    | 8 | 5  | 4 |
|   House 1    | 4 | 11 | 7 |
|   House 2    | 9 | 7  | 9 |
|   House 3    | 6 | 4  | 5 |
|   House 4    | 2 | 2  | 2 |
+--------------+---+----+---+
The reservation prices for houses are
+--------------+----+---+---+---+---+
| House Number | 0  | 1 | 2 | 3 | 4 |
+--------------+----+---+---+---+---+
|    Price     | 10 | 1 | 0 | 1 | 1 |
+--------------+----+---+---+---+---+
The minimum increment of bid price is 1.00


Auction starts successfully


Round 1
The bid information is
+--------------+-----+-------+
| House Number |  1  |   2   |
+--------------+-----+-------+
|    Buyer     | [1] | [0 2] |
+--------------+-----+-------+
The bid prices for houses are
+--------------+----+---+---+---+---+
| House Number | 0  | 1 | 2 | 3 | 4 |
+--------------+----+---+---+---+---+
|    Price     | 10 | 2 | 1 | 1 | 1 |
+--------------+----+---+---+---+---+
The winners are
[1, 0]
The losers are
[2]


Round 2
The bid information is
+--------------+-----+-----+
| House Number |  1  |  2  |
+--------------+-----+-----+
|    Buyer     | [1] | [2] |
+--------------+-----+-----+
The bid prices for houses are
+--------------+----+---+---+---+---+
| House Number | 0  | 1 | 2 | 3 | 4 |
+--------------+----+---+---+---+---+
|    Price     | 10 | 2 | 2 | 1 | 1 |
+--------------+----+---+---+---+---+
```

(continues on next page)

(continued from previous page)

```
The winners are
[1, 2]
The losers are
[0]


Round 3
The bid information is
+--------------+-----+-----+
| House Number |  1  |  2  |
+--------------+-----+-----+
|    Buyer     | [1] | [0] |
+--------------+-----+-----+
The bid prices for houses are
+--------------+----+---+---+---+---+
| House Number | 0  | 1 | 2 | 3 | 4 |
+--------------+----+---+---+---+---+
|    Price     | 10 | 2 | 3 | 1 | 1 |
+--------------+----+---+---+---+---+
The winners are
[1, 0]
The losers are
[2]


Round 4
The bid information is
+--------------+-----+-----+
| House Number |  1  |  2  |
+--------------+-----+-----+
|    Buyer     | [1] | [2] |
+--------------+-----+-----+
The bid prices for houses are
+--------------+----+---+---+---+---+
| House Number | 0  | 1 | 2 | 3 | 4 |
+--------------+----+---+---+---+---+
|    Price     | 10 | 2 | 4 | 1 | 1 |
+--------------+----+---+---+---+---+
The winners are
[1, 2]
The losers are
[0]


Round 5
The bid information is
+--------------+-----+-----+
| House Number |  1  |  2  |
+--------------+-----+-----+
|    Buyer     | [1] | [0] |
+--------------+-----+-----+
The bid prices for houses are
+--------------+----+---+---+---+---+
| House Number | 0  | 1 | 2 | 3 | 4 |
+--------------+----+---+---+---+---+
|    Price     | 10 | 2 | 5 | 1 | 1 |
```

(continues on next page)

(continued from previous page)

```
+--------------+----+---+---+---+---+
The winners are
[1, 0]
The losers are
[2]


Round 6
The bid information is
+--------------+-----+-----+
| House Number |  1  |  2  |
+--------------+-----+-----+
|    Buyer     | [2] | [0] |
+--------------+-----+-----+
The bid prices for houses are
+--------------+----+---+---+---+---+
| House Number | 0  | 1 | 2 | 3 | 4 |
+--------------+----+---+---+---+---+
|    Price     | 10 | 3 | 5 | 1 | 1 |
+--------------+----+---+---+---+---+
The winners are
[2, 0]
The losers are
[1]


Round 7
The bid information is
+--------------+-----+-----+
| House Number |  1  |  2  |
+--------------+-----+-----+
|    Buyer     | [1] | [0] |
+--------------+-----+-----+
The bid prices for houses are
+--------------+----+---+---+---+---+
| House Number | 0  | 1 | 2 | 3 | 4 |
+--------------+----+---+---+---+---+
|    Price     | 10 | 4 | 5 | 1 | 1 |
+--------------+----+---+---+---+---+
The winners are
[1, 0]
The losers are
[2]


Round 8
The bid information is
+--------------+-----+-----+
| House Number |  1  |  2  |
+--------------+-----+-----+
|    Buyer     | [1] | [2] |
+--------------+-----+-----+
The bid prices for houses are
+--------------+----+---+---+---+---+
| House Number | 0  | 1 | 2 | 3 | 4 |
+--------------+----+---+---+---+---+
```

(continues on next page)

(continued from previous page)

```
|    Price     | 10 | 4 | 6 | 1 | 1 |
+--------------+----+---+---+---+---+
The winners are
[1, 2]
The losers are
[0]


Round 9
The bid information is
+--------------+-----+-----+-----+
| House Number |  1  |  2  |  3  |
+--------------+-----+-----+-----+
|    Buyer     | [1] | [2] | [0] |
+--------------+-----+-----+-----+
The bid prices for houses are
+--------------+----+---+---+---+---+
| House Number | 0  | 1 | 2 | 3 | 4 |
+--------------+----+---+---+---+---+
|    Price     | 10 | 4 | 6 | 2 | 1 |
+--------------+----+---+---+---+---+
The winners are
[1, 2, 0]
The losers are
[]


The auction ends because every buyer gets one house.


Auction ends


The final result is as follows


The allocation plan is
+--------------+-----+-----+-----+
| House Number |  1  |  2  |  3  |
+--------------+-----+-----+-----+
|    Buyer     | [1] | [2] | [0] |
+--------------+-----+-----+-----+
The bid prices for houses are
+--------------+----+---+---+---+---+
| House Number | 0  | 1 | 2 | 3 | 4 |
+--------------+----+---+---+---+---+
|    Price     | 10 | 4 | 6 | 2 | 1 |
+--------------+----+---+---+---+---+
The winners are
[1, 2, 0]
The losers are
[]
The houses unsold are
[0, 4]
The total revenue is 12.00
```

5. reservation prices are so high that the auction can't start

```
r4 = np.array([15,15,15])

auction_6 = ascending_bid_auction(v, r4, ϵ)

auction_6.start_auction()
```

```
The Ascending Bid Auction for Houses


Basic Information: 3 houses, 4 buyers
The valuation matrix is as follows
+--------------+---+----+---+---+
| Buyer Number | 0 | 1  | 2 | 3 |
+--------------+---+----+---+---+
|   House 0    | 8 | 5  | 9 | 4 |
|   House 1    | 4 | 11 | 7 | 4 |
|   House 2    | 9 | 7  | 6 | 4 |
+--------------+---+----+---+---+
The reservation prices for houses are
+--------------+----+----+----+
| House Number | 0  | 1  | 2  |
+--------------+----+----+----+
|    Price     | 15 | 15 | 15 |
+--------------+----+----+----+
The minimum increment of bid price is 1.00


The auction can not start because of high reservation prices
```

7.10 A Groves-Clarke Mechanism

We now decribe an alternative way for society to allocate n houses to m possible buyers in a way that maximizes total value across all potential buyers.

We continue to assume that each buyer can purchase at most one house.

The mechanism is a very special case of a Groves-Clarke mechanism [Groves, 1973], [Clarke, 1971].

Its special structure substantially simplifies writing Python code to find an optimal allocation.

Our mechanims works like this.

- The values V_{ij} are private information to person j
- The mechanism makes each person j willing to tell a social planner his private values $V_{i,j}$ for all $i = 1, \ldots, n$.
- The social planner asks all potential bidders to tell the planner their private values V_{ij}
- The social planner tells no one these, but uses them to allocate houses and set prices
- The mechanism is designed in a way that makes all prospective buyers want to tell the planner their private values
  - truth telling is a dominant strategy for each potential buyer
- The planner finds a house, bidder pair with highest private value by computing $(\tilde{i}, \tilde{j}) = \text{argmax}(V_{ij})$
- The planner assigns house $\tilde{i}$ to buyer $\tilde{j}$
- The planner charges buyer $\tilde{j}$ the price $\max_{-\tilde{j}} V_{\tilde{i},j}$, where $-\tilde{j}$ means all j's except $\tilde{j}$.

- The planner creates a matrix of private values for the remaining houses $-\tilde{i}$ by deleting row (i.e., house) $\tilde{i}$ and column (i.e., buyer) $\tilde{j}$ from V.
  - (But in doing this, the planner keeps track of the real names of the bidders and the houses).
- The planner returns to the original step and repeat it.
- The planner iterates until all n houses are allocated and the charges for all n houses are set.

7.11 An Example Solved by Hand

Let's see how our Groves-Clarke algorithm would work for the following simple matrix V matrix of private values

$$V = \begin{bmatrix} 10 & 9 & 8 & 7 & 6 \\ 9 & 9 & 7 & 6 & 6 \\ 8 & 6 & 6 & 9 & 4 \\ 7 & 5 & 6 & 4 & 9 \end{bmatrix}$$

Remark: In the first step, when the highest private value corresponds to more than one house, bidder pairs, we choose the pair with the highest sale price. If a highest sale price corresponds to two or more pairs with highest private values, we randomly choose one.

```
np.random.seed(666)

V_orig = np.array([[10, 9, 8, 7, 6],  # record the origianl values
                   [9, 9, 7, 6, 6],
                   [8, 6, 6, 9, 4],
                   [7, 5, 6, 4, 9]])
V = np.copy(V_orig)  # used iteratively
n, m = V.shape
p = np.zeros(n) # prices of houses
Q = np.zeros((n, m)) # keep record the status of houses and buyers
```

First assignment

First, we find house, bidder pair with highest private value.

```
i, j = np.where(V==np.max(V))
i, j
```

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

So, house 0 will be sold to buyer 0 at a price of 9. We then update the sale price of house 0 and the status matrix Q.

```
p[i] = np.max(np.delete(V[i, :], j))
Q[i, j] = 1
p, Q
```

```
(array([9., 0., 0., 0.]),
 array([[1., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.]]))
```

Then we remove row 0 and column 0 from V. To keep the real number of houses and buyers, we set this row and this column to -1, which will have the same result as removing them since $V \geq 0$.

```
V[i, :] = -1
V[:, j] = -1
V
```

```
array([[-1, -1, -1, -1, -1],
       [-1,  9,  7,  6,  6],
       [-1,  6,  6,  9,  4],
       [-1,  5,  6,  4,  9]])
```

Second assignment

We find house, bidder pair with the highest private value again.

```
i, j = np.where(V==np.max(V))
i, j
```

```
(array([1, 2, 3]), array([1, 3, 4]))
```

In this special example, there are three pairs (1, 1), (2, 3) and (3, 4) with the highest private value. To solve this problem, we choose the one with highest sale price.

```
p_candidate = np.zeros(len(i))
for k in range(len(i)):
    p_candidate[k] = np.max(np.delete(V[i[k], :], j[k]))
k, = np.where(p_candidate==np.max(p_candidate))
i, j = i[k], j[k]
i, j
```

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

So, house 1 will be sold to buyer 1 at a price of 7. We update matrices.

```
p[i] = np.max(np.delete(V[i, :], j))
Q[i, j] = 1
V[i, :] = -1
V[:, j] = -1
p, Q, V
```

```
(array([9., 7., 0., 0.]),
 array([[1., 0., 0., 0., 0.],
        [0., 1., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.]]),
 array([[-1, -1, -1, -1, -1],
        [-1, -1, -1, -1, -1],
        [-1, -1,  6,  9,  4],
        [-1, -1,  6,  4,  9]]))
```

Third assignment

```
i, j = np.where(V==np.max(V))
i, j
```

```
(array([2, 3]), array([3, 4]))
```

In this special example, there are two pairs (2, 3) and (3, 4) with the highest private value.

To resolve the assignment, we choose the one with highest sale price.

```
p_candidate = np.zeros(len(i))
for k in range(len(i)):
    p_candidate[k] = np.max(np.delete(V[i[k], :], j[k]))
k, = np.where(p_candidate==np.max(p_candidate))
i, j = i[k], j[k]
i, j
```

```
(array([2, 3]), array([3, 4]))
```

The two pairs even have the same sale price.

We randomly choose one pair.

```
k = np.random.choice(len(i))
i, j = i[k], j[k]
i, j
```

```
(2, 3)
```

Finally, house 2 will be sold to buyer 3.

We update matrices accordingly.

```
p[i] = np.max(np.delete(V[i, :], j))
Q[i, j] = 1
V[i, :] = -1
V[:, j] = -1
p, Q, V
```

```
(array([9., 7., 6., 0.]),
 array([[1., 0., 0., 0., 0.],
        [0., 1., 0., 0., 0.],
        [0., 0., 0., 1., 0.],
        [0., 0., 0., 0., 0.]]),
 array([[-1, -1, -1, -1, -1],
        [-1, -1, -1, -1, -1],
        [-1, -1, -1, -1, -1],
        [-1, -1,  6, -1,  9]]))
```

Fourth assignment

```
i, j = np.where(V==np.max(V))
i, j
```

```
(array([3]), array([4]))
```

House 3 will be sold to buyer 4.

The final outcome follows.

```
p[i] = np.max(np.delete(V[i, :], j))
Q[i, j] = 1
V[i, :] = -1
V[:, j] = -1
S = V_orig*Q - np.diag(p)@Q
p, Q, V, S
```

```
(array([9., 7., 6., 6.]),
 array([[1., 0., 0., 0., 0.],
        [0., 1., 0., 0., 0.],
        [0., 0., 0., 1., 0.],
        [0., 0., 0., 0., 1.]]),
 array([[-1, -1, -1, -1, -1],
        [-1, -1, -1, -1, -1],
        [-1, -1, -1, -1, -1],
        [-1, -1, -1, -1, -1]]),
 array([[1., 0., 0., 0., 0.],
        [0., 2., 0., 0., 0.],
        [0., 0., 0., 3., 0.],
        [0., 0., 0., 0., 3.]]))
```

7.12 Another Python Class

It is efficient to assemble our calculations in a single Python Class.

```
class GC_Mechanism:

    def __init__(self, V):
        """
        Implementation of the special Groves Clarke Mechanism for house auction.

        Parameters:
        ----------
        V: 2 dimensional private value matrix

        """

        self.V_orig = V.copy()
        self.V = V.copy()
        self.n, self.m = self.V.shape
        self.p = np.zeros(self.n)
        self.Q = np.zeros((self.n, self.m))
        self.S = np.copy(self.Q)

    def find_argmax(self):
        """
        Find the house-buyer pair with the highest value.
```

(continues on next page)

(continued from previous page)

```
        When the highest private value corresponds to more than one house, bidder␣
↪pairs,
        we choose the pair with the highest sale price.
        Moreoever, if the highest sale price corresponds to two or more pairs with␣
↪highest private value,
        We randomly choose one.

        Parameters:
        ----------
        V: 2 dimensional private value matrix with -1 indicating revomed rows and␣
↪columns

        Returns:
        ----------
        i: the index of the sold house

        j: the index of the buyer

        """
        i, j = np.where(self.V==np.max(self.V))

        if (len(i)>1):
            p_candidate = np.zeros(len(i))
            for k in range(len(i)):
                p_candidate[k] = np.max(np.delete(self.V[i[k], :], j[k]))
            k, = np.where(p_candidate==np.max(p_candidate))
            i, j = i[k], j[k]

            if (len(i)>1):
                k = np.random.choice(len(i))
                k = np.array([k])
                i, j = i[k], j[k]
        return i, j

    def update_status(self, i, j):
        self.p[i] = np.max(np.delete(self.V[i, :], j))
        self.Q[i, j] = 1
        self.V[i, :] = -1
        self.V[:, j] = -1

    def calculate_surplus(self):
        self.S = self.V_orig*self.Q - np.diag(self.p)@self.Q

    def start(self):
        while (np.max(self.V)>=0):
            i, j = self.find_argmax()
            self.update_status(i, j)
            print("House %i is sold to buyer %i at price %i"%(i[0], j[0], self.
↪p[i[0]]))
            print("\n")
        self.calculate_surplus()
        print("Prices of house:\n", self.p)
        print("\n")
        print("The status matrix:\n", self.Q)
        print("\n")
        print("The surplus matrix:\n", self.S)
```

```
np.random.seed(666)

V_orig = np.array([[10, 9, 8, 7, 6],
                   [9, 9, 7, 6, 6],
                   [8, 6, 6, 9, 4],
                   [7, 5, 6, 4, 9]])
gc_mechanism = GC_Mechanism(V_orig)
gc_mechanism.start()
```

```
House 0 is sold to buyer 0 at price 9


House 1 is sold to buyer 1 at price 7


House 2 is sold to buyer 3 at price 6
```

```
House 3 is sold to buyer 4 at price 6


Prices of house:
 [9. 7. 6. 6.]


The status matrix:
 [[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]


The surplus matrix:
 [[1. 0. 0. 0. 0.]
 [0. 2. 0. 0. 0.]
 [0. 0. 0. 3. 0.]
 [0. 0. 0. 0. 3.]]
```

7.12.1 Elaborations

Here we use some additional notation designed to conform with standard notation in parts of the VCG literature.

We want to verify that our pseudo code is indeed a **pivot mechanism**, also called a **VCG** (Vickrey-Clarke-Groves) mechanism.

- The mechanism is named after [Groves, 1973], [Clarke, 1971], and [Vickrey, 1961].

To prepare for verifying this, we add some notation.

Let X be the set of feasible allocations of houses under the protocols above (i.e., at most one house to each person).

Let $X(v)$ be the allocation that the mechanism chooses for matrix v of private values.

The mechanism maps a matrix v of private values into an $x \in X$.

Let $v_j(x)$ be the value that person j attaches to allocation $x \in X$.

Let $\check{t}_j(v)$ the payment that the mechanism charges person j.

The VCG mechanism chooses the allocation

$$X(v) = \text{argmax}_{x \in X} \sum_{j=1}^{m} v_j(x) \tag{7.1}$$

and charges person j a "social cost"

$$\check{t}_j(v) = \max_{x \in X} \sum_{k \neq j} v_k(x) - \sum_{k \neq j} v_k(X(v)) \tag{7.2}$$

In our setting, equation (7.1) says that the VCG allocation allocates houses to maximize the total value of the successful prospective buyers.

In our setting, equation (7.2) says that the mechanism charges people for the externality that their presence in society imposes on other prospective buyers.

Thus, notice that according to equation (7.2):

- unsuccessful prospective buyers pay 0 because removing them from "society" would not affect the allocation chosen by the mechanim
- successful prospective buyers pay the difference between the total value society could achieve without them present and the total value that others present in society do achieve under the mechanism.

The generalized second-price auction described in our pseudo code above does indeed satisfy (1). We want to compute $\check{t}_j$ for $j = 1, \ldots, m$ and compare with p_j from the second price auction.

7.12.2 Social Cost

Using the GC_Mechanism class, we can calculate the social cost of each buyer.

Let's see a simpler example with private value matrix

$$V = \begin{bmatrix} 10 & 9 & 8 & 7 & 6 \\ 9 & 8 & 7 & 6 & 6 \\ 8 & 7 & 6 & 5 & 4 \end{bmatrix}$$

To begin with, we implement the GC mechanism and see the outcome.

```
np.random.seed(666)

V_orig = np.array([[10, 9, 8, 7, 6],
                   [9, 8, 7, 6, 6],
                   [8, 7, 6, 5, 4]])
gc_mechanism = GC_Mechanism(V_orig)
gc_mechanism.start()
```

```
House 0 is sold to buyer 0 at price 9


House 1 is sold to buyer 1 at price 7


House 2 is sold to buyer 2 at price 5


Prices of house:
```

(continues on next page)

(continued from previous page)

```
 [9. 7. 5.]


The status matrix:
 [[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]]


The surplus matrix:
 [[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]]
```

We exclude buyer 0 and calculate the allocation.

```
V_exc_0 = np.copy(V_orig)
V_exc_0[:, 0] = -1
V_exc_0
gc_mechanism_exc_0 = GC_Mechanism(V_exc_0)
gc_mechanism_exc_0.start()
```

```
House 0 is sold to buyer 1 at price 8


House 1 is sold to buyer 2 at price 6


House 2 is sold to buyer 3 at price 4


Prices of house:
 [8. 6. 4.]


The status matrix:
 [[0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]]


The surplus matrix:
 [[-0.  1.  0.  0.  0.]
 [-0.  0.  1.  0.  0.]
 [-0.  0.  0.  1.  0.]]
```

Calculate the social cost of buyer 0.

```
print("The social cost of buyer 0:",
     np.sum(gc_mechanism_exc_0.Q*gc_mechanism_exc_0.V_orig)-np.sum(np.delete(gc_
 ↪mechanism.Q*gc_mechanism.V_orig, 0, axis=1)))
```

```
The social cost of buyer 0: 7.0
```

Repeat this process for buyer 1 and buyer 2

```
V_exc_1 = np.copy(V_orig)
V_exc_1[:, 1] = -1
V_exc_1
gc_mechanism_exc_1 = GC_Mechanism(V_exc_1)
gc_mechanism_exc_1.start()

print("\nThe social cost of buyer 1:",
     np.sum(gc_mechanism_exc_1.Q*gc_mechanism_exc_1.V_orig)-np.sum(np.delete(gc_
 ↪mechanism.Q*gc_mechanism.V_orig, 1, axis=1)))
```

```
House 0 is sold to buyer 0 at price 8


House 1 is sold to buyer 2 at price 6


House 2 is sold to buyer 3 at price 4


Prices of house:
 [8. 6. 4.]


The status matrix:
 [[1. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]]


The surplus matrix:
 [[ 2. -0.  0.  0.  0.]
 [ 0. -0.  1.  0.  0.]
 [ 0. -0.  0.  1.  0.]]

The social cost of buyer 1: 6.0
```

```
V_exc_2 = np.copy(V_orig)
V_exc_2[:, 2] = -1
V_exc_2
gc_mechanism_exc_2 = GC_Mechanism(V_exc_2)
gc_mechanism_exc_2.start()

print("\nThe social cost of buyer 2:",
     np.sum(gc_mechanism_exc_2.Q*gc_mechanism_exc_2.V_orig)-np.sum(np.delete(gc_
 ↪mechanism.Q*gc_mechanism.V_orig, 2, axis=1)))
```

```
House 0 is sold to buyer 0 at price 9


House 1 is sold to buyer 1 at price 6


House 2 is sold to buyer 3 at price 4
```

(continues on next page)

(continued from previous page)

```
Prices of house:
 [9. 6. 4.]


The status matrix:
 [[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 0. 1. 0.]]


The surplus matrix:
 [[ 1.  0. -0.  0.  0.]
 [ 0.  2. -0.  0.  0.]
 [ 0.  0. -0.  1.  0.]]

The social cost of buyer 2: 5.0
```

Part III

Rational Expectation Models

CHAPTER

EIGHT

CASS-KOOPMANS MODEL

Contents

8.1 Overview

This lecture and *Cass-Koopmans Competitive Equilibrium* describe a model that Tjalling Koopmans [Koopmans, 1965] and David Cass [Cass, 1965] used to analyze optimal growth.

The model can be viewed as an extension of the model of Robert Solow described in an earlier lecture but adapted to make the saving rate be a choice.

(Solow assumed a constant saving rate determined outside the model.)

We describe two versions of the model, one in this lecture and the other in *Cass-Koopmans Competitive Equilibrium*.

Together, the two lectures illustrate what is, in fact, a more general connection between a **planned economy** and a decentralized economy organized as a **competitive equilibrium**.

This lecture is devoted to the planned economy version.

In the planned economy, there are

- no prices
- no budget constraints

Instead there is a dictator that tells people

- what to produce

- what to invest in physical capital
- who is to consume what and when

The lecture uses important ideas including

- A min-max problem for solving a planning problem.
- A **shooting algorithm** for solving difference equations subject to initial and terminal conditions.
- A **turnpike** property that describes optimal paths for long but finite-horizon economies.

Let's start with some standard imports:

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
from numba import njit, float64
from numba.experimental import jitclass
import numpy as np
```

8.2 The Model

Time is discrete and takes values $t = 0, 1, \ldots, T$ where T is finite.

(We'll eventually study a limiting case in which $T = +\infty$)

A single good can either be consumed or invested in physical capital.

The consumption good is not durable and depreciates completely if not consumed immediately.

The capital good is durable but depreciates.

We let C_t be the total consumption of a nondurable consumption good at time t.

Let K_t be the stock of physical capital at time t.

Let $\vec{C} = \{C_0, \ldots, C_T\}$ and $\vec{K} = \{K_0, \ldots, K_{T+1}\}$.

8.2.1 Digression: Aggregation Theory

We use a concept of a representative consumer to be thought of as follows.

There is a unit mass of identical consumers indexed by $\omega \in [0, 1]$.

Consumption of consumer ω is $c(\omega)$.

Aggregate consumption is

$$C = \int_0^1 c(\omega) d\omega$$

Consider a welfare problem that chooses an allocation $\{c(\omega)\}$ across consumers to maximize

$$\int_0^1 u(c(\omega)) d\omega$$

where $u(\cdot)$ is a concave utility function with $u' > 0, u'' < 0$ and maximization is subject to

$$C = \int_0^1 c(\omega) d\omega. \tag{8.1}$$

Form a Lagrangian $L = \int_0^1 u(c(\omega))d\omega + \lambda[C - \int_0^1 c(\omega)d\omega]$.

Differentiate under the integral signs with respect to each ω to obtain the first-order necessary conditions

$$u'(c(\omega)) = \lambda.$$

These conditions imply that $c(\omega)$ equals a constant c that is independent of ω.

To find c, use feasibility constraint (8.1) to conclude that

$$c(\omega) = c = C.$$

This line of argument indicates the special *aggregation theory* that lies beneath outcomes in which a representative consumer consumes amount C.

It appears often in aggregate economics.

We shall use this aggregation theory here and also in this lecture *Cass-Koopmans Competitive Equilibrium*.

An Economy

A representative household is endowed with one unit of labor at each t and likes the consumption good at each t.

The representative household inelastically supplies a single unit of labor N_t at each t, so that $N_t = 1$ for all $t \in \{0, 1, \dots, T\}$.

The representative household has preferences over consumption bundles ordered by the utility functional:

$$U(\vec{C}) = \sum_{t=0}^{T} \beta^t \frac{C_t^{1-\gamma}}{1-\gamma} \tag{8.2}$$

where $\beta \in (0, 1)$ is a discount factor and $\gamma > 0$ governs the curvature of the one-period utility function.

Larger γ's imply more curvature.

Note that

$$u(C_t) = \frac{C_t^{1-\gamma}}{1-\gamma} \tag{8.3}$$

satisfies $u' > 0, u'' < 0$.

$u' > 0$ asserts that the consumer prefers more to less.

$u'' < 0$ asserts that marginal utility declines with increases in C_t.

We assume that $K_0 > 0$ is an exogenous initial capital stock.

There is an economy-wide production function

$$F(K_t, N_t) = AK_t^{\alpha} N_t^{1-\alpha} \tag{8.4}$$

with $0 < \alpha < 1$, $A > 0$.

A feasible allocation $\vec{C}, \vec{K}$ satisfies

$$C_t + K_{t+1} \leq F(K_t, N_t) + (1-\delta)K_t \quad \text{for all } t \in \{0, 1, \dots, T\} \tag{8.5}$$

where $\delta \in (0, 1)$ is a depreciation rate of capital.

8.3 Planning Problem

A planner chooses an allocation $\{\vec{C}, \vec{K}\}$ to maximize (8.2) subject to (8.5).

Let $\vec{\mu} = \{\mu_0, \dots, \mu_T\}$ be a sequence of nonnegative **Lagrange multipliers**.

To find an optimal allocation, form a Lagrangian

$$\mathcal{L}(\vec{C}, \vec{K}, \vec{\mu}) = \sum_{t=0}^{T} \beta^t \left\{ u(C_t) + \mu_t \left(F(K_t, 1) + (1-\delta)K_t - C_t - K_{t+1} \right) \right\} \tag{8.6}$$

and pose the following min-max problem:

$$\min_{\vec{\mu}} \max_{\vec{C}, \vec{K}} \mathcal{L}(\vec{C}, \vec{K}, \vec{\mu}) \tag{8.7}$$

- **Extremization** means maximization with respect to $\vec{C}, \vec{K}$ and minimization with respect to $\vec{\mu}$.
- Our problem satisfies conditions that assure that second-order conditions are satisfied at an allocation that satisfies the first-order necessary conditions that we are about to compute.

Before computing first-order conditions, we present some handy formulas.

8.3.1 Useful Properties of Linearly Homogeneous Production Function

The following technicalities will help us.

Notice that

$$F(K_t, N_t) = AK_t^{\alpha} N_t^{1-\alpha} = N_t A \left(\frac{K_t}{N_t} \right)^{\alpha}$$

Define the **output per-capita production function**

$$\frac{F(K_t, N_t)}{N_t} \equiv f\left(\frac{K_t}{N_t} \right) = A \left(\frac{K_t}{N_t} \right)^{\alpha}$$

whose argument is **capital per-capita**.

It is useful to recall the following calculations for the marginal product of capital

$$\begin{aligned} \frac{\partial F(K_t, N_t)}{\partial K_t} &= \frac{\partial N_t f\left(\frac{K_t}{N_t}\right)}{\partial K_t} \\ &= N_t f'\left(\frac{K_t}{N_t} \right) \frac{1}{N_t} \quad \text{(Chain rule)} \\ &= f'\left(\frac{K_t}{N_t} \right) \Big|_{N_t = 1} \\ &= f'(K_t) \end{aligned} \tag{8.8}$$

and the marginal product of labor

$$\begin{aligned} \frac{\partial F(K_t, N_t)}{\partial N_t} &= \frac{\partial N_t f\left(\frac{K_t}{N_t}\right)}{\partial N_t} \quad \text{(Product rule)} \\ &= f\left(\frac{K_t}{N_t} \right) + N_t f'\left(\frac{K_t}{N_t} \right) \frac{-K_t}{N_t^2} \quad \text{(Chain rule)} \\ &= f\left(\frac{K_t}{N_t} \right) - \frac{K_t}{N_t} f'\left(\frac{K_t}{N_t} \right) \Big|_{N_t = 1} \\ &= f(K_t) - f'(K_t) K_t \end{aligned}$$

(Here we are using that $N_t = 1$ for all t, so that $K_t = \frac{K_t}{N_t}$.)

8.3.2 First-order necessary conditions

We now compute **first-order necessary conditions** for extremization of Lagrangian (8.6):

$$C_t: \quad u'(C_t) - \mu_t = 0 \quad \text{for all} \quad t = 0, 1, \ldots, T \tag{8.9}$$

$$K_t: \quad \beta\mu_t \left[(1-\delta) + f'(K_t)\right] - \mu_{t-1} = 0 \quad \text{for all} \quad t = 1, 2, \ldots, T \tag{8.10}$$

$$\mu_t: \quad F(K_t, 1) + (1-\delta)K_t - C_t - K_{t+1} = 0 \quad \text{for all} \quad t = 0, 1, \ldots, T \tag{8.11}$$

$$K_{T+1}: \quad -\mu_T \leq 0, \ \leq 0 \text{ if } K_{T+1} = 0; \ = 0 \text{ if } K_{T+1} > 0 \tag{8.12}$$

In computing (8.10) we recognize that K_t appears in both the time t and time $t-1$ feasibility constraints (8.5).

Restrictions (8.12) come from differentiating with respect to K_{T+1} and applying the following **Karush-Kuhn-Tucker condition** (KKT) (see Karush-Kuhn-Tucker conditions):

$$\mu_T K_{T+1} = 0 \tag{8.13}$$

Combining (8.9) and (8.10) gives

$$\beta u'\left(C_t\right)\left[(1-\delta) + f'\left(K_t\right)\right] - u'\left(C_{t-1}\right) = 0 \quad \text{for all } t = 1, 2, \ldots, T+1$$

which can be rearranged to become

$$\beta u'\left(C_{t+1}\right)\left[(1-\delta) + f'\left(K_{t+1}\right)\right] = u'\left(C_t\right) \quad \text{for all } t = 0, 1, \ldots, T \tag{8.14}$$

Applying the inverse marginal utility of consumption function on both sides of the above equation gives

$$C_{t+1} = u'^{-1}\left(\left(\frac{\beta}{u'(C_t)}[f'(K_{t+1}) + (1-\delta)]\right)^{-1}\right)$$

which for our utility function (8.3) becomes the consumption **Euler equation**

$$C_{t+1} = \left(\beta C_t^{\gamma}[f'(K_{t+1}) + (1-\delta)]\right)^{1/\gamma}$$

which we can combine with the feasibility constraint (8.5) to get

$$\begin{aligned} C_{t+1} &= C_t\left(\beta[f'(F(K_t, 1) + (1-\delta)K_t - C_t) + (1-\delta)]\right)^{1/\gamma} \\ K_{t+1} &= F(K_t, 1) + (1-\delta)K_t - C_t. \end{aligned}$$

This is a pair of non-linear first-order difference equations that map C_t, K_t into C_{t+1}, K_{t+1} and that an optimal sequence $\vec{C}, \vec{K}$ must satisfy.

It must also satisfy the initial condition that K_0 is given and $K_{T+1} = 0$.

Below we define a `jitclass` that stores parameters and functions that define our economy.

```
planning_data = [
    ('γ', float64),     # Coefficient of relative risk aversion
    ('β', float64),     # Discount factor
    ('δ', float64),     # Depreciation rate on capital
    ('α', float64),     # Return to capital per capita
    ('A', float64)      # Technology
]
```

```
@jitclass(planning_data)
class PlanningProblem():

    def __init__(self, γ=2, β=0.95, δ=0.02, α=0.33, A=1):

        self.γ, self.β = γ, β
        self.δ, self.α, self.A = δ, α, A

    def u(self, c):
        '''
        Utility function
        ASIDE: If you have a utility function that is hard to solve by hand
        you can use automatic or symbolic differentiation
        See https://github.com/HIPS/autograd
        '''
        γ = self.γ

        return c ** (1 - γ) / (1 - γ) if γ!= 1 else np.log(c)

    def u_prime(self, c):
        'Derivative of utility'
        γ = self.γ

        return c ** (-γ)

    def u_prime_inv(self, c):
        'Inverse of derivative of utility'
        γ = self.γ

        return c ** (-1 / γ)

    def f(self, k):
        'Production function'
        α, A = self.α, self.A

        return A * k ** α

    def f_prime(self, k):
        'Derivative of production function'
        α, A = self.α, self.A

        return α * A * k ** (α - 1)

    def f_prime_inv(self, k):
        'Inverse of derivative of production function'
        α, A = self.α, self.A

        return (k / (A * α)) ** (1 / (α - 1))

    def next_k_c(self, k, c):
        ''''
        Given the current capital Kt and an arbitrary feasible
        consumption choice Ct, computes Kt+1 by state transition law
        and optimal Ct+1 by Euler equation.
        '''
        β, δ = self.β, self.δ
        u_prime, u_prime_inv = self.u_prime, self.u_prime_inv
```

(continues on next page)

(continued from previous page)

```
        f, f_prime = self.f, self.f_prime

        k_next = f(k) + (1 - δ) * k - c
        c_next = u_prime_inv(u_prime(c) / (β * (f_prime(k_next) + (1 - δ))))

        return k_next, c_next
```

We can construct an economy with the Python code:

```
pp = PlanningProblem()
```

8.4 Shooting Algorithm

We use **shooting** to compute an optimal allocation $\vec{C}, \vec{K}$ and an associated Lagrange multiplier sequence $\vec{\mu}$.

First-order necessary conditions (8.9), (8.10), and (8.11) for the planning problem form a system of **difference equations** with two boundary conditions:

- K_0 is a given **initial condition** for capital
- $K_{T+1} = 0$ is a **terminal condition** for capital that we deduced from the first-order necessary condition for K_{T+1} the KKT condition (8.13)

We have no initial condition for the Lagrange multiplier μ_0.

If we did, our job would be easy:

- Given μ_0 and k_0, we could compute c_0 from equation (8.9) and then k_1 from equation (8.11) and μ_1 from equation (8.10).
- We could continue in this way to compute the remaining elements of $\vec{C}, \vec{K}, \vec{\mu}$.

However, we woujld not be assured that the Kuhn-Tucker condition (8.13) would be satisfied.

Furthermore, we don't have an initial condition for μ_0.

So this won't work.

Indeed, part of our task is to compute the **optimal** value of μ_0.

To compute μ_0 and the other objects we want, a simple modification of the above procedure will work.

It is called the **shooting algorithm**.

It is an instance of a **guess and verify** algorithm that consists of the following steps:

- Guess an initial Lagrange multiplier μ_0.
- Apply the **simple algorithm** described above.
- Compute K_{T+1} and check whether it equals zero.
- If $K_{T+1} = 0$, we have solved the problem.
- If $K_{T+1} > 0$, lower μ_0 and try again.
- If $K_{T+1} < 0$, raise μ_0 and try again.

The following Python code implements the shooting algorithm for the planning problem.

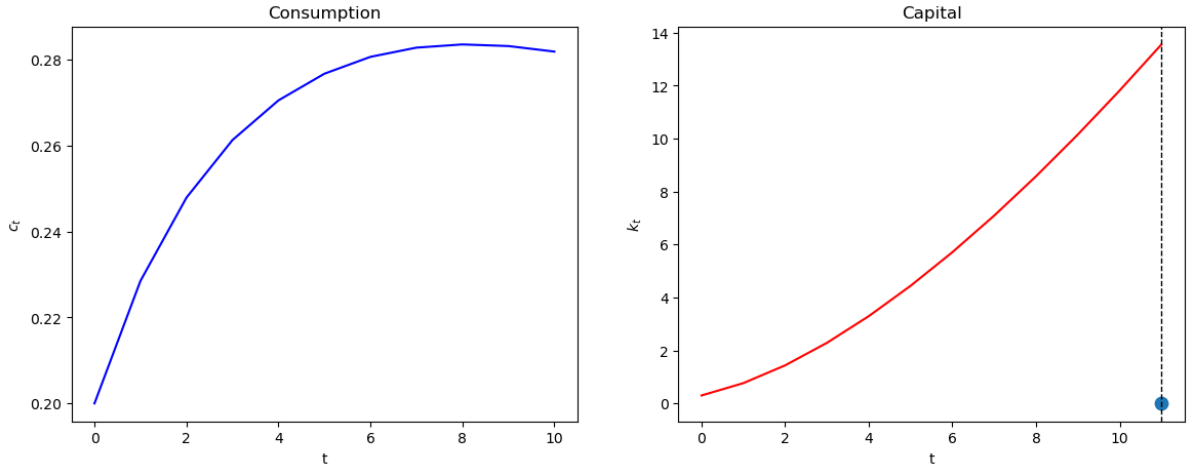
(Actually, we modified the preceding algorithm slightly by starting with a guess for c_0 instead of μ_0 in the following code.)

```
@njit
def shooting(pp, c0, k0, T=10):
    '''
    Given the initial condition of capital k0 and an initial guess
    of consumption c0, computes the whole paths of c and k
    using the state transition law and Euler equation for T periods.
    '''
    if c0 > pp.f(k0):
        print("initial consumption is not feasible")

        return None

    # initialize vectors of c and k
    c_vec = np.empty(T+1)
    k_vec = np.empty(T+2)

    c_vec[0] = c0
    k_vec[0] = k0

    for t in range(T):
        k_vec[t+1], c_vec[t+1] = pp.next_k_c(k_vec[t], c_vec[t])

    k_vec[T+1] = pp.f(k_vec[T]) + (1 - pp.δ) * k_vec[T] - c_vec[T]

    return c_vec, k_vec
```

We'll start with an incorrect guess.

```
paths = shooting(pp, 0.2, 0.3, T=10)
```

```
fig, axs = plt.subplots(1, 2, figsize=(14, 5))

colors = ['blue', 'red']
titles = ['Consumption', 'Capital']
ylabels = ['$c_t$', '$k_t$']

T = paths[0].size - 1
for i in range(2):
    axs[i].plot(paths[i], c=colors[i])
    axs[i].set(xlabel='t', ylabel=ylabels[i], title=titles[i])

axs[1].scatter(T+1, 0, s=80)
axs[1].axvline(T+1, color='k', ls='--', lw=1)

plt.show()
```



Evidently, our initial guess for μ_0 is too high, so initial consumption too low.

We know this because we miss our $K_{T+1} = 0$ target on the high side.

Now we automate things with a search-for-a-good μ_0 algorithm that stops when we hit the target $K_{t+1} = 0$.

We use a **bisection method**.

We make an initial guess for C_0 (we can eliminate μ_0 because C_0 is an exact function of μ_0).

We know that the lowest C_0 can ever be is 0 and that the largest it can be is initial output $f(K_0)$.

Guess C_0 and shoot forward to $T + 1$.

If $K_{T+1} > 0$, we take it to be our new **lower** bound on C_0.

If $K_{T+1} < 0$, we take it to be our new **upper** bound.

Make a new guess for C_0 that is halfway between our new upper and lower bounds.

Shoot forward again, iterating on these steps until we converge.

When K_{T+1} gets close enough to 0 (i.e., within an error tolerance bounds), we stop.

```
@njit
def bisection(pp, c0, k0, T=10, tol=1e-4, max_iter=500, k_ter=0, verbose=True):

    # initial boundaries for guess c0
    c0_upper = pp.f(k0)
    c0_lower = 0

    i = 0
    while True:
        c_vec, k_vec = shooting(pp, c0, k0, T)
        error = k_vec[-1] - k_ter

        # check if the terminal condition is satisfied
        if np.abs(error) < tol:
            if verbose:
                print('Converged successfully on iteration ', i+1)
            return c_vec, k_vec

        i += 1
        if i == max_iter:
```

(continues on next page)

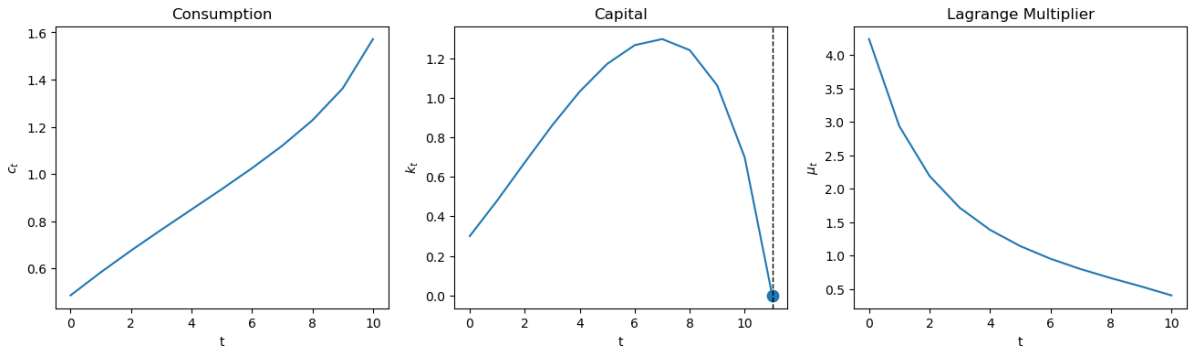
(continued from previous page)

```
            if verbose:
                print('Convergence failed.')
            return c_vec, k_vec

        # if iteration continues, updates boundaries and guess of c0
        if error > 0:
            c0_lower = c0
        else:
            c0_upper = c0

        c0 = (c0_lower + c0_upper) / 2
```

```
def plot_paths(pp, c0, k0, T_arr, k_ter=0, k_ss=None, axs=None):

    if axs is None:
        fix, axs = plt.subplots(1, 3, figsize=(16, 4))
    ylabels = ['$c_t$', '$k_t$', '$\mu_t$']
    titles = ['Consumption', 'Capital', 'Lagrange Multiplier']

    c_paths = []
    k_paths = []
    for T in T_arr:
        c_vec, k_vec = bisection(pp, c0, k0, T, k_ter=k_ter, verbose=False)
        c_paths.append(c_vec)
        k_paths.append(k_vec)

        μ_vec = pp.u_prime(c_vec)
        paths = [c_vec, k_vec, μ_vec]

        for i in range(3):
            axs[i].plot(paths[i])
            axs[i].set(xlabel='t', ylabel=ylabels[i], title=titles[i])

        # Plot steady state value of capital
        if k_ss is not None:
            axs[1].axhline(k_ss, c='k', ls='--', lw=1)

        axs[1].axvline(T+1, c='k', ls='--', lw=1)
        axs[1].scatter(T+1, paths[1][-1], s=80)

    return c_paths, k_paths
```

Now we can solve the model and plot the paths of consumption, capital, and Lagrange multiplier.

```
plot_paths(pp, 0.3, 0.3, [10]);
```



8.5 Setting Initial Capital to Steady State Capital

When $T \to +\infty$, the optimal allocation converges to steady state values of C_t and K_t.

It is instructive to set K_0 equal to the $\lim_{T \to +\infty} K_t$, which we'll call steady state capital.

In a steady state $K_{t+1} = K_t = \bar{K}$ for all very large t.

Evalauating feasibility constraint (8.5) at $\bar{K}$ gives

$$f(\bar{K}) - \delta \bar{K} = \bar{C} \tag{8.15}$$

Substituting $K_t = \bar{K}$ and $C_t = \bar{C}$ for all t into (8.14) gives

$$1 = \beta \frac{u'(\bar{C})}{u'(\bar{C})} [f'(\bar{K}) + (1 - \delta)]$$

Defining $\beta = \frac{1}{1+\rho}$, and cancelling gives

$$1 + \rho = 1[f'(\bar{K}) + (1 - \delta)]$$

Simplifying gives

$$f'(\bar{K}) = \rho + \delta$$

and

$$\bar{K} = f'^{-1}(\rho + \delta)$$

For production function (8.4), this becomes

$$\alpha \bar{K}^{\alpha - 1} = \rho + \delta$$

As an example, after setting $\alpha = .33$, $\rho = 1/\beta - 1 = 1/(19/20) - 1 = 20/19 - 19/19 = 1/19$, $\delta = 1/50$, we get

$$\bar{K} = \left(\frac{\frac{33}{100}}{\frac{1}{50} + \frac{1}{19}} \right)^{\frac{67}{100}} \approx 9.57583$$

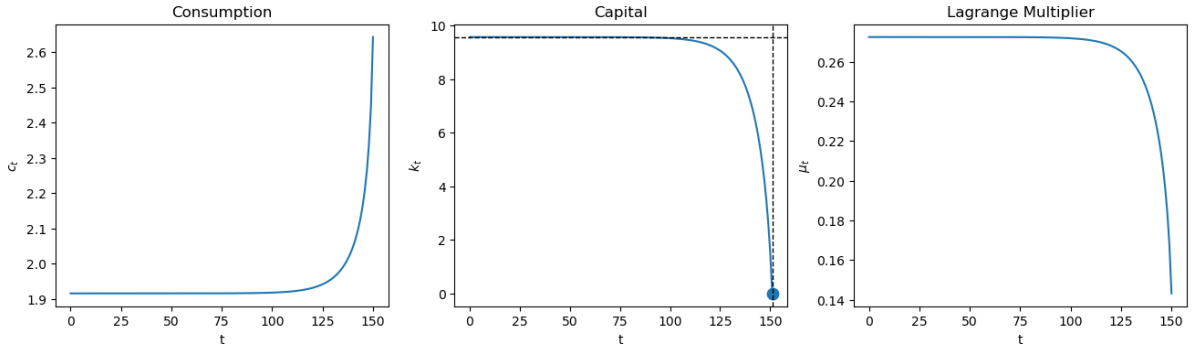
Let's verify this with Python and then use this steady state $\bar{K}$ as our initial capital stock K_0.

```
ρ = 1 / pp.β - 1
k_ss = pp.f_prime_inv(ρ+pp.δ)

print(f'steady state for capital is: {k_ss}')
```

```
steady state for capital is: 9.57583816331462
```

Now we plot

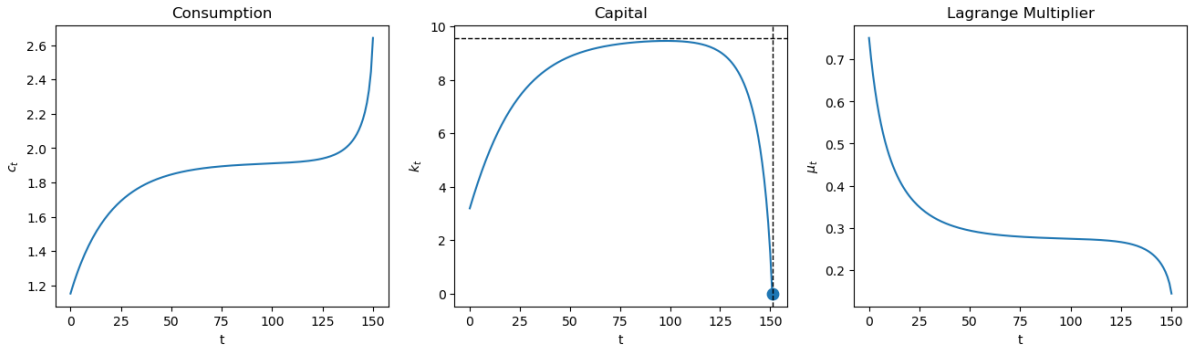
```
plot_paths(pp, 0.3, k_ss, [150], k_ss=k_ss);
```



Evidently, with a large value of T, K_t stays near K_0 until t approaches T closely.

Let's see what the planner does when we set K_0 below $\bar{K}$.

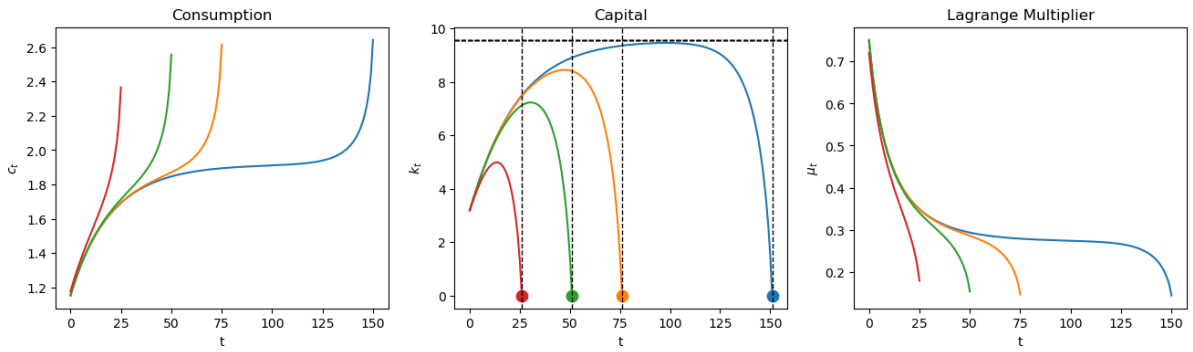
```
plot_paths(pp, 0.3, k_ss/3, [150], k_ss=k_ss);
```



Notice how the planner pushes capital toward the steady state, stays near there for a while, then pushes K_t toward the terminal value $K_{T+1} = 0$ when t closely approaches T.

The following graphs compare optimal outcomes as we vary T.

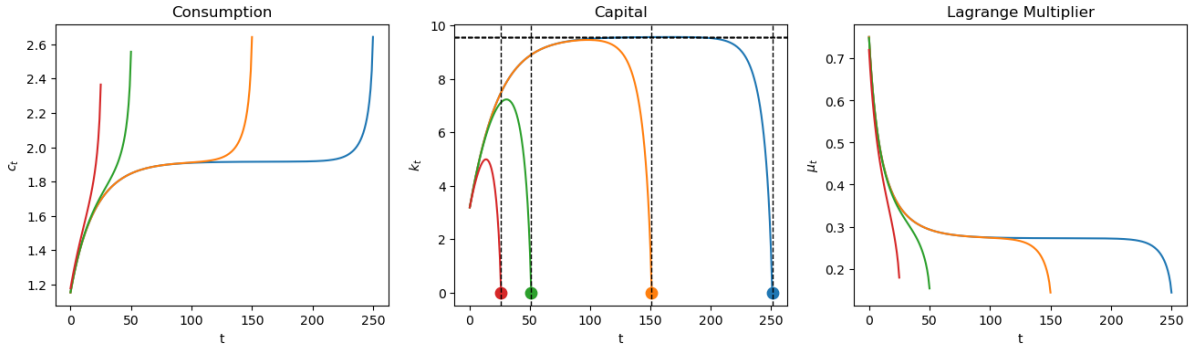
```
plot_paths(pp, 0.3, k_ss/3, [150, 75, 50, 25], k_ss=k_ss);
```



8.6 A Turnpike Property

The following calculation indicates that when T is very large, the optimal capital stock stays close to its steady state value most of the time.

```
plot_paths(pp, 0.3, k_ss/3, [250, 150, 50, 25], k_ss=k_ss);
```



In the above graphs, different colors are associated with different horizons T.

Notice that as the horizon increases, the planner keeps K_t closer to the steady state value $\bar{K}$ for longer.

This pattern reflects a **turnpike** property of the steady state.

A rule of thumb for the planner is

- from K_0, push K_t toward the steady state and stay close to the steady state until time approaches T.

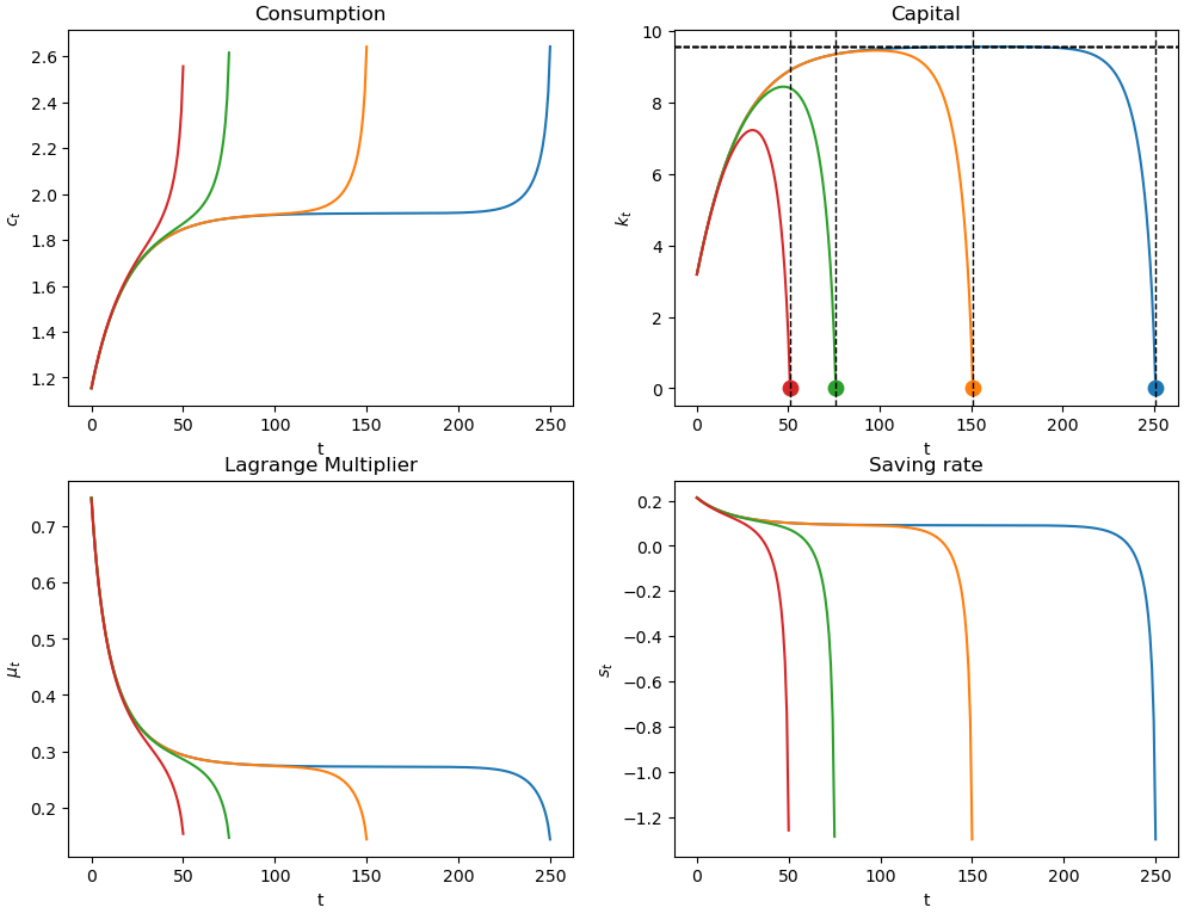
The planner accomplishes this by adjusting the saving rate $\frac{f(K_t)-C_t}{f(K_t)}$ over time.

Let's calculate and plot the saving rate.

```
@njit
def saving_rate(pp, c_path, k_path):
    'Given paths of c and k, computes the path of saving rate.'
    production = pp.f(k_path[:-1])

    return (production - c_path) / production
```

```
def plot_saving_rate(pp, c0, k0, T_arr, k_ter=0, k_ss=None, s_ss=None):

    fix, axs = plt.subplots(2, 2, figsize=(12, 9))

    c_paths, k_paths = plot_paths(pp, c0, k0, T_arr, k_ter=k_ter, k_ss=k_ss, axs=axs.
↪flatten())

    for i, T in enumerate(T_arr):
        s_path = saving_rate(pp, c_paths[i], k_paths[i])
        axs[1, 1].plot(s_path)

    axs[1, 1].set(xlabel='t', ylabel='$s_t$', title='Saving rate')

    if s_ss is not None:
        axs[1, 1].hlines(s_ss, 0, np.max(T_arr), linestyle='--')
```

```
plot_saving_rate(pp, 0.3, k_ss/3, [250, 150, 75, 50], k_ss=k_ss)
```



8.7 A Limiting Infinite Horizon Economy

We want to set $T = +\infty$.

The appropriate thing to do is to replace terminal condition (8.12) with

$$\lim_{T \to +\infty} \beta^T u'(C_T) K_{T+1} = 0,$$

a condition that will be satisfied by a path that converges to an optimal steady state.

We can approximate the optimal path by starting from an arbitrary initial K_0 and shooting towards the optimal steady state K at a large but finite $T + 1$.

In the following code, we do this for a large T and plot consumption, capital, and the saving rate.

We know that in the steady state that the saving rate is constant and that $\bar{s} = \frac{f(\bar{K}) - \bar{C}}{f(\bar{K})}$.

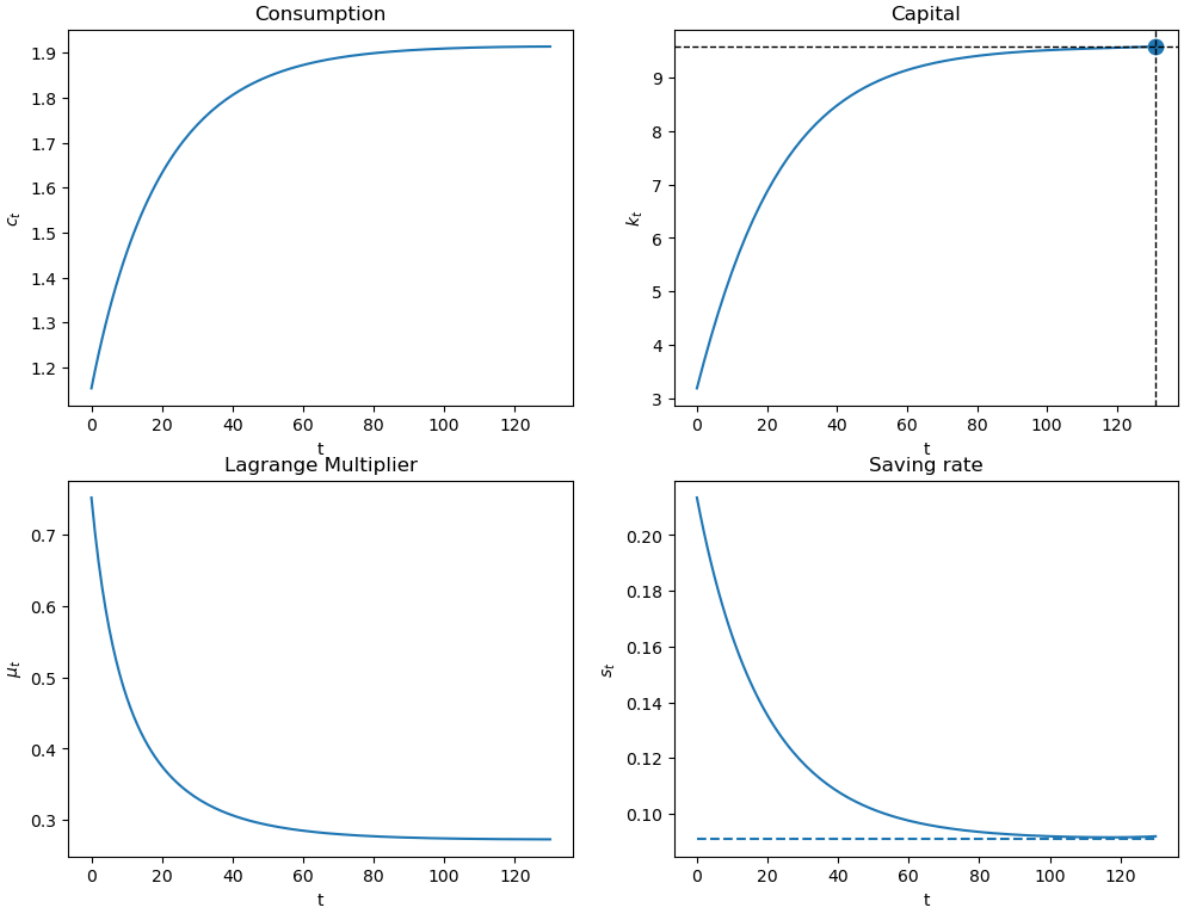
From (8.15) the steady state saving rate equals

$$\bar{s} = \frac{\delta \bar{K}}{f(\bar{K})}$$

The steady state saving rate $\bar{S} = \bar{s} f(\bar{K})$ is the amount required to offset capital depreciation each period.

We first study optimal capital paths that start below the steady state.

```
# steady state of saving rate
s_ss = pp.δ * k_ss / pp.f(k_ss)

plot_saving_rate(pp, 0.3, k_ss/3, [130], k_ter=k_ss, k_ss=k_ss, s_ss=s_ss)
```



Since $K_0 < \bar{K}$, $f'(K_0) > \rho + \delta$.

The planner chooses a positive saving rate that is higher than the steady state saving rate.

Note that $f''(K) < 0$, so as K rises, $f'(K)$ declines.

The planner slowly lowers the saving rate until reaching a steady state in which $f'(K) = \rho + \delta$.

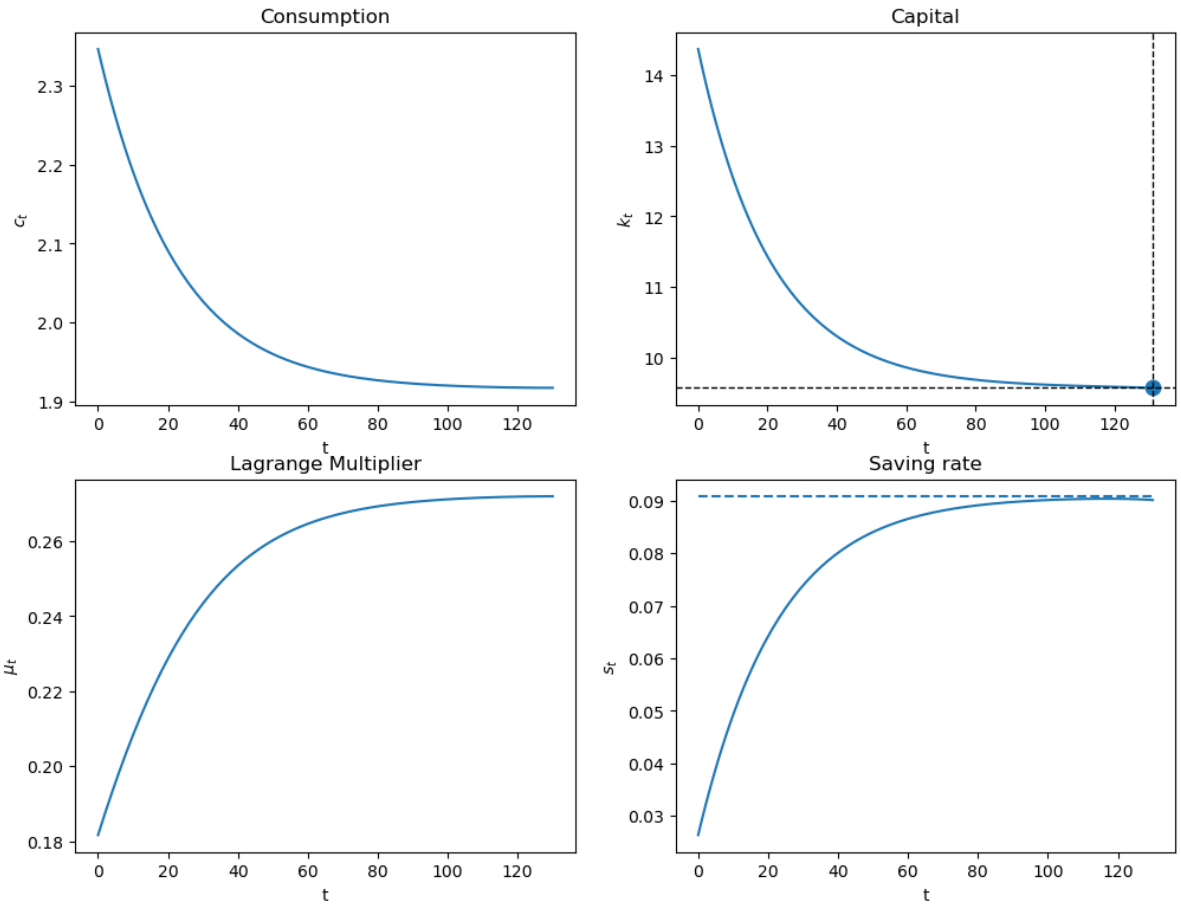
8.7.1 Exercise

Exercise 8.7.1

- Plot the optimal consumption, capital, and saving paths when the initial capital level begins at 1.5 times the steady state level as we shoot towards the steady state at $T = 130$.
- Why does the saving rate respond as it does?

Solution to Exercise 8.7.1

```
plot_saving_rate(pp, 0.3, k_ss*1.5, [130], k_ter=k_ss, k_ss=k_ss, s_ss=s_ss)
```



8.8 Concluding Remarks

In *Cass-Koopmans Competitive Equilibrium*, we study a decentralized version of an economy with exactly the same technology and preference structure as deployed here.

In that lecture, we replace the planner of this lecture with Adam Smith's **invisible hand**.

In place of quantity choices made by the planner, there are market prices that are set by a *deus ex machina* from outside the model, a so-called invisible hand.

Equilibrium market prices must reconcile distinct decisions that are made independently by a representative household and a representative firm.

The relationship between a command economy like the one studied in this lecture and a market economy like that studied in *Cass-Koopmans Competitive Equilibrium* is a foundational topic in general equilibrium theory and welfare economics.

CHAPTER

NINE

CASS-KOOPMANS COMPETITIVE EQUILIBRIUM

Contents

9.1 Overview

This lecture continues our analysis in this lecture *Cass-Koopmans Planning Model* about the model that Tjalling Koopmans [Koopmans, 1965] and David Cass [Cass, 1965] used to study optimal capital accumulation.

This lecture illustrates what is, in fact, a more general connection between a **planned economy** and an economy organized as a competitive equilibrium or a **market economy**.

The earlier lecture *Cass-Koopmans Planning Model* studied a planning problem and used ideas including

- A Lagrangian formulation of the planning problem that leads to a system of difference equations.
- A **shooting algorithm** for solving difference equations subject to initial and terminal conditions.
- A **turnpike** property that describes optimal paths for long-but-finite horizon economies.

The present lecture uses additional ideas including

- Hicks-Arrow prices, named after John R. Hicks and Kenneth Arrow.
- A connection between some Lagrange multipliers from the planning problem and the Hicks-Arrow prices.
- A **Big K , little k** trick widely used in macroeconomic dynamics.
  - We shall encounter this trick in this lecture and also in this lecture.
- A non-stochastic version of a theory of the **term structure of interest rates**.

- An intimate connection between two ways to organize an economy, namely:
  - **socialism** in which a central planner commands the allocation of resources, and
  - **competitive markets** in which competitive equilibrium **prices** induce individual consumers and producers to choose a socially optimal allocation as unintended consequences of their selfish decisions

Let's start with some standard imports:

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
from numba import njit, float64
from numba.experimental import jitclass
import numpy as np
```

9.2 Review of Cass-Koopmans Model

The physical setting is identical with that in *Cass-Koopmans Planning Model*.

Time is discrete and takes values $t = 0, 1, \ldots, T$.

Output of a single good can either be consumed or invested in physical capital.

The capital good is durable but partially depreciates each period at a constant rate.

We let C_t be a nondurable consumption good at time t.

Let K_t be the stock of physical capital at time t.

Let $\vec{C} = \{C_0, \ldots, C_T\}$ and $\vec{K} = \{K_0, \ldots, K_{T+1}\}$.

A representative household is endowed with one unit of labor at each t and likes the consumption good at each t.

The representative household inelastically supplies a single unit of labor N_t at each t, so that $N_t = 1$ for all $t \in \{0, 1, \ldots, T\}$.

The representative household has preferences over consumption bundles ordered by the utility functional:

$$U(\vec{C}) = \sum_{t=0}^{T} \beta^t \frac{C_t^{1-\gamma}}{1-\gamma}$$

where $\beta \in (0, 1)$ is a discount factor and $\gamma > 0$ governs the curvature of the one-period utility function.

We assume that $K_0 > 0$.

There is an economy-wide production function

$$F(K_t, N_t) = AK_t^{\alpha} N_t^{1-\alpha}$$

with $0 < \alpha < 1$, $A > 0$.

A feasible allocation $\vec{C}, \vec{K}$ satisfies

$$C_t + K_{t+1} \leq F(K_t, N_t) + (1-\delta)K_t \quad \text{for all } t \in \{0, 1, \ldots, T\}$$

where $\delta \in (0, 1)$ is a depreciation rate of capital.

9.2.1 Planning Problem

In this lecture *Cass-Koopmans Planning Model*, we studied a problem in which a planner chooses an allocation $\{\vec{C}, \vec{K}\}$ to maximize (8.2) subject to (8.5).

The allocation that solves the planning problem reappears in a competitive equilibrium, as we shall see below.

9.3 Competitive Equilibrium

We now study a decentralized version of the economy.

It shares the same technology and preference structure as the planned economy studied in this lecture *Cass-Koopmans Planning Model*.

But now there is no planner.

There are (unit masses of) price-taking consumers and firms.

Market prices are set to reconcile distinct decisions that are made separately by a representative consumer and a representative firm.

There is a representative consumer who has the same preferences over consumption plans as did a consumer in the planned economy.

Instead of being told what to consume and save by a planner, a consumer (also known as a *household*) chooses for itself subject to a budget constraint.

- At each time t, the consumer receives wages and rentals of capital from a firm – these comprise its **income** at time t.
- The consumer decides how much income to allocate to consumption or to savings.
- The household can save either by acquiring additional physical capital (it trades one for one with time t consumption) or by acquiring claims on consumption at dates other than t.
- The household owns physical capital and labor and rents them to the firm.
- The household consumes, supplies labor, and invests in physical capital.
- A profit-maximizing representative firm operates the production technology.
- The firm rents labor and capital each period from the representative household and sells its output each period to the household.
- The representative household and the representative firm are both **price takers** who believe that prices are not affected by their choices

---

Note: Again, we can think of there being unit measures of identical representative consumers and identical representative firms.

9.4 Market Structure

The representative household and the representative firm are both price takers.

The household owns both factors of production, namely, labor and physical capital.

Each period, the firm rents both factors from the household.

There is a **single** grand competitive market in which a household trades date 0 goods for goods at all other dates $t = 1, 2, \ldots, T$.

9.4.1 Prices

There are sequences of prices $\{w_t, \eta_t\}_{t=0}^T = \{\vec{w}, \vec{\eta}\}$ where

- w_t is a wage, i.e., a rental rate, for labor at time t
- η_t is a rental rate for capital at time t

In addition there is a vector $\{q_t^0\}$ of intertemporal prices where

- q_t^0 is the price at time 0 of one unit of the good at date t.

We call $\{q_t^0\}_{t=0}^T$ a vector of **Hicks-Arrow prices**, named after the 1972 economics Nobel prize winners.

Because is a **relative price**. the unit of account in terms of which the prices q_t^0 are stated is; we are free to re-normalize them by multiplying all of them by a positive scalar, say $\lambda > 0$.

Units of q_t^0 could be set so that they are

$$\frac{\text{number of time 0 goods}}{\text{number of time t goods}}$$

In this case, we would be taking the time 0 consumption good to be the **numeraire**.

9.5 Firm Problem

At time t a representative firm hires labor $\tilde{n}_t$ and capital $\tilde{k}_t$.

The firm's profits at time t are

$$F(\tilde{k}_t, \tilde{n}_t) - w_t \tilde{n}_t - \eta_t \tilde{k}_t$$

where w_t is a wage rate at t and η_t is the rental rate on capital at t.

As in the planned economy model

$$F(\tilde{k}_t, \tilde{n}_t) = A \tilde{k}_t^\alpha \tilde{n}_t^{1-\alpha}$$

9.5.1 Zero Profit Conditions

Zero-profits conditions for capital and labor are

$$F_k(\tilde{k}_t, \tilde{n}_t) = \eta_t$$

and

$$F_n(\tilde{k}_t, \tilde{n}_t) = w_t \tag{9.1}$$

These conditions emerge from a no-arbitrage requirement.

To describe this no-arbitrage profits reasoning, we begin by applying a theorem of Euler about linearly homogenous functions.

The theorem applies to the Cobb-Douglas production function because we it displays constant returns to scale:

$$\alpha F(\tilde{k}_t, \tilde{n}_t) = F(\alpha\tilde{k}_t, \alpha\tilde{n}_t)$$

for $\alpha \in (0, 1)$.

Taking partial derivatives $\frac{\partial}{\partial\alpha}$ on both sides of the above equation gives

$$F(\tilde{k}_t, \tilde{n}_t) = \frac{\partial F}{\partial \tilde{k}_t}\tilde{k}_t + \frac{\partial F}{\partial \tilde{n}_t}\tilde{n}_t$$

Rewrite the firm's profits as

$$\frac{\partial F}{\partial \tilde{k}_t}\tilde{k}_t + \frac{\partial F}{\partial \tilde{n}_t}\tilde{n}_t - w_t\tilde{n}_t - \eta_t k_t$$

or

$$\left(\frac{\partial F}{\partial \tilde{k}_t} - \eta_t\right)\tilde{k}_t + \left(\frac{\partial F}{\partial \tilde{n}_t} - w_t\right)\tilde{n}_t$$

Because F is homogeneous of degree 1, it follows that $\frac{\partial F}{\partial \tilde{k}_t}$ and $\frac{\partial F}{\partial \tilde{n}_t}$ are homogeneous of degree 0 and therefore fixed with respect to $\tilde{k}_t$ and $\tilde{n}_t$.

If $\frac{\partial F}{\partial \tilde{k}_t} > \eta_t$, then the firm makes positive profits on each additional unit of $\tilde{k}_t$, so it would want to make $\tilde{k}_t$ arbitrarily large.

But setting $\tilde{k}_t = +\infty$ is not physically feasible, so **equilibrium** prices must take values that present the firm with no such arbitrage opportunity.

A similar argument applies if $\frac{\partial F}{\partial \tilde{n}_t} > w_t$.

If $\frac{\partial \tilde{k}_t}{\partial \tilde{k}_t} < \eta_t$, the firm would want to set $\tilde{k}_t$ to zero, which is not feasible.

It is convenient to define $\vec{w} = \{w_0, \ldots, w_T\}$ and $\vec{\eta} = \{\eta_0, \ldots, \eta_T\}$.

9.6 Household Problem

A representative household lives at $t = 0, 1, \ldots, T$.

At t, the household rents 1 unit of labor and k_t units of capital to a firm and receives income

$$w_t 1 + \eta_t k_t$$

At t the household allocates its income to the following purchases between the following two categories:

- consumption c_t
- net investment $k_{t+1} - (1-\delta)k_t$

Here $(k_{t+1} - (1-\delta)k_t)$ is the household's net investment in physical capital and $\delta \in (0,1)$ is again a depreciation rate of capital.

In period t, the consumer is free to purchase more goods to be consumed and invested in physical capital than its income from supplying capital and labor to the firm, provided that in some other periods its income exceeds its purchases.

A consumer's net excess demand for time t consumption goods is the gap

$$e_t \equiv (c_t + (k_{t+1} - (1-\delta)k_t)) - (w_t 1 + \eta_t k_t)$$

Let $\vec{c} = \{c_0, \ldots, c_T\}$ and let $\vec{k} = \{k_1, \ldots, k_{T+1}\}$.

k_0 is given to the household.

The household faces a **single** budget constraint that requires that the present value of the household's net excess demands must be zero:

$$\sum_{t=0}^{T} q_t^0 e_t \leq 0$$

or

$$\sum_{t=0}^{T} q_t^0 \left(c_t + (k_{t+1} - (1-\delta)k_t)\right) \leq \sum_{t=0}^{T} q_t^0 (w_t 1 + \eta_t k_t)$$

The household faces price system $\{q_t^0, w_t, \eta_t\}$ as a price-taker and chooses an allocation to solve the constrained optimization problem:

$$\max_{\vec{c},\vec{k}} \sum_{t=0}^{T} \beta^t u(c_t)$$

$$\text{subject to } \sum_{t=0}^{T} q_t^0 \left(c_t + (k_{t+1} - (1-\delta)k_t) - (w_t - \eta_t k_t)\right) \leq 0$$

Components of a **price system** have the following units:

- w_t is measured in units of the time t good per unit of time t labor hired
- η_t is measured in units of the time t good per unit of time t capital hired
- q_t^0 is measured in units of a numeraire per unit of the time t good

9.6.1 Definitions

- A **price system** is a sequence $\{q_t^0, \eta_t, w_t\}_{t=0}^T = \{\vec{q}, \vec{\eta}, \vec{w}\}$.
- An **allocation** is a sequence $\{c_t, k_{t+1}, n_t = 1\}_{t=0}^T = \{\vec{c}, \vec{k}, \vec{n}\}$.
- A **competitive equilibrium** is a price system and an allocation with the following properties:
  - Given the price system, the allocation solves the household's problem.
  - Given the price system, the allocation solves the firm's problem.

The vision here is that an equilibrium price system and allocation are determined once and for all.

In effect, we imagine that all trades occur just before time 0.

9.7 Computing a Competitive Equilibrium

We compute a competitive equilibrium by using a **guess and verify** approach.

- We **guess** equilibrium price sequences $\{\vec{q}, \vec{\eta}, \vec{w}\}$.
- We then **verify** that at those prices, the household and the firm choose the same allocation.

9.7.1 Guess for Price System

In this lecture *Cass-Koopmans Planning Model*, we computed an allocation $\{\vec{C}, \vec{K}, \vec{N}\}$ that solves a planning problem.

We use that allocation to construct a guess for the equilibrium price system.

Note: This allocation will constitute the **Big** K to be in the present instance of the **Big** K **, little** k trick that we'll apply to a competitive equilibrium in the spirit of this lecture and this lecture.

In particular, we shall use the following procedure:

- obtain first-order conditions for the representative firm and the representative consumer.
- from these equations, obtain a new set of equations by replacing the firm's choice variables $\tilde{k}, \tilde{n}$ and the consumer's choice variables with the quantities $\vec{C}, \vec{K}$ that solve the planning problem.
- solve the resulting equations for $\{\vec{q}, \vec{\eta}, \vec{w}\}$ as functions of $\vec{C}, \vec{K}$.
- verify that at these prices, $c_t = C_t, k_t = \tilde{k}_t = K_t, \tilde{n}_t = 1$ for $t = 0, 1, \ldots, T$.

Thus, we guess that for $t = 0, \ldots, T$:

$$q_t^0 = \beta^t u'(C_t) \tag{9.2}$$

$$w_t = f(K_t) - K_t f'(K_t) \tag{9.3}$$

$$\eta_t = f'(K_t) \tag{9.4}$$

At these prices, let capital chosen by the household be

$$k_t^*(\vec{q}, \vec{w}, \vec{\eta}), \quad t \geq 0 \tag{9.5}$$

and let the allocation chosen by the firm be

$$\tilde{k}_t^*(\vec{q}, \vec{w}, \vec{\eta}), \quad t \geq 0$$

and so on.

If our guess for the equilibrium price system is correct, then it must occur that

$$k_t^* = \tilde{k}_t^* \tag{9.6}$$

$$1 = \tilde{n}_t^* \tag{9.7}$$

$$c_t^* + k_{t+1}^* - (1 - \delta) k_t^* = F(\tilde{k}_t^*, \tilde{n}_t^*)$$

We shall verify that for $t = 0, \ldots, T$ allocations chosen by the household and the firm both equal the allocation that solves the planning problem:

$$k_t^* = \tilde{k}_t^* = K_t, \tilde{n}_t = 1, c_t^* = C_t \tag{9.8}$$

9.7.2 Verification Procedure

Our approach is firsts to stare at first-order necessary conditions for optimization problems of the household and the firm.

At the price system we have guessed, we'll then verify that both sets of first-order conditions are satisfied at the allocation that solves the planning problem.

9.7.3 Household's Lagrangian

To solve the household's problem, we formulate the Lagrangian

$$\mathcal{L}(\vec{c}, \vec{k}, \lambda) = \sum_{t=0}^{T} \beta^t u(c_t) + \lambda \left(\sum_{t=0}^{T} q_t^0 \left(((1-\delta)k_t - w_t) + \eta_t k_t - c_t - k_{t+1} \right) \right)$$

and attack the min-max problem:

$$\min_{\lambda} \max_{\vec{c}, \vec{k}} \mathcal{L}(\vec{c}, \vec{k}, \lambda)$$

First-order conditions are

$$c_t : \quad \beta^t u'(c_t) - \lambda q_t^0 = 0 \quad t = 0, 1, \ldots, T \tag{9.9}$$

$$k_t : \quad -\lambda q_t^0 \left[(1-\delta) + \eta_t \right] + \lambda q_{t-1}^0 = 0 \quad t = 1, 2, \ldots, T+1 \tag{9.10}$$

$$\lambda : \quad \left(\sum_{t=0}^{T} q_t^0 \left(c_t + (k_{t+1} - (1-\delta)k_t) - w_t - \eta_t k_t \right) \right) \leq 0 \tag{9.11}$$

$$k_{T+1} : \quad -\lambda q_0^{T+1} \leq 0, \ \leq 0 \text{ if } k_{T+1} = 0; \ = 0 \text{ if } k_{T+1} > 0 \tag{9.12}$$

Now we plug in our guesses of prices and do some algebra in the hope of recovering all first-order necessary conditions (8.9)-(8.12) for the planning problem from this lecture *Cass-Koopmans Planning Model*.

Combining (9.9) and (9.2), we get:

$$u'(C_t) = \mu_t$$

which is (8.9).

Combining (9.10), (9.2), and (9.4), we get:

$$-\lambda \beta^t \mu_t \left[(1-\delta) + f'(K_t) \right] + \lambda \beta^{t-1} \mu_{t-1} = 0 \tag{9.13}$$

Rewriting (9.13) by dividing by λ on both sides (which is nonzero since u'>0) we get:

$$\beta^t \mu_t [(1 - \delta + f'(K_t)] = \beta^{t-1} \mu_{t-1}$$

or

$$\beta \mu_t [(1 - \delta + f'(K_t)] = \mu_{t-1}$$

which is (8.10).

Combining (9.11), (9.2), (9.3) and (9.4) after multiplying both sides of (9.11) by λ, we get

$$\sum_{t=0}^{T} \beta^t \mu_t \left(C_t + (K_{t+1} - (1-\delta)K_t) - f(K_t) + K_t f'(K_t) - f'(K_t) K_t \right) \leq 0$$

which simplifies to

$$\sum_{t=0}^{T} \beta^t \mu_t \left(C_t + K_{t+1} - (1-\delta) K_t - F(K_t, 1) \right) \leq 0$$

Since $\beta^t \mu_t > 0$ for $t = 0, \ldots, T$, it follows that

$$C_t + K_{t+1} - (1-\delta) K_t - F(K_t, 1) = 0 \quad \text{for all } t \text{ in } \{0, 1, \ldots, T\}$$

which is (8.11).

Combining (9.12) and (9.2), we get:

$$-\beta^{T+1} \mu_{T+1} \leq 0$$

Dividing both sides by β^{T+1} gives

$$-\mu_{T+1} \leq 0$$

which is (8.12) for the planning problem.

Thus, at our guess of the equilibrium price system, the allocation that solves the planning problem also solves the problem faced by a representative household living in a competitive equilibrium.

9.7.4 Representative Firm's Problem

We now turn to the problem faced by a firm in a competitive equilibrium:

If we plug (9.8) into (9.1) for all t, we get

$$\frac{\partial F(K_t, 1)}{\partial K_t} = f'(K_t) = \eta_t$$

which is (9.4).

If we now plug (9.8) into (9.1) for all t, we get:

$$\frac{\partial F(\tilde{K}_t, 1)}{\partial \tilde{L}_t} = f(K_t) - f'(K_t) K_t = w_t$$

which is exactly (9.5).

Thus, at our guess for the equilibrium price system, the allocation that solves the planning problem also solves the problem faced by a firm within a competitive equilibrium.

By (9.6) and (9.7) this allocation is identical to the one that solves the consumer's problem.

Note: Because budget sets are affected only by relative prices, $\{q_t^0\}$ is determined only up to multiplication by a positive constant.

Normalization: We are free to choose a $\{q_t^0\}$ that makes $\lambda = 1$ so that we are measuring q_t^0 in units of the marginal utility of time 0 goods.

We will plot q, w, η below to show these equilibrium prices induce the same aggregate movements that we saw earlier in the planning problem.

To proceed, we bring in Python code that *Cass-Koopmans Planning Model* used to solve the planning problem

First let's define a `jitclass` that stores parameters and functions the characterize an economy.

```
planning_data = [
    ('γ', float64),     # Coefficient of relative risk aversion
    ('β', float64),     # Discount factor
    ('δ', float64),     # Depreciation rate on capital
    ('α', float64),     # Return to capital per capita
    ('A', float64)      # Technology
]
```

```
@jitclass(planning_data)
class PlanningProblem():

    def __init__(self, γ=2, β=0.95, δ=0.02, α=0.33, A=1):

        self.γ, self.β = γ, β
        self.δ, self.α, self.A = δ, α, A

    def u(self, c):
        '''
        Utility function
        ASIDE: If you have a utility function that is hard to solve by hand
        you can use automatic or symbolic differentiation
        See https://github.com/HIPS/autograd
        '''
        γ = self.γ

        return c ** (1 - γ) / (1 - γ) if γ!= 1 else np.log(c)

    def u_prime(self, c):
        'Derivative of utility'
        γ = self.γ

        return c ** (-γ)

    def u_prime_inv(self, c):
        'Inverse of derivative of utility'
        γ = self.γ

        return c ** (-1 / γ)

    def f(self, k):
        'Production function'
        α, A = self.α, self.A

        return A * k ** α

    def f_prime(self, k):
        'Derivative of production function'
        α, A = self.α, self.A

        return α * A * k ** (α - 1)

    def f_prime_inv(self, k):
        'Inverse of derivative of production function'
        α, A = self.α, self.A

        return (k / (A * α)) ** (1 / (α - 1))
```

(continues on next page)

(continued from previous page)

```
    def next_k_c(self, k, c):
        '''
        Given the current capital Kt and an arbitrary feasible
        consumption choice Ct, computes Kt+1 by state transition law
        and optimal Ct+1 by Euler equation.
        '''
        β, δ = self.β, self.δ
        u_prime, u_prime_inv = self.u_prime, self.u_prime_inv
        f, f_prime = self.f, self.f_prime

        k_next = f(k) + (1 - δ) * k - c
        c_next = u_prime_inv(u_prime(c) / (β * (f_prime(k_next) + (1 - δ))))

        return k_next, c_next
```

```
@njit
def shooting(pp, c0, k0, T=10):
    '''
    Given the initial condition of capital k0 and an initial guess
    of consumption c0, computes the whole paths of c and k
    using the state transition law and Euler equation for T periods.
    '''
    if c0 > pp.f(k0):
        print("initial consumption is not feasible")

        return None

    # initialize vectors of c and k
    c_vec = np.empty(T+1)
    k_vec = np.empty(T+2)

    c_vec[0] = c0
    k_vec[0] = k0

    for t in range(T):
        k_vec[t+1], c_vec[t+1] = pp.next_k_c(k_vec[t], c_vec[t])

    k_vec[T+1] = pp.f(k_vec[T]) + (1 - pp.δ) * k_vec[T] - c_vec[T]

    return c_vec, k_vec
```

```
@njit
def bisection(pp, c0, k0, T=10, tol=1e-4, max_iter=500, k_ter=0, verbose=True):

    # initial boundaries for guess c0
    c0_upper = pp.f(k0)
    c0_lower = 0

    i = 0
    while True:
        c_vec, k_vec = shooting(pp, c0, k0, T)
        error = k_vec[-1] - k_ter

        # check if the terminal condition is satisfied
```

(continues on next page)

(continued from previous page)

```
        if np.abs(error) < tol:
            if verbose:
                print('Converged successfully on iteration ', i+1)
            return c_vec, k_vec

        i += 1
        if i == max_iter:
            if verbose:
                print('Convergence failed.')
            return c_vec, k_vec

        # if iteration continues, updates boundaries and guess of c0
        if error > 0:
            c0_lower = c0
        else:
            c0_upper = c0

        c0 = (c0_lower + c0_upper) / 2
```

```
pp = PlanningProblem()

# Steady states
ρ = 1 / pp.β - 1
k_ss = pp.f_prime_inv(ρ+pp.δ)
c_ss = pp.f(k_ss) - pp.δ * k_ss
```

The above code from this lecture *Cass-Koopmans Planning Model* lets us compute an optimal allocation for the planning problem.

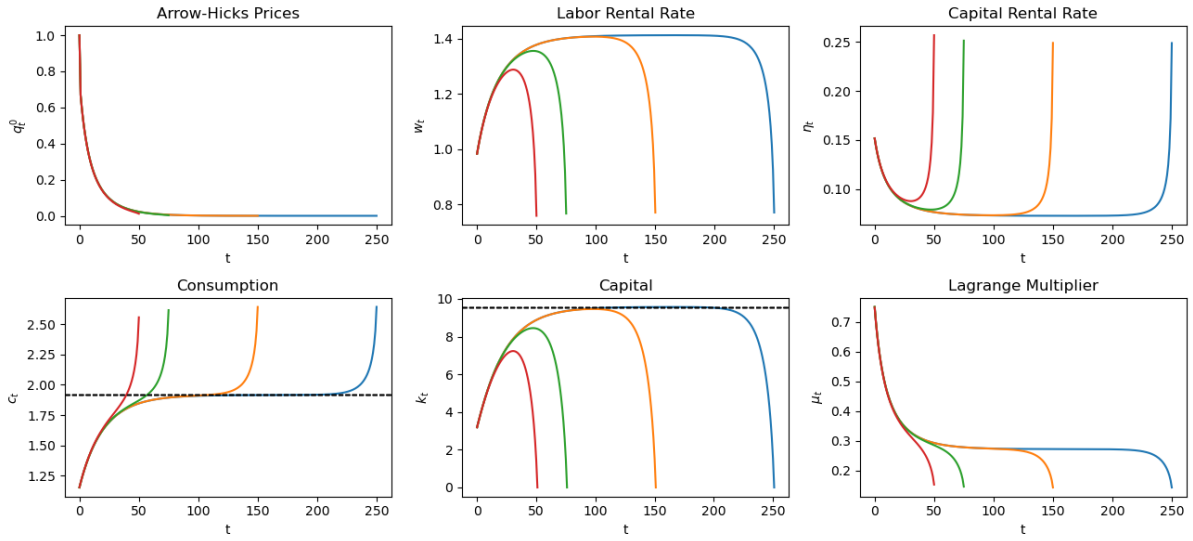
- from the preceding analysis, we know that it will also be an allocation associated with a competitive equilibium.

Now we're ready to bring in Python code that we require to compute additional objects that appear in a competitive equilibrium.

```
@njit
def q(pp, c_path):
    # Here we choose numeraire to be u'(c_0) -- this is q^(t_0)_t
    T = len(c_path) - 1
    q_path = np.ones(T+1)
    q_path[0] = 1
    for t in range(1, T+1):
        q_path[t] = pp.β ** t * pp.u_prime(c_path[t])
    return q_path

@njit
def w(pp, k_path):
    w_path = pp.f(k_path) - k_path * pp.f_prime(k_path)
    return w_path

@njit
def η(pp, k_path):
    η_path = pp.f_prime(k_path)
    return η_path
```

Now we calculate and plot for each T

```
T_arr = [250, 150, 75, 50]

fix, axs = plt.subplots(2, 3, figsize=(13, 6))
titles = ['Arrow-Hicks Prices', 'Labor Rental Rate', 'Capital Rental Rate',
          'Consumption', 'Capital', 'Lagrange Multiplier']
ylabels = ['$q_t^0$', '$w_t$', '$\eta_t$', '$c_t$', '$k_t$', '$\mu_t$']

for T in T_arr:
    c_path, k_path = bisection(pp, 0.3, k_ss/3, T, verbose=False)
    μ_path = pp.u_prime(c_path)

    q_path = q(pp, c_path)
    w_path = w(pp, k_path)[:-1]
    η_path = η(pp, k_path)[:-1]
    paths = [q_path, w_path, η_path, c_path, k_path, μ_path]

    for i, ax in enumerate(axs.flatten()):
        ax.plot(paths[i])
        ax.set(title=titles[i], ylabel=ylabels[i], xlabel='t')
        if titles[i] == 'Capital':
            ax.axhline(k_ss, lw=1, ls='--', c='k')
        if titles[i] == 'Consumption':
            ax.axhline(c_ss, lw=1, ls='--', c='k')

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

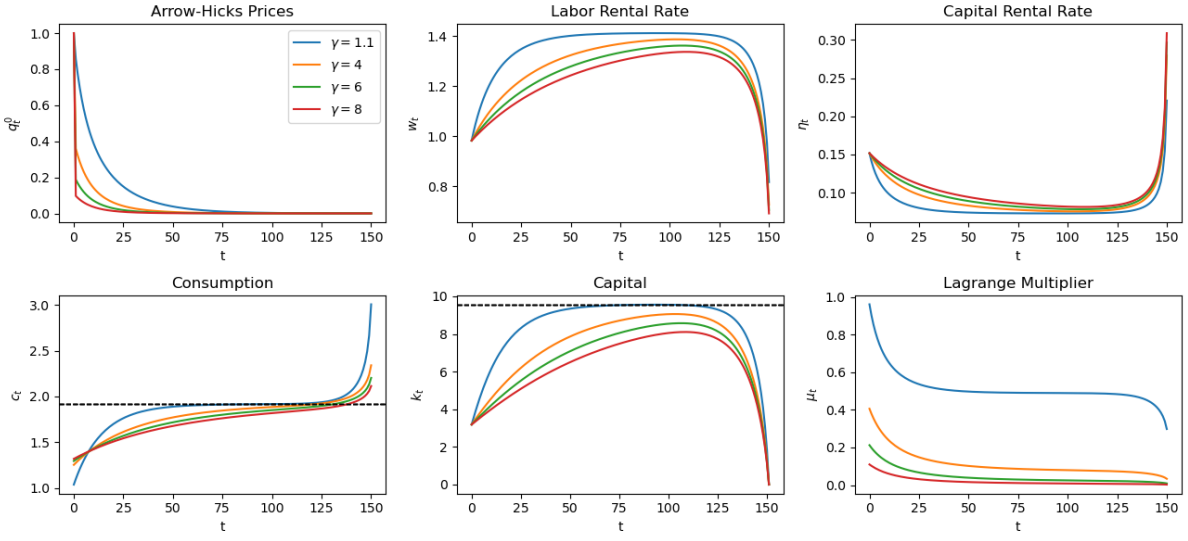


Varying Curvature

Now we see how our results change if we keep T constant, but allow the curvature parameter, γ to vary, starting with K_0 below the steady state.

We plot the results for $T = 150$

```
T = 150
γ_arr = [1.1, 4, 6, 8]

fix, axs = plt.subplots(2, 3, figsize=(13, 6))

for γ in γ_arr:
    pp_γ = PlanningProblem(γ=γ)
    c_path, k_path = bisection(pp_γ, 0.3, k_ss/3, T, verbose=False)
    μ_path = pp_γ.u_prime(c_path)

    q_path = q(pp_γ, c_path)
    w_path = w(pp_γ, k_path)[:-1]
    η_path = η(pp_γ, k_path)[:-1]
    paths = [q_path, w_path, η_path, c_path, k_path, μ_path]

    for i, ax in enumerate(axs.flatten()):
        ax.plot(paths[i], label=f'$\gamma = {γ}$')
        ax.set(title=titles[i], ylabel=ylabels[i], xlabel='t')
        if titles[i] == 'Capital':
            ax.axhline(k_ss, lw=1, ls='--', c='k')
        if titles[i] == 'Consumption':
            ax.axhline(c_ss, lw=1, ls='--', c='k')

axs[0, 0].legend()
plt.tight_layout()
plt.show()
```



Adjusting γ means adjusting how much individuals prefer to smooth consumption.

Higher γ means individuals prefer to smooth more resulting in slower convergence to a steady state allocation.

Lower γ means individuals prefer to smooth less, resulting in faster convergence to a steady state allocation.

9.8 Yield Curves and Hicks-Arrow Prices

We return to Hicks-Arrow prices and calculate how they are related to **yields** on loans of alternative maturities.

This will let us plot a **yield curve** that graphs yields on bonds of maturities $j = 1, 2, ...$ against $j = 1, 2, ...$

We use the following formulas.

A **yield to maturity** on a loan made at time t_0 that matures at time $t > t_0$

$$r_{t_0,t} = -\frac{\log q_t^{t_0}}{t - t_0}$$

A Hicks-Arrow price system for a base-year $t_0 \leq t$ satisfies

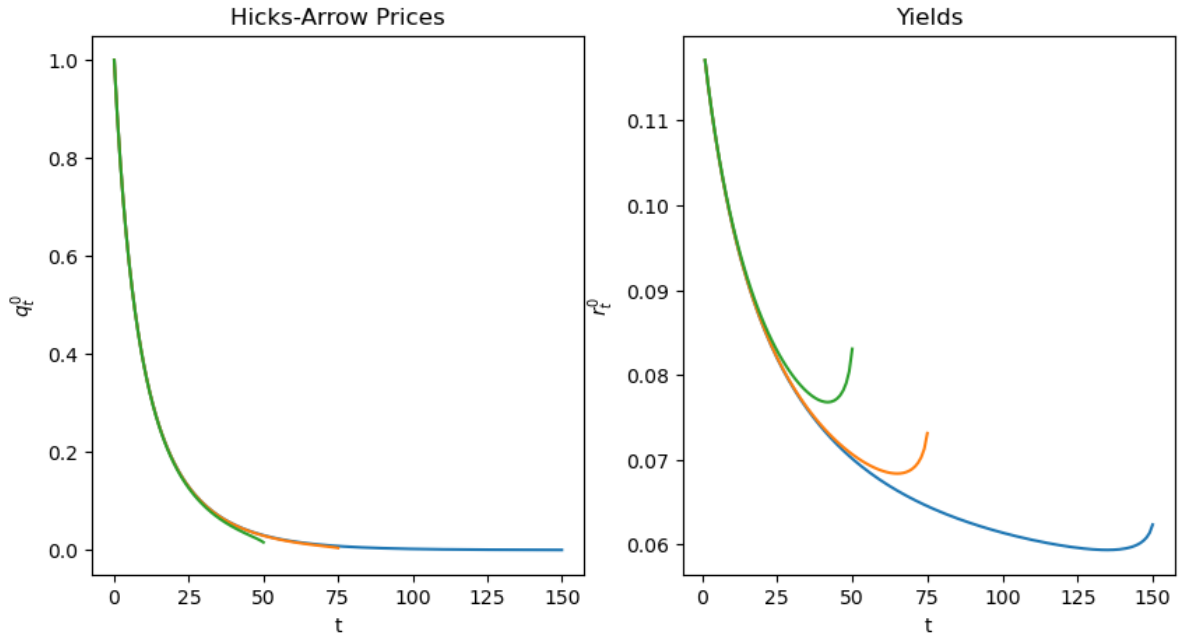
$$q_t^{t_0} = \beta^{t-t_0} \frac{u'(c_t)}{u'(c_{t_0})} = \beta^{t-t_0} \frac{c_t^{-\gamma}}{c_{t_0}^{-\gamma}}$$

We redefine our function for q to allow arbitrary base years, and define a new function for r, then plot both.

We begin by continuing to assume that $t_0 = 0$ and plot things for different maturities $t = T$, with K_0 below the steady state

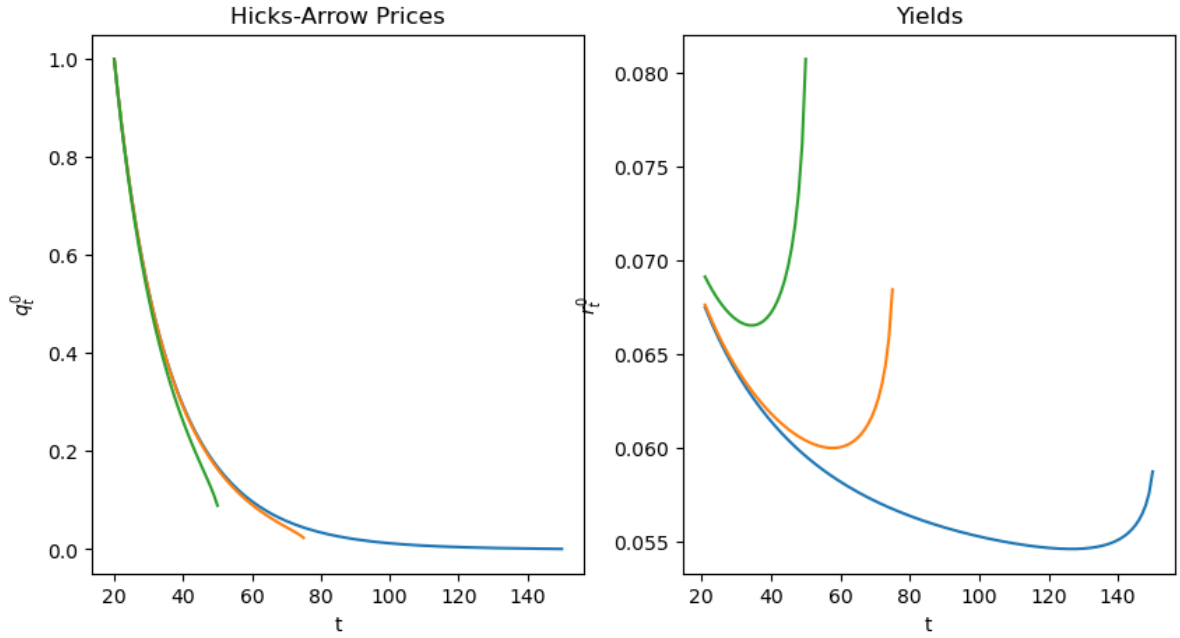
```
@njit
def q_generic(pp, t0, c_path):
    # simplify notations
    β = pp.β
    u_prime = pp.u_prime

    T = len(c_path) - 1
    q_path = np.zeros(T+1-t0)
    q_path[0] = 1
    for t in range(t0+1, T+1):
        q_path[t-t0] = β ** (t-t0) * u_prime(c_path[t]) / u_prime(c_path[t0])
    return q_path

@njit
def r(pp, t0, q_path):
    '''Yield to maturity'''
    r_path = - np.log(q_path[1:]) / np.arange(1, len(q_path))
    return r_path

def plot_yield_curves(pp, t0, c0, k0, T_arr):

    fig, axs = plt.subplots(1, 2, figsize=(10, 5))

    for T in T_arr:
        c_path, k_path = bisection(pp, c0, k0, T, verbose=False)
        q_path = q_generic(pp, t0, c_path)
        r_path = r(pp, t0, q_path)

        axs[0].plot(range(t0, T+1), q_path)
        axs[0].set(xlabel='t', ylabel='$q_t^0$', title='Hicks-Arrow Prices')

        axs[1].plot(range(t0+1, T+1), r_path)
        axs[1].set(xlabel='t', ylabel='$r_t^0$', title='Yields')
```

```
T_arr = [150, 75, 50]
plot_yield_curves(pp, 0, 0.3, k_ss/3, T_arr)
```



Now we plot when $t_0 = 20$

```
plot_yield_curves(pp, 20, 0.3, k_ss/3, T_arr)
```



CHAPTER

TEN

RATIONAL EXPECTATIONS EQUILIBRIUM

Contents

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

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

```
!pip install quantecon
```

10.1 Overview

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

To illustrate it, we describe a linear quadratic version of a model due to Lucas and Prescott [Lucas and Prescott, 1971].

That 1971 paper is one of a small number of research articles that ignited a *rational expectations revolution*.

We follow Lucas and Prescott by employing a setting that is readily "Bellmanized" (i.e., susceptible to being formulated as a dynamic programming problems.

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

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

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

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

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

Except that for us

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

Let's start with some standard imports:

```
import matplotlib.pyplot as plt
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
```

10.1.1 The Big Y, little y Trick

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

The following setting justifies the concept of a representative firm that stands in for a large number of other firms too.

There is a uniform unit measure of identical firms named $\omega \in \Omega = [0, 1]$.

The output of firm ω is $y(\omega)$.

The output of all firms is $Y = \int_0^1 y(\omega) d\,\omega$.

All firms end up choosing to produce the same output, so that at the end of the day $y(\omega) = y$ and $Y = y = \int_0^1 y(\omega) d\,\omega$.

This setting allows us to speak of a representative firm that chooses to produce y.

We want to impose that

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

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

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

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

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

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

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

Each of these firms ends up producing and selling output $y(\omega) = y$.

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

$$p = a_0 - a_1 Y \tag{10.1}$$

where

- $a_i > 0$ for $i = 0, 1$

- $Y = \int_0^1 y(\omega) d\omega$ is the market-wide level of output

For convenience, we'll often just write y instead of $y(\omega)$ when we are describing the choice problem of an individual firm $\omega \in \Omega$.

Each firm has a total cost function

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

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

Using (10.1), we can express the problem of the representative firm as

$$\max_y \left[(a_0 - a_1 Y) y - c_1 y - 0.5 c_2 y^2 \right] \tag{10.2}$$

In posing problem (10.2), 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 (10.2).

This assures that the firm is a price taker.

The first-order condition for problem (10.2) is

$$a_0 - a_1 Y - c_1 - c_2 y = 0 \tag{10.3}$$

At this point, *but not before*, we substitute $Y = y$ into (10.3) to obtain the following linear equation

$$a_0 - c_1 - (a_1 + c_2) Y = 0 \tag{10.4}$$

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 (10.1).

10.1.2 Related Planning Problem

Define **consumer surplus** as the area under the inverse demand curve:

$$S_c(Y) = \int_0^Y (a_0 - a_1 s) ds = a_o Y - \frac{a_1}{2} Y^2.$$

Define the social cost of production as

$$S_p(Y) = c_1 Y + \frac{c_2}{2} Y^2$$

Consider the planning problem

$$\max_Y [S_c(Y) - S_p(Y)]$$

The first-order necessary condition for the planning problem is equation (10.4).

Thus, a Y that solves (10.4) is a competitive equilibrium output as well as an output that solves the planning problem.

This type of outcome provides an intellectual justification for liking a competitive equilibrium.

10.1.3 Further Reading

References for this lecture include

- [Lucas and Prescott, 1971]
- [Sargent, 1987], chapter XIV
- [Ljungqvist and Sargent, 2018], chapter 7

10.2 Rational Expectations Equilibrium

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

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

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

Hence each firm wants to forecast future total industry output.

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

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

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

10.2.1 Competitive Equilibrium with Adjustment Costs

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

Each firm sell output $y_t(\omega) = y_t$.

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

$$p_t = a_0 - a_1 Y_t \tag{10.5}$$

where

- $a_i > 0$ for $i = 0, 1$
- $Y_t = \int_0^1 y_t(\omega) d\omega = y_t$ is the market-wide level of output

The Firm's Problem

Each firm is a price taker.

While it faces no uncertainty, it does face adjustment costs

In particular, it chooses a production plan to maximize

$$\sum_{t=0}^{\infty} \beta^t r_t \tag{10.6}$$

where

$$r_t := p_t y_t - \frac{\gamma(y_{t+1} - y_t)^2}{2}, \qquad y_0 \text{ given} \tag{10.7}$$

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 (10.5), the firm's incentive to forecast the market price translates into an incentive to forecast aggregate output Y_t.

Aggregate output depends on the choices of other firms.

The output $y_t(\omega)$ of a single firm ω has a negligible effect on aggregate output $\int_0^1 y_t(\omega) d\omega$.

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

Representative Firm's Beliefs

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

$$Y_{t+1} = H(Y_t) \tag{10.8}$$

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 (10.8) 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

$$v(y, Y) = \max_{y'} \left\{ a_0 y - a_1 y Y - \frac{\gamma(y' - y)^2}{2} + \beta v(y', H(Y)) \right\} \tag{10.9}$$

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

$$y_{t+1} = h(y_t, Y_t) \tag{10.10}$$

where

$$h(y, Y) := \text{argmax}_{y'} \left\{ a_0 y - a_1 y Y - \frac{\gamma(y' - y)^2}{2} + \beta v(y', H(Y)) \right\} \tag{10.11}$$

Evidently v and h both depend on H.

Characterization with First-Order Necessary Conditions

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

The first-order necessary condition for choosing y' is

$$-\gamma(y' - y) + \beta v_y(y', H(Y)) = 0 \tag{10.12}$$

An important useful envelope result of Benveniste-Scheinkman [Benveniste and Scheinkman, 1979] implies that to differentiate v with respect to y we can naively differentiate the right side of (10.9), giving

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

Substituting this equation into (10.12) gives the *Euler equation*

$$-\gamma(y_{t+1} - y_t) + \beta[a_0 - a_1 Y_{t+1} + \gamma(y_{t+2} - y_{t+1})] = 0 \tag{10.13}$$

The firm optimally sets an output path that satisfies (10.13), taking (10.8) as given, and subject to

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

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

A representative firm's decision rule solves the difference equation (10.13) subject to the given initial condition y_0 and the transversality condition.

Note that solving the Bellman equation (10.9) for v and then h in (10.11) yields a decision rule that automatically imposes both the Euler equation (10.13) and the transversality condition.

The Actual Law of Motion for Output

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

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

$$Y_{t+1} = h(Y_t, Y_t) \tag{10.14}$$

Thus, when firms believe that the law of motion for market-wide output is (10.8), their optimizing behavior makes the actual law of motion be (10.14).

10.2.2 Definition of Rational Expectations Equilibrium

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

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

Thus, a rational expectations equilibrium equates the perceived and actual laws of motion (10.8) and (10.14).

Fixed Point Characterization

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

The mapping Φ is the composition of two mappings, the first of which maps a perceived law of motion into a decision rule via (10.9)–(10.11), the second of which maps a decision rule into an actual law via (10.14).

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

10.3 Computing an Equilibrium

Now let's compute a rational expectations equilibrium.

10.3.1 Failure of Contractivity

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

Unfortunately, the mapping Φ is not a contraction.

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

There are examples in which these iterations diverge.

Fortunately, another method works here.

The method exploits a connection between equilibrium and Pareto optimality expressed in the fundamental theorems of welfare economics (see, e.g, [Mas-Colell *et al.*, 1995]).

Lucas and Prescott [Lucas and Prescott, 1971] used this method to construct a rational expectations equilibrium.

Some details follow.

10.3.2 A Planning Problem Approach

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

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

Optimal quantities from the planning problem are rational expectations equilibrium quantities.

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

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

$$s(Y_t, Y_{t+1}) := \int_0^{Y_t} (a_0 - a_1 x)\, dx - \frac{\gamma (Y_{t+1} - Y_t)^2}{2} \tag{10.15}$$

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

[1] A literature that studies whether models populated with agents who learn can converge to rational expectations equilibria features iterations on a modification of the mapping Φ that can be approximated as $\gamma\Phi + (1-\gamma)I$. Here I is the identity operator and $\gamma \in (0,1)$ is a *relaxation parameter*. See [Marcet and Sargent, 1989] and [Evans and Honkapohja, 2001] for statements and applications of this approach to establish conditions under which collections of adaptive agents who use least squares learning to converge to a rational expectations equilibrium.

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

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

subject to an initial condition for Y_0.

10.3.3 Solution of Planning Problem

Evaluating the integral in (10.15) 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

$$V(Y) = \max_{Y'} \left\{ a_0 Y - \frac{a_1}{2} Y^2 - \frac{\gamma (Y' - Y)^2}{2} + \beta V(Y') \right\} \tag{10.16}$$

The associated first-order condition is

$$-\gamma (Y' - Y) + \beta V'(Y') = 0 \tag{10.17}$$

Applying the same Benveniste-Scheinkman formula gives

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

Substituting this into equation (10.17) and rearranging leads to the Euler equation

$$\beta a_0 + \gamma Y_t - [\beta a_1 + \gamma (1 + \beta)] Y_{t+1} + \gamma \beta Y_{t+2} = 0 \tag{10.18}$$

10.3.4 Key Insight

Return to equation (10.13) and set $y_t = Y_t$ for all t.

A small amount of algebra will convince you that when $y_t = Y_t$, equations (10.18) and (10.13) are identical.

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

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

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

It follows that for this example we can compute equilibrium quantities by forming the optimal linear regulator problem corresponding to the Bellman equation (10.16).

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

Structure of the Law of Motion

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

$$Y_{t+1} = \kappa_0 + \kappa_1 Y_t \tag{10.19}$$

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 (10.9) 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

$$y_{t+1} = h_0 + h_1 y_t + h_2 Y_t \tag{10.20}$$

Hence a rational expectations equilibrium will be defined by the parameters $(\kappa_0, \kappa_1, h_0, h_1, h_2)$ in (10.19)–(10.20).

10.4 Exercises

Exercise 10.4.1

Consider the firm problem *described above*.

Let the firm's belief function H be as given in (10.19).

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 (10.20) and give the values for each h_j.

If there were a unit measure of identical competitive firms all behaving according to (10.20), what would (10.20) imply for the *actual* law of motion (10.8) for market supply.

Solution to Exercise 10.4.1

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 $u_t = -Fx_t$, or

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

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

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

Here's our solution

```
# Model parameters

a0 = 100
a1 = 0.05
β = 0.95
γ = 10.0

# Beliefs

κ0 = 95.5
κ1 = 0.95

# Formulate the LQ problem

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

# Solve for the optimal policy

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

print(out1)
print(out2)
```

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

The implication is that

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

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

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

Exercise 10.4.2

Consider the following κ_0, κ_1 pairs as candidates for the aggregate law of motion component of a rational expectations equilibrium (see (10.19)).

Extending the program that you wrote for Exercise 10.4.1, determine which if any satisfy *the definition* of a rational expectations equilibrium

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

Describe an iterative algorithm that uses the program that you wrote for Exercise 10.4.1 to compute a rational expectations equilibrium.

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

Solution to Exercise 10.4.2

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

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

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

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

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

The following code implements this test

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

for κ0, κ1 in candidates:

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

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

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

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

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

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

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

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

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

Exercise 10.4.3

Recall the planner's problem *described above*

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

Solution to Exercise 10.4.3

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
```

(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()

# 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.

Exercise 10.4.4

A monopolist faces the industry demand curve (10.5) and chooses $\{Y_t\}$ to maximize $\sum_{t=0}^{\infty} \beta^t r_t$ where

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

Formulate this problem as an LQ problem.

Compute the optimal policy using the same parameters as Exercise 10.4.2.

In particular, solve for the parameters in

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

Compare your results with Exercise 10.4.2 – comment.

Solution to Exercise 10.4.4

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

ELEVEN

STABILITY IN LINEAR RATIONAL EXPECTATIONS MODELS

Contents

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

```
!pip install quantecon
```

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
import quantecon as qe
from sympy import init_printing, symbols, Matrix
init_printing()
```

11.1 Overview

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

We study a rational expectations version of Philip Cagan's model [Cagan, 1956] linking the price level to the money supply.

Cagan did not use a rational expectations version of his model, but Sargent [Sargent, 1977] did.

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

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

In this lecture, we'll encounter:

- a convenient formula for the expectation of geometric sum of future values of a variable
- a way of solving an expectational difference equation by mapping it into a vector first-order difference equation and appropriately manipulating an eigen decomposition of the transition matrix in order to impose stability
- a way to use a Big K, little k argument to allow apparent feedback from endogenous to exogenous variables within a rational expectations equilibrium
- a use of eigenvector decompositions of matrices that allowed Blanchard and Khan (1981) [Blanchard and Kahn, 1980] and Whiteman (1983) [Whiteman, 1983] to solve a class of linear rational expectations models
- how to use **SymPy** to get analytical formulas for some key objects comprising a rational expectations equilibrium

Matrix decompositions employed here are described in more depth in this lecture Lagrangian formulations.

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

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

11.2 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.

11.2.1 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

$$(1 - \lambda L)y_t = u_t, \forall t \tag{11.1}$$

has solutions

$$y_t = (1 - \lambda L)^{-1}u_t + k\lambda^t \tag{11.2}$$

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 (11.2) 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 (11.1) as

$$y_{t-1} = \lambda^{-1}y_t - \lambda^{-1}u_t, \forall t \tag{11.3}$$

or

$$(1 - \lambda^{-1}L^{-1})y_t = -\lambda^{-1}u_{t+1}. \tag{11.4}$$

A solution is

$$y_t = -\lambda^{-1}\left(\frac{1}{1 - \lambda^{-1}L^{-1}}\right)u_{t+1} + k\lambda^t \tag{11.5}$$

for any k.

To verify that this is a solution, check the consequences of operating on both sides of equation (11.5) by $(1 - \lambda L)$ and compare to equation (11.1).

For any bounded $\{u_t\}$ sequence, solution (11.2) exists for $|\lambda| < 1$ because the **distributed lag** in u converges.

Solution (11.5) exists when $|\lambda| > 1$ because the **distributed lead** in u converges.

When $|\lambda| > 1$, the distributed lag in u in (11.2) may diverge, in which case a solution of this form does not exist.

The distributed lead in u in (11.5) need not converge when $|\lambda| < 1$.

11.2.2 Second Order

Now consider the second order difference equation

$$(1 - \lambda_1 L)(1 - \lambda_2 L)y_{t+1} = u_t \tag{11.6}$$

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 (11.6). Using insights from our analysis of the first-order equation, operate on both sides of (11.6) by the forward inverse of $(1 - \lambda_2 L)$ to rewrite equation (11.6) as

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

or

$$y_{t+1} = \lambda_1 y_t - \lambda_2^{-1} \sum_{j=0}^{\infty} \lambda_2^{-j} u_{t+j+1}. \tag{11.7}$$

Thus, we obtained equation (11.7) by solving a stable root (in this case λ_1) **backward**, and an unstable root (in this case λ_2) **forward**.

Equation (11.7) 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**

11.3 Illustration: Cagan's Model

Now let's use linear difference equations to represent and solve Sargent's [Sargent, 1977] rational expectations version of Cagan's model [Cagan, 1956] that connects the price level to the public's anticipations of future money supplies.

Cagan did not use a rational expectations version of his model, but Sargent [Sargent, 1977]

Let

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

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

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

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

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

$$p_t = (1 - \lambda) m_t + \lambda p_{t+1} \tag{11.8}$$

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 (11.8) forward gives

$$p_t = (1 - \lambda) \sum_{j=0}^{\infty} \lambda^j m_{t+j}, \tag{11.9}$$

which is the unique **stable** solution of difference equation (11.8) among a class of more general solutions

$$p_t = (1-\lambda)\sum_{j=0}^{\infty}\lambda^j m_{t+j} + c\lambda^{-t} \tag{11.10}$$

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

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

Equation (11.10) attributes **perfect foresight** about the money supply sequence to the holders of real balances.

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

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

$$\begin{aligned} m_t &= Gx_t \\ x_{t+1} &= Ax_t \end{aligned} \tag{11.11}$$

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 (11.11) 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 (11.8) that obeys the system comprised of (11.8)-(11.11).

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

This requires that we shut down the term $c\lambda^{-t}$ in equation (11.10) above by setting $c = 0$

The solution we are after is

$$p_t = Fx_t \tag{11.12}$$

where

$$F = (1-\lambda)G(I - \lambda A)^{-1} \tag{11.13}$$

Note: As mentioned above, an *explosive solution* of difference equation (11.8) can be constructed by adding to the right hand of (11.12) a sequence $c\lambda^{-t}$ where c is an arbitrary positive constant.

11.4 Some Python Code

We'll construct examples that illustrate (11.11).

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

$$m_{t+1} = \alpha + \rho_1 m_t + \rho_2 m_{t-1} \tag{11.14}$$

that is parameterized by ρ_1, ρ_2, α

To capture this parameterization with system (11.9) 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 (11.12) and (11.13).

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

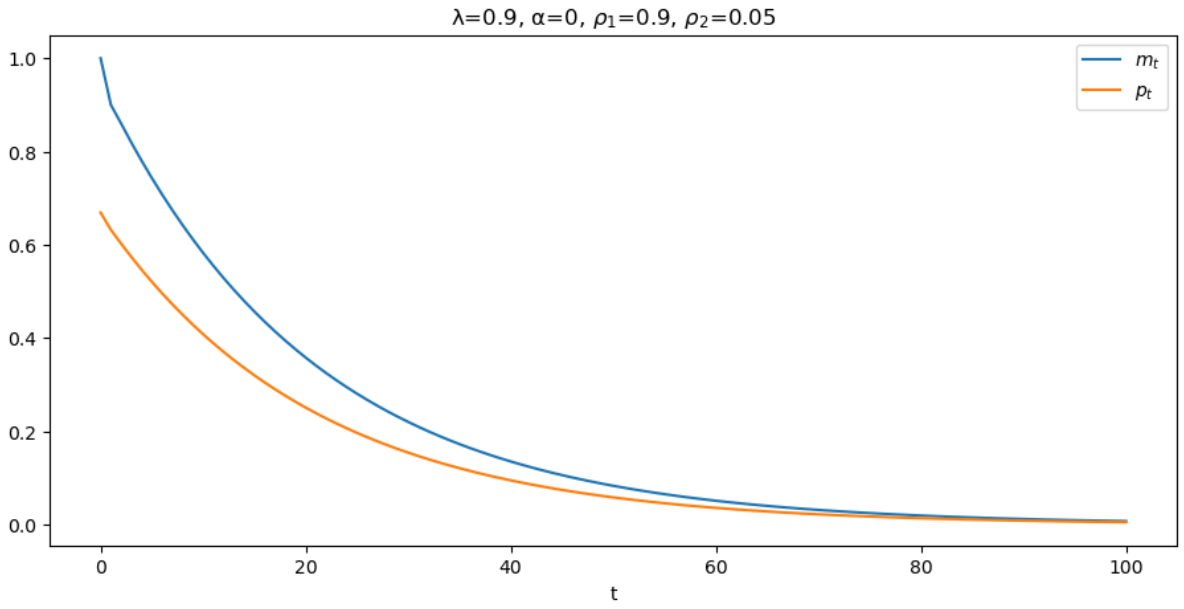
```
F=  [[0.         0.66889632 0.03010033]]
```

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

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

T = 100 # length of simulation

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

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

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

    x = A @ x_old

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

    x_old = x
```

```
/tmp/ipykernel_6712/4040936680.py:9: DeprecationWarning: Conversion of an array␣
 ↪with ndim > 0 to a scalar is deprecated, and will error in future. Ensure you␣
 ↪extract a single element from your array before performing this operation.␣
 ↪(Deprecated NumPy 1.25.)
  m_seq[0] = G @ x0
/tmp/ipykernel_6712/4040936680.py:10: DeprecationWarning: Conversion of an array␣
 ↪with ndim > 0 to a scalar is deprecated, and will error in future. Ensure you␣
 ↪extract a single element from your array before performing this operation.␣
 ↪(Deprecated NumPy 1.25.)
  p_seq[0] = F @ x0
/tmp/ipykernel_6712/4040936680.py:18: DeprecationWarning: Conversion of an array␣
 ↪with ndim > 0 to a scalar is deprecated, and will error in future. Ensure you␣
 ↪extract a single element from your array before performing this operation.␣
 ↪(Deprecated NumPy 1.25.)
  m_seq[t+1] = G @ x
/tmp/ipykernel_6712/4040936680.py:19: DeprecationWarning: Conversion of an array␣
 ↪with ndim > 0 to a scalar is deprecated, and will error in future. Ensure you␣
 ↪extract a single element from your array before performing this operation.␣
 ↪(Deprecated NumPy 1.25.)
  p_seq[t+1] = F @ x
```

```
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 (11.9), 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

11.5 Alternative Code

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

The following code block performs the calculation with that code.

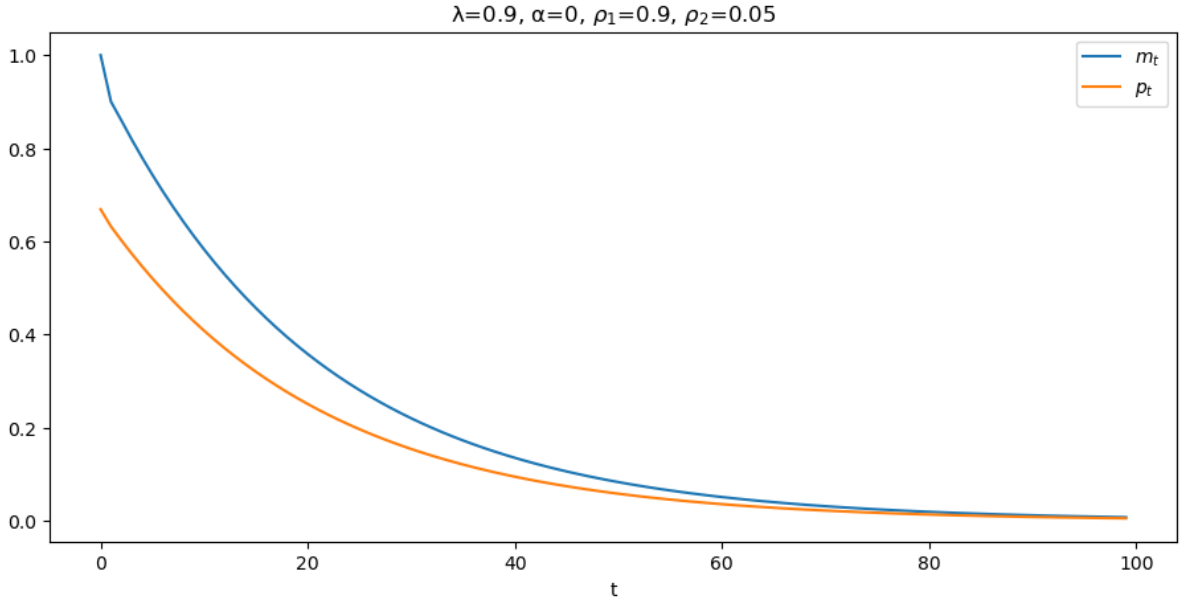
```
# construct a LinearStateSpace instance

# 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}')
```

(continues on next page)

(continued from previous page)

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



11.5.1 Special Case

To simplify our presentation in ways that will let focus on an important idea, in the above second-order difference equation (11.14) 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

$$m_{t+1} = \rho m_t \tag{11.15}$$

and the state x_t becomes

$$x_t = m_t.$$

Consequently, we can set $G = 1, A = \rho$ making our formula (11.13) 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.

11.6 Another Perspective

Above, we imposed stability or non-explosiveness on the solution of the key difference equation (11.8) 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 (11.8) and (11.15) to form the system

$$\begin{bmatrix} m_{t+1} \\ p_{t+1} \end{bmatrix} = \begin{bmatrix} \rho & 0 \\ -(1-\lambda)/\lambda & \lambda^{-1} \end{bmatrix} \begin{bmatrix} m_t \\ p_t \end{bmatrix} \tag{11.16}$$

or

$$y_{t+1} = Hy_t, \quad t \geq 0 \tag{11.17}$$

where

$$H = \begin{bmatrix} \rho & 0 \\ -(1-\lambda)/\lambda & \lambda^{-1} \end{bmatrix}. \tag{11.18}$$

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 (11.17) 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

$$y_0^* = \begin{bmatrix} y_{1,0}^* \\ 0 \end{bmatrix} \tag{11.19}$$

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 (11.19) 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 (11.19) 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

$$Q^{21} m_0 + Q^{22} p_0 = 0 \tag{11.20}$$

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

Solving this equation for p_0, we find

$$p_0 = -(Q^{22})^{-1} Q^{21} m_0. \tag{11.21}$$

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

11.6.1 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

$$p_0 = Q_{21} Q_{11}^{-1} m_0. \tag{11.22}$$

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

$$p_t = Q_{21} Q_{11}^{-1} m_t. \tag{11.23}$$

To implement formula (11.23), 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

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

11.6.2 Remarks about Feedback

We have expressed (11.16) 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 (11.16) 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.

11.7 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

$$m_{t+1} = \rho m_t + \delta p_t \tag{11.24}$$

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 (11.8) continues to describe equality between the demand and supply of money.

We assume that equations (11.8) and (11.24) govern $y_t \equiv \begin{bmatrix} m_t \\ p_t \end{bmatrix}$ for $t \geq 0$.

The transition matrix H in the law of motion

$$y_{t+1} = Hy_t$$

now becomes

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

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

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.

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

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

    return H

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

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

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

    return eigvals
```

```
H_eigvals()
```

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

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

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

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

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

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

A sufficiently small positive δ also causes no problem.

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

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

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

For example,

```
H_eigvals(δ=0.2)
```

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

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

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

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

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

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

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

        print("both eigenvalues exceed unity in modulus")

        return None

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

    return p0
```

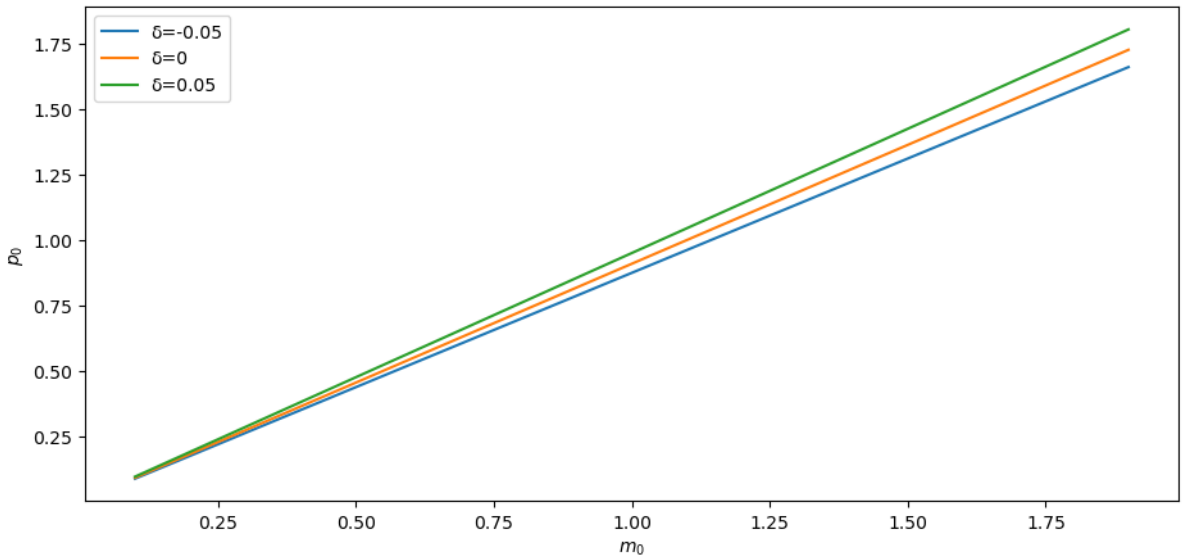
Let's plot how the solution p_0 changes as m_0 changes for different settings of δ.

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

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

(continues on next page)

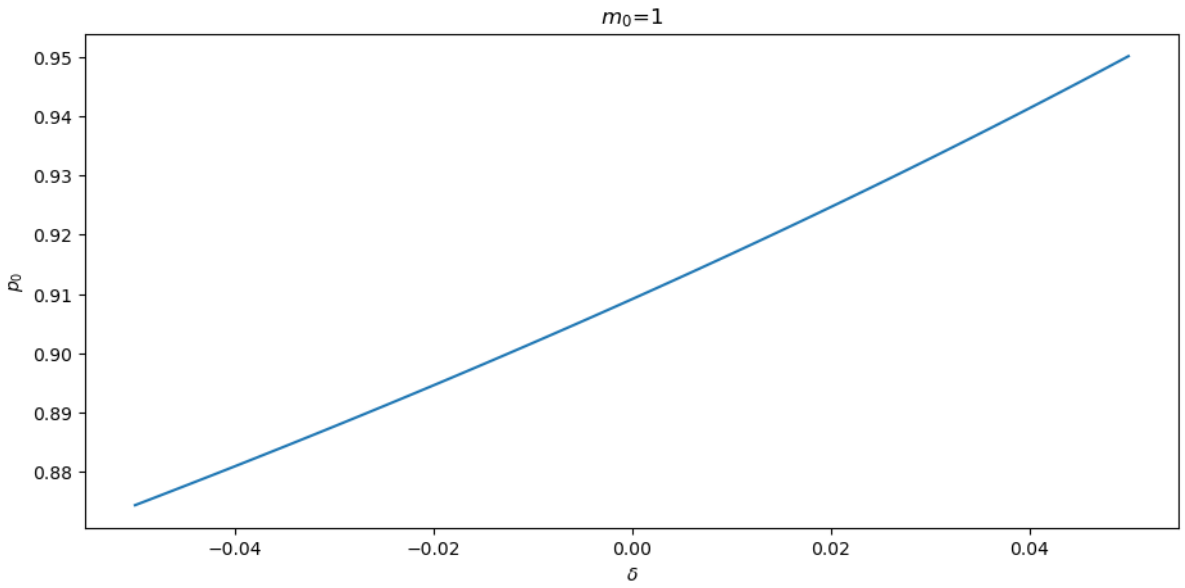
(continued from previous page)

```
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
```

11.8 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 (11.9) 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 (11.13) 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 (11.11), (11.12), and (11.13) 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 (11.13) 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
```

$$0.950124378879109$$

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

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

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

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

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

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

$$(0.95012437887911,\ 0.950124378879109)$$

11.9 Fun with SymPy

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

It puts SymPy to work on our model.

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

We start by generating an H with nonzero δ.

```
λ, δ, ρ = symbols('λ, δ, ρ')
```

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

```
H1
```

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

```
H1.eigenvals()
```

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

```
H1.eigenvects()
```

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

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()
```

$$\left\{\frac{1}{\lambda} : 1,\ \rho : 1\right\}$$

```
H2.eigenvects()
```

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

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

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

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

    vec.append(eigvec[0])

    if eigval == ρ:
        ind = i

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

```
Q
```

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

Q^{-1}

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

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

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

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

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

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

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

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

CHAPTER

TWELVE

MARKOV PERFECT EQUILIBRIUM

Contents

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

```
!pip install quantecon
```

12.1 Overview

This lecture describes the concept of Markov perfect equilibrium.

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

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

We will focus on settings with

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

Other references include chapter 7 of [Ljungqvist and Sargent, 2018].

Let's start with some standard imports:

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

12.2 Background

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

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

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

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

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

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

Well known examples include

- Choice of price, output, location or capacity for firms in an industry (e.g., [Ericson and Pakes, 1995], [Ryan, 2012], [Doraszelski and Satterthwaite, 2010]).
- Rate of extraction from a shared natural resource, such as a fishery (e.g., [Levhari and Mirman, 1980], [Van Long, 2011]).

Let's examine a model of the first type.

12.2.1 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

$$p = a_0 - a_1(q_1 + q_2) \tag{12.1}$$

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 (12.1) 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:

$$\pi_i = pq_i - \gamma(\hat{q}_i - q_i)^2, \quad \gamma > 0, \tag{12.2}$$

Substituting the inverse demand curve (12.1) into (12.2) lets us express the one-period payoff as

$$\pi_i(q_i, q_{-i}, \hat{q}_i) = a_0 q_i - a_1 q_i^2 - a_1 q_i q_{-i} - \gamma(\hat{q}_i - q_i)^2, \tag{12.3}$$

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

$$v_i(q_i, q_{-i}) = \max_{\hat{q}_i} \{\pi_i(q_i, q_{-i}, \hat{q}_i) + \beta v_i(\hat{q}_i, f_{-i}(q_{-i}, q_i))\} \tag{12.4}$$

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 (12.4).
- The maximizer on the right side of (12.4) 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.

12.2.2 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

$$v_i^{j+1}(q_i, q_{-i}) = \max_{\hat{q}_i} \left\{\pi_i(q_i, q_{-i}, \hat{q}_i) + \beta v_i^j(\hat{q}_i, f_{-i}(q_{-i}, q_i))\right\} \tag{12.5}$$

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.

12.3 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.

12.3.1 Coupled Linear Regulator Problems

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

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

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

$$\sum_{t=t_0}^{t_1-1} \beta^{t-t_0} \{x_t' R_i x_t + u_{it}' Q_i u_{it} + u_{-it}' S_i u_{-it} + 2x_t' W_i u_{it} + 2u_{-it}' M_i u_{it}\} \tag{12.6}$$

while the state evolves according to

$$x_{t+1} = Ax_t + B_1 u_{1t} + B_2 u_{2t} \tag{12.7}$$

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 $n \times k_i$

12.3.2 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 (12.6) and (12.7), then player 1's problem becomes minimization of

$$\sum_{t=t_0}^{t_1-1} \beta^{t-t_0} \left\{ x_t' \Pi_{1t} x_t + u_{1t}' Q_1 u_{1t} + 2 u_{1t}' \Gamma_{1t} x_t \right\} \tag{12.8}$$

subject to

$$x_{t+1} = \Lambda_{1t} x_t + B_1 u_{1t}, \tag{12.9}$$

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

$$F_{1t} = (Q_1 + \beta B_1' P_{1t+1} B_1)^{-1} (\beta B_1' P_{1t+1} \Lambda_{1t} + \Gamma_{1t}) \tag{12.10}$$

where P_{1t} solves the matrix Riccati difference equation

$$P_{1t} = \Pi_{1t} - (\beta B_1' P_{1t+1} \Lambda_{1t} + \Gamma_{1t})' (Q_1 + \beta B_1' P_{1t+1} B_1)^{-1} (\beta B_1' P_{1t+1} \Lambda_{1t} + \Gamma_{1t}) + \beta \Lambda_{1t}' P_{1t+1} \Lambda_{1t} \tag{12.11}$$

Similarly, decision rules that solve player 2's problem are

$$F_{2t} = (Q_2 + \beta B_2' P_{2t+1} B_2)^{-1} (\beta B_2' P_{2t+1} \Lambda_{2t} + \Gamma_{2t}) \tag{12.12}$$

where P_{2t} solves

$$P_{2t} = \Pi_{2t} - (\beta B_2' P_{2t+1} \Lambda_{2t} + \Gamma_{2t})'(Q_2 + \beta B_2' P_{2t+1} B_2)^{-1}(\beta B_2' P_{2t+1} \Lambda_{2t} + \Gamma_{2t}) + \beta \Lambda_{2t}' P_{2t+1} \Lambda_{2t} \tag{12.13}$$

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 (12.10), (12.11), (12.12), and (12.13), 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 (12.10) contain F_{2t}
- some terms on the right-hand side of (12.12) contain F_{1t}

we need to solve these $k_1 + k_2$ equations simultaneously.

Key Insight

A key insight is that equations (12.10) and (12.12) 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 (12.11) and (12.13).

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.

12.3.3 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.

12.4 Application

Let's use these procedures to treat some applications, starting with the duopoly model.

12.4.1 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 (12.3).

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:

$$x_{t+1} = (A - B_1 F_1 - B_1 F_2) x_t \tag{12.14}$$

12.4.2 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

```
import numpy as np
import quantecon as qe

# Parameters
a0 = 10.0
a1 = 2.0
β = 0.96
γ = 12.0

# In LQ form
A = np.eye(3)
B1 = np.array([[0.], [1.], [0.]])
B2 = np.array([[0.], [0.], [1.]])


R1 = [[      0.,      -a0 / 2,           0.],
      [-a0 / 2.,           a1,      a1 / 2.],
      [       0,      a1 / 2.,           0.]]

R2 = [[      0.,           0.,       -a0 / 2],
      [      0.,           0.,       a1 / 2.],
      [-a0 / 2,       a1 / 2.,            a1]]

Q1 = Q2 = γ
S1 = S2 = W1 = W2 = M1 = M2 = 0.0
```

(continues on next page)

(continued from previous page)

```
# Solve using QE's nnash function
F1, F2, P1, P2 = qe.nnash(A, B1, B2, R1, R2, Q1,
                          Q2, S1, S2, W1, W2, M1,
                          M2, beta=β)

# Display policies
print("Computed policies for firm 1 and firm 2:\n")
print(f"F1 = {F1}")
print(f"F2 = {F2}")
print("\n")
```

```
Computed policies for firm 1 and firm 2:

F1 = [[-0.66846615  0.29512482  0.07584666]]
F2 = [[-0.66846615  0.07584666  0.29512482]]
```

Running the code produces the following output.

One way to see that F_i is indeed optimal for firm i taking F_2 as given is to use QuantEcon.py's LQ class.

In particular, let's take F2 as computed above, plug it into (12.8) and (12.9) 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
```

12.4.3 Dynamics

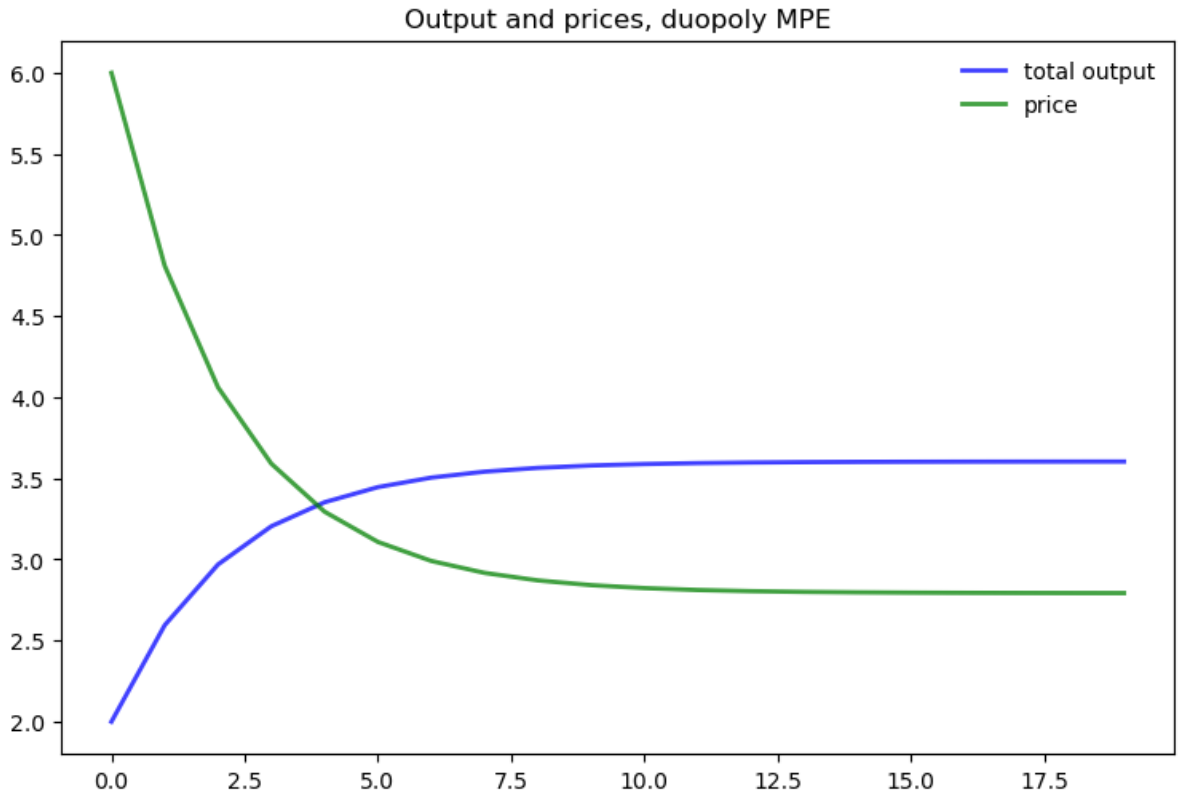
Let's now investigate the dynamics of price and output in this simple duopoly model under the MPE policies.

Given our optimal policies $F1$ and $F2$, the state evolves according to (12.14).

The following program

- imports $F1$ and $F2$ from the previous program along with all parameters.
- computes the evolution of x_t using (12.14).
- 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 $q_{10} = q_{20} = 1.0$.

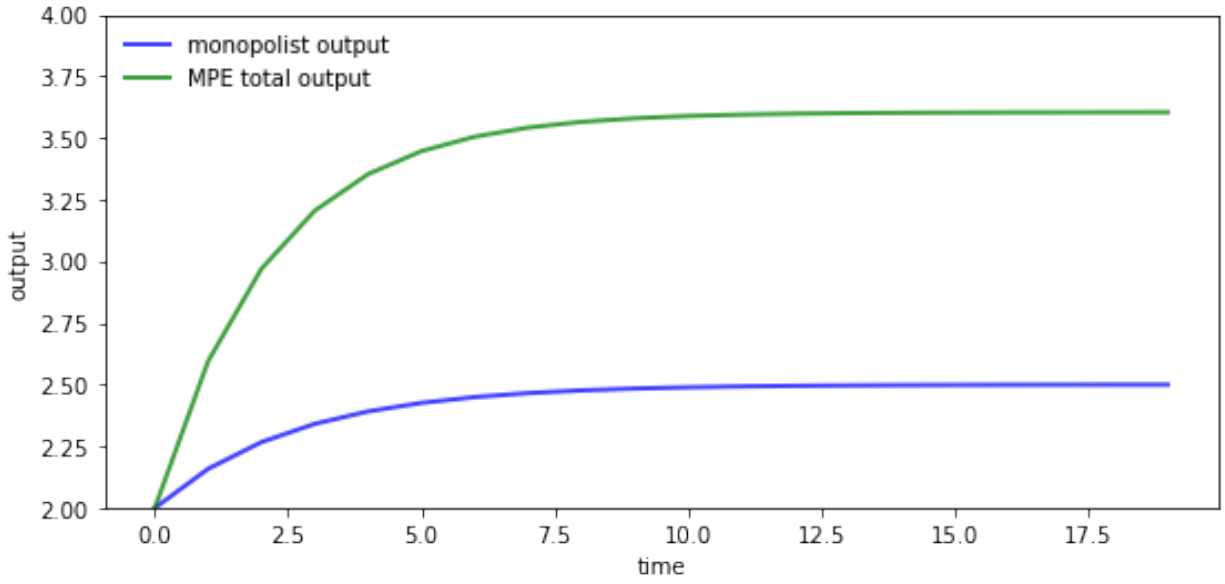
To gain some perspective we can compare this to what happens in the monopoly case.

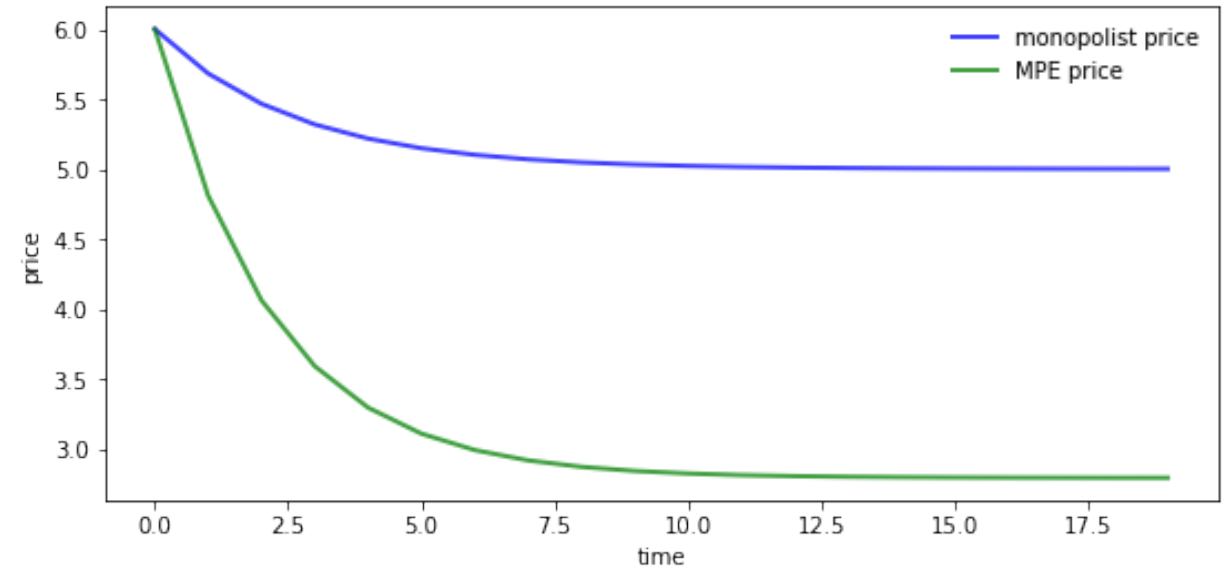
The first panel in the next figure compares output of the monopolist and industry output under the MPE, as a function of time.

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.


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

As expected, output is higher and prices are lower under duopoly than monopoly.

12.5 Exercises

Exercise 12.5.1

Replicate the *pair of figures* showing the comparison of output and prices for the monopolist and duopoly under MPE.

Parameters are as in duopoly_mpe.py and you can use that code to compute MPE policies under duopoly.

The optimal policy in the monopolist case can be computed using QuantEcon.py's LQ class.

Solution to Exercise 12.5.1

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.],
      [-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 $x_{t+1} = A x_t + B u_t$.

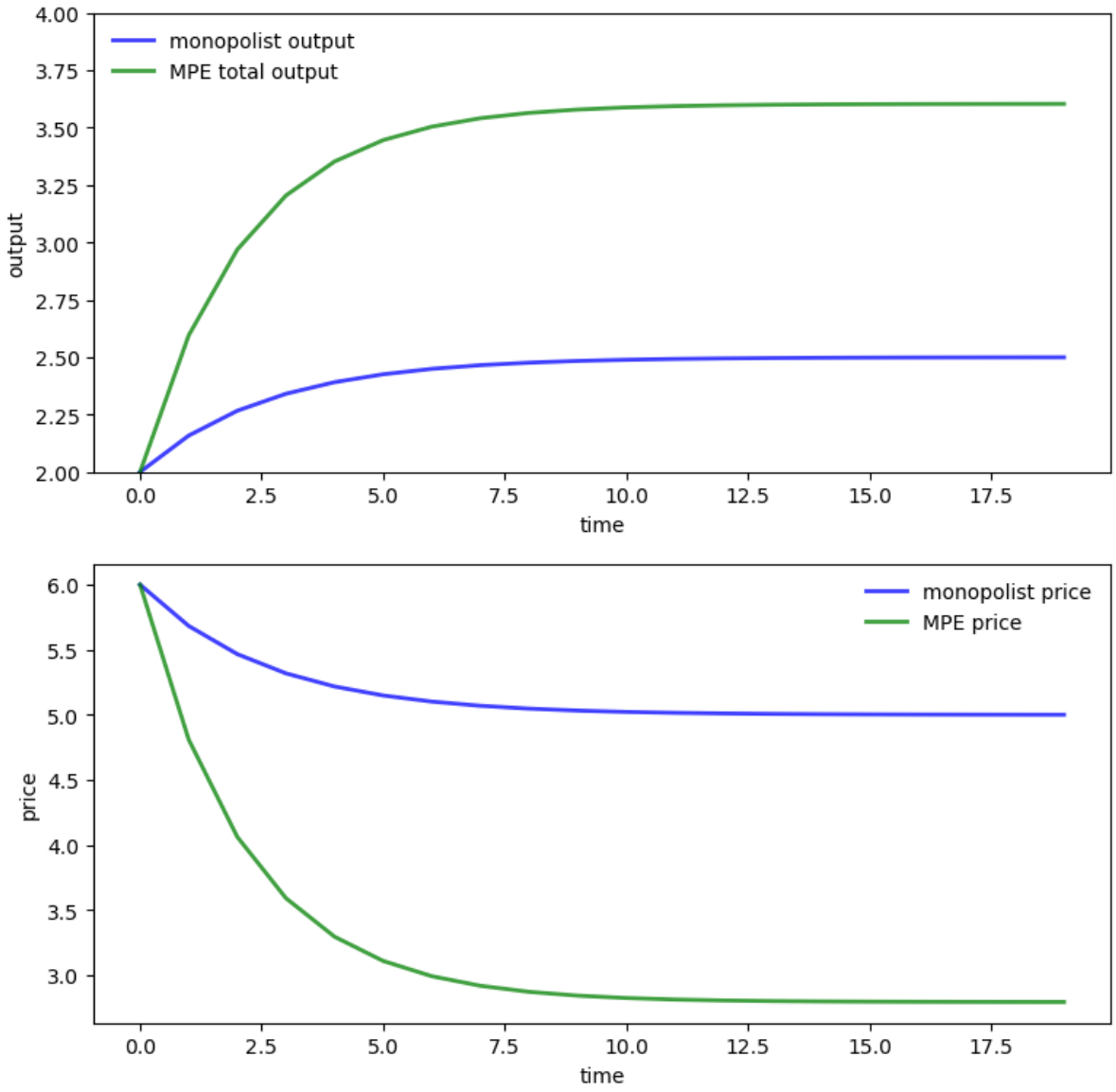
We solve for the optimal policy $u_t = -F x_t$ and track the resulting dynamics of $\{q_t\}$, starting at $q_0 = 2.0$.

```
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
```

```
/tmp/ipykernel_6324/1048199155.py:13: DeprecationWarning: Conversion of an array␣
↪with ndim > 0 to a scalar is deprecated, and will error in future. Ensure you␣
↪extract a single element from your array before performing this operation.␣
↪(Deprecated NumPy 1.25.)
  qm[i] = float(x) + q_bar
```

Let's have a look at the different time paths

```
fig, axes = plt.subplots(2, 1, figsize=(9, 9))

ax = axes[0]
ax.plot(qm, 'b-', lw=2, alpha=0.75, label='monopolist output')
ax.plot(q, 'g-', lw=2, alpha=0.75, label='MPE total output')
ax.set(ylabel="output", xlabel="time", ylim=(2, 4))
ax.legend(loc='upper left', frameon=0)

ax = axes[1]
ax.plot(pm, 'b-', lw=2, alpha=0.75, label='monopolist price')
ax.plot(p, 'g-', lw=2, alpha=0.75, label='MPE price')
ax.set(ylabel="price", xlabel="time")
ax.legend(loc='upper right', frameon=0)
plt.show()
```



Exercise 12.5.2

In this exercise, we consider a slightly more sophisticated duopoly problem.

It takes the form of infinite horizon linear-quadratic game proposed by Judd [Judd, 1990].

Two firms set prices and quantities of two goods interrelated through their demand curves.

Relevant variables are defined as follows:

- I_{it} = inventories of firm i at beginning of t
- q_{it} = production of firm i during period t
- p_{it} = price charged by firm i during period t
- S_{it} = sales made by firm i during period t
- E_{it} = costs of production of firm i during period t

- C_{it} = costs of carrying inventories for firm i during t

The firms' cost functions are

- $C_{it} = c_{i1} + c_{i2} I_{it} + 0.5 c_{i3} I_{it}^2$
- $E_{it} = e_{i1} + e_{i2} q_{it} + 0.5 e_{i3} q_{it}^2$ where e_{ij}, c_{ij} are positive scalars

Inventories obey the laws of motion

$$I_{i,t+1} = (1 - \delta) I_{it} + q_{it} - S_{it}$$

Demand is governed by the linear schedule

$$S_t = D p_{it} + b$$

where

- $S_t = \begin{bmatrix} S_{1t} & S_{2t} \end{bmatrix}'$
- D is a 2×2 negative definite matrix and
- b is a vector of constants

Firm i maximizes the undiscounted sum

$$\lim_{T \to \infty} \frac{1}{T} \sum_{t=0}^{T} (p_{it} S_{it} - E_{it} - C_{it})$$

We can convert this to a linear-quadratic problem by taking

$$u_{it} = \begin{bmatrix} p_{it} \\ q_{it} \end{bmatrix} \quad \text{and} \quad x_t = \begin{bmatrix} I_{1t} \\ I_{2t} \\ 1 \end{bmatrix}$$

Decision rules for price and quantity take the form $u_{it} = -F_i x_t$.

The Markov perfect equilibrium of Judd's model can be computed by filling in the matrices appropriately.

The exercise is to calculate these matrices and compute the following figures.

The first figure shows the dynamics of inventories for each firm when the parameters are

```
δ = 0.02
D = np.array([[-1, 0.5], [0.5, -1]])
b = np.array([25, 25])
c1 = c2 = np.array([1, -2, 1])
e1 = e2 = np.array([10, 10, 3])
```

Inventories trend to a common steady state.

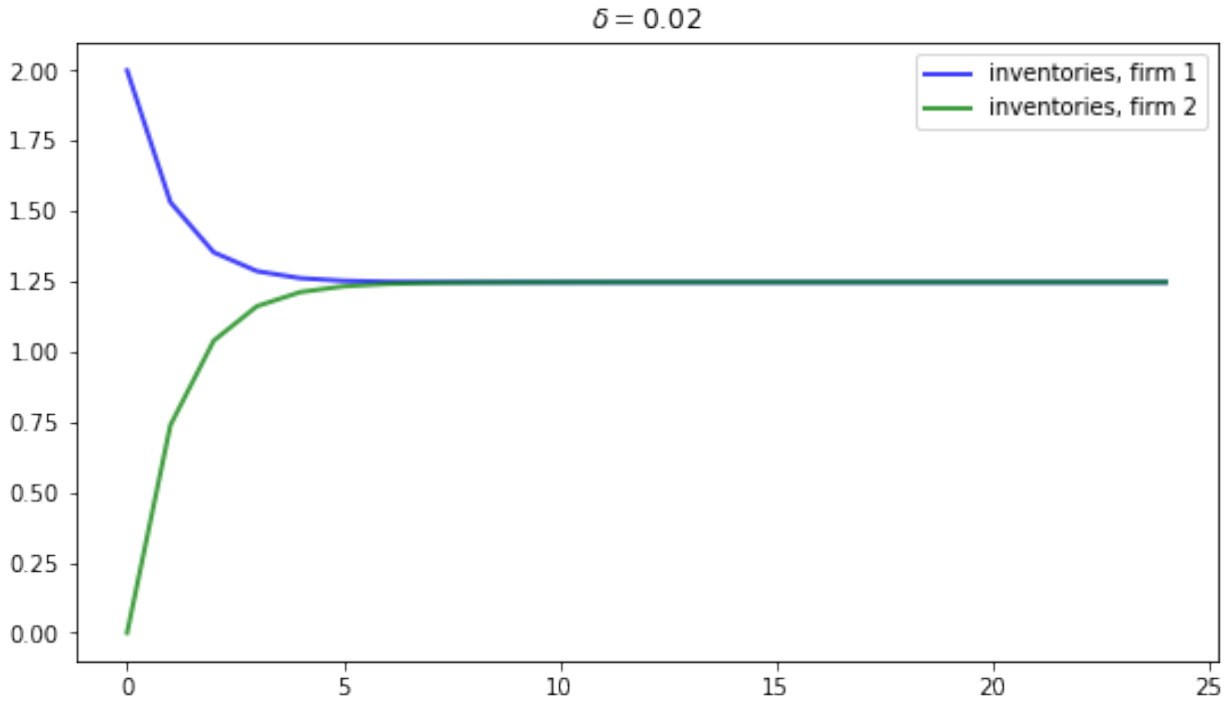
If we increase the depreciation rate to $\delta = 0.05$, then we expect steady state inventories to fall.

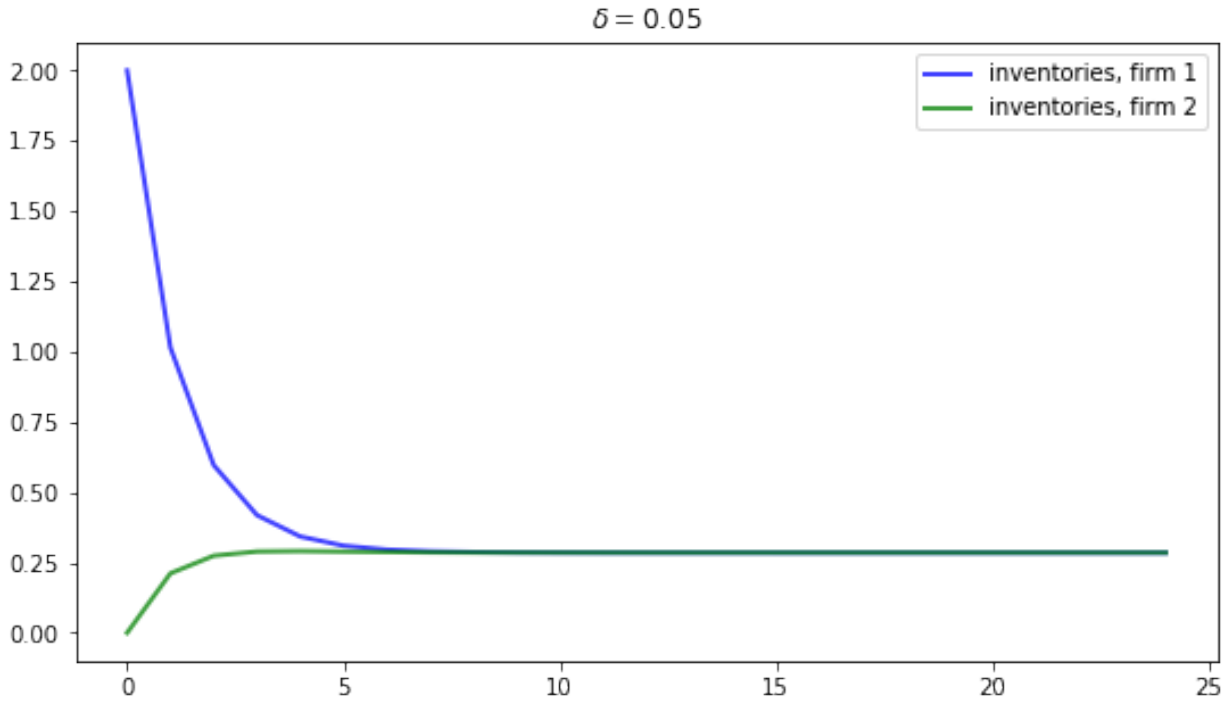
This is indeed the case, as the next figure shows

In this exercise, reproduce the figure when $\delta = 0.02$.

Solution to Exercise 12.5.2

We treat the case $\delta = 0.02$


$\delta = 0.02$
inventories, firm 1
inventories, firm 2
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0
5
10
15
20
25


$\delta = 0.05$
inventories, firm 1
inventories, firm 2
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0
5
10
15
20
25

```
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,
```

(continues on next page)

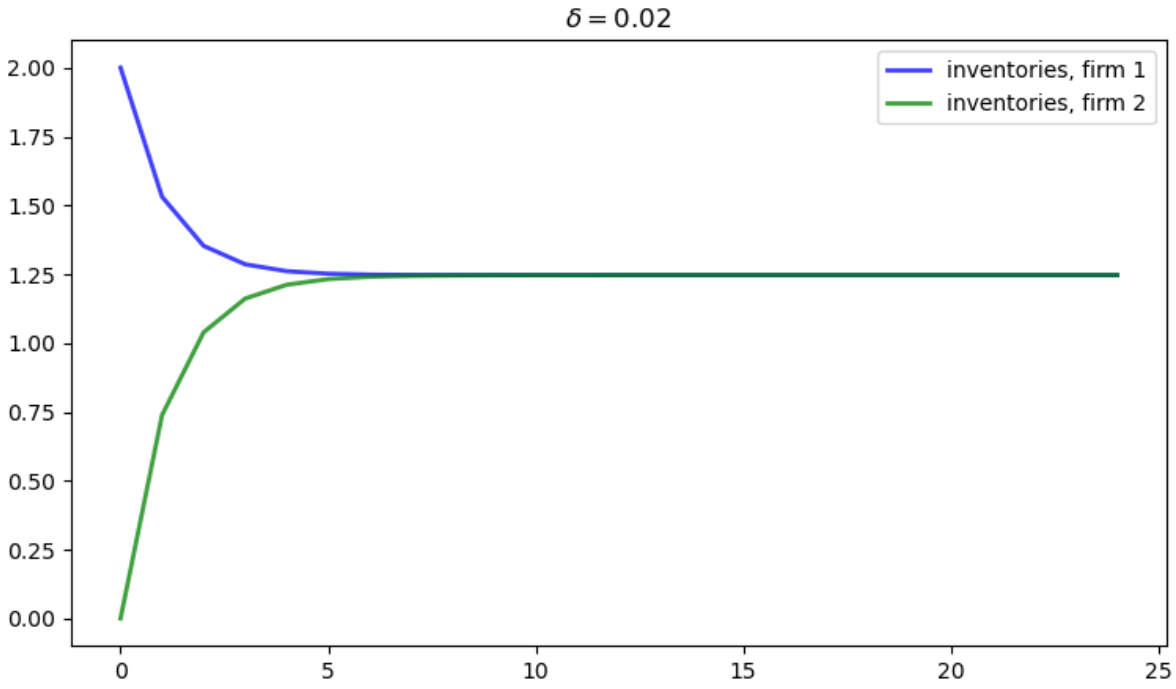
(continued from previous page)

```
                            S2, W1, W2, M1, M2)

print("\nFirm 1's feedback rule:\n")
print(F1)

print("\nFirm 2's feedback rule:\n")
print(F2)
```

```
Firm 1's feedback rule:

[[ 2.43666582e-01  2.72360627e-02 -6.82788293e+00]
 [ 3.92370734e-01  1.39696451e-01 -3.77341073e+01]]

Firm 2's feedback rule:

[[ 2.72360627e-02  2.43666582e-01 -6.82788293e+00]
 [ 1.39696451e-01  3.92370734e-01 -3.77341073e+01]]
```

Now let's look at the dynamics of inventories, and reproduce the graph corresponding to $\delta = 0.02$

```
AF = A - B1 @ F1 - B2 @ F2
n = 25
x = np.empty((3, n))
x[:, 0] = 2, 0, 1
for t in range(n-1):
    x[:, t+1] = AF @ x[:, t]
I1 = x[0, :]
I2 = x[1, :]
fig, ax = plt.subplots(figsize=(9, 5))
ax.plot(I1, 'b-', lw=2, alpha=0.75, label='inventories, firm 1')
ax.plot(I2, 'g-', lw=2, alpha=0.75, label='inventories, firm 2')
ax.set_title(rf'$\delta = {δ}$')
ax.legend()
plt.show()
```


$\delta = 0.02$
inventories, firm 1
inventories, firm 2
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0
5
10
15
20
25

CHAPTER

THIRTEEN

KNOWING THE FORECASTS OF OTHERS

Contents

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

```
!pip install --upgrade quantecon
!conda install -y -c plotly plotly plotly-orca
```

13.1 Introduction

Robert E. Lucas, Jr. [Robert E. Lucas, 1975], Kenneth Kasa [Kasa, 2000], and Robert Townsend [Townsend, 1983] showed that putting decision makers into environments in which they want to infer persistent hidden state variables from equilibrium prices and quantities can elongate and amplify impulse responses to aggregate shocks.

This provides a promising way to think about amplification mechanisms in business cycle models.

Townsend [Townsend, 1983] noted that living in such environments makes decision makers want to forecast forecasts of others.

This theme has been pursued for situations in which decision makers' imperfect information forces them to pursue an infinite recursion that involves forming beliefs about the beliefs of others (e.g., [Allen *et al.*, 2002]).

Lucas [Robert E. Lucas, 1975] side stepped having decision makers forecast the forecasts of other decision makers by assuming that they simply **pool their information** before forecasting.

A **pooling equilibrium** like Lucas's plays a prominent role in this lecture.

Because he didn't assume such pooling, [Townsend, 1983] confronted the forecasting the forecasts of others problem.

To formulate the problem recursively required that Townsend define a decision maker's **state** vector.

Townsend concluded that his original model required an intractable infinite dimensional state space.

Therefore, he constructed a more manageable approximating model in which a hidden Markov component of a demand shock is revealed to all firms after a fixed, finite number of periods.

In this lecture, we illustrate again the theme that **finding the state is an art** by showing how to formulate Townsend's original model in terms of a low-dimensional state space.

We show that Townsend's model shares equilibrium prices and quantities with those that prevail in a pooling equilibrium.

That finding emerged from a line of research about Townsend's model that built on [Pearlman *et al.*, 1986] and that culminated in [Pearlman and Sargent, 2005] .

Rather than directly deploying the [Pearlman *et al.*, 1986] machinery here, we shall instead implement a sneaky **guess-and-verify** tactic.

- We first compute a pooling equilibrium and represent it as an instance of a linear state-space system provided by the Python class `quantecon.LinearStateSpace`.
- Leaving the state-transition equation for the pooling equilibrium unaltered, we alter the observation vector for a firm to match what it is in Townsend's original model. So rather than directly observing the signal received by firms in the other industry, a firm sees the equilibrium price of the good produced by the other industry.
- We compute a population linear least squares regression of the noisy signal at time t that firms in the other industry would receive in a pooling equilibrium on time t information that a firm receives in Townsend's original model.
- The R^2 in this regression equals 1.
- That verifies that a firm's information set in Townsend's original model equals its information set in a pooling equilibrium.
- Therefore, equilibrium prices and quantities in Townsend's original model equal those in a pooling equilibrium.

13.1.1 A Sequence of Models

We proceed by describing a sequence of models of two industries that are linked in a single way:

- shocks to the demand curves for their products have a common component.

The models are simplified versions of Townsend's [Townsend, 1983].

Townsend's is a model of a rational expectations equilibrium in which firms want to **forecast forecasts of others**.

In Townsend's model, firms condition their forecasts on observed endogenous variables whose equilibrium laws of motion are determined by their own forecasting functions.

We shall assemble model components progressively in ways that can help us to appreciate the structure of the **pooling equilibrium** that ultimately interests us.

While keeping all other aspects of the model the same, we shall study consequences of alternative assumptions about what decision makers observe.

Technically, this lecture deploys concepts and tools that appear in First Look at Kalman Filter and Rational Expectations Equilibrium.

13.2 The Setting

We cast all variables in terms of deviations from means.

Therefore, we omit constants from inverse demand curves and other functions.

Firms in industry $i = 1, 2$ use a single factor of production, capital k_t^i, to produce output of a single good, y_t^i.

Firms bear quadratic costs of adjusting their capital stocks.

A representative firm in industry i has production function $y_t^i = fk_t^i$, $f > 0$.

The firm acts as a price taker with respect to output price P_t^i, and maximizes

$$E_0^i \sum_{t=0}^{\infty} \beta^t \left\{ P_t^i f k_t^i - .5h(k_{t+1}^i - k_t^i)^2 \right\}, \quad h > 0. \tag{13.1}$$

Demand in industry i is described by the inverse demand curve

$$P_t^i = -bY_t^i + \theta_t + \epsilon_t^i, \quad b > 0, \tag{13.2}$$

where P_t^i is the price of good i at t, $Y_t^i = fK_t^i$ is output in market i, θ_t is a persistent component of a demand shock that is common across the two industries, and ϵ_t^i is an industry specific component of the demand shock that is i.i.d. and whose time t marginal distribution is $\mathcal{N}(0, \sigma_\epsilon^2)$.

We assume that θ_t is governed by

$$\theta_{t+1} = \rho\theta_t + v_t \tag{13.3}$$

where $\{v_t\}$ is an i.i.d. sequence of Gaussian shocks, each with mean zero and variance σ_v^2.

To simplify notation, we'll study a special case by setting $h = f = 1$.

Costs of adjusting their capital stocks impart to firms an incentive to forecast the price of the good that they sell.

Throughout, we use the **rational expectations** equilibrium concept presented in this lecture Rational Expectations Equilibrium.

We let capital letters denote market wide objects and lower case letters denote objects chosen by a representative firm.

In each industry, a competitive equilibrium prevails.

To rationalize the big K, little k connection, we can think of there being a continuum of firms in industry i, with each firm being indexed by $\omega \in [0, 1]$ and $K^i = \int_0^1 k^i(\omega) d\omega$.

In equilibrium, $k_t^i = K_t^i$, but we must distinguish between k_t^i and K_t^i when we pose the firm's optimization problem.

13.3 Tactics

We shall compute equilibrium laws of motion for capital in industry i under a sequence of assumptions about what a representative firm observes.

Successive members of this sequence make a representative firm's information more and more obscure.

We begin with the most information, then gradually withdraw information in a way that approaches and eventually reaches the Townsend-like information structure that we are ultimately interested in.

Thus, we shall compute equilibria under the following alternative information structures:

- **Perfect foresight:** future values of θ_t, ϵ_t^i are observed in industry i.
- **Observed history of stochastic** θ_t: $\{\theta_t, \epsilon_t^i\}$ are realizations from a stochastic process; current and past values of each are observed at time t but future values are not.
- **One noise-ridden observation on** θ_t: values of $\{\theta_t, \epsilon_t^i\}$ separately are never observed. However, at time t, a history w^t of scalar noise-ridden observations on θ_t is observed at time t.
- **Two noise-ridden observations on** θ_t: values of $\{\theta_t, \epsilon_t^i\}$ separately are never observed. However, at time t, a history w^t of *two* noise-ridden observations on θ_t is observed at time t.

Successive computations build one on previous ones.

We proceed by first finding an equilibrium under perfect foresight.

To compute an equilibrium with current and past but not future values of θ_t observed, we use a *certainty equivalence principle* to justify modifying the perfect foresight equilibrium by replacing future values of $\theta_s, \epsilon_s^i, s \geq t$ with mathematical expectations conditioned on θ_t.

This provides the equilibrium when θ_t is observed at t but future θ_{t+j} and ϵ_{t+j}^i are not observed.

To find an equilibrium when a history w^t observations of a **single** noise-ridden θ_t is observed, we again apply a certainty equivalence principle and replace future values of the random variables $\theta_s, \epsilon_s^i, s \geq t$ with their mathematical expectations conditioned on w^t.

To find an equilibrium when a history w^t of **two** noisy signals on θ_t is observed, we replace future values of the random variables $\theta_s, \epsilon_s^i, s \geq t$ with their mathematical expectations conditioned on history w^t.

We call the equilibrium with two noise-ridden observations on θ_t a **pooling equilibrium**.

- It corresponds to an arrangement in which at the beginning of each period firms in industries 1 and 2 somehow get together and share information about current values of their noisy signals on θ.

We want ultimately to compare outcomes in a pooling equilibrium with an equilibrium under the following alternative information structure for a firm in industry i that originally interested Townsend [Townsend, 1983]:

- **Firm** i**'s noise-ridden signal on** θ_t **and the price in industry** $-i$, a firm in industry i observes a history w^t of *one* noise-ridden signal on θ_t and a history of industry $-i$'s price is observed. (Here $-i$ means ``not i".)

With this information structure, a representative firm i sees the price as well as the aggregate endogenous state variable Y_t^i in its own industry.

That allows it to infer the total demand shock $\theta_t + \epsilon_t^i$.

However, at time t, the firm sees only P_t^{-i} and does not see Y_t^{-i}, so that a firm in industry i does not **directly** observe $\theta_t + \epsilon_t^{-i}$.

Nevertheless, it will turn out that equilibrium prices and quantities in this equilibrium equal their counterparts in a pooling equilibrium because firms in industry i are able to infer the noisy signal about the demand shock received by firms in industry $-i$.

We shall verify this assertion by using a guess and verify tactic that involves running a least squares regression and inspecting its R^2.[1]

[1] [Pearlman and Sargent, 2005] verified this assertion using a different tactic, namely, by constructing analytic formulas for an equilibrium under the incomplete information structure and confirming that they match the pooling equilibrium formulas derived here.

13.4 Equilibrium Conditions

It is convenient to solve a firm's problem without uncertainty by forming the Lagrangian:

$$J = \sum_{t=0}^{\infty} \beta^t \left\{ P_t^i k_t^i - .5(\mu_t^i)^2 + \phi_t^i \left[k_t^i + \mu_t^i - k_{t+1}^i \right] \right\}$$

where $\{\phi_t^i\}$ is a sequence of Lagrange multipliers on the transition law $k_{t+1}^i = k_t^i + \mu_t^i$.

First order conditions for the nonstochastic problem are

$$\begin{aligned} \phi_t^i &= \beta \phi_{t+1}^i + \beta P_{t+1}^i \\ \mu_t^i &= \phi_t^i. \end{aligned} \tag{13.4}$$

Substituting the demand function (13.2) for P_t^i, imposing the condition $k_t^i = K_t^i$ that makes representative firm be representative, and using definition (13.6) of g_t^i, the Euler equation (13.4) lagged by one period can be expressed as $-bk_t^i + \theta_t + \epsilon_t^i + (k_{t+1}^i - k_t^i) - g_t^i = 0$ or

$$k_{t+1}^i = (b+1)k_t^i - \theta_t - \epsilon_t^i + g_t^i \tag{13.5}$$

where we define g_t^i by

$$g_t^i = \beta^{-1}(k_t^i - k_{t-1}^i) \tag{13.6}$$

We can write Euler equation (13.4) as:

$$g_t^i = P_t^i + \beta g_{t+1}^i \tag{13.7}$$

In addition, we have the law of motion for θ_t, (13.3), and the demand equation (13.2).

In summary, with perfect foresight, equilibrium conditions for industry i comprise the following system of difference equations:

$$\begin{aligned} k_{t+1}^i &= (1+b)k_t^i - \epsilon_t^i - \theta_t + g_t^i \\ \theta_{t+1} &= \rho \theta_t + v_t \\ g_{t+1}^i &= \beta^{-1}(g_t^i - P_t^i) \\ P_t^i &= -bk_t^i + \epsilon_t^i + \theta_t \end{aligned} \tag{13.8}$$

Without perfect foresight, the same system prevails except that the following equation replaces the third equation of (13.8):

$$g_{t+1,t}^i = \beta^{-1}(g_t^i - P_t^i)$$

where $x_{t+1,t}$ denotes the mathematical expectation of x_{t+1} conditional on information at time t.

13.4.1 Equilibrium under perfect foresight

Our first step is to compute the equilibrium law of motion for k_t^i under perfect foresight.

Let L be the lag operator.[2]

Equations (13.7) and (13.5) imply the second order difference equation in k_t^i:[3]

$$\left[(L^{-1} - (1+b))(1 - \beta L^{-1}) + b \right] k_t^i = \beta L^{-1} \epsilon_t^i + \beta L^{-1} \theta_t. \tag{13.9}$$

[2] See [Sargent, 1987], especially chapters IX and XIV, for principles that guide solving some roots backwards and others forwards.

[3] As noted by [Sargent, 1987], this difference equation is the Euler equation for a planning problem that maximizes the discounted sum of consumer plus producer surplus.

Factor the polynomial in L on the left side as:

$$-\beta[L^{-2} - (\beta^{-1} + (1+b))L^{-1} + \beta^{-1}] = \tilde{\lambda}^{-1}(L^{-1} - \tilde{\lambda})(1 - \tilde{\lambda}\beta L^{-1})$$

where $|\tilde{\lambda}| < 1$ is the smaller root and λ is the larger root of $(\lambda - 1)(\lambda - 1/\beta) = b\lambda$.

Therefore, (13.9) can be expressed as

$$\tilde{\lambda}^{-1}(L^{-1} - \tilde{\lambda})(1 - \tilde{\lambda}\beta L^{-1})k_t^i = \beta L^{-1}\epsilon_t^i + \beta L^{-1}\theta_t.$$

Solving the stable root backwards and the unstable root forwards gives

$$k_{t+1}^i = \tilde{\lambda}k_t^i + \frac{\tilde{\lambda}\beta}{1 - \tilde{\lambda}\beta L^{-1}}(\epsilon_{t+1}^i + \theta_{t+1}).$$

Recall that we have already set $k^i = K^i$ at the appropriate point in the argument, namely, *after* having derived the first-order necessary conditions for a representative firm in industry i.

Thus, under perfect foresight the equilibrium capital stock in industry i satisfies

$$k_{t+1}^i = \tilde{\lambda}k_t^i + \sum_{j=1}^{\infty}(\tilde{\lambda}\beta)^j(\epsilon_{t+j}^i + \theta_{t+j}). \tag{13.10}$$

Next, we shall investigate consequences of replacing future values of $(\epsilon_{t+j}^i + \theta_{t+j})$ in equation (13.10) with alternative forecasting schemes.

In particular, we shall compute equilibrium laws of motion for capital under alternative assumptions about information available to firms in market i.

13.5 Equilibrium with θ_t stochastic but observed at t

If future θ's are unknown at t, it is appropriate to replace all random variables on the right side of (13.10) with their conditional expectations based on the information available to decision makers in market i.

For now, we assume that this information set is $I_t^p = \begin{bmatrix}\theta^t & \epsilon^{it}\end{bmatrix}$, where z^t represents the semi-infinite history of variable z_s up to time t.

Later we shall give firms less information.

To obtain an appropriate counterpart to (13.10) under our current assumption about information, we apply a certainty equivalence principle.

In particular, it is appropriate to take (13.10) and replace each term $(\epsilon_{t+j}^i + \theta_{t+j})$ on the right side with $E[(\epsilon_{t+j}^i + \theta_{t+j})|\theta^t]$.

After using (13.3) and the i.i.d. assumption about $\{\epsilon_t^i\}$, this gives

$$k_{t+1}^i = \tilde{\lambda}k_t^i + \frac{\tilde{\lambda}\beta\rho}{1 - \tilde{\lambda}\beta\rho}\theta_t$$

or

$$k_{t+1}^i = \tilde{\lambda}k_t^i + \frac{\rho}{\lambda - \rho}\theta_t \tag{13.11}$$

where $\lambda \equiv (\beta\tilde{\lambda})^{-1}$.

For our purposes, it is convenient to represent the equilibrium $\{k_t^i\}_t$ process recursively as

$$\begin{aligned} k_{t+1}^i &= \tilde{\lambda}k_t^i + \frac{1}{\lambda - \rho}\hat{\theta}_{t+1} \\ \hat{\theta}_{t+1} &= \rho\theta_t \\ \theta_{t+1} &= \rho\theta_t + v_t. \end{aligned} \tag{13.12}$$

13.5.1 Filtering

One noisy signal

We get closer to the original Townsend model that interests us by now assuming that firms in market i do not observe θ_t.

Instead they observe a history w^t of noisy signals at time t.

In particular, assume that

$$\begin{aligned} w_t &= \theta_t + e_t \\ \theta_{t+1} &= \rho\theta_t + v_t \end{aligned} \tag{13.13}$$

where e_t and v_t are mutually independent i.i.d. Gaussian shock processes with means of zero and variances σ_e^2 and σ_v^2, respectively.

Define

$$\hat{\theta}_{t+1} = E(\theta_{t+1}|w^t)$$

where $w^t = [w_t, w_{t-1}, \dots, w_0]$ denotes the history of the w_s process up to and including t.

Associated with the state-space representation (13.13) is the time-invariant *innovations representation*

$$\begin{aligned} \hat{\theta}_{t+1} &= \rho\hat{\theta}_t + \kappa a_t \\ w_t &= \hat{\theta}_t + a_t \end{aligned} \tag{13.14}$$

where $a_t \equiv w_t - E(w_t|w^{t-1})$ is the *innovations* process in w_t and the Kalman gain κ is

$$\kappa = \frac{\rho p}{p + \sigma_e^2} \tag{13.15}$$

and where p satisfies the Riccati equation

$$p = \sigma_v^2 + \frac{p\rho^2\sigma_e^2}{\sigma_e^2 + p}. \tag{13.16}$$

State-reconstruction error

Define the state *reconstruction error* $\tilde{\theta}_t$ by

$$\tilde{\theta}_t = \theta_t - \hat{\theta}_t.$$

Then $p = E\tilde{\theta}_t^2$.

Equations (13.13) and (13.14) imply

$$\tilde{\theta}_{t+1} = (\rho - \kappa)\tilde{\theta}_t + v_t - ke_t. \tag{13.17}$$

Notice that we can express $\hat{\theta}_{t+1}$ as

$$\hat{\theta}_{t+1} = [\rho\theta_t + v_t] + [\kappa e_t - (\rho - \kappa)\tilde{\theta}_t - v_t], \tag{13.18}$$

where the first term in braces equals θ_{t+1} and the second term in braces equals $-\tilde{\theta}_{t+1}$.

We can express (13.11) as

$$k_{t+1}^i = \tilde{\lambda}k_t^i + \frac{1}{\lambda - \rho}E\theta_{t+1}|\theta^t. \tag{13.19}$$

An application of a certainty equivalence principle asserts that when only w^t is observed, a corresponding equilibrium $\{k_t^i\}$ process can be found by replacing the information set θ^t with w^t in (13.19).

Making this substitution and using (13.18) leads to

$$k_{t+1}^i = \tilde{\lambda} k_t^i + \frac{\rho}{\lambda - \rho}\theta_t + \frac{\kappa}{\lambda - \rho} e_t - \frac{\rho - \kappa}{\lambda - \rho}\tilde{\theta}_t. \tag{13.20}$$

Simplifying equation (13.18), we also have

$$\hat{\theta}_{t+1} = \rho\theta_t + \kappa e_t - (\rho - \kappa)\tilde{\theta}_t. \tag{13.21}$$

Equations (13.20), (13.21) describe an equilibrium when w^t is observed.

13.5.2 A new state variable

Relative to (13.11), the equilibrium acquires a **new state variable**, namely, the θ–reconstruction error, $\tilde{\theta}_t$.

For a subsequent argument, by using (13.15), it is convenient to write (13.20) as

$$k_{t+1}^i = \tilde{\lambda} k_t^i + \frac{\rho}{\lambda - \rho}\theta_t + \frac{1}{\lambda - \rho}\frac{p\rho}{p + \sigma_e^2} e_t - \frac{1}{\lambda - \rho}\frac{\rho\sigma_e^2}{p + \sigma_e^2}\tilde{\theta}_t \tag{13.22}$$

In summary, when decision makers in market i observe a semi-infinite history w^t of noisy signals w_t on θ_t at t, we an equilibrium law of motion for k_t^i can be represented as

$$\begin{aligned} k_{t+1}^i &= \tilde{\lambda} k_t^i + \frac{1}{\lambda - \rho}\hat{\theta}_{t+1} \\ \hat{\theta}_{t+1} &= \rho\theta_t + \frac{\rho p}{p + \sigma_e^2} e_t - \frac{\rho\sigma_e^2}{p + \sigma_e^2}\tilde{\theta}_t \\ \tilde{\theta}_{t+1} &= \frac{\rho\sigma_e^2}{p + \sigma_e^2}\tilde{\theta}_t - \frac{p\rho}{p + \sigma_e^2} e_t + v_t \\ \theta_{t+1} &= \rho\theta_t + v_t. \end{aligned} \tag{13.23}$$

13.5.3 Two Noisy Signals

We now construct a **pooling equilibrium** by assuming that at time t a firm in industry i receives a vector w_t of *two* noisy signals on θ_t:

$$\begin{aligned} \theta_{t+1} &= \rho\theta_t + v_t \\ w_t &= \begin{bmatrix} 1 \\ 1 \end{bmatrix} \theta_t + \begin{bmatrix} e_{1t} \\ e_{2t} \end{bmatrix} \end{aligned}$$

To justify that we are constructing is a **pooling equilibrium** we can assume that

$$\begin{bmatrix} e_{1t} \\ e_{2t} \end{bmatrix} = \begin{bmatrix} \epsilon_t^1 \\ \epsilon_t^2 \end{bmatrix}$$

so that a firm in industry i observes the noisy signals on that θ_t presented to firms in both industries i and $-i$.

The pertinent innovations representation now becomes

$$\begin{aligned} \hat{\theta}_{t+1} &= \rho\hat{\theta}_t + \kappa a_t \\ w_t &= \begin{bmatrix} 1 \\ 1 \end{bmatrix} \hat{\theta}_t + a_t \end{aligned} \tag{13.24}$$

where $a_t \equiv w_t - E[w_t|w^{t-1}]$ is a (2×1) vector of innovations in w_t and κ is now a (1×2) vector of Kalman gains.

Formulas for the Kalman filter imply that

$$\kappa = \frac{\rho p}{2p + \sigma_e^2} \begin{bmatrix} 1 & 1 \end{bmatrix} \tag{13.25}$$

where $p = E\tilde{\theta}_t \tilde{\theta}_t^T$ now satisfies the Riccati equation

$$p = \sigma_v^2 + \frac{p\rho^2 \sigma_e^2}{2p + \sigma_e^2}. \tag{13.26}$$

Thus, when a representative firm in industry i observes *two* noisy signals on θ_t, we can express the equilibrium law of motion for capital recursively as

$$\begin{aligned} k_{t+1}^i &= \tilde{\lambda} k_t^i + \frac{1}{\lambda - \rho} \hat{\theta}_{t+1} \\ \hat{\theta}_{t+1} &= \rho \theta_t + \frac{\rho p}{2p + \sigma_e^2}(e_{1t} + e_{2t}) - \frac{\rho \sigma_e^2}{2p + \sigma_e^2} \tilde{\theta}_t \\ \tilde{\theta}_{t+1} &= \frac{\rho \sigma_e^2}{2p + \sigma_e^2} \tilde{\theta}_t - \frac{p\rho}{2p + \sigma_e^2}(e_{1t} + e_{2t}) + v_t \\ \theta_{t+1} &= \rho \theta_t + v_t. \end{aligned} \tag{13.27}$$

Below, by using a guess-and-verify tactic, we shall show that outcomes in this **pooling equilibrium** equal those in an equilibrium under the alternative information structure that interested Townsend [Townsend, 1983] but that originally seemed too challenging to compute.[4]

13.6 Guess-and-Verify Tactic

As a preliminary step we shall take our recursive representation (13.23) of an equilibrium in industry i with one noisy signal on θ_t and perform the following steps:

- Compute λ and $\tilde{\lambda}$ by posing a root-finding problem and solving it with `numpy.roots`
- Compute p by forming the appropriate discrete Riccati equation and then solving it using `quantecon.solve_discrete_riccati`
- Add a *measurement equation* for $P_t^i = bk_t^i + \theta_t + e_t$, $\theta_t + e_t$, and e_t to system (13.23).
- Write the resulting system in state-space form and encode it using `quantecon.LinearStateSpace`
- Use methods of the `quantecon.LinearStateSpace` to compute impulse response functions of k_t^i with respect to shocks v_t, e_t.

After analyzing the one-noisy-signal structure in this way, by making appropriate modifications we shall analyze the two-noisy-signal structure.

We proceed to analyze first the one-noisy-signal structure and then the two-noisy-signal structure.

[4] [Pearlman and Sargent, 2005] verify the same claim by applying machinery of [Pearlman *et al.*, 1986].

13.7 Equilibrium with One Noisy Signal on θ_t

13.7.1 Step 1: Solve for $\tilde{\lambda}$ and λ

1. Cast $(\lambda - 1)\left(\lambda - \frac{1}{\beta}\right) = b\lambda$ as $p(\lambda) = 0$ where p is a polynomial function of λ.
2. Use `numpy.roots` to solve for the roots of p
3. Verify $\lambda \approx \frac{1}{\beta\tilde{\lambda}}$

Note that $p(\lambda) = \lambda^2 - \left(1 + b + \frac{1}{\beta}\right)\lambda + \frac{1}{\beta}$.

13.7.2 Step 2: Solve for p

1. Cast $p = \sigma_v^2 + \frac{p\rho^2\sigma_e^2}{2p+\sigma_e^2}$ as a discrete matrix Riccati equation.
2. Use `quantecon.solve_discrete_riccati` to solve for p
3. Verify $p \approx \sigma_v^2 + \frac{p\rho^2\sigma_e^2}{2p+\sigma_e^2}$

Note that:

$$
\begin{aligned}
A &= \begin{bmatrix} \rho \end{bmatrix} \\
B &= \begin{bmatrix} \sqrt{2} \end{bmatrix} \\
R &= \begin{bmatrix} \sigma_e^2 \end{bmatrix} \\
Q &= \begin{bmatrix} \sigma_v^2 \end{bmatrix} \\
N &= \begin{bmatrix} 0 \end{bmatrix}
\end{aligned}
$$

13.7.3 Step 3: Represent the system using `quantecon.LinearStateSpace`

We use the following representation for constructing the `quantecon.LinearStateSpace` instance.

$$
\underbrace{\begin{bmatrix} e_{t+1} \\ k^i_{t+1} \\ \tilde{\theta}_{t+1} \\ P_{t+1} \\ \theta_{t+1} \\ v_{t+1} \end{bmatrix}}_{x_{t+1}} = \underbrace{\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{\kappa}{\lambda-\rho} & \tilde{\lambda} & \frac{-1}{\lambda-\rho}\frac{\kappa\sigma_e^2}{p} & 0 & \frac{\rho}{\lambda-\rho} & 0 \\ -\kappa & 0 & \frac{\kappa\sigma_e^2}{p} & 0 & 0 & 1 \\ \frac{b\kappa}{\lambda-\rho} & b\tilde{\lambda} & \frac{-b}{\lambda-\rho}\frac{\kappa\sigma_e^2}{p} & 0 & \frac{b\rho}{\lambda-\rho}+\rho & 1 \\ 0 & 0 & 0 & 0 & \rho & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} e_t \\ k^i_t \\ \tilde{\theta}_t \\ P_t \\ \theta_t \\ v_t \end{bmatrix}}_{x_t} + \underbrace{\begin{bmatrix} \sigma_e & 0 \\ 0 & 0 \\ 0 & 0 \\ \sigma_e & 0 \\ 0 & 0 \\ 0 & \sigma_v \end{bmatrix}}_{C} \begin{bmatrix} z_{1,t+1} \\ z_{2,t+1} \end{bmatrix}
$$

$$
\underbrace{\begin{bmatrix} P_t \\ e_t + \theta_t \\ e_t \end{bmatrix}}_{y_t} = \underbrace{\begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}}_{G} \underbrace{\begin{bmatrix} e_t \\ k^i_t \\ \tilde{\theta}_t \\ P_t \\ \theta_t \\ v_t \end{bmatrix}}_{x_t} + \underbrace{\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}}_{H} w_{t+1}
$$

$$
\begin{bmatrix} z_{1,t+1} \\ z_{2,t+1} \\ w_{t+1} \end{bmatrix} \sim \mathcal{N}(0, I)
$$

$$
\kappa = \frac{\rho p}{p + \sigma_e^2}
$$

This representation includes extraneous variables such as P_t in the state vector.

We formulate things in this way because it allows us easily to compute covariances of these variables with other components of the state vector (step 5 above) by using the `stationary_distributions` method of the `LinearStateSpace` class.

```
import numpy as np
import quantecon as qe
import plotly.graph_objects as go
import plotly.offline as pyo
from statsmodels.regression.linear_model import OLS
from IPython.display import display, Latex, Image

pyo.init_notebook_mode(connected=True)
```

```
β = 0.9  # Discount factor
ρ = 0.8  # Persistence parameter for the hidden state
b = 1.5  # Demand curve parameter
σ_v = 0.5  # Standard deviation of shock to ϑ_t
σ_e = 0.6  # Standard deviation of shocks to w_t
```

```
# Compute λ
poly = np.array([1, -(1 + β + b) / β, 1 / β])
roots_poly = np.roots(poly)
λ_tilde = roots_poly.min()
λ = roots_poly.max()
```

```
# Verify that λ = (βλ_tilde) ^ (-1)
tol = 1e-12
np.max(np.abs(λ - 1 / (β * λ_tilde))) < tol
```

```
True
```

```
A_ricc = np.array([[ρ]])
B_ricc = np.array([[1.]])
R_ricc = np.array([[σ_e ** 2]])
Q_ricc = np.array([[σ_v ** 2]])
N_ricc = np.zeros((1, 1))
p = qe.solve_discrete_riccati(A_ricc, B_ricc, Q_ricc, R_ricc, N_ricc).item()

p_one = p  # Save for comparison later
```

```
# Verify that p = σ_v ^ 2 + p * ρ ^ 2 - (ρ * p) ^ 2 / (p + σ_e ** 2)
tol = 1e-12
np.abs(p - (σ_v ** 2 + p * ρ ** 2 - (ρ * p) ** 2 / (p + σ_e ** 2))) < tol
```

```
True
```

```
κ = ρ * p / (p + σ_e ** 2)
κ_prod = κ * σ_e ** 2 / p
```

(continues on next page)

(continued from previous page)

```
κ_one = κ  # Save for comparison later

A_lss = np.array([[0., 0., 0., 0., 0., 0.],
                  [κ / (λ - ρ), λ_tilde, -κ_prod / (λ - ρ), 0., ρ / (λ - ρ), 0.],
                  [-κ, 0., κ_prod, 0., 0., 1.],
                  [b * κ / (λ - ρ) , b * λ_tilde, -b * κ_prod / (λ - ρ), 0., b * ρ /␣
 ↪(λ - ρ) + ρ, 1.],
                  [0., 0., 0., 0., ρ, 1.],
                  [0., 0., 0., 0., 0., 0.]])

C_lss = np.array([[σ_e, 0.],
                  [0.,  0.],
                  [0.,  0.],
                  [σ_e, 0.],
                  [0., 0.],
                  [0., σ_v]])

G_lss = np.array([[0., 0., 0., 1., 0., 0.],
                  [1., 0., 0., 0., 1., 0.],
                  [1., 0., 0., 0., 0., 0.]])
```

```
mu_0 = np.array([0., 0., 0., 0., 0., 0.])

lss = qe.LinearStateSpace(A_lss, C_lss, G_lss, mu_0=mu_0)
```

```
ts_length = 100_000
x, y = lss.simulate(ts_length, random_state=1)
```

```
# Verify that two ways of computing P_t match
np.max(np.abs(np.array([[1., b, 0., 0., 1., 0.]]) @ x - x[3])) < 1e-12
```

```
True
```

13.7.4 Step 4: Compute impulse response functions

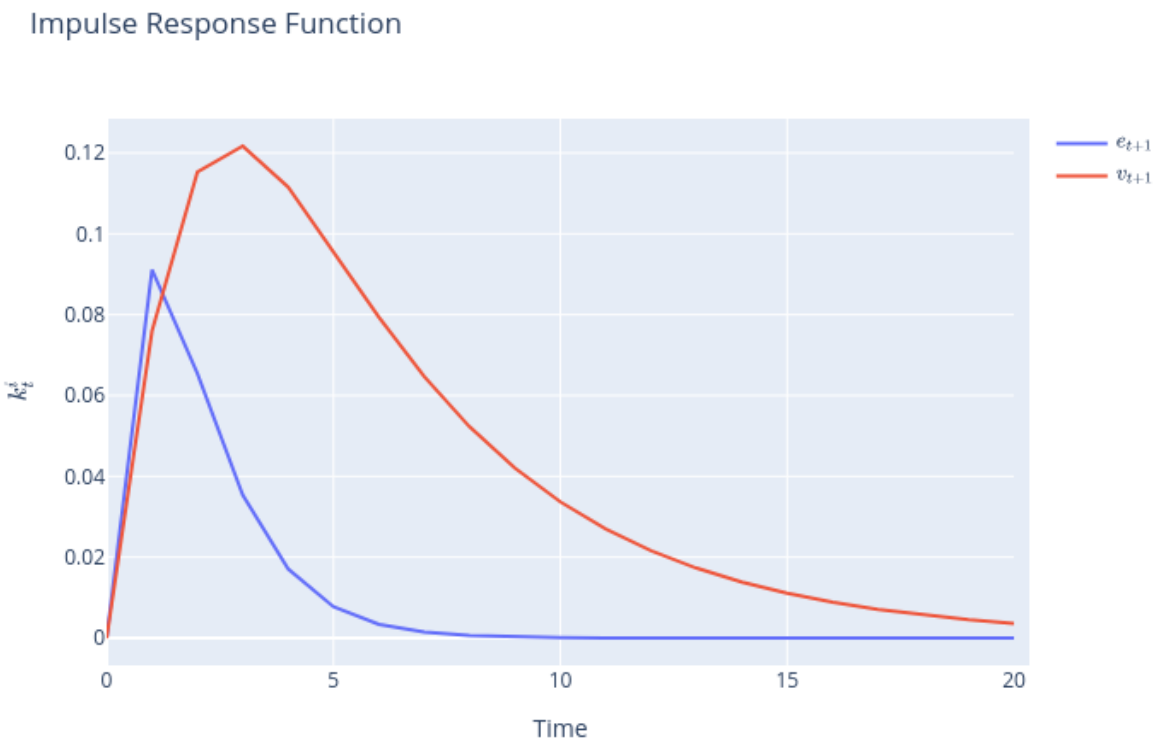
To compute impulse response functions of k_t^i, we use the `impulse_response` method of the `quantecon.LinearStateSpace` class and plot outcomes.

```
xcoef, ycoef = lss.impulse_response(j=21)
data = np.array([xcoef])[0, :, 1, :]

fig = go.Figure(data=go.Scatter(y=data[:-1, 0], name=r'$e_{t+1}$'))
fig.add_trace(go.Scatter(y=data[1:, 1], name=r'$v_{t+1}$'))
fig.update_layout(title=r'Impulse Response Function',
                  xaxis_title='Time',
                  yaxis_title=r'$k^{i}_{t}$')
fig1 = fig
# Export to PNG file
Image(fig1.to_image(format="png"))
```

(continues on next page)

(continued from previous page)

```
# fig1.show() will provide interactive plot when running
# notebook locally
```



13.7.5 Step 5: Compute stationary covariance matrices and population regressions

We compute stationary covariance matrices by calling the `stationary_distributions` method of the `quantecon.LinearStateSpace` class.

By appropriately decomposing the covariance matrix of the state vector, we obtain ingredients of pertinent population regression coefficients.

Define

$$\Sigma_x = \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix}$$

where Σ_{11} is the covariance matrix of dependent variables and Σ_{22} is the covariance matrix of independent variables.

Regression coefficients are $\beta = \Sigma_{21}\Sigma_{22}^{-1}$.

To verify an instance of a law of large numbers computation, we construct a long simulation of the state vector and for the resulting sample compute the ordinary least-squares estimator of β that we shall compare with corresponding population regression coefficients.

```
_, _, Σ_x, Σ_y, Σ_yx = lss.stationary_distributions()

Σ_11 = Σ_x[0, 0]
Σ_12 = Σ_x[0, 1:4]
Σ_21 = Σ_x[1:4, 0]
Σ_22 = Σ_x[1:4, 1:4]

reg_coeffs = Σ_12 @ np.linalg.inv(Σ_22)

print('Regression coefficients (e_t on k_t, P_t, \\tilde{\\theta_t})')
print('------------------------------')
print(r'k_t:', reg_coeffs[0])
print(r'\tilde{\theta_t}:', reg_coeffs[1])
print(r'P_t:', reg_coeffs[2])
```

```
Regression coefficients (e_t on k_t, P_t, \tilde{\theta_t})
------------------------------
k_t: -3.275556845219769
\tilde{\theta_t}: -0.9649461170475457
P_t: 0.9649461170475457
```

```
# Compute R squared
R_squared = reg_coeffs @ Σ_x[1:4, 1:4] @ reg_coeffs  / Σ_x[0, 0]
R_squared
```

```
0.9649461170475461
```

```
# Verify that the computed coefficients are close to least squares estimates
model = OLS(x[0], x[1:4].T)
reg_res = model.fit()
np.max(np.abs(reg_coeffs - reg_res.params)) < 1e-2
```

```
True
```

```
# Verify that R_squared matches least squares estimate
np.abs(reg_res.rsquared - R_squared) < 1e-2
```

```
True
```

```
# Verify that 9_t + e_t can be recovered
model = OLS(y[1], x[1:4].T)
reg_res = model.fit()
np.abs(reg_res.rsquared - 1.) < 1e-6
```

```
True
```

13.8 Equilibrium with Two Noisy Signals on θ_t

Steps 1, 4, and 5 are identical to those for the one-noisy-signal structure.

Step 2 requires a straightforward modification.

For step 3, we construct the following state-space representation so that we can get our hands on all of the random processes that we require in order to compute a regression of the noisy signal about θ from the other industry that a firm receives directly in a pooling equilibrium against information that a firm would receive in Townsend's original model.

For this purpose, we include equilibrium goods prices from both industries in the state vector:

$$
\underbrace{\begin{bmatrix} e_{1,t+1} \\ e_{2,t+1} \\ k^i_{t+1} \\ \tilde{\theta}_{t+1} \\ P^1_{t+1} \\ P^2_{t+1} \\ \theta_{t+1} \\ v_{t+1} \end{bmatrix}}_{x_{t+1}} = \underbrace{\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{\kappa}{\lambda-\rho} & \frac{\kappa}{\lambda-\rho} & \tilde{\lambda} & \frac{-1}{\lambda-\rho}\frac{\kappa\sigma_e^2}{p} & 0 & 0 & \frac{\rho}{\lambda-\rho} & 0 \\ -\kappa & -\kappa & 0 & \frac{\kappa\sigma_e^2}{p} & 0 & 0 & 0 & 1 \\ \frac{b\kappa}{\lambda-\rho} & \frac{b\kappa}{\lambda-\rho} & b\tilde{\lambda} & \frac{-b}{\lambda-\rho}\frac{\kappa\sigma_e^2}{p} & 0 & 0 & \frac{b\rho}{\lambda-\rho}+\rho & 1 \\ \frac{b\kappa}{\lambda-\rho} & \frac{b\kappa}{\lambda-\rho} & b\tilde{\lambda} & \frac{-b}{\lambda-\rho}\frac{\kappa\sigma_e^2}{p} & 0 & 0 & \frac{b\rho}{\lambda-\rho}+\rho & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & \rho & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} e_{1,t} \\ e_{2,t} \\ k^i_t \\ \tilde{\theta}_t \\ P^1_t \\ P^2_t \\ \theta_t \\ v_t \end{bmatrix}}_{x_t} + \underbrace{\begin{bmatrix} \sigma_e & 0 & 0 \\ 0 & \sigma_e & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ \sigma_e & 0 & 0 \\ 0 & \sigma_e & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \sigma_v \end{bmatrix}}_{C} \begin{bmatrix} z_{1,t+1} \\ z_{2,t+1} \\ z_{3,t+1} \end{bmatrix}
$$

$$
\underbrace{\begin{bmatrix} P^1_t \\ P^2_t \\ e_{1,t}+\theta_t \\ e_{2,t}+\theta_t \\ e_{1,t} \\ e_{2,t} \end{bmatrix}}_{y_t} = \underbrace{\begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}}_{G} \underbrace{\begin{bmatrix} e_{1,t} \\ e_{2,t} \\ k^i_t \\ \tilde{\theta}_t \\ P^1_t \\ P^2_t \\ \theta_t \\ v_t \end{bmatrix}}_{x_t} + \underbrace{\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}}_{H} w_{t+1}
$$

$$
\begin{bmatrix} z_{1,t+1} \\ z_{2,t+1} \\ z_{3,t+1} \\ w_{t+1} \end{bmatrix} \sim \mathcal{N}(0, I)
$$

$$
\kappa = \frac{\rho p}{2p + \sigma_e^2}
$$

```
A_ricc = np.array([[ρ]])
B_ricc = np.array([[np.sqrt(2)]])
R_ricc = np.array([[σ_e ** 2]])
Q_ricc = np.array([[σ_v ** 2]])
N_ricc = np.zeros((1, 1))
p = qe.solve_discrete_riccati(A_ricc, B_ricc, Q_ricc, R_ricc, N_ricc).item()

p_two = p  # Save for comparison later
```

```
# Verify that p = σ_v^2 + (ρp^2σ_e^2) / (2p + σ_e^2)
tol = 1e-12
np.abs(p - (σ_v ** 2 + p * ρ ** 2 * σ_e ** 2 / (2 * p + σ_e ** 2))) < tol
```

```
True
```

```
κ = ρ * p / (2 * p + σ_e ** 2)
κ_prod = κ * σ_e ** 2 / p

κ_two = κ  # Save for comparison later

A_lss = np.array([[0., 0., 0., 0., 0., 0., 0., 0.],
                  [0., 0., 0., 0., 0., 0., 0., 0.],
                  [κ / (λ - ρ), κ / (λ - ρ), λ_tilde, -κ_prod / (λ - ρ), 0., 0., ρ /␣
 ↪(λ - ρ), 0.],
                  [-κ, -κ, 0., κ_prod, 0., 0., 0., 1.],
                  [b * κ / (λ - ρ), b * κ / (λ - ρ), b * λ_tilde, -b * κ_prod / (λ -␣
 ↪ρ), 0., 0., b * ρ / (λ - ρ) + ρ, 1.],
                  [b * κ / (λ - ρ), b * κ / (λ - ρ), b * λ_tilde, -b * κ_prod / (λ -␣
 ↪ρ), 0., 0., b * ρ / (λ - ρ) + ρ, 1.],
                  [0., 0., 0., 0., 0., 0., ρ, 1.],
                  [0., 0., 0., 0., 0., 0., 0., 0.]])

C_lss = np.array([[σ_e, 0., 0.],
                  [0., σ_e, 0.],
                  [0., 0.,  0.],
                  [0., 0.,  0.],
                  [σ_e, 0., 0.],
                  [0., σ_e, 0.],
                  [0., 0., 0.],
                  [0., 0., σ_v]])

G_lss = np.array([[0., 0., 0., 0., 1., 0., 0., 0.],
                  [0., 0, 0, 0., 0., 1., 0., 0.],
                  [1., 0., 0., 0., 0., 0., 1., 0.],
                  [0., 1., 0., 0., 0., 0., 1., 0.],
                  [1., 0., 0., 0., 0., 0., 0., 0.],
                  [0., 1., 0., 0., 0., 0., 0., 0.]])
```

```
mu_0 = np.array([0., 0., 0., 0., 0., 0., 0., 0.])

lss = qe.LinearStateSpace(A_lss, C_lss, G_lss, mu_0=mu_0)
```

```
ts_length = 100_000
x, y = lss.simulate(ts_length, random_state=1)
```

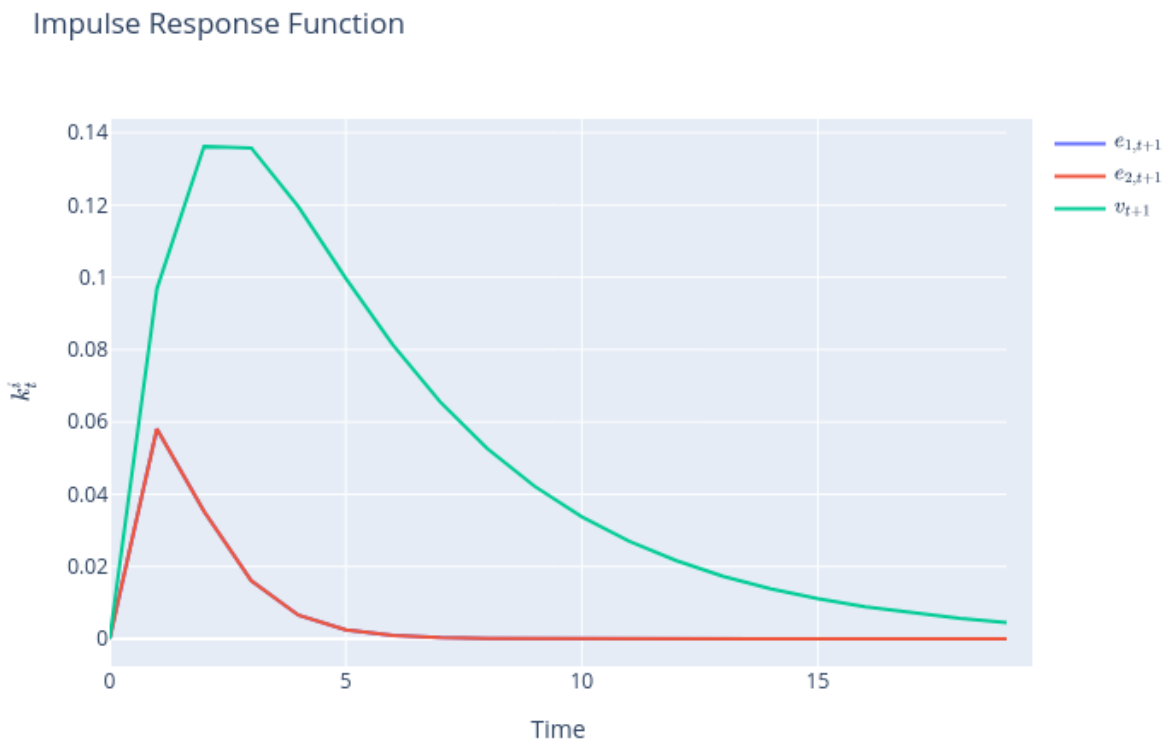
```
xcoef, ycoef = lss.impulse_response(j=20)
```

```
data = np.array([xcoef])[0, :, 2, :]

fig = go.Figure(data=go.Scatter(y=data[:-1, 0], name=r'$e_{1,t+1}$'))
fig.add_trace(go.Scatter(y=data[:-1, 1], name=r'$e_{2,t+1}$'))
fig.add_trace(go.Scatter(y=data[1:, 2], name=r'$v_{t+1}$'))
fig.update_layout(title=r'Impulse Response Function',
                  xaxis_title='Time',
                  yaxis_title=r'$k^{i}_{t}$')
fig2=fig
# Export to PNG file
Image(fig2.to_image(format="png"))
# fig2.show() will provide interactive plot when running
```

(continues on next page)

(continued from previous page)

```
# notebook locally
```




```
_, _, Σ_x, Σ_y, Σ_yx = lss.stationary_distributions()

Σ_11 = Σ_x[1, 1]
Σ_12 = Σ_x[1, 2:5]
Σ_21 = Σ_x[2:5, 1]
Σ_22 = Σ_x[2:5, 2:5]

reg_coeffs = Σ_12 @ np.linalg.inv(Σ_22)

print('Regression coefficients (e_{2,t} on k_t, P^{1}_t, \\tilde{\\theta_t})')
print('------------------------------')
print(r'k_t:', reg_coeffs[0])
print(r'\tilde{\theta_t}:', reg_coeffs[1])
print(r'P_t:', reg_coeffs[2])
```

```
Regression coefficients (e_{2,t} on k_t, P^{1}_t, \tilde{\theta_t})
------------------------------
k_t: 0.0
\tilde{\theta_t}: 0.0
P_t: 0.0
```

```
# Compute R squared
```

(continues on next page)

(continued from previous page)

```
R_squared = reg_coeffs @ Σ_x[2:5, 2:5] @ reg_coeffs  / Σ_x[1, 1]
R_squared
```

```
0.0
```

```
# Verify that the computed coefficients are close to least squares estimates
model = OLS(x[1], x[2:5].T)
reg_res = model.fit()
np.max(np.abs(reg_coeffs - reg_res.params)) < 1e-2
```

```
True
```

```
# Verify that R_squared matches least squares estimate
np.abs(reg_res.rsquared - R_squared) < 1e-2
```

```
True
```

```
_, _, Σ_x, Σ_y, Σ_yx = lss.stationary_distributions()

Σ_11 = Σ_x[1, 1]
Σ_12 = Σ_x[1, 2:6]
Σ_21 = Σ_x[2:6, 1]
Σ_22 = Σ_x[2:6, 2:6]

reg_coeffs = Σ_12 @ np.linalg.inv(Σ_22)

print('Regression coefficients (e_{2,t} on k_t, P^{1}_t, P^{2}_t, \\tilde{\\theta_t})
 ↪')
print('------------------------------')
print(r'k_t:', reg_coeffs[0])
print(r'\tilde{\theta_t}:', reg_coeffs[1])
print(r'P^{1}_t:', reg_coeffs[2])
print(r'P^{2}_t:', reg_coeffs[3])
```

```
Regression coefficients (e_{2,t} on k_t, P^{1}_t, P^{2}_t, \tilde{\theta_t})
------------------------------
k_t: -3.1373589171035627
\tilde{\theta_t}: -0.9242343967443672
P^{1}_t: -0.037882801627816154
P^{2}_t: 0.9621171983721835
```

```
# Compute R squared
R_squared = reg_coeffs @ Σ_x[2:6, 2:6] @ reg_coeffs  / Σ_x[1, 1]
R_squared
```

```
0.9621171983721837
```

13.9 Key Step

Now we come to the key step for verifying that equilibrium outcomes for prices and quantities are identical in the pooling equilibrium original model that led Townsend to deduce an infinite-dimensional state space.

We accomplish this by computing a population linear least squares regression of the noisy signal that firms in the other industry receive in a pooling equilibrium on time t information that a firm would receive in Townsend's original model.

Let's compute the regression and stare at the R^2:

```
# Verify that 9_t + e^{2}_t can be recovered

# 9_t + e^{2}_t on k^{i}_t, P^{1}_t, P^{2}_t, \\tilde{\\theta_t}


model = OLS(y[1], x[2:6].T)
reg_res = model.fit()
np.abs(reg_res.rsquared - 1.) < 1e-6
```

```
True
```

```
reg_res.rsquared
```

```
1.0
```

The R^2 in this regression equals 1.

That verifies that a firm's information set in Townsend's original model equals its information set in a pooling equilibrium.

Therefore, equilibrium prices and quantities in Townsend's original model equal those in a pooling equilibrium.

13.10 An observed common shock benchmark

For purposes of comparison, it is useful to construct a model in which demand disturbance in both industries still both share have a common persistent component θ_t, but in which the persistent component θ is observed each period.

In this case, firms share the same information immediately and have no need to deploy signal-extraction techniques.

Thus, consider a version of our model in which histories of both ϵ_t^i and θ_t are observed by a representative firm.

In this case, the firm's optimal decision rule is described by

$$k_{t+1}^i = \tilde{\lambda} k_t^i + \frac{1}{\lambda - \rho} \hat{\theta}_{t+1}$$

where $\hat{\theta}_{t+1} = E_t \theta_{t+1}$ is given by

$$\hat{\theta}_{t+1} = \rho \theta_t$$

Thus, the firm's decision rule can be expressed

$$k_{t+1}^i = \tilde{\lambda} k_t^i + \frac{\rho}{\lambda - \rho}\theta_t$$

Consequently, when a history $\theta_s, s \leq t$ is observed without noise, the following state space system prevails:

$$\begin{bmatrix} \theta_{t+1} \\ k_{t+1}^i \end{bmatrix} = \begin{bmatrix} \rho & 0 \\ \frac{\rho}{\lambda-\rho} & \tilde{\lambda} \end{bmatrix} \begin{bmatrix} \theta_t \\ k_t^i \end{bmatrix} + \begin{bmatrix} \sigma_v \\ 0 \end{bmatrix} z_{1,t+1}$$
$$\begin{bmatrix} \theta_t \\ k_t^i \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \theta_t \\ k_t^i \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} z_{1,t+1}$$

where $z_{t,t+1}$ is a scalar iid standardized Gaussian process.

As usual, the system can be written as

$$x_{t+1} = Ax_t + Cz_{t+1}$$
$$y_t = Gx_t + Hw_{t+1}$$

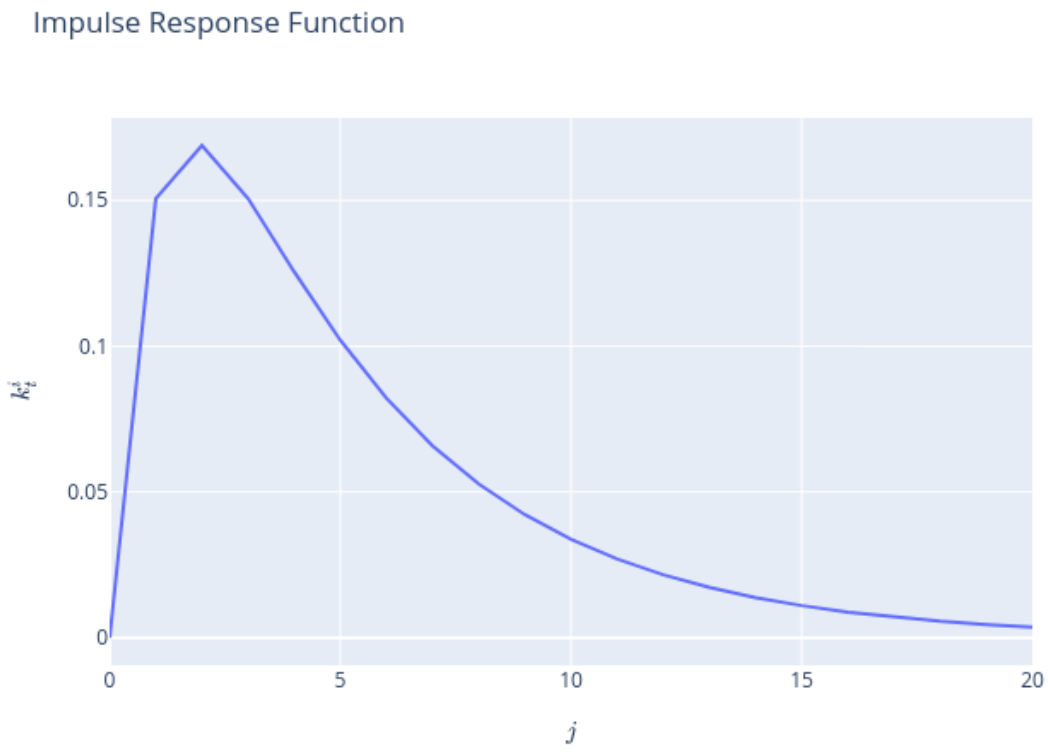
In order once again to use the quantecon class `quantecon.LinearStateSpace`, let's form pertinent state-space matrices

```
Ao_lss = np.array([[ρ, 0.],
                   [ρ / (λ - ρ), λ_tilde]])

Co_lss = np.array([[σ_v], [0.]])

Go_lss = np.identity(2)
```

```
muo_0 = np.array([0., 0.])

lsso = qe.LinearStateSpace(Ao_lss, Co_lss, Go_lss, mu_0=muo_0)
```

Now let's form and plot an impulse response function of k_t^i to shocks v_t to θ_{t+1}

```
xcoef, ycoef = lsso.impulse_response(j=21)
data = np.array([ycoef])[0, :, 1, :]

fig = go.Figure(data=go.Scatter(y=data[:-1, 0], name=r'$z_{t+1}$'))
fig.update_layout(title=r'Impulse Response Function',
                  xaxis_title= r'lag $j$',
                  yaxis_title=r'$k^{i}_{t}$')
fig3 = fig
# Export to PNG file
Image(fig3.to_image(format="png"))
# fig1.show() will provide interactive plot when running
# notebook locally
```

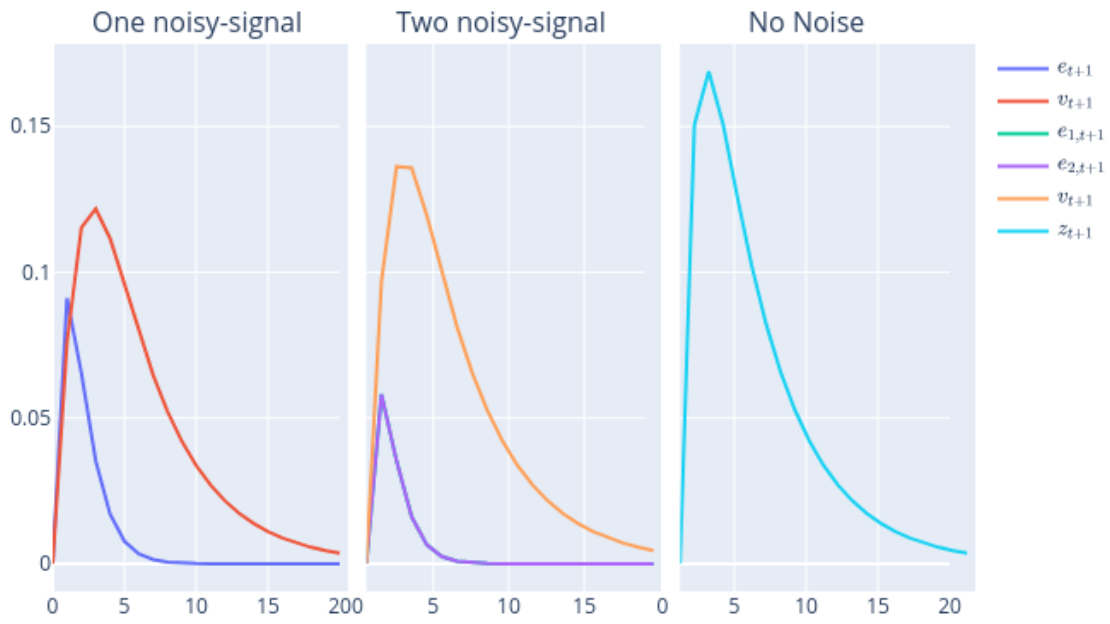


13.11 Comparison of All Signal Structures

It is enlightening side by side to plot impulse response functions for capital for the two noisy-signal information structures and the noiseless signal on θ that we have just presented.

Please remember that the two-signal structure corresponds to the **pooling equilibrium** and also **Townsend's original model**.

```
fig_comb = go.Figure(data=[
                        *fig1.data,
                        *fig2.update_traces(xaxis='x2', yaxis='y2').data,
                        *fig3.update_traces(xaxis='x3', yaxis='y3').data
                    ]).set_subplots(1, 3,
                           subplot_titles=("One noisy-signal",
                                           "Two noisy-signal",
                                           "No Noise"),
                           horizontal_spacing=0.02,
                           shared_yaxes=True)
# Export to PNG file
Image(fig_comb.to_image(format="png"))
# fig_comb.show() # will provide interactive plot when running
# notebook locally
```



The three panels in the graph above show that

- responses of k_t^i to shocks v_t to the hidden Markov demand state θ_t process are **largest** in the no-noisy-signal structure in which the firm observes θ_t at time t
- responses of k_t^i to shocks v_t to the hidden Markov demand state θ_t process are **smaller** in the two-noisy-signal structure
- responses of k_t^i to shocks v_t to the hidden Markov demand state θ_t process are **smallest** in the one-noisy-signal structure

With respect to the iid demand shocks e_t the graphs show that

- responses of k_t^i to shocks e_t to the hidden Markov demand state θ_t process are **smallest** (i.e., nonexistent) in the no-noisy-signal structure in which the firm observes θ_t at time t
- responses of k_t^i to shocks e_t to the hidden Markov demand state θ_t process are **larger** in the two-noisy-signal structure
- responses of k_t^i to idiosyncratic *own-market* noise-shocks e_t are **largest** in the one-noisy-signal structure

Among other things, these findings indicate that time series correlations and coherences between outputs in the two industries are higher in the two-noisy-signals or **pooling** model than they are in the one-noisy signal model.

The enhanced influence of the shocks v_t to the hidden Markov demand state θ_t process that emerges from the two-noisy-signal model relative to the one-noisy-signal model is a symptom of a lower equilibrium hidden-state reconstruction error variance in the two-signal model:

```
display(Latex('$\\textbf{Reconstruction error variances}$'))
display(Latex(f'One-noise structure: {round(p_one, 6)}'))
display(Latex(f'Two-noise structure: {round(p_two, 6)}'))
```

Reconstruction error variances

$$One - noisestructure : 0.36618$$

$$Two - noisestructure : 0.324062$$

Kalman gains for the two structures are

```
display(Latex('$\\textbf{Kalman Gains}$'))
display(Latex(f'One noisy-signal structure: {round(κ_one, 6)}'))
display(Latex(f'Two noisy-signals structure: {round(κ_two, 6)}'))
```

Kalman Gains

$$Onenoisy - signalstructure : 0.403404$$

$$Twonoisy - signalsstructure : 0.25716$$

Another lesson that comes from the preceding three-panel graph is that the presence of iid noise ϵ_t^i in industry i generates a response in k_t^{-i} in the two-noisy-signal structure, but not in the one-noisy-signal structure.

13.12 Notes on History of the Problem

To truncate what he saw as an intractable, infinite dimensional state space, Townsend constructed an approximating model in which the common hidden Markov demand shock is revealed to all firms after a fixed number of periods.

Thus,

- Townsend wanted to assume that at time t firms in industry i observe $k_t^i, Y_t^i, P_t^i, (P^{-i})^t$, where $(P^{-i})^t$ is the history of prices in the other market up to time t.
- Because that turned out to be too challenging, Townsend made a sensible alternative assumption that eased his calculations: that after a large number S of periods, firms in industry i observe the hidden Markov component of the demand shock θ_{t-S}.

Townsend argued that the more manageable model could do a good job of approximating the intractable model in which the Markov component of the demand shock remains unobserved for ever.

By applying technical machinery of [Pearlman *et al.*, 1986], [Pearlman and Sargent, 2005] showed that there is a recursive representation of the equilibrium of the perpetually and symmetrically uninformed model that Townsend wanted to solve [Townsend, 1983].

A reader of [Pearlman and Sargent, 2005] will notice that their representation of the equilibrium of Townsend's model exactly matches that of the **pooling equilibrium** presented here.

We have structured our notation in this lecture to faciliate comparison of the **pooling equilibrium** constructed here with the equilibrium of Townsend's model reported in [Pearlman and Sargent, 2005].

The computational method of [Pearlman and Sargent, 2005] is recursive: it enlists the Kalman filter and invariant subspace methods for solving systems of Euler equations[5] .

[5] See [Anderson *et al.*, 1996] for an account of invariant subspace methods.

As [Singleton, 1987], [Kasa, 2000], and [Sargent, 1991] also found, the equilibrium is fully revealing: observed prices tell participants in industry i all of the information held by participants in market $-i$ ($-i$ means not i).

This means that higher-order beliefs play no role: observing equilibrium prices in effect lets decision makers pool their information sets[6] .

The disappearance of higher order beliefs means that decision makers in this model do not really face a problem of forecasting the forecasts of others.

Because those forecasts are the same as their own, they know them.

13.12.1 Further historical remarks

Sargent [Sargent, 1991] proposed a way to compute an equilibrium without making Townsend's approximation.

Extending the reasoning of [Muth, 1960], Sargent noticed that it is possible to summarize the relevant history with a low dimensional object, namely, a small number of current and lagged forecasting errors.

Positing an equilibrium in a space of perceived laws of motion for endogenous variables that takes the form of a vector autoregressive, moving average, Sargent described an equilibrium as a fixed point of a mapping from the perceived law of motion to the actual law of motion of that form.

Sargent worked in the time domain and proceeded to guess and verify the appropriate orders of the autoregressive and moving average pieces of the equilibrium representation.

By working in the frequency domain [Kasa, 2000] showed how to discover the appropriate orders of the autoregressive and moving average parts, and also how to compute an equilibrium.

The [Pearlman and Sargent, 2005] recursive computational method, which stays in the time domain, also discovered appropriate orders of the autoregressive and moving average pieces.

In addition, by displaying equilibrium representations in the form of [Pearlman *et al.*, 1986], [Pearlman and Sargent, 2005] showed how the moving average piece is linked to the innovation process of the hidden persistent component of the demand shock.

That scalar innovation process is the additional state variable contributed by the problem of extracting a signal from equilibrium prices that decision makers face in Townsend's model.

---

[6] See [Allen *et al.*, 2002] for a discussion of information assumptions needed to create a situation in which higher order beliefs appear in equilibrium decision rules. A way to read our findings in light of [Allen *et al.*, 2002] is that, relative to the number of signals agents observe, Townsend's section 8 model has too few random shocks to get higher order beliefs to play a role.

Part IV

Other

CHAPTER

FOURTEEN

TROUBLESHOOTING

Contents

This page is for readers experiencing errors when running the code from the lectures.

14.1 Fixing Your Local Environment

The basic assumption of the lectures is that code in a lecture should execute whenever

1. it is executed in a Jupyter notebook and
2. the notebook is running on a machine with the latest version of Anaconda Python.

You have installed Anaconda, haven't you, following the instructions in this lecture?

Assuming that you have, the most common source of problems for our readers is that their Anaconda distribution is not up to date.

Here's a useful article on how to update Anaconda.

Another option is to simply remove Anaconda and reinstall.

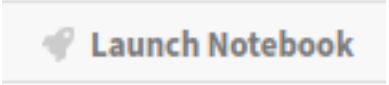
You also need to keep the external code libraries, such as QuantEcon.py up to date.

For this task you can either

- use conda install -y quantecon on the command line, or
- execute !conda install -y quantecon within a Jupyter notebook.

If your local environment is still not working you can do two things.

First, you can use a remote machine instead, by clicking on the Launch Notebook icon available for each lecture



Second, you can report an issue, so we can try to fix your local set up.

We like getting feedback on the lectures so please don't hesitate to get in touch.

14.2 Reporting an Issue

One way to give feedback is to raise an issue through our issue tracker.

Please be as specific as possible. Tell us where the problem is and as much detail about your local set up as you can provide.

Another feedback option is to use our discourse forum.

Finally, you can provide direct feedback to contact@quantecon.org

CHAPTER

FIFTEEN

REFERENCES

CHAPTER

SIXTEEN

EXECUTION STATISTICS

This table contains the latest execution statistics.

Document	Modified	Method	Run Time (s)	Status
aiyagari	2024-05-01 00:15	cache	15.62	✅
arellano	2024-05-01 00:17	cache	82.62	✅
cass_koopmans_1	2024-05-01 00:17	cache	6.83	✅
cass_koopmans_2	2024-05-01 00:17	cache	6.03	✅
coase	2024-05-01 00:17	cache	9.82	✅
house_auction	2024-05-01 00:17	cache	4.25	✅
intro	2024-05-01 00:17	cache	0.97	✅
knowing_forecasts_of_others	2024-05-01 00:19	cache	91.69	✅
markov_perf	2024-05-01 00:19	cache	5.57	✅
matsuyama	2024-05-01 00:59	cache	2406.79	✅
rational_expectations	2024-05-01 00:59	cache	4.9	✅
re_with_feedback	2024-05-01 00:59	cache	8.36	✅
status	2024-05-01 00:17	cache	0.97	✅
troubleshooting	2024-05-01 00:17	cache	0.97	✅
two_auctions	2024-05-01 00:59	cache	14.49	✅
uncertainty_traps	2024-05-01 01:00	cache	2.51	✅
zreferences	2024-05-01 00:17	cache	0.97	✅

These lectures are built on `linux` instances through `github actions` so are executed using the following hardware specifications

BIBLIOGRAPHY

[Aiy94] S Rao Aiyagari. Uninsured Idiosyncratic Risk and Aggregate Saving. *The Quarterly Journal of Economics*, 109(3):659–684, 1994.

[AMS02] Franklin Allen, Stephen Morris, and Hyun Song Shin. Beauty contests, bubbles, and iterated expectations in asset markets. *mimeo*, 2002.

[AHMS96] Evan Anderson, Lars Peter Hansen, Ellen R. McGrattan, and Thomas J. Sargent. Mechanics of forming and estimating dynamic linear economies. In Hans M. Amman, David A. Kendrick, and John Rust, editors, *Handbook of computational economics*, 171–252. Elsevier Science, North-Holland, 1996.

[Are08] Cristina Arellano. Default risk and income fluctuations in emerging economies. *The American Economic Review*, pages 690–712, 2008.

[BBZ15] Jess Benhabib, Alberto Bisin, and Shenghao Zhu. The wealth distribution in bewley economies with capital income risk. *Journal of Economic Theory*, 159:489–515, 2015.

[BS79] L M Benveniste and J A Scheinkman. On the Differentiability of the Value Function in Dynamic Models of Economics. *Econometrica*, 47(3):727–732, 1979.

[Bew77] Truman Bewley. The permanent income hypothesis: a theoretical formulation. *Journal of Economic Theory*, 16(2):252–292, 1977.

[BK80] Olivier Jean Blanchard and Charles M Kahn. The Solution of Linear Difference Models under Rational Expectations. *Econometrica*, 48(5):1305–1311, July 1980.

[Cag56] Philip Cagan. The monetary dynamics of hyperinflation. In Milton Friedman, editor, *Studies in the Quantity Theory of Money*, pages 25–117. University of Chicago Press, Chicago, 1956.

[Cas65] David Cass. Optimum growth in an aggregative model of capital accumulation. *Review of Economic Studies*, 32(3):233–240, 1965.

[Cla71] E. Clarke. Multipart pricing of public goods. *Public Choice*, 8:19–33, 1971.

[Coa37] Ronald Harry Coase. The nature of the firm. *economica*, 4(16):386–405, 1937.

[DJ92] Raymond J Deneckere and Kenneth L Judd. Cyclical and chaotic behavior in a dynamic equilibrium model, with implications for fiscal policy. *Cycles and chaos in economic equilibrium*, pages 308–329, 1992.

[DS10] Ulrich Doraszelski and Mark Satterthwaite. Computable markov-perfect industry dynamics. *The RAND Journal of Economics*, 41(2):215–243, 2010.

[EP95] Richard Ericson and Ariel Pakes. Markov-perfect industry dynamics: a framework for empirical work. *The Review of Economic Studies*, 62(1):53–82, 1995.

[EH01] G W Evans and S Honkapohja. *Learning and Expectations in Macroeconomics*. Frontiers of Economic Research. Princeton University Press, 2001.

[FSTD15] Pablo Fajgelbaum, Edouard Schaal, and Mathieu Taschereau-Dumouchel. Uncertainty traps. Technical Report, National Bureau of Economic Research, 2015.

[Gro73] T. Groves. Incentives in teams. *Econometrica*, 41:617–631, 1973.

[HL96] John Heaton and Deborah J Lucas. Evaluating the effects of incomplete markets on risk sharing and asset pricing. *Journal of Political Economy*, pages 443–487, 1996.

[HK85] Elhanan Helpman and Paul Krugman. *Market structure and international trade*. MIT Press Cambridge, 1985.

[Jud90] K L Judd. Cournot versus bertrand: a dynamic resolution. Technical Report, Hoover Institution, Stanford University, 1990.

[Jud85] Kenneth L Judd. On the performance of patents. *Econometrica*, pages 567–585, 1985.

[Kas00] Kenneth Kasa. Forecasting the forecasts of others in the frequency domain. *Review of Economic Dynamics*, 3:726–756, 2000.

[KNS18] Tomoo Kikuchi, Kazuo Nishimura, and John Stachurski. Span of control, transaction costs, and the structure of production chains. *Theoretical Economics*, 13(2):729–760, 2018.

[Koo65] Tjalling C. Koopmans. On the concept of optimal economic growth. In Tjalling C. Koopmans, editor, *The Economic Approach to Development Planning*, pages 225–287. Chicago, 1965.

[LM80] David Levhari and Leonard J Mirman. The great fish war: an example using a dynamic cournot-nash solution. *The Bell Journal of Economics*, pages 322–334, 1980.

[LS18] L Ljungqvist and T J Sargent. *Recursive Macroeconomic Theory*. MIT Press, 4 edition, 2018.

[LP71] Robert E Lucas, Jr. and Edward C Prescott. Investment under uncertainty. *Econometrica: Journal of the Econometric Society*, pages 659–681, 1971.

[MS89] Albert Marcet and Thomas J Sargent. Convergence of Least-Squares Learning in Environments with Hidden State Variables and Private Information. *Journal of Political Economy*, 97(6):1306–1322, 1989.

[MCWG95] A Mas-Colell, M D Whinston, and J R Green. *Microeconomic Theory*. Volume 1. Oxford University Press, 1995.

[Mut60] John F Muth. Optimal properties of exponentially weighted forecasts. *Journal of the american statistical association*, 55(290):299–306, 1960.

[PCL86] Joseph Pearlman, David Currie, and Paul Levine. Rational Expectations Models with Private Information. *Economic Modelling*, 3(2):90–105, 1986.

[PS05] Joseph G. Pearlman and Thomas J. Sargent. Knowing the Forecasts of Others. *Review of Economic Dynamics*, 8(2):480–497, April 2005. URL: https://ideas.repec.org/a/red/issued/v8y2005i2p480-497.html, doi:10.1016/j.red.2004.10.011.

[REL75] Jr. Robert E. Lucas. An equilibrium model of the business cycle. *Journal of Political Economy*, 83:1113–1144, 1975.

[Rya12] Stephen P Ryan. The costs of environmental regulation in a concentrated industry. *Econometrica*, 80(3):1019–1061, 2012.

[Sar77] Thomas J Sargent. The Demand for Money During Hyperinflations under Rational Expectations: I. *International Economic Review*, 18(1):59–82, February 1977.

[Sar87] Thomas J Sargent. *Macroeconomic Theory*. Academic Press, New York, 2nd edition, 1987.

[Sar91] Thomas J. Sargent. Equilibrium with signal extraction from endogenous variables. *Journal of Economic Dynamics and Control*, 15:245–273, 1991.

[Sin87] Kenneth J. Singleton. Asset prices in a time-series model with disparately informed competitive traders. In William A. Barnett and Kenneth J. Singleton, editors, *New Apprroaches to Monetary Economics*. Cambridge University Press, 1987.

[Tow83] Robert M. Townsend. Forecasting the forecasts of others. *Journal of Political Economy*, 91:546–588, 1983.

[VL11] Ngo Van Long. Dynamic games in the economics of natural resources: a survey. *Dynamic Games and Applications*, 1(1):115–148, 2011.

[Vic61] W. Vickrey. Counterspeculation, auctions, and competitive sealed tenders. *Journal of Finance*, 16:8–37, 1961.

[Whi83] Charles Whiteman. *Linear Rational Expectations Models: A User's Guide*. University of Minnesota Press, Minneapolis, Minnesota, 1983.

[YS05] G Alastair Young and Richard L Smith. *Essentials of statistical inference*. Cambridge University Press, 2005.

INDEX