Part III

Multiple Agent Models

CHAPTER

FOURTEEN

ROBUST MARKOV PERFECT EQUILIBRIUM

Contents

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

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

14.1 Overview

This lecture describes a Markov perfect equilibrium with robust agents.

We focus on special settings with

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

These specifications simplify calculations and allow us to give a simple example that illustrates basic forces.

This lecture is based on ideas described in chapter 15 of [HS08a] and in Markov perfect equilibrium and *Robustness*.

Let's start with some standard imports:

```
import numpy as np
import quantecon as qe
from scipy.linalg import solve
import matplotlib.pyplot as plt
%matplotlib inline
```

14.1.1 Basic Setup

Decisions of two agents affect the motion of a state vector that appears as an argument of payoff functions of both agents.

As described in Markov perfect equilibrium, when decision-makers have no concerns about the robustness of their decision rules to misspecifications of the state dynamics, a Markov perfect equilibrium can be computed via backward recursion on two sets of equations

- a pair of Bellman equations, one for each agent.
- a pair of equations that express linear decision rules for each agent as functions of that agent's continuation value function as well as parameters of preferences and state transition matrices.

This lecture shows how a similar equilibrium concept and similar computational procedures apply when we impute concerns about robustness to both decision-makers.

A Markov perfect equilibrium with robust agents will be characterized by

- a pair of Bellman equations, one for each agent.
- a pair of equations that express linear decision rules for each agent as functions of that agent's continuation value function as well as parameters of preferences and state transition matrices.
- a pair of equations that express linear decision rules for worst-case shocks for each agent as functions of that agent's continuation value function as well as parameters of preferences and state transition matrices.

Below, we'll construct a robust firms version of the classic duopoly model with adjustment costs analyzed in Markov perfect equilibrium.

14.2 Linear Markov Perfect Equilibria with Robust Agents

As we saw in Markov perfect equilibrium, the study of Markov perfect equilibria in dynamic 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.

14.2.1 Modified Coupled Linear Regulator Problems

We consider a general linear quadratic regulator game with two players, each of whom fears model misspecifications.

We often call the players agents.

The agents share a common baseline model for the transition dynamics of the state vector

- this is a counterpart of a 'rational expectations' assumption of shared beliefs

But now one or more agents doubt that the baseline model is correctly specified.

The agents express the possibility that their baseline specification is incorrect by adding a contribution Cv_{it} to the time t transition law for the state

- C is the usual *volatility matrix* that appears in stochastic versions of optimal linear regulator problems.
- v_{it} is a possibly history-dependent vector of distortions to the dynamics of the state that agent i uses to represent misspecification of the original model.

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 a sequence $\{u_{-it}\}$ as given and chooses a sequence $\{u_{it}\}$ to minimize and $\{v_{it}\}$ to maximize

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

while thinking that the state evolves according to

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

Here

- x_t is an $n \times 1$ state vector, u_{it} is a $k_i \times 1$ vector of controls for player i, and
- v_{it} is an $h \times 1$ vector of distortions to the state dynamics that concern 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 $n \times k_i$
- C is $n \times h$
- $\theta_i \in [\underline{\theta}_i, +\infty]$ is a scalar multiplier parameter of player i

If $\theta_i = +\infty$, player i completely trusts the baseline model.

If $\theta_i <_\infty$, player i suspects that some other unspecified model actually governs the transition dynamics.

The term $\theta_i v_{it}' v_{it}$ is a time t contribution to an entropy penalty that an (imaginary) loss-maximizing agent inside agent i's mind charges for distorting the law of motion in a way that harms agent i.

- the imaginary loss-maximizing agent helps the loss-minimizing agent by helping him construct bounds on the behavior of his decision rule over a large **set** of alternative models of state transition dynamics.

14.2.2 Computing Equilibrium

We formulate a linear robust 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.

Player i's malevolent alter ego employs decision rules $v_{it} = K_{it} x_t$ where K_{it} is an $h \times n$ matrix.

A robust Markov perfect equilibrium is a pair of sequences $\{F_{1t}, F_{2t}\}$ and a pair of sequences $\{K_{1t}, K_{2t}\}$ over $t = t_0, \ldots, t_1 - 1$ that satisfy

- $\{F_{1t}, K_{1t}\}$ solves player 1's robust decision problem, taking $\{F_{2t}\}$ as given, and
- $\{F_{2t}, K_{2t}\}$ solves player 2's robust decision problem, taking $\{F_{1t}\}$ as given.

If we substitute $u_{2t} = -F_{2t} x_t$ into (1) and (2), then player 1's problem becomes minimization-maximization of

$$\sum_{t=t_0}^{t_1-1} \beta^{t-t_0} \left\{x_t' \Pi_{1t} x_t + u_{1t}' Q_1 u_{1t} + 2u_{1t}' \Gamma_{1t} x_t - \theta_1 v_{1t}' v_{1t}\right\} \tag{3}$$

subject to

$$x_{t+1} = \Lambda_{1t} x_t + B_1 u_{1t} + C v_{1t} \tag{4}$$

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 robust dynamic programming problem of the type studied in the *Robustness* lecture, which can be solved by working backward.

Maximization with respect to distortion v_{1t} leads to the following version of the $\mathcal{D}$ operator from the *Robustness* lecture, namely

$$\mathcal{D}_1(P) := P + PC(\theta_1 I - C'PC)^{-1} C'P \tag{5}$$

The matrix F_{1t} in the policy rule $u_{1t} = -F_{1t} x_t$ that solves agent 1's problem satisfies

$$F_{1t} = (Q_1 + \beta B'_1 \mathcal{D}_1(P_{1t+1}) B_1)^{-1} (\beta B'_1 \mathcal{D}_1(P_{1t+1}) \Lambda_{1t} + \Gamma_{1t}) \tag{6}$$

where P_{1t} solves the matrix Riccati difference equation

$$P_{1t} = \Pi_{1t} - (\beta B'_1 \mathcal{D}_1(P_{1t+1}) \Lambda_{1t} + \Gamma_{1t})' (Q_1 + \beta B'_1 \mathcal{D}_1(P_{1t+1}) B_1)^{-1} (\beta B'_1 \mathcal{D}_1(P_{1t+1}) \Lambda_{1t} + \Gamma_{1t}) + \beta \Lambda'_{1t} \mathcal{D}_1(P_{1t+1}) \Lambda_{1t} \tag{7}$$

Similarly, the policy that solves player 2's problem is

$$F_{2t} = (Q_2 + \beta B'_2 \mathcal{D}_2(P_{2t+1}) B_2)^{-1} (\beta B'_2 \mathcal{D}_2(P_{2t+1}) \Lambda_{2t} + \Gamma_{2t}) \tag{8}$$

where P_{2t} solves

$$P_{2t} = \Pi_{2t} - (\beta B'_2 \mathcal{D}_2(P_{2t+1}) \Lambda_{2t} + \Gamma_{2t})' (Q_2 + \beta B'_2 \mathcal{D}_2(P_{2t+1}) B_2)^{-1} (\beta B'_2 \mathcal{D}_2(P_{2t+1}) \Lambda_{2t} + \Gamma_{2t}) + \beta \Lambda'_{2t} \mathcal{D}_2(P_{2t+1}) \Lambda_{2t} \tag{9}$$

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 (6), (7), (8), and (9), and "work backwards" from time $t_1 - 1$.

Since we're working backwards, P_{1t+1} and P_{2t+1} are taken as given at each stage.

Moreover, since

- some terms on the right-hand side of (6) contain F_{2t}
- some terms on the right-hand side of (8) contain F_{1t}

we need to solve these $k_1 + k_2$ equations simultaneously.

14.2.3 Key Insight

As in Markov perfect equilibrium, a key insight here is that equations (6) and (8) 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 (7) and (9).

Notice how j's control law F_{jt} is a function of $\{F_{is}, s \geq t, i \neq j\}$.

Thus, agent i's choice of $\{F_{it}; t = t_0, \ldots, t_1 - 1\}$ influences agent j's choice of control laws.

However, in the Markov perfect equilibrium of this game, each agent is assumed to ignore the influence that his choice exerts on the other agent's choice.

After these equations have been solved, we can also deduce associated sequences of worst-case shocks.

14.2.4 Worst-case Shocks

For agent i the maximizing or worst-case shock v_{it} is

$$v_{it} = K_{it} x_t$$

where

$$K_{it} = \theta_i^{-1} (I - \theta_i^{-1} C' P_{i,t+1} C)^{-1} C' P_{i,t+1} (A - B_1 F_{it} - B_2 F_{2t})$$

14.2.5 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.

14.2.6 Implementation

We use the function nnash_robust to compute a Markov perfect equilibrium of the infinite horizon linear quadratic dynamic game with robust planers in the manner described above.

14.3 Application

14.3.1 A Duopoly Model

Without concerns for robustness, the model is identical to the duopoly model from the Markov perfect equilibrium lecture.

To begin, we briefly review the structure of that model.

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

$$p = a_0 - a_1 (q_1 + q_2) \tag{10}$$

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 (10) 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:

$$\pi_i = p q_i - \gamma (\hat{q}_i - q_i)^2, \quad \gamma > 0, \tag{11}$$

Substituting the inverse demand curve (10) into (11) lets us express the one-period payoff as

$$\pi_i(q_i, q_{-i}, \hat{q}_i) = a_0 q_i - a_1 q_i^2 - a_1 q_i q_{-i} - \gamma (\hat{q}_i - q_i)^2, \tag{12}$$

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}).

This completes our review of the duopoly model without concerns for robustness.

Now we activate robustness concerns of both firms.

To map a robust version of the duopoly model into coupled robust linear-quadratic dynamic programming problems, we again 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 (11) for the two firms in the duopoly model.

The law of motion for the state x_t is $x_{t+1} = Ax_t + B_1 u_{1t} + B_2 u_{2t}$ where

$$A := \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad B_1 := \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad B_2 := \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

A robust 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:

$$x_{t+1} = (A - B_1 F_1 - B_1 F_2) x_t \tag{13}$$

14.3.2 Parameters and Solution

Consider the duopoly model with parameter values of:

- $a_0 = 10$
- $a_1 = 2$
- $\beta = 0.96$
- $\gamma = 12$

From these, we computed the infinite horizon MPE without robustness using the code

```
import numpy as np
import quantecon as qe

# Parameters
a0 = 10.0
a1 = 2.0
β = 0.96
γ = 12.0
```

(continues on next page)

(continued from previous page)

```python
# In LQ form
A = np.eye(3)
B1 = np.array([[0.], [1.], [0.]])
B2 = np.array([[0.], [0.], [1.]])


R1 = [[      0.,     -a0 / 2,          0.],
      [-a0 / 2.,          a1,     a1 / 2.],
      [       0,     a1 / 2.,          0.]]

R2 = [[     0.,           0.,      -a0 / 2],
      [     0.,           0.,      a1 / 2.],
      [-a0 / 2,      a1 / 2.,           a1]]

Q1 = Q2 = γ
S1 = S2 = W1 = W2 = M1 = M2 = 0.0

# Solve using QE's nnash function
F1, F2, P1, P2 = qe.nnash(A, B1, B2, R1, R2, Q1,
                          Q2, S1, S2, W1, W2, M1,
                          M2, beta=β)

# Display policies
print("Computed policies for firm 1 and firm 2:\n")
print(f"F1 = {F1}")
print(f"F2 = {F2}")
print("\n")
```

```
Computed policies for firm 1 and firm 2:

F1 = [[-0.66846615  0.29512482  0.07584666]]
F2 = [[-0.66846615  0.07584666  0.29512482]]
```

Markov Perfect Equilibrium with Robustness

We add robustness concerns to the Markov Perfect Equilibrium model by extending the function `qe.nnash` (link) into a robustness version by adding the maximization operator $\mathcal{D}(P)$ into the backward induction.

The MPE with robustness function is `nnash_robust`.

The function's code is as follows

```python
def nnash_robust(A, C, B1, B2, R1, R2, Q1, Q2, S1, S2, W1, W2, M1, M2,
                 θ1, θ2, beta=1.0, tol=1e-8, max_iter=1000):

    """
    Compute the limit of a Nash linear quadratic dynamic game with
    robustness concern.

    In this problem, player i minimizes
    .. math::
        \sum_{t=0}^{\infty}
        \left\{
            x_t' r_i x_t + 2 x_t' w_i
```

(continues on next page)

(continued from previous page)

```
        u_{it} +u_{it}' q_i u_{it} + u_{jt}' s_i u_{jt} + 2 u_{jt}'
        m_i u_{it}
    \right\}
subject to the law of motion
.. math::
    x_{it+1} = A x_t + b_1 u_{1t} + b_2 u_{2t} + C w_{it+1}
and a perceived control law :math:`u_j(t) = - f_j x_t` for the other
player.

The player i also concerns about the model misspecification,
and maximizes
.. math::
    \sum_{t=0}^{\infty}
    \left\{
        \beta^{t+1} \theta_{i} w_{it+1}'w_{it+1}
    \right\}

The solution computed in this routine is the :math:`f_i` and
:math:`P_i` of the associated double optimal linear regulator
problem.

Parameters
----------
A : scalar(float) or array_like(float)
    Corresponds to the MPE equations, should be of size (n, n)
C : scalar(float) or array_like(float)
    As above, size (n, c), c is the size of w
B1 : scalar(float) or array_like(float)
    As above, size (n, k_1)
B2 : scalar(float) or array_like(float)
    As above, size (n, k_2)
R1 : scalar(float) or array_like(float)
    As above, size (n, n)
R2 : scalar(float) or array_like(float)
    As above, size (n, n)
Q1 : scalar(float) or array_like(float)
    As above, size (k_1, k_1)
Q2 : scalar(float) or array_like(float)
    As above, size (k_2, k_2)
S1 : scalar(float) or array_like(float)
    As above, size (k_1, k_1)
S2 : scalar(float) or array_like(float)
    As above, size (k_2, k_2)
W1 : scalar(float) or array_like(float)
    As above, size (n, k_1)
W2 : scalar(float) or array_like(float)
    As above, size (n, k_2)
M1 : scalar(float) or array_like(float)
    As above, size (k_2, k_1)
M2 : scalar(float) or array_like(float)
    As above, size (k_1, k_2)
θ1 : scalar(float)
        Robustness parameter of player 1
θ2 : scalar(float)
        Robustness parameter of player 2
beta : scalar(float), optional(default=1.0)
      Discount factor
```

(continues on next page)

(continued from previous page)

```
    tol : scalar(float), optional(default=1e-8)
         This is the tolerance level for convergence
    max_iter : scalar(int), optional(default=1000)
        This is the maximum number of iterations allowed

    Returns
    -------
    F1 : array_like, dtype=float, shape=(k_1, n)
        Feedback law for agent 1
    F2 : array_like, dtype=float, shape=(k_2, n)
        Feedback law for agent 2
    P1 : array_like, dtype=float, shape=(n, n)
        The steady-state solution to the associated discrete matrix
        Riccati equation for agent 1
    P2 : array_like, dtype=float, shape=(n, n)
        The steady-state solution to the associated discrete matrix
        Riccati equation for agent 2
    """

    # Unload parameters and make sure everything is a matrix
    params = A, C, B1, B2, R1, R2, Q1, Q2, S1, S2, W1, W2, M1, M2
    params = map(np.asmatrix, params)
    A, C, B1, B2, R1, R2, Q1, Q2, S1, S2, W1, W2, M1, M2 = params


    # Multiply A, B1, B2 by sqrt(β) to enforce discounting
    A, B1, B2 = [np.sqrt(β) * x for x in (A, B1, B2)]

    # Initial values
    n = A.shape[0]
    k_1 = B1.shape[1]
    k_2 = B2.shape[1]

    v1 = np.eye(k_1)
    v2 = np.eye(k_2)
    P1 = np.eye(n) * 1e-5
    P2 = np.eye(n) * 1e-5
    F1 = np.random.randn(k_1, n)
    F2 = np.random.randn(k_2, n)


    for it in range(max_iter):
        # Update
        F10 = F1
        F20 = F2

        I = np.eye(C.shape[1])

        # D1(P1)
        # Note: INV1 may not be solved if the matrix is singular
        INV1 = solve(θ1 * I - C.T @ P1 @ C, I)
        D1P1 =  P1 + P1 @ C @ INV1 @ C.T @ P1

        # D2(P2)
        # Note: INV2 may not be solved if the matrix is singular
        INV2 = solve(θ2 * I - C.T @ P2 @ C, I)
        D2P2 =  P2 + P2 @ C @ INV2 @ C.T @ P2
```

(continues on next page)

(continued from previous page)

```
        G2 = solve(Q2 + B2.T @ D2P2 @ B2, v2)
        G1 = solve(Q1 + B1.T @ D1P1 @ B1, v1)
        H2 = G2 @ B2.T @ D2P2
        H1 = G1 @ B1.T @ D1P1

        # Break up the computation of F1, F2
        F1_left = v1 - (H1 @ B2 + G1 @ M1.T) @ (H2 @ B1 + G2 @ M2.T)
        F1_right = H1 @ A + G1 @ W1.T - \
                   (H1 @ B2 + G1 @ M1.T) @ (H2 @ A + G2 @ W2.T)
        F1 = solve(F1_left, F1_right)
        F2 = H2 @ A + G2 @ W2.T - (H2 @ B1 + G2 @ M2.T) @ F1

        Λ1 = A - B2 @ F2
        Λ2 = A - B1 @ F1
        Π1 = R1 + F2.T @ S1 @ F2
        Π2 = R2 + F1.T @ S2 @ F1
        Γ1 = W1.T - M1.T @ F2
        Γ2 = W2.T - M2.T @ F1

        # Compute P1 and P2
        P1 = Π1 - (B1.T @ D1P1 @ Λ1 + Γ1).T @ F1 + \
             Λ1.T @ D1P1 @ Λ1
        P2 = Π2 - (B2.T @ D2P2 @ Λ2 + Γ2).T @ F2 + \
             Λ2.T @ D2P2 @ Λ2

        dd = np.max(np.abs(F10 - F1)) + np.max(np.abs(F20 - F2))

        if dd < tol:  # success!
            break

    else:
        raise ValueError(f'No convergence: Iteration limit of {max_iter} \
            reached in nnash')

    return F1, F2, P1, P2
```

14.3.3 Some Details

Firm i wants to minimize

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

where

$$x_t := \begin{bmatrix} 1 \\ q_{1t} \\ q_{2t} \end{bmatrix} \quad \text{and} \quad u_{it} := q_{i,t+1} - q_{it}, \quad i = 1,2$$

and

$$R_1 := \begin{bmatrix} 0 & -\frac{a_0}{2} & 0 \\ -\frac{a_0}{2} & a_1 & \frac{a_1}{2} \\ 0 & \frac{a_1}{2} & 0 \end{bmatrix}, \quad R_2 := \begin{bmatrix} 0 & 0 & -\frac{a_0}{2} \\ 0 & 0 & \frac{a_1}{2} \\ -\frac{a_0}{2} & \frac{a_1}{2} & a_1 \end{bmatrix}, \quad Q_1 = Q_2 = \gamma, \quad S_1 = S_2 = 0, \quad W_1 = W_2 = 0, \quad M_1 = M_2 = 0$$

The parameters of the duopoly model are:

- $a_0 = 10$
- $a_1 = 2$
- $\beta = 0.96$
- $\gamma = 12$

```
# 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
```

Consistency Check

We first conduct a comparison test to check if `nnash_robust` agrees with `qe.nnash` in the non-robustness case in which each $\theta_i \approx +\infty$

```
# 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)

# Solve using nnash_robust
F1r, F2r, P1r, P2r = nnash_robust(A, np.zeros((3, 1)), B1, B2, R1, R2, Q1,
                                  Q2, S1, S2, W1, W2, M1, M2, 1e-10,
                                  1e-10, beta=β)

print('F1 and F1r should be the same: ', np.allclose(F1, F1r))
print('F2 and F2r should be the same: ', np.allclose(F1, F1r))
print('P1 and P1r should be the same: ', np.allclose(P1, P1r))
print('P2 and P2r should be the same: ', np.allclose(P1, P1r))
```

```
F1 and F1r should be the same:  True
F2 and F2r should be the same:  True
P1 and P1r should be the same:  True
P2 and P2r should be the same:  True
```

We can see that the results are consistent across the two functions.

Comparative Dynamics under Baseline Transition Dynamics

We want to compare the dynamics of price and output under the baseline MPE model with those under the baseline model under the robust decision rules within the robust MPE.

This means that we simulate the state dynamics under the MPE equilibrium **closed-loop** transition matrix

$$A^o = A - B_1 F_1 - B_2 F_2$$

where F_1 and F_2 are the firms' robust decision rules within the robust markov_perfect equilibrium

- by simulating under the baseline model transition dynamics and the robust MPE rules we are in assuming that at the end of the day firms' concerns about misspecification of the baseline model do not materialize.
- a short way of saying this is that misspecification fears are all 'just in the minds' of the firms.
- simulating under the baseline model is a common practice in the literature.
- note that *some* assumption about the model that actually governs the data has to be made in order to create a simulation.
- later we will describe the (erroneous) beliefs of the two firms that justify their robust decisions as best responses to transition laws that are distorted relative to the baseline model.

After simulating x_t under the baseline transition dynamics and robust decision rules $F_i, i = 1, 2$, we extract and plot industry output $q_t = q_{1t} + q_{2t}$ and price $p_t = a_0 - a_1 q_t$.

Here we set the robustness and volatility matrix parameters as follows:

- $\theta_1 = 0.02$
- $\theta_2 = 0.04$
- $C = \begin{pmatrix} 0 \\ 0.01 \\ 0.01 \end{pmatrix}$

Because we have set $\theta_1 < \theta_2 < +\infty$ we know that

- both firms fear that the baseline specification of the state transition dynamics are incorrect.
- firm 1 fears misspecification more than firm 2.

```
# Robustness parameters and matrix
C = np.asmatrix([[0], [0.01], [0.01]])
θ1 = 0.02
θ2 = 0.04
n = 20


# Solve using nnash_robust
F1r, F2r, P1r, P2r = nnash_robust(A, C, B1, B2, R1, R2, Q1,
                                  Q2, S1, S2, W1, W2, M1, M2,
                                  θ1, θ2, beta=β)



# MPE output and price
AF = A - B1 @ F1 - B2 @ F2
x = np.empty((3, n))
x[:, 0] = 1, 1, 1
for t in range(n - 1):
```

(continues on next page)

(continued from previous page)

```
    x[:, t + 1] = AF @ x[:, t]
q1 = x[1, :]
q2 = x[2, :]
q = q1 + q2       # Total output, MPE
p = a0 - a1 * q   # Price, MPE


# RMPE output and price
AO = A - B1 @ F1r - B2 @ F2r
xr = np.empty((3, n))
xr[:, 0] = 1, 1, 1
for t in range(n - 1):
    xr[:, t+1] = AO @ xr[:, t]
qr1 = xr[1, :]
qr2 = xr[2, :]
qr = qr1 + qr2      # Total output, RMPE
pr = a0 - a1 * qr   # Price, RMPE

# RMPE heterogeneous beliefs output and price
I = np.eye(C.shape[1])
INV1 = solve(θ1 * I - C.T @ P1 @ C, I)
K1 =  P1 @ C @ INV1 @ C.T @ P1 @ AO
AOCK1 = AO + C.T @ K1

INV2 = solve(θ2 * I - C.T @ P2 @ C, I)
K2 =  P2 @ C @ INV2 @ C.T @ P2 @ AO
AOCK2 = AO + C.T @ K2
xrp1 = np.empty((3, n))
xrp2 = np.empty((3, n))
xrp1[:, 0] = 1, 1, 1
xrp2[:, 0] = 1, 1, 1
for t in range(n - 1):
    xrp1[:, t + 1] = AOCK1 @ xrp1[:, t]
    xrp2[:, t + 1] = AOCK2 @ xrp2[:, t]
qrp11 = xrp1[1, :]
qrp12 = xrp1[2, :]
qrp21 = xrp2[1, :]
qrp22 = xrp2[2, :]
qrp1 = qrp11 + qrp12       # Total output, RMPE from player 1's belief
qrp2 = qrp21 + qrp22       # Total output, RMPE from player 2's belief
prp1 = a0 - a1 * qrp1      # Price, RMPE from player 1's belief
prp2 = a0 - a1 * qrp2      # Price, RMPE from player 2's belief
```

The following code prepares graphs that compare market-wide output $q_{1t} + q_{2t}$ and the price of the good p_t under equilibrium decision rules $F_i, i = 1, 2$ from an ordinary Markov perfect equilibrium and the decision rules under a Markov perfect equilibrium with robust firms with multiplier parameters $\theta_i, i = 1, 2$ set as described above.

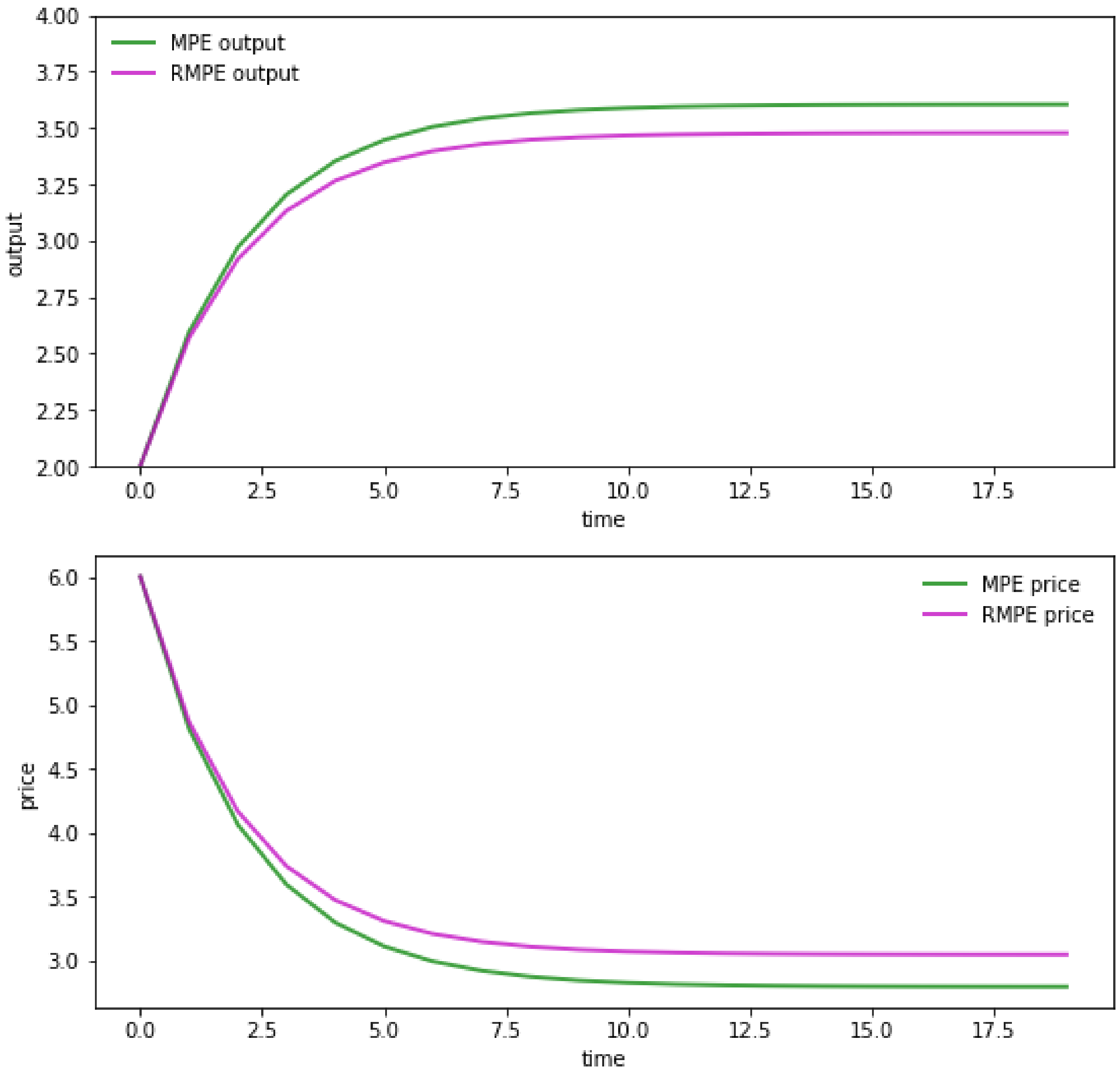
Both industry output and price are under the transition dynamics associated with the baseline model; only the decision rules F_i differ across the two equilibrium objects presented.

```
fig, axes = plt.subplots(2, 1, figsize=(9, 9))

ax = axes[0]
ax.plot(q, 'g-', lw=2, alpha=0.75, label='MPE output')
ax.plot(qr, 'm-', lw=2, alpha=0.75, label='RMPE output')
ax.set(ylabel="output", xlabel="time", ylim=(2, 4))
ax.legend(loc='upper left', frameon=0)
```

(continues on next page)

(continued from previous page)

```
ax = axes[1]
ax.plot(p, 'g-', lw=2, alpha=0.75, label='MPE price')
ax.plot(pr, 'm-', lw=2, alpha=0.75, label='RMPE price')
ax.set(ylabel="price", xlabel="time")
ax.legend(loc='upper right', frameon=0)
plt.show()
```

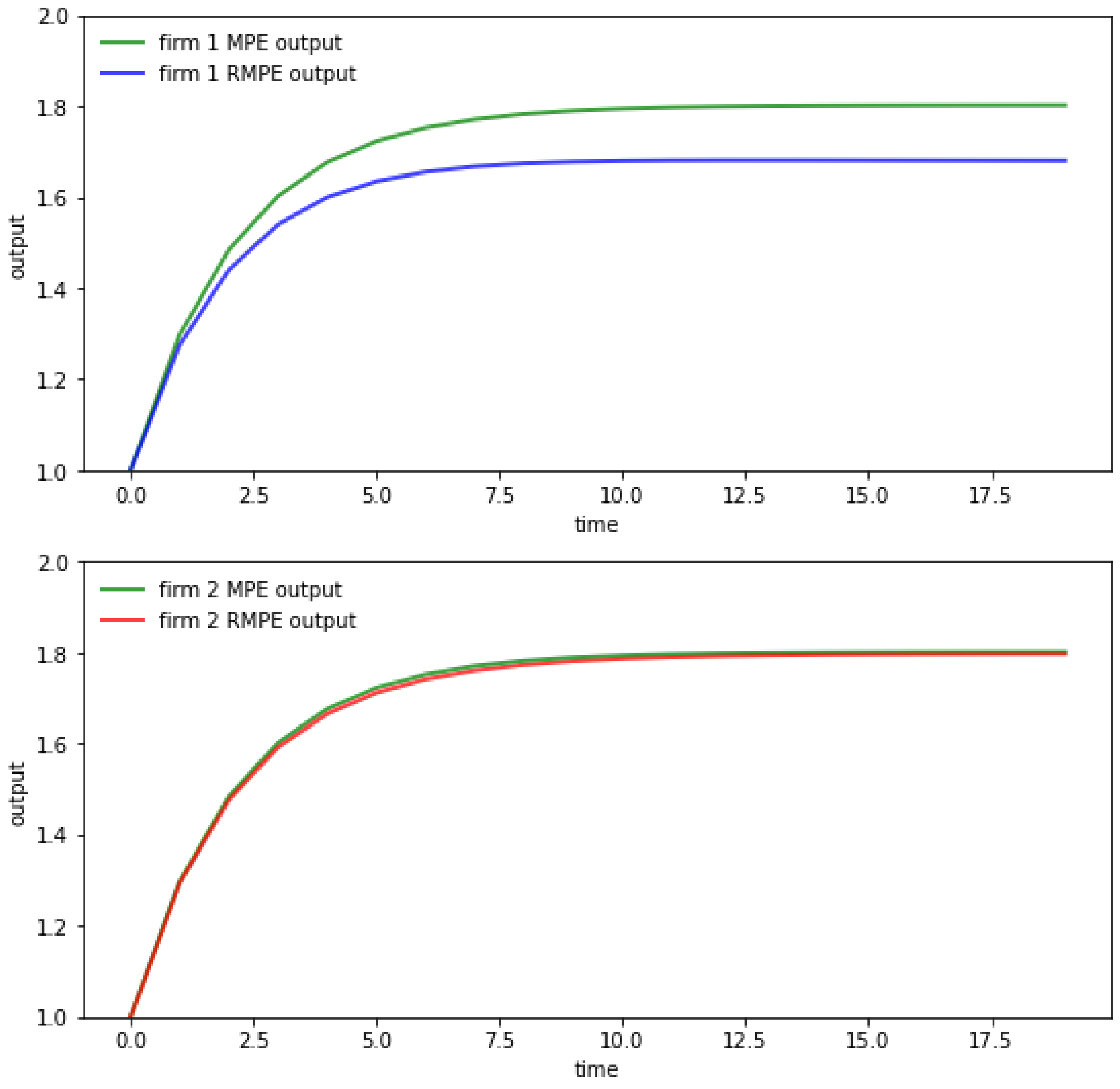


Under the dynamics associated with the baseline model, the price path is higher with the Markov perfect equilibrium robust decision rules than it is with decision rules for the ordinary Markov perfect equilibrium.

So is the industry output path.

To dig a little beneath the forces driving these outcomes, we want to plot q_{1t} and q_{2t} in the Markov perfect equilibrium with robust firms and to compare them with corresponding objects in the Markov perfect equilibrium without robust firms

```python
fig, axes = plt.subplots(2, 1, figsize=(9, 9))

ax = axes[0]
ax.plot(q1, 'g-', lw=2, alpha=0.75, label='firm 1 MPE output')
ax.plot(qr1, 'b-', lw=2, alpha=0.75, label='firm 1 RMPE output')
ax.set(ylabel="output", xlabel="time", ylim=(1, 2))
ax.legend(loc='upper left', frameon=0)

ax = axes[1]
ax.plot(q2, 'g-', lw=2, alpha=0.75, label='firm 2 MPE output')
ax.plot(qr2, 'r-', lw=2, alpha=0.75, label='firm 2 RMPE output')
ax.set(ylabel="output", xlabel="time", ylim=(1, 2))
ax.legend(loc='upper left', frameon=0)
plt.show()
```



Evidently, firm 1's output path is substantially lower when firms are robust firms while firm 2's output path is virtually the same as it would be in an ordinary Markov perfect equilibrium with no robust firms.

Recall that we have set $\theta_1 = .02$ and $\theta_2 = .04$, so that firm 1 fears misspecification of the baseline model substantially more than does firm 2

- but also please notice that firm 2's behavior in the Markov perfect equilibrium with robust firms responds to the decision rule $F_1 x_t$ employed by firm 1.
- thus it is something of a coincidence that its output is almost the same in the two equilibria.

Larger concerns about misspecification induce firm 1 to be more cautious than firm 2 in predicting market price and the output of the other firm.

To explore this, we study next how *ex-post* the two firms' beliefs about state dynamics differ in the Markov perfect equilibrium with robust firms.

(by *ex-post* we mean *after* extremization of each firm's intertemporal objective)

Heterogeneous Beliefs

As before, let $A^o = A - B\_1F\_1^r - B\_2F\_2^r$, where in a robust MPE, F_i^r is a robust decision rule for firm i.

Worst-case forecasts of x_t starting from $t = 0$ differ between the two firms.

This means that worst-case forecasts of industry output $q_{1t} + q_{2t}$ and price p_t also differ between the two firms.

To find these worst-case beliefs, we compute the following three "closed-loop" transition matrices

- A^o
- $A^o + CK\_1$
- $A^o + CK\_2$

We call the first transition law, namely, A^o, the baseline transition under firms' robust decision rules.

We call the second and third worst-case transitions under robust decision rules for firms 1 and 2.

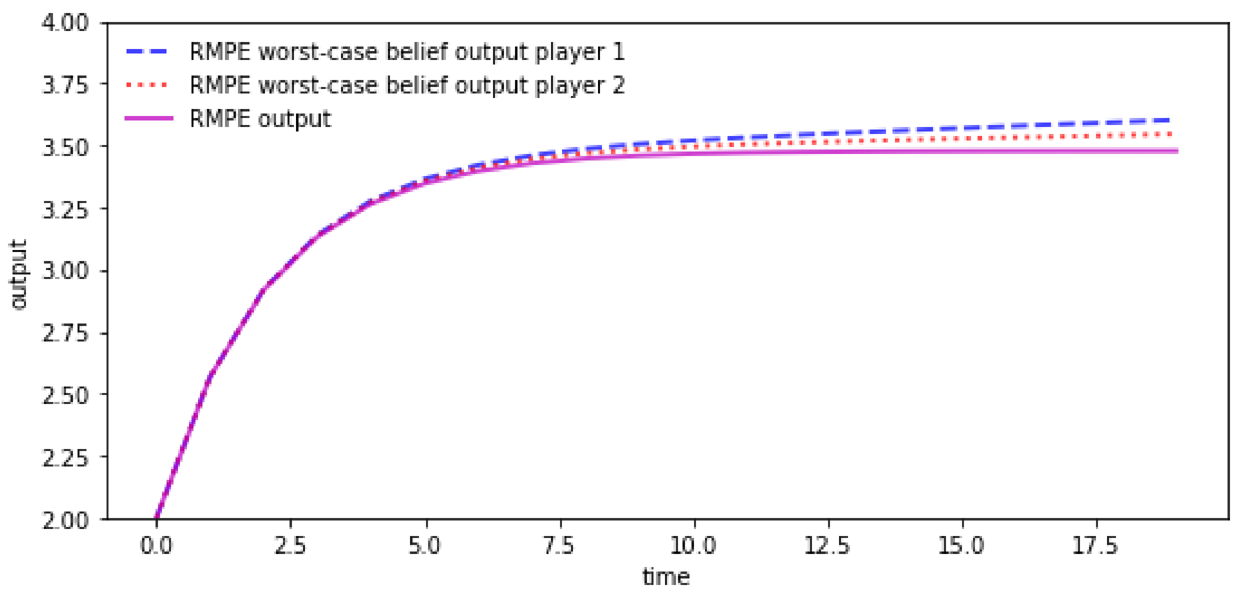
From $\{x_t\}$ paths generated by each of these transition laws, we pull off the associated price and total output sequences.

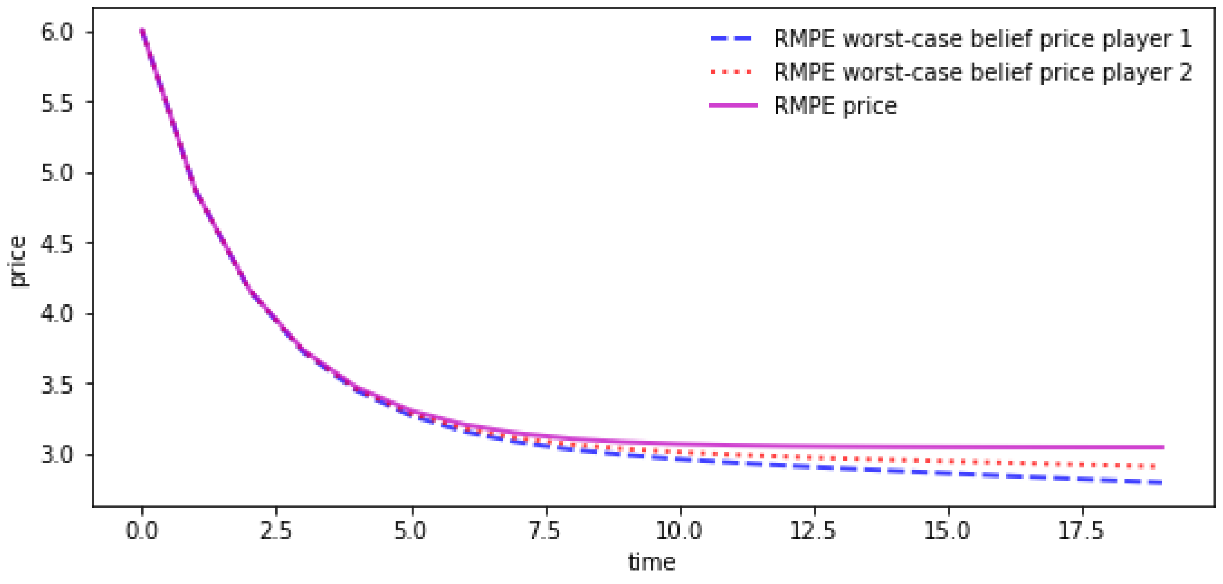
The following code plots them

```
print('Baseline Robust transition matrix AO is: \n', np.round(AO, 3))
print('Player 1\'s worst-case transition matrix AOCK1 is: \n', \
np.round(AOCK1, 3))
print('Player 2\'s worst-case transition matrix AOCK2 is: \n', \
np.round(AOCK2, 3))
```

```
Baseline Robust transition matrix AO is:
 [[ 1.     0.     0.   ]
 [ 0.666  0.682 -0.074]
 [ 0.671 -0.071  0.694]]
Player 1's worst-case transition matrix AOCK1 is:
 [[ 0.998  0.002  0.   ]
 [ 0.664  0.685 -0.074]
 [ 0.669 -0.069  0.694]]
Player 2's worst-case transition matrix AOCK2 is:
 [[ 0.999  0.     0.001]
 [ 0.665  0.683 -0.073]
 [ 0.67  -0.071  0.695]]
```

```
# == Plot == #
fig, axes = plt.subplots(2, 1, figsize=(9, 9))

ax = axes[0]
ax.plot(qrp1, 'b--', lw=2, alpha=0.75,
    label='RMPE worst-case belief output player 1')
ax.plot(qrp2, 'r:', lw=2, alpha=0.75,
    label='RMPE worst-case belief output player 2')
ax.plot(qr, 'm-', lw=2, alpha=0.75, label='RMPE output')
ax.set(ylabel="output", xlabel="time", ylim=(2, 4))
ax.legend(loc='upper left', frameon=0)

ax = axes[1]
ax.plot(prp1, 'b--', lw=2, alpha=0.75,
    label='RMPE worst-case belief price player 1')
ax.plot(prp2, 'r:', lw=2, alpha=0.75,
    label='RMPE worst-case belief price player 2')
ax.plot(pr, 'm-', lw=2, alpha=0.75, label='RMPE price')
ax.set(ylabel="price", xlabel="time")
ax.legend(loc='upper right', frameon=0)
plt.show()
```





We see from the above graph that under robustness concerns, player 1 and player 2 have heterogeneous beliefs about total output and the goods price even though they share the same baseline model and information

- firm 1 thinks that total output will be higher and price lower than does firm 2
- this leads firm 1 to produce less than firm 2

These beliefs justify (or **rationalize**) the Markov perfect equilibrium robust decision rules.

This means that the robust rules are the unique **optimal** rules (or best responses) to the indicated worst-case transition dynamics.

([HS08a] discuss how this property of robust decision rules is connected to the concept of *admissibility* in Bayesian statistical decision theory)

CHAPTER

FIFTEEN

DEFAULT RISK AND INCOME FLUCTUATIONS

Contents

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

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

15.1 Overview

This lecture computes versions of Arellano's [Are08] 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
import random

from numba import jit, int64, float64
from numba.experimental import jitclass
%matplotlib inline
```

15.2 Structure

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

15.2.1 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

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t u(c_t) \tag{1}$$

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 (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.

15.2.2 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

$$c = y + B - q(B', y)B' \tag{2}$$

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.

15.2.3 Financial Markets

Foreign creditors

- are risk neutral
- know the domestic output stochastic process $\{y_t\}$ and observe $y_t, y_{t-1}, \dots$, 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

$$q = \frac{1 - \delta}{1 + r} \tag{3}$$

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

15.2.4 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.

15.2.5 Reentering International Credit Market

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

15.3 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

$$\delta(B', y) := \int \mathbb{1}\{v_c(B', y') < v_d(y')\} p(y, y') dy' \tag{4}$$

Given zero profits for foreign creditors in equilibrium, we can combine (3) and (4) to pin down the bond price function:

$$q(B', y) = \frac{1 - \delta(B', y)}{1 + r} \tag{5}$$

15.3.1 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 (5)

15.4 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 [Are08] 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 value function $v(B, y)$ and price function $q(B', y)$.
2. At each pair (B, y),
   - update the value of defaulting $v_d(y)$.
   - update the value of continuing $v_c(B, y)$.
3. Update the value function $v(B, y)$, the default rule, the implied ex ante default probability, and the price function.
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 Tauchen's quadrature method.

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

We start by defining the data structure that will help us compile the class (for more information on why we do this, see the lecture on numba.)

```python
# Define the data information for the jitclass
arellano_data = [
    ('B', float64[:]), ('P', float64[:, :]), ('y', float64[:]),
    ('β', float64), ('γ', float64), ('r', float64),
    ('ρ', float64), ('η', float64), ('θ', float64),
    ('def_y', float64[:])
]

# Define utility function
@jit(nopython=True)
def u(c, γ):
    return c**(1-γ)/(1-γ)
```

We then define our `jitclass` that will store various parameters and contain the code that can apply the Bellman operators and determine the savings policy given prices and value functions

```python
@jitclass(arellano_data)
class Arellano_Economy:
    """
    Arellano 2008 deals with a small open economy whose government
    invests in foreign assets in order to smooth the consumption of
    domestic households. Domestic households receive a stochastic
    path of income.

    Parameters
    ----------
    B : vector(float64)
        A grid for bond holdings
    P : matrix(float64)
        The transition matrix for a country's output
    y : vector(float64)
        The possible output states
```

(continues on next page)

(continued from previous page)

```
    β : float
        Time discounting parameter
    γ : float
        Risk-aversion parameter
    r : float
        int lending rate
    ρ : float
        Persistence in the income process
    η : float
        Standard deviation of the income process
    θ : float
        Probability of re-entering financial markets in each period
    """

    def __init__(
            self, B, P, y,
            β=0.953, γ=2.0, r=0.017,
            ρ=0.945, η=0.025, θ=0.282
        ):

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

        # Compute the mean output
        self.def_y = np.minimum(0.969 * np.mean(y), y)

    def bellman_default(self, iy, EVd, EV):
        """
        The RHS of the Bellman equation when the country is in a
        defaulted state on their debt
        """
        # Unpack certain parameters for simplification
        β, γ, θ = self.β, self.γ, self.θ

        # Compute continuation value
        zero_ind = len(self.B) // 2
        cont_value = θ * EV[iy, zero_ind] + (1 - θ) * EVd[iy]

        return u(self.def_y[iy], γ) + β*cont_value

    def bellman_nondefault(self, iy, iB, q, EV, iB_tp1_star=-1):
        """
        The RHS of the Bellman equation when the country is not in a
        defaulted state on their debt
        """
        # Unpack certain parameters for simplification
        β, γ, θ = self.β, self.γ, self.θ
        B, y = self.B, self.y

        # Compute the RHS of Bellman equation
        if iB_tp1_star < 0:
            iB_tp1_star = self.compute_savings_policy(iy, iB, q, EV)
        c = max(y[iy] - q[iy, iB_tp1_star]*B[iB_tp1_star] + B[iB], 1e-14)

        return u(c, γ) + β*EV[iy, iB_tp1_star]
```

(continues on next page)

(continued from previous page)

```
    def compute_savings_policy(self, iy, iB, q, EV):
        """
        Finds the debt/savings that maximizes the value function
        for a particular state given prices and a value function
        """
        # Unpack certain parameters for simplification
        β, γ, θ = self.β, self.γ, self.θ
        B, y = self.B, self.y

        # Compute the RHS of Bellman equation
        current_max = -1e14
        iB_tp1_star = 0
        for iB_tp1, B_tp1 in enumerate(B):
            c = max(y[iy] - q[iy, iB_tp1]*B[iB_tp1] + B[iB], 1e-14)
            m = u(c, γ) + β*EV[iy, iB_tp1]

            if m > current_max:
                iB_tp1_star = iB_tp1
                current_max = m

        return iB_tp1_star
```

We can now write a function that will use this class to compute the solution to our model

```
@jit(nopython=True)
def solve(model, tol=1e-8, maxiter=10_000):
    """
    Given an Arellano_Economy type, this function computes the optimal
    policy and value functions
    """
    # Unpack certain parameters for simplification
    β, γ, r, θ = model.β, model.γ, model.r, model.θ
    B = np.ascontiguousarray(model.B)
    P, y = np.ascontiguousarray(model.P), np.ascontiguousarray(model.y)
    nB, ny = B.size, y.size

    # Allocate space
    iBstar = np.zeros((ny, nB), int64)
    default_prob = np.zeros((ny, nB))
    default_states = np.zeros((ny, nB))
    q = np.full((ny, nB), 0.95)
    Vd = np.zeros(ny)
    Vc, V, Vupd = np.zeros((ny, nB)), np.zeros((ny, nB)), np.zeros((ny, nB))

    it = 0
    dist = 10.0
    while (it < maxiter) and (dist > tol):

        # Compute expectations used for this iteration
        EV = P@V
        EVd = P@Vd

        for iy in range(ny):
            # Update value function for default state
            Vd[iy] = model.bellman_default(iy, EVd, EV)
```

(continues on next page)

(continued from previous page)

```
            for iB in range(nB):
                # Update value function for non-default state
                iBstar[iy, iB] = model.compute_savings_policy(iy, iB, q, EV)
                Vc[iy, iB] = model.bellman_nondefault(iy, iB, q, EV, iBstar[iy, iB])

        # Once value functions are updated, can combine them to get
        # the full value function
        Vd_compat = np.reshape(np.repeat(Vd, nB), (ny, nB))
        Vupd[:, :] = np.maximum(Vc, Vd_compat)

        # Can also compute default states and update prices
        default_states[:, :] = 1.0 * (Vd_compat > Vc)
        default_prob[:, :] = P @ default_states
        q[:, :] = (1 - default_prob) / (1 + r)

        # Check tolerance etc...
        dist = np.max(np.abs(Vupd - V))
        V[:, :] = Vupd[:, :]
        it += 1

    return V, Vc, Vd, iBstar, default_prob, default_states, q
```

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

```
def simulate(model, T, default_states, iBstar, q, y_init=None, B_init=None):
    """
    Simulates the Arellano 2008 model of sovereign debt

    Parameters
    ----------
    model: Arellano_Economy
        An instance of the Arellano model with the corresponding parameters
    T: integer
        The number of periods that the model should be simulated
    default_states: array(float64, 2)
        A matrix of 0s and 1s that denotes whether the country was in
        default on their debt in that period (default = 1)
    iBstar: array(float64, 2)
        A matrix which specifies the debt/savings level that a country holds
        during a given state
    q: array(float64, 2)
        A matrix that specifies the price at which a country can borrow/save
        for a given state
    y_init: integer
        Specifies which state the income process should start in
    B_init: integer
        Specifies which state the debt/savings state should start

    Returns
    -------
    y_sim: array(float64, 1)
        A simulation of the country's income
    B_sim: array(float64, 1)
        A simulation of the country's debt/savings
    q_sim: array(float64, 1)
        A simulation of the price required to have an extra unit of
        consumption in the following period
```

(continues on next page)

(continued from previous page)

```
    default_sim: array(bool, 1)
        A simulation of whether the country was in default or not
    """
    # Find index i such that Bgrid[i] is approximately 0
    zero_B_index = np.searchsorted(model.B, 0.0)

    # Set initial conditions
    in_default = False
    max_y_default = 0.969 * np.mean(model.y)
    if y_init == None:
        y_init = np.searchsorted(model.y, model.y.mean())
    if B_init == None:
        B_init = zero_B_index

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

    # Allocate memory for remaining outputs
    Bi = B_init
    B_sim = np.empty(T)
    y_sim = np.empty(T)
    q_sim = np.empty(T)
    default_sim = np.empty(T, dtype=bool)

    # Perform simulation
    for t in range(T):
        yi = y_sim_indices[t]

        # Fill y/B for today
        if not in_default:
            y_sim[t] = model.y[yi]
        else:
            y_sim[t] = np.minimum(model.y[yi], max_y_default)
        B_sim[t] = model.B[Bi]
        default_sim[t] = in_default

        # Check whether in default and branch depending on that state
        if not in_default:
            if default_states[yi, Bi] > 1e-4:
                in_default=True
                Bi_next = zero_B_index
            else:
                Bi_next = iBstar[yi, Bi]
        else:
            Bi_next = zero_B_index
            if np.random.rand() < model.θ:
                in_default=False

        # Fill in states
        q_sim[t] = q[yi, Bi_next]
        Bi = Bi_next

    return y_sim, B_sim, q_sim, default_sim
```

15.5 Results

Let's start by trying to replicate the results obtained in [Are08].

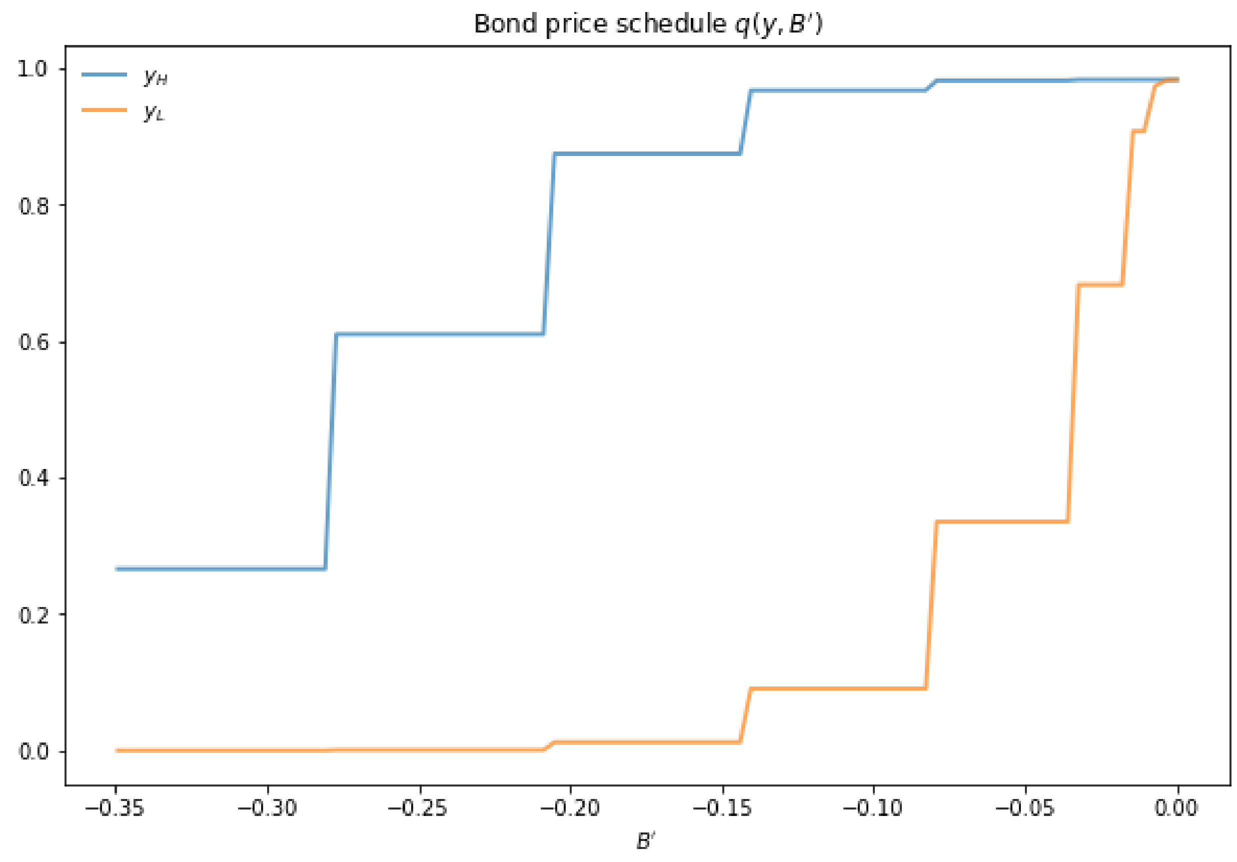
In what follows, all results are computed using Arellano's parameter values.

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.



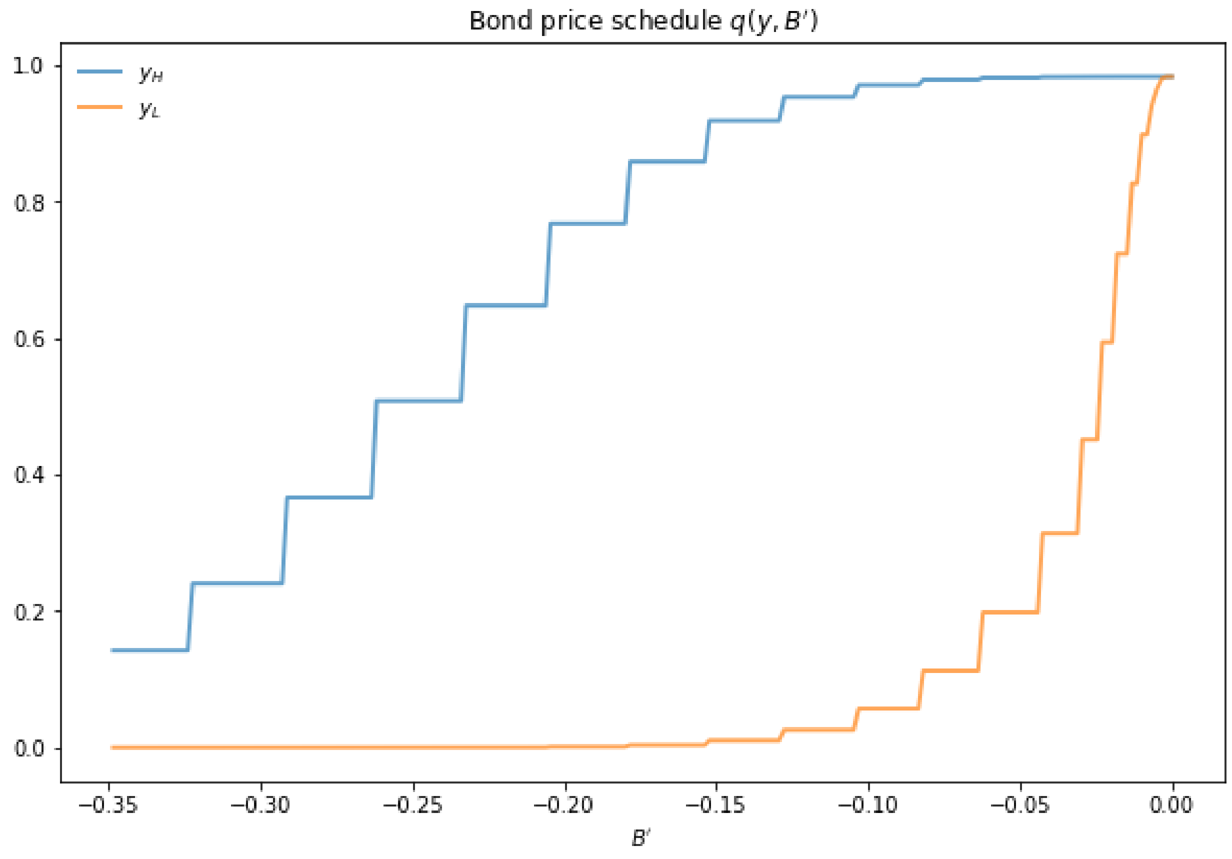
- y_L is 5% below the mean of the y grid values
- y_H is 5% above the mean of the y grid values

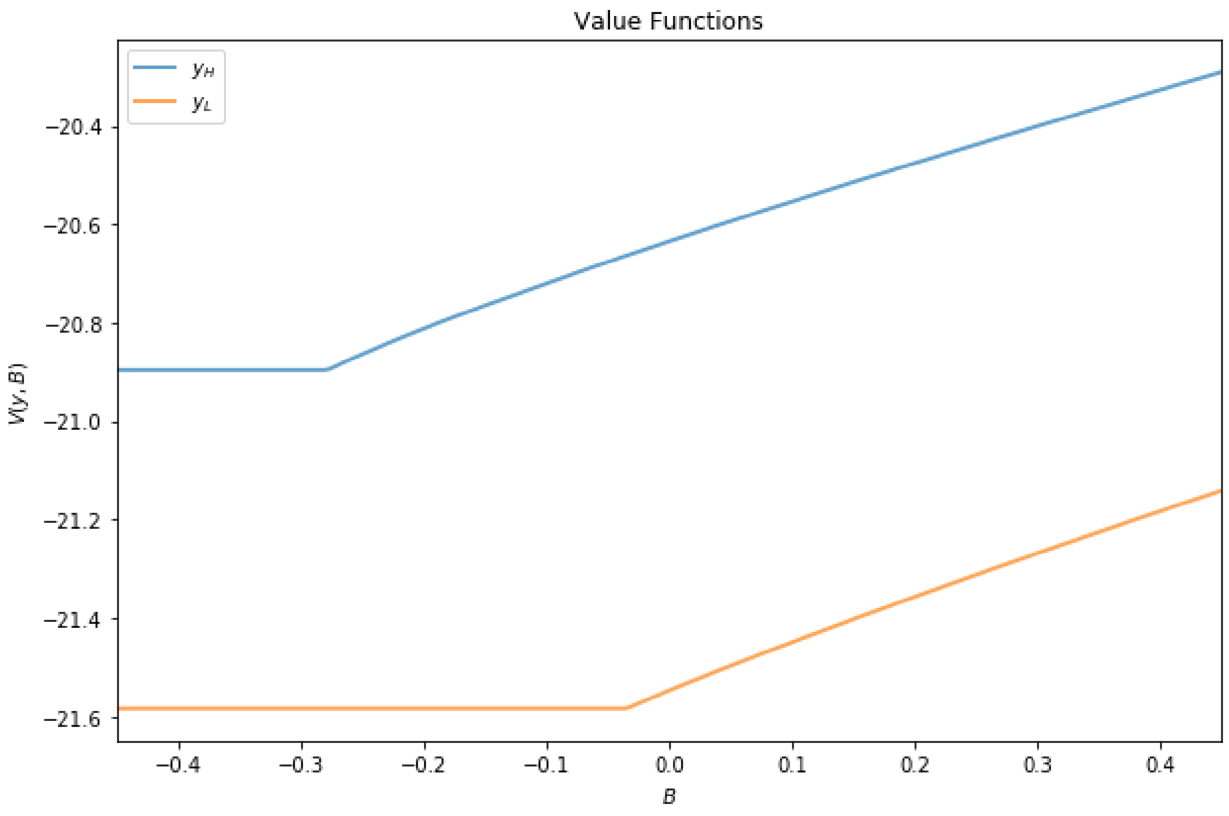
The grid used to compute this figure was relatively coarse (`ny, nB = 21, 251`) in order to match Arrelano's findings.

Here's the same relationships computed on a finer grid (`ny, nB = 51, 551`)

In either case, 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.

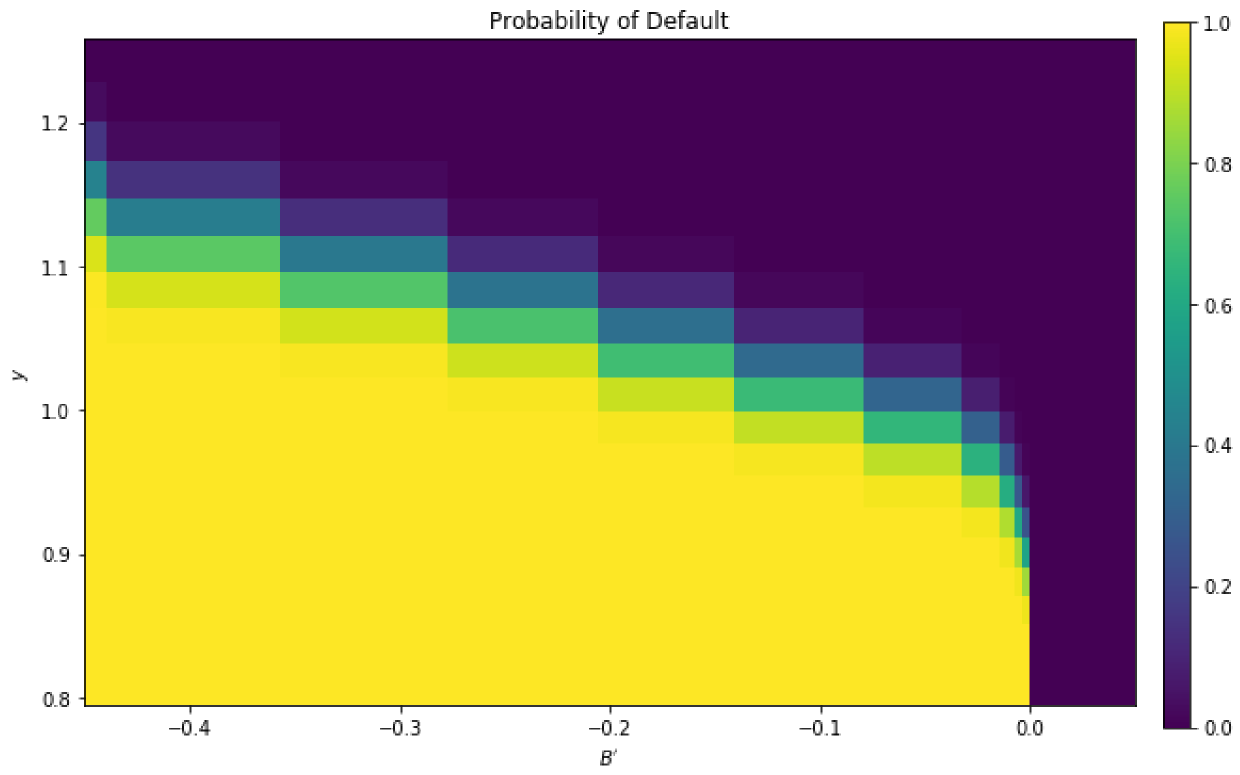

Bond price schedule $q(y, B')$
y_H
y_L
1.0
0.8
0.6
0.4
0.2
0.0
−0.35
−0.30
−0.25
−0.20
−0.15
−0.10
−0.05
0.00
B'


Value Functions
y_H
y_L
$V(y, B)$
−20.4
−20.6
−20.8
−21.0
−21.2
−21.4
−21.6
−0.4
−0.3
−0.2
−0.1
0.0
0.1
0.2
0.3
0.4
B

The next figure plots value functions and replicates the right hand panel of Figure 4 of [Are08].

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

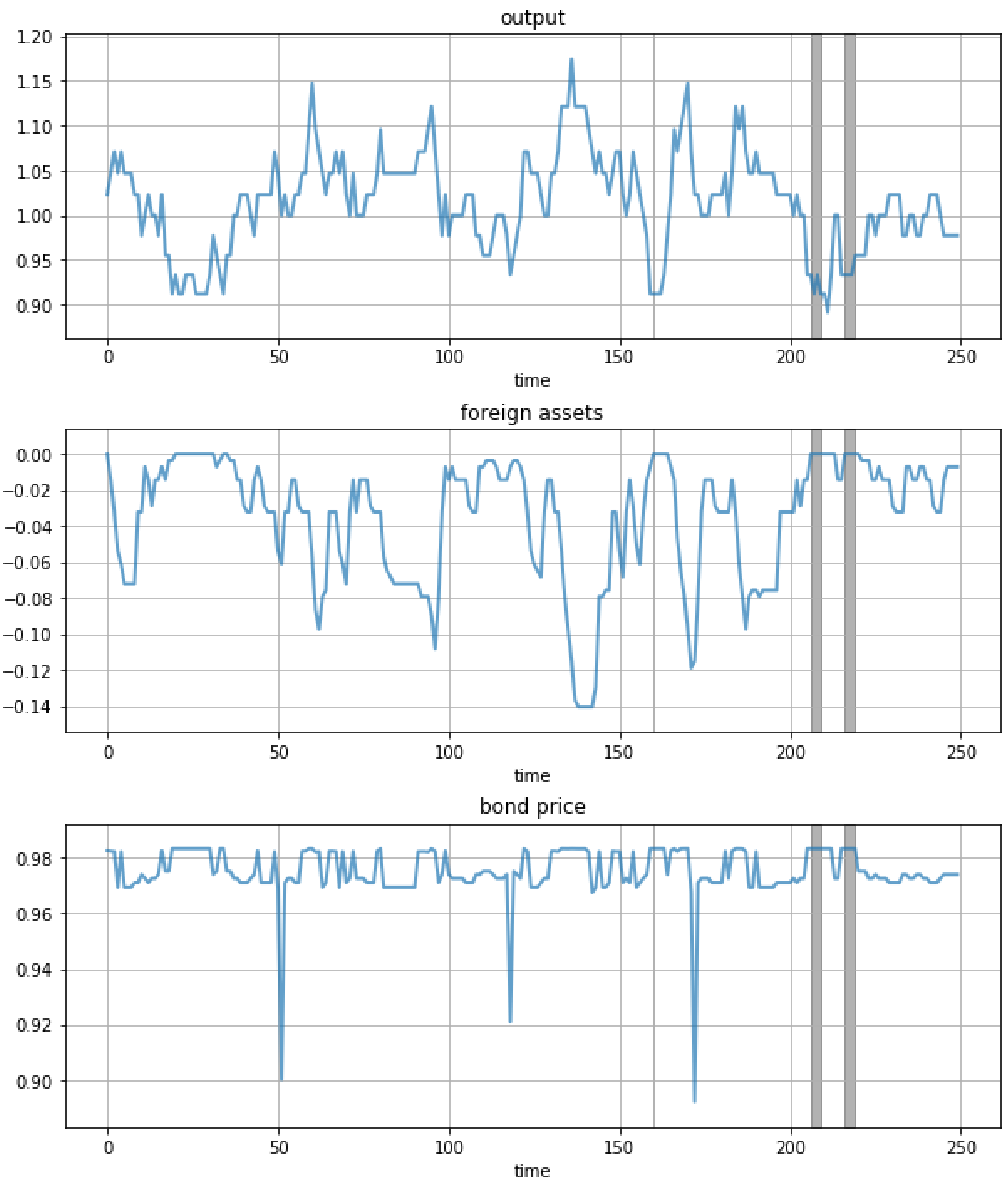
Periods of relative stability are followed by sharp spikes in the discount rate on government debt.

15.6 Exercises

15.6.1 Exercise 1

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

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


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
time
bond price
0.98
0.96
0.94
0.92
0.90
time

15.7 Solutions

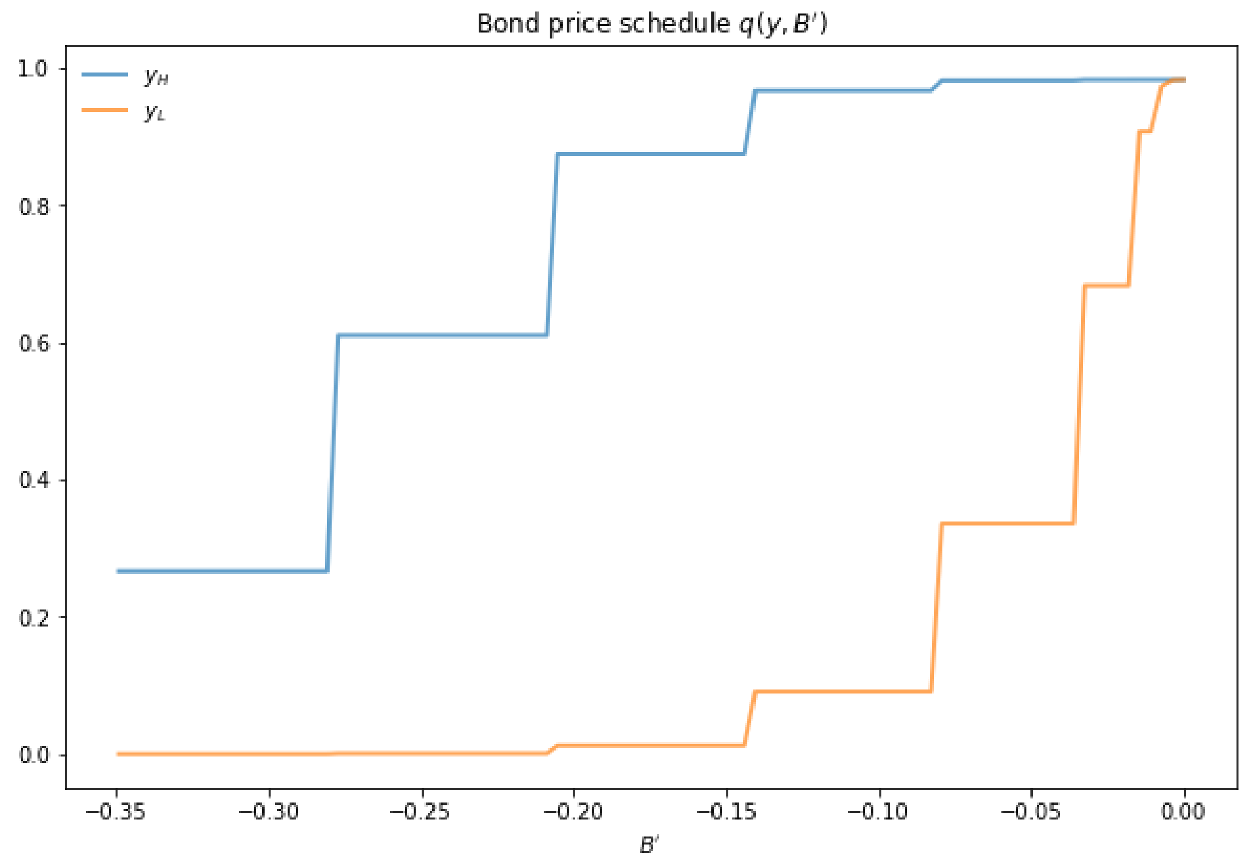
Compute the value function, policy and equilibrium prices

```python
β, γ, r = 0.953, 2.0, 0.017
ρ, η, θ = 0.945, 0.025, 0.282
ny = 21
nB = 251
Bgrid = np.linspace(-0.45, 0.45, nB)
mc = qe.markov.tauchen(ρ, η, 0, 3, ny)
ygrid, P = np.exp(mc.state_values), mc.P

ae = Arellano_Economy(
    Bgrid, P, ygrid, β=β, γ=γ, r=r, ρ=ρ, η=η, θ=θ
)
```

```python
V, Vc, Vd, iBstar, default_prob, default_states, q = solve(ae)
```

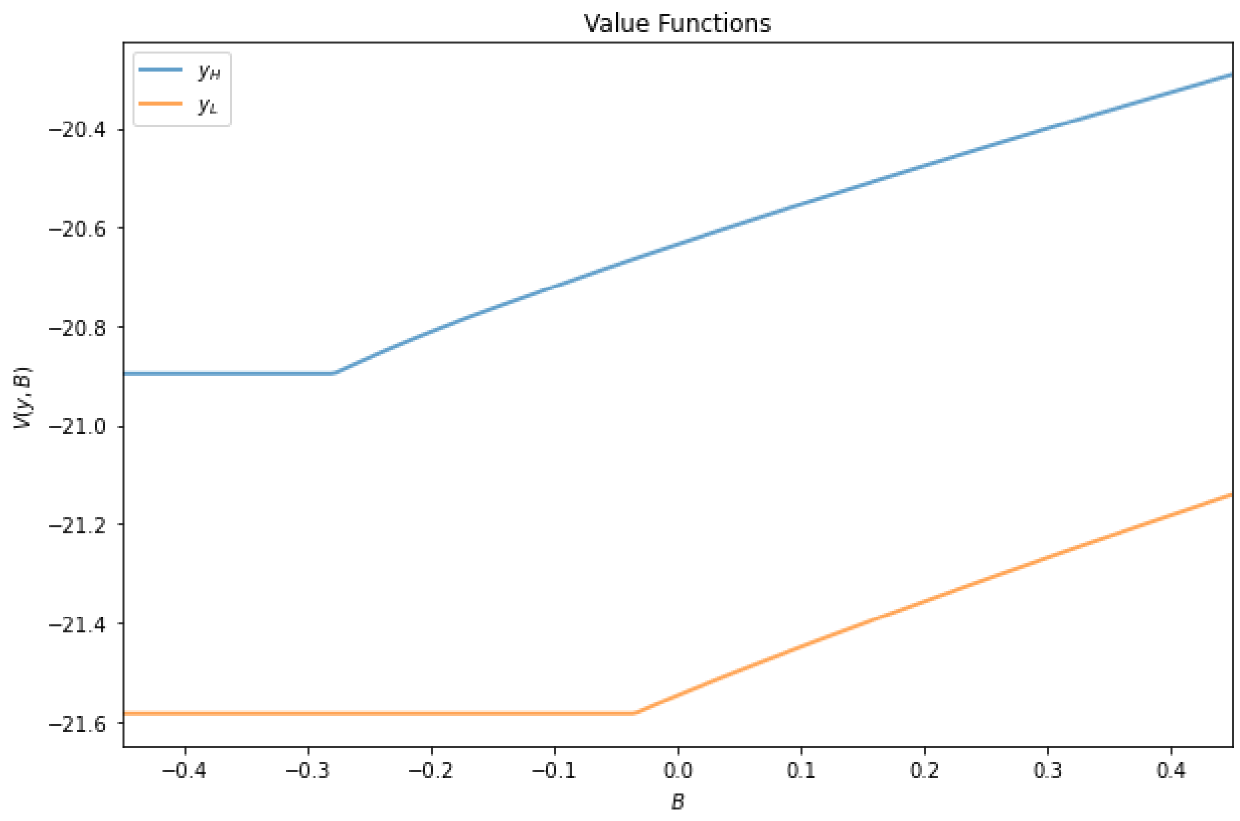
Compute the bond price schedule as seen in figure 3 of Arellano (2008)

```python
# Create "Y High" and "Y Low" values as 5% devs from mean
high, low = np.mean(ae.y) * 1.05, np.mean(ae.y) * .95
iy_high, iy_low = (np.searchsorted(ae.y, 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 in range(nB):
    b = ae.B[i]
    if -0.35 <= b <= 0:  # To match fig 3 of Arellano
        x.append(b)
        q_low.append(q[iy_low, i])
        q_high.append(q[iy_high, i])
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

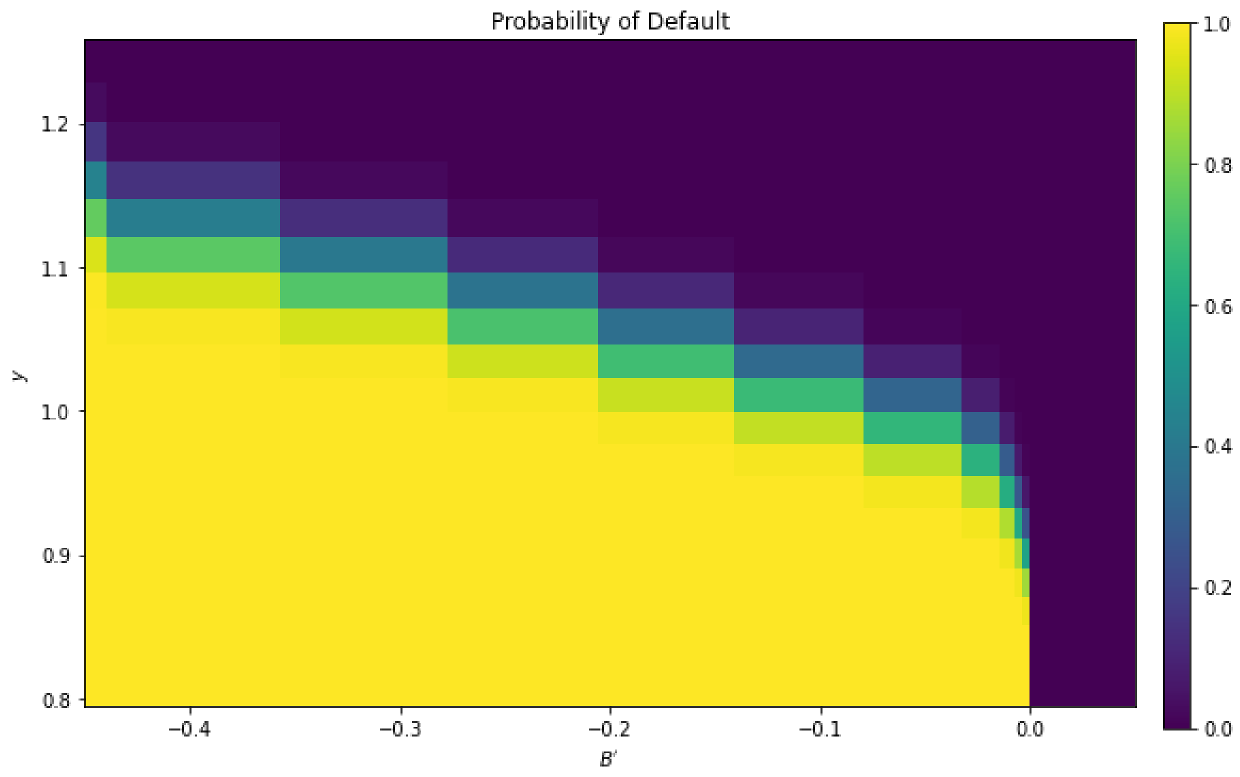
```
# Create "Y High" and "Y Low" values as 5% devs from mean
high, low = np.mean(ae.y) * 1.05, np.mean(ae.y) * .95
iy_high, iy_low = (np.searchsorted(ae.y, x) for x in (high, low))

fig, ax = plt.subplots(figsize=(10, 6.5))
ax.set_title("Value Functions")
ax.plot(ae.B, V[iy_high], label="$y_H$", lw=2, alpha=0.7)
ax.plot(ae.B, 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(ae.B.min(), ae.B.max())
plt.show()
```



Draw a heat map for default probability

```
xx, yy = ae.B, ae.y
zz = default_prob

# Create figure
fig, ax = plt.subplots(figsize=(10, 6.5))
hm = ax.pcolormesh(xx, yy, zz)
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()
```

```
<ipython-input-11-3377b9e4ed2e>:6: MatplotlibDeprecationWarning: shading='flat' when␣
↪X and Y have the same dimensions as C is deprecated since 3.3.  Either specify the␣
↪corners of the quadrilaterals with X and Y, or pass shading='auto', 'nearest' or
↪'gouraud', or set rcParams['pcolor.shading'].  This will become an error two minor␣
↪releases later.
  hm = ax.pcolormesh(xx, yy, zz)
```



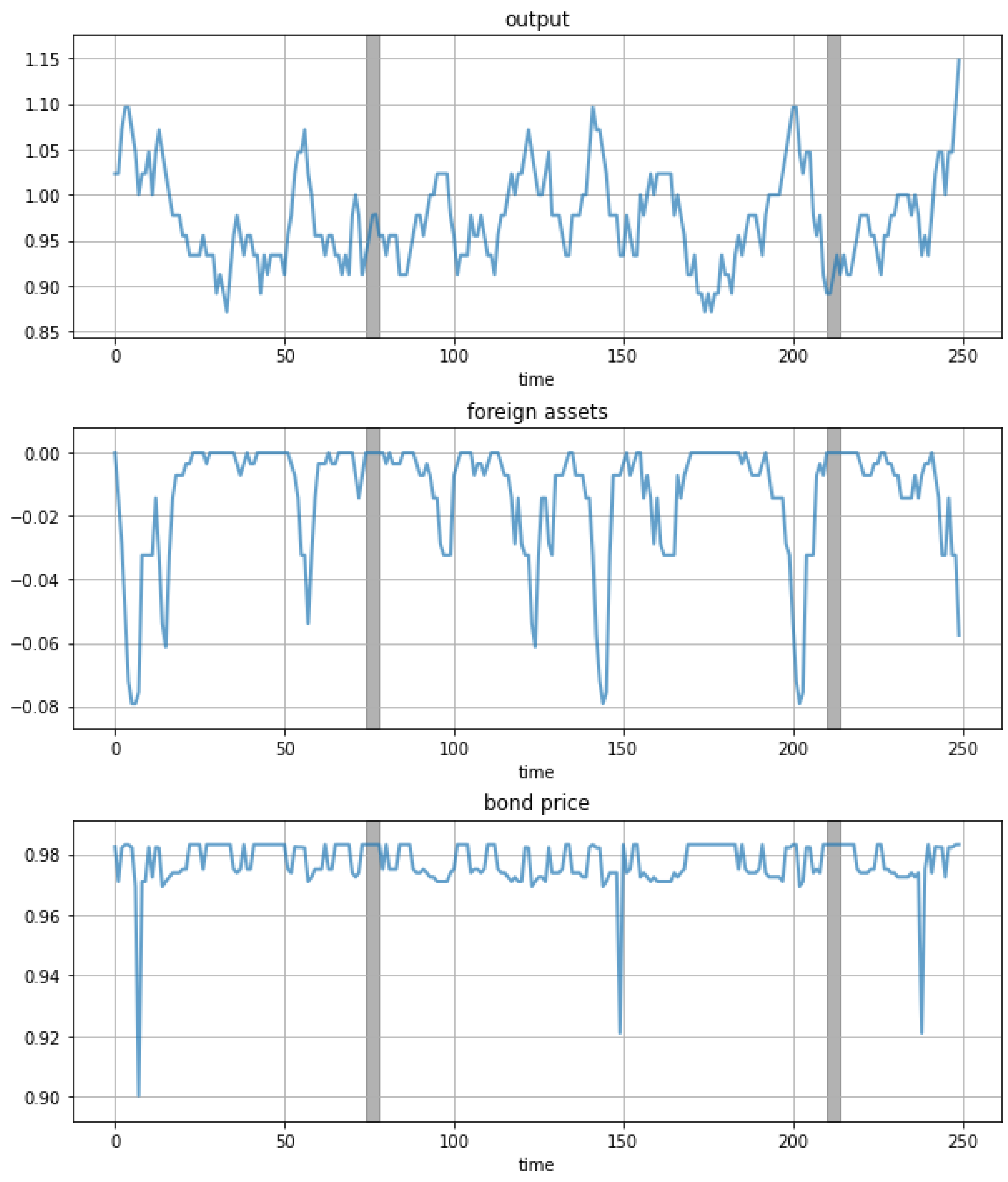
Plot a time series of major variables simulated from the model

```
T = 250

np.random.seed(42)
y_vec, B_vec, q_vec, default_vec = simulate(ae, T, default_states, iBstar, q)

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

plot_series = (y_vec, B_vec, q_vec)
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)
```

(continues on next page)

(continued from previous page)

```
    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)
    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
0.98
0.96
0.94
0.92
0.90
0
50
100
150
200
250
time

CHAPTER

SIXTEEN

GLOBALIZATION AND CYCLES

Contents

16.1 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
import seaborn as sns
```

(continues on next page)

(continued from previous page)

```
from numba import jit, vectorize
from ipywidgets import interact
```

16.1.1 Background

The model builds on work by Judd [Jud85], Deneckner and Judd [DJ92] and Helpman and Krugman [HK85] 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.

16.2 Key Ideas

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

16.2.1 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.

16.2.2 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.

16.3 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 $t = 0, 1, \ldots$.

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, $\Omega_t = \Omega^c_t + \Omega^m_t$.

16.3.1 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

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

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.

16.3.2 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

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

16.3.3 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 $F : \mathbb{R}^2_+ \rightarrow \mathbb{R}^2_+$ 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

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

16.4 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

```python
@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)

```python
    """
    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

    Returns
```

(continues on next page)

(continued from previous page)

```
    -------
    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

    return time_2_sync
```

(continues on next page)

(continued from previous page)

```python
# == 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
        ----------
        n1_0 : scalar(Float)
            Initial normalized measure of firms in country one
        n2_0 : scalar(Float)
```

(continues on next page)

(continued from previous page)

```python
            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
        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)

```python
        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
```

16.4.1 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.

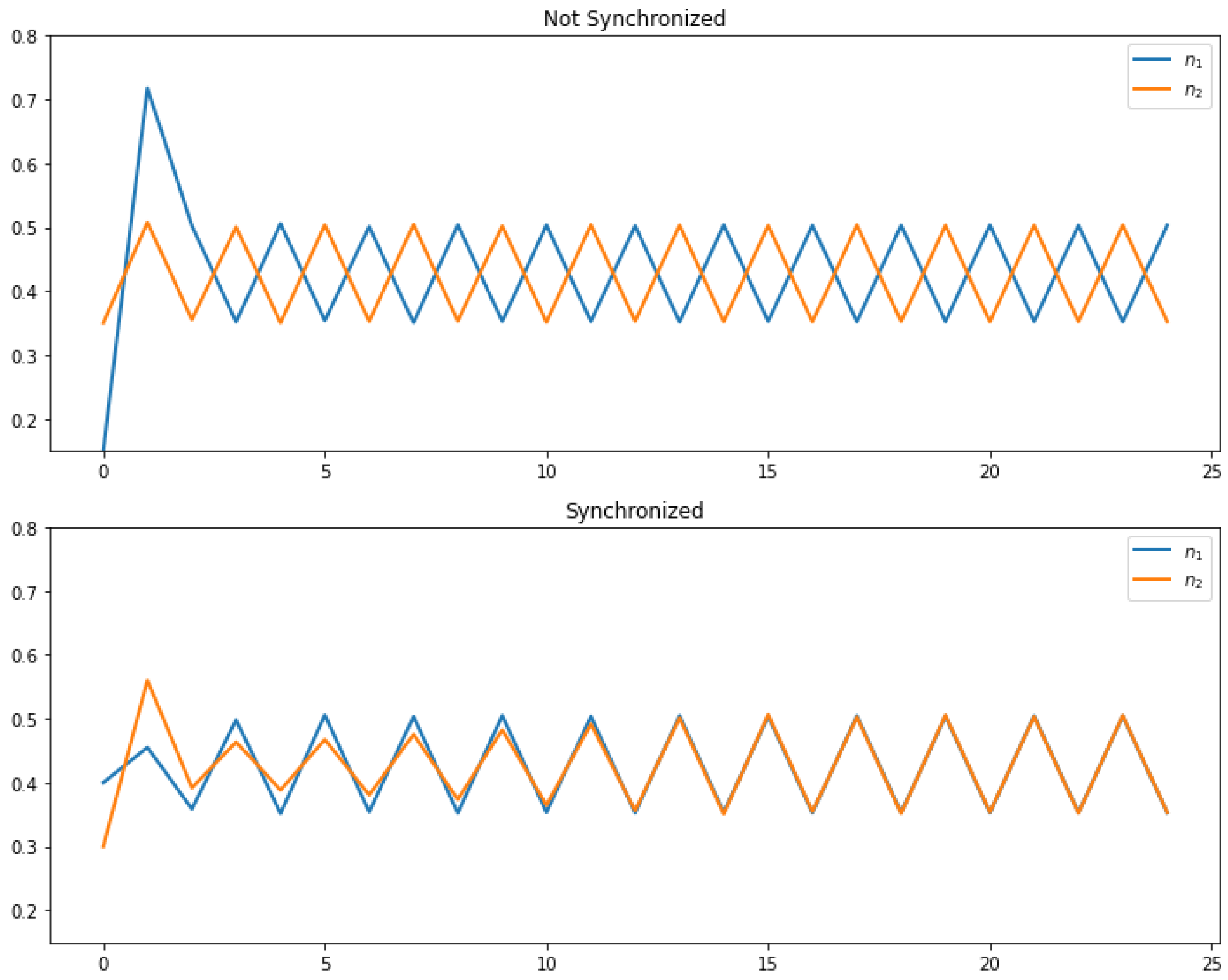
Here's the function

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

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

(continues on next page)

(continued from previous page)

```
    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.

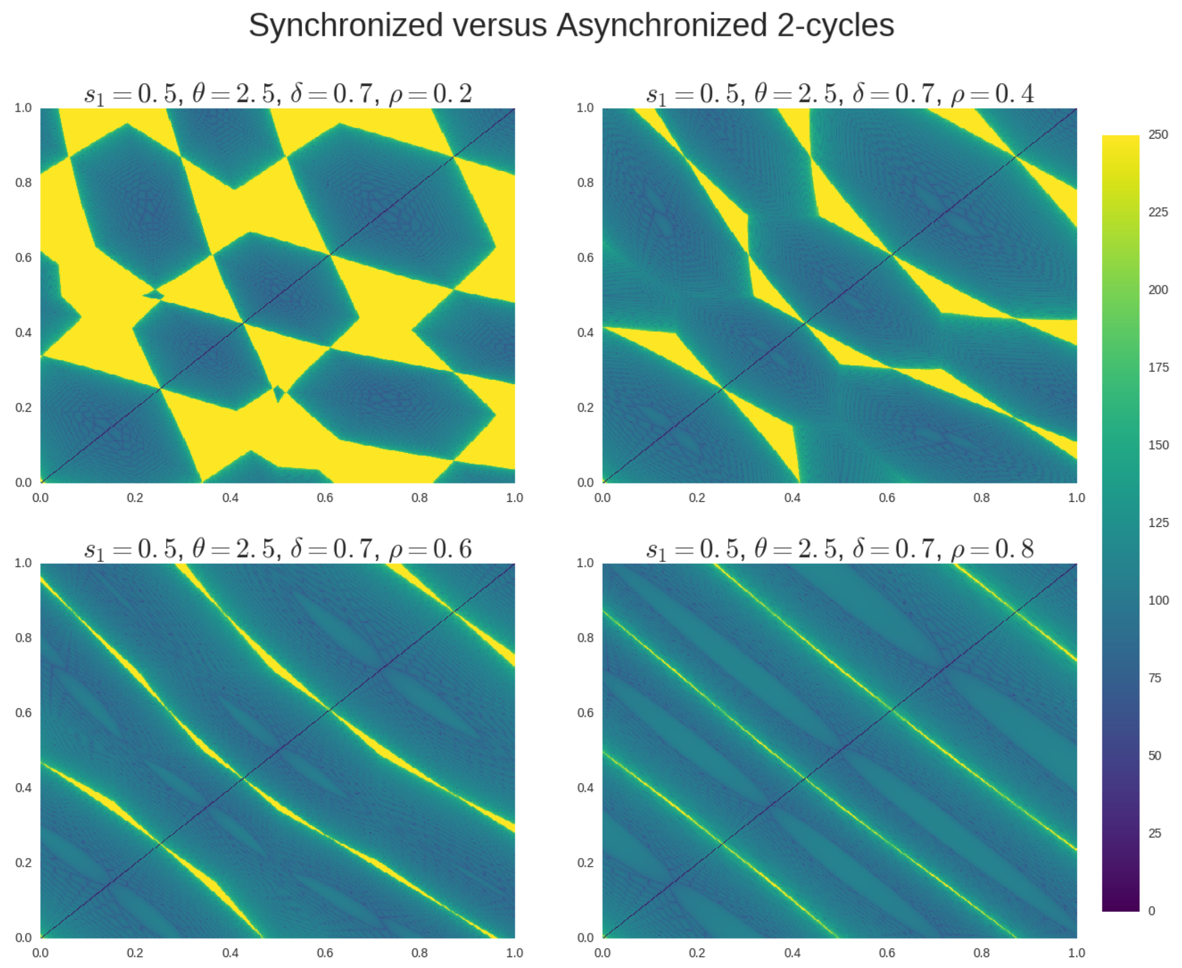
16.4.2 Basin of Attraction

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.

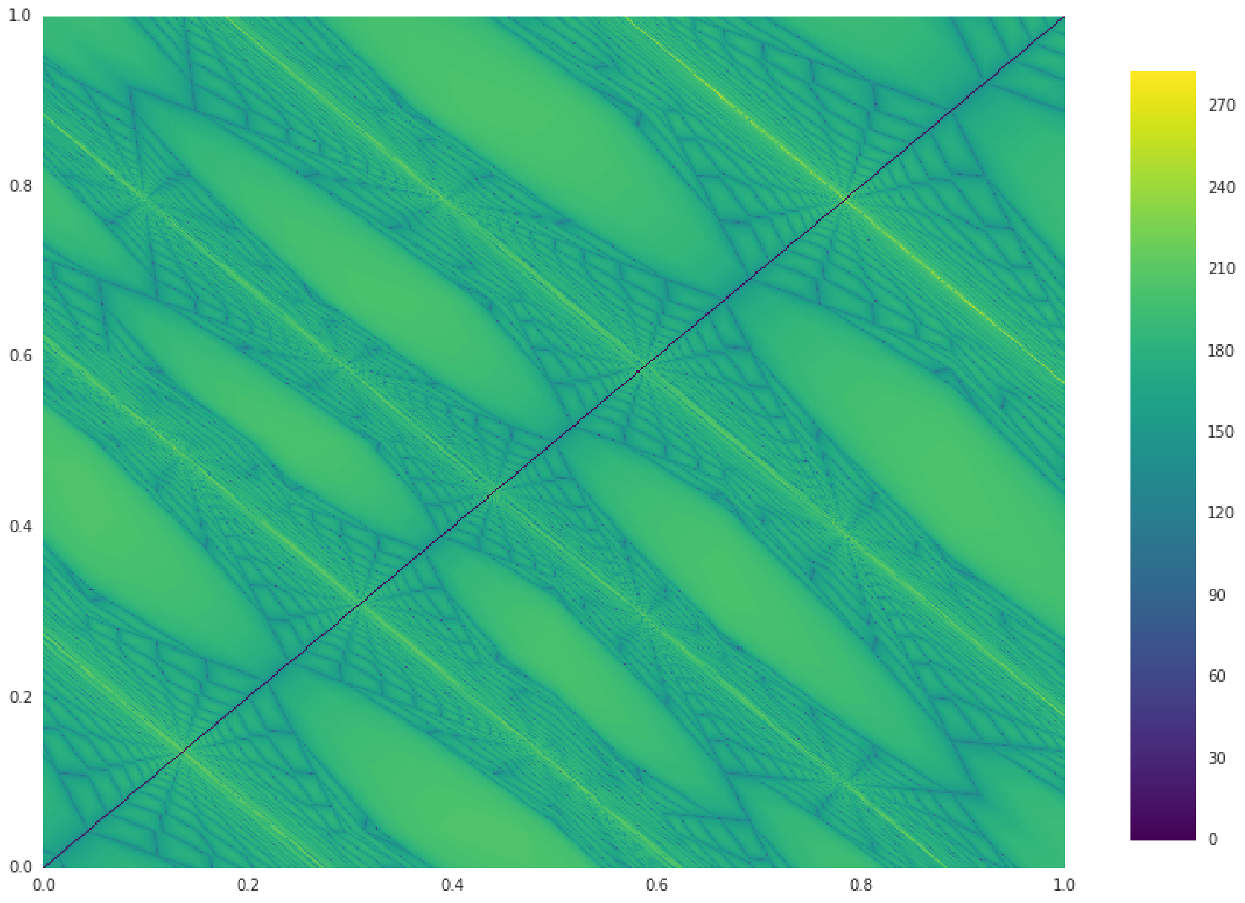


As you can see, larger values of ρ translate to more synchronization.

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

16.5 Exercises

16.5.1 Exercise 1

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

16.6 Solutions

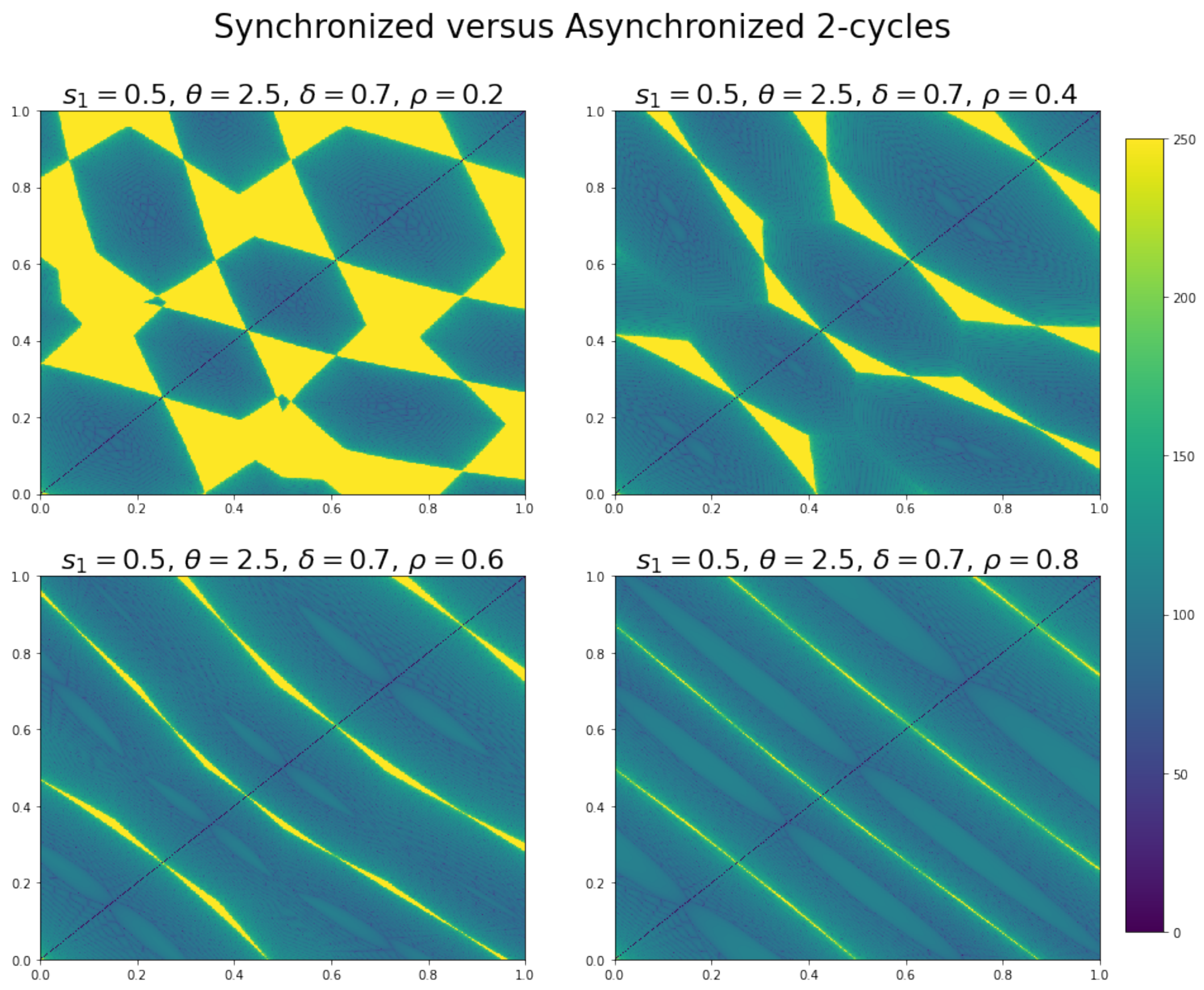
```
def plot_attraction_basis(s1=0.5, θ=2.5, δ=0.7, ρ=0.2, npts=250, ax=None):
    if ax is None:
        fig, ax = plt.subplots()

    # 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",
```

(continues on next page)

(continued from previous page)

```
              x=0.475, y=0.915, size=26)
plt.show()
```

```
<ipython-input-4-193e7524cdc9>:9: MatplotlibDeprecationWarning: shading='flat' when X␣
↪and Y have the same dimensions as C is deprecated since 3.3.  Either specify the␣
↪corners of the quadrilaterals with X and Y, or pass shading='auto', 'nearest' or
↪'gouraud', or set rcParams['pcolor.shading'].  This will become an error two minor␣
↪releases later.
  cf = ax.pcolormesh(unitrange, unitrange, ab, cmap="viridis")
```



16.6.1 Interactive Version

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,
               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

SEVENTEEN

COASE'S THEORY OF THE FIRM

Contents

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

```
!pip install interpolation
```

17.1 Overview

In 1937, Ronald Coase wrote a brilliant essay on the nature of the firm [Coa37].

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
```

17.1.1 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.

17.1.2 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…" ([Coa37], 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.

17.1.3 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.

17.1.4 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 [KNS18].

17.2 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.

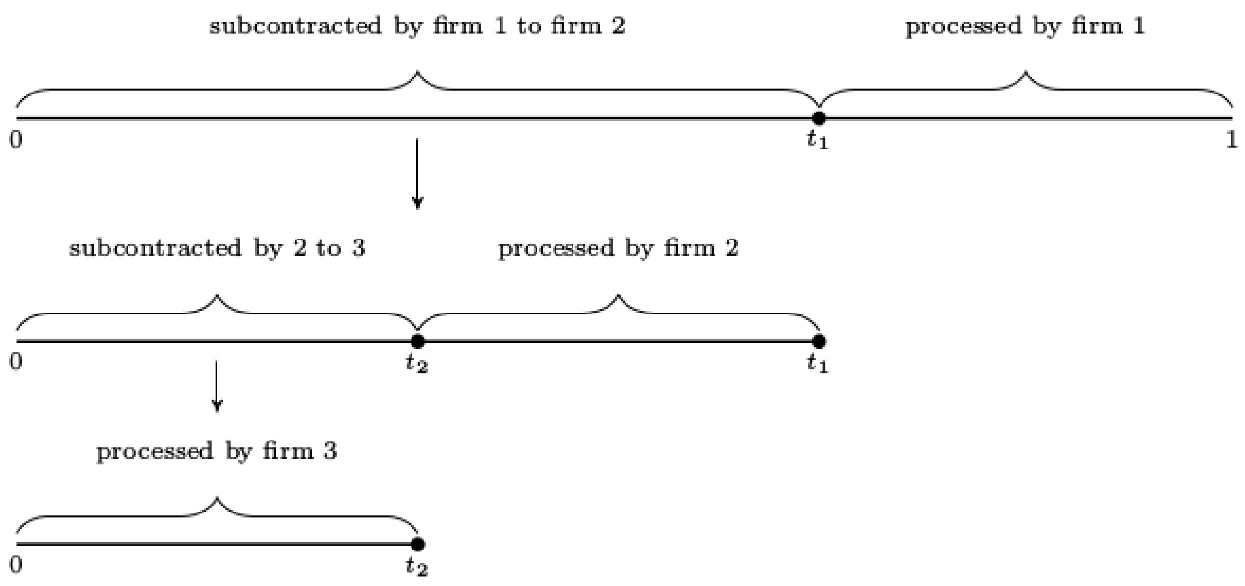
The stages are indexed by $t \in [0, 1]$, with $t = 0$ indicating that no tasks have been undertaken and $t = 1$ indicating that the good is complete.

17.2.1 Subcontracting

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

In this example,

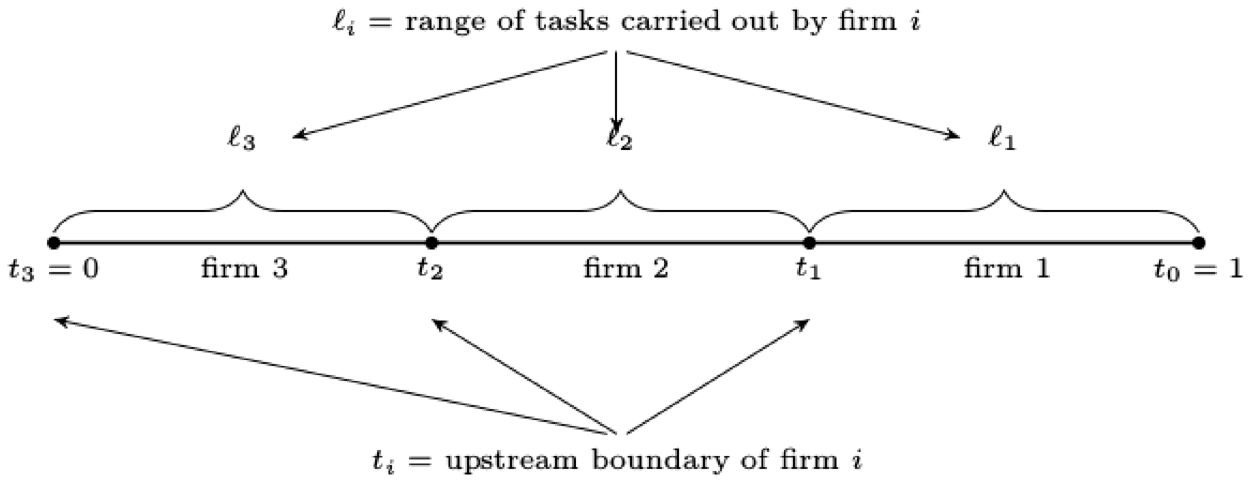
- Firm 1 receives a contract to sell one unit of the completed good to a final buyer.
- Firm 1 then forms a contract with firm 2 to purchase the partially completed good at stage t_1, with the intention of implementing the remaining $1 - t_1$ tasks in-house (i.e., processing from stage t_1 to stage 1).
- 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.


ℓ_i = range of tasks carried out by firm i
ℓ_3
ℓ_2
ℓ_1
$t_3 = 0$
firm 3
t_2
firm 2
t_1
firm 1
$t_0 = 1$
t_i = upstream boundary of firm 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.

17.2.2 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.

17.3 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.

17.3.1 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

17.3.2 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

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

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

$$\pi_i = p(t_{i-1}) - c(\ell_i) - \delta p(t_i) \tag{2}$$

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 (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.

17.4 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?

17.4.1 A Fixed Point Method

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

$$Tp(s) = \min_{t \leq s} \{c(s - t) + \delta p(t)\} \quad \text{for all} \quad s \in [0, 1]. \tag{3}$$

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 [KNS18] 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

$$t^*(s) := \text{ the solution to } \min_{t \leq s} \{c(s-t) + \delta p^*(t)\} \tag{4}$$

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 (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

$$t_0^* = 1 \quad \text{and} \quad t_i^* = t^*(t_{i-1}) \tag{5}$$

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

$$n^* := \inf\{i \in \mathbb{N} \,:\, t_i^* = 0\} \tag{6}$$

The task allocation corresponding to (5) is given by $\ell_i^* := t_{i-1}^* - t_i^*$ for all i.

In [KNS18] it is shown that

1. The value n^* in (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

$$p^*(s) = \min_{t \leq s} \{c(s-t) + \delta p^*(t)\} \quad \text{for all} \quad s \in [0,1] \tag{7}$$

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

17.4.2 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 [KNS18] it is shown that t^* and ℓ^* are increasing and continuous, while p^* is continuously differentiable at all $s \in (0, 1)$ with

$$(p^*)'(s) = c'(\ell^*(s)) \tag{8}$$

Equation (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

$$\delta (p^*)'(t^*(s)) = c'(s - t^*(s)) \tag{9}$$

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 (8) and (9) and evaluating at $s = t_i$, we see that active firms that are adjacent satisfy

$$\delta \, c'(\ell^*_{i+1}) = c'(\ell^*_i) \tag{10}$$

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.

17.5 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 (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)

```python
             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 $p_0 = c$

```python
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.

```python
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
    f = lambda t: delta * p_function(t) + c(s - t)
```

(continues on next page)

(continued from previous page)

```
    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 $s = 1$.

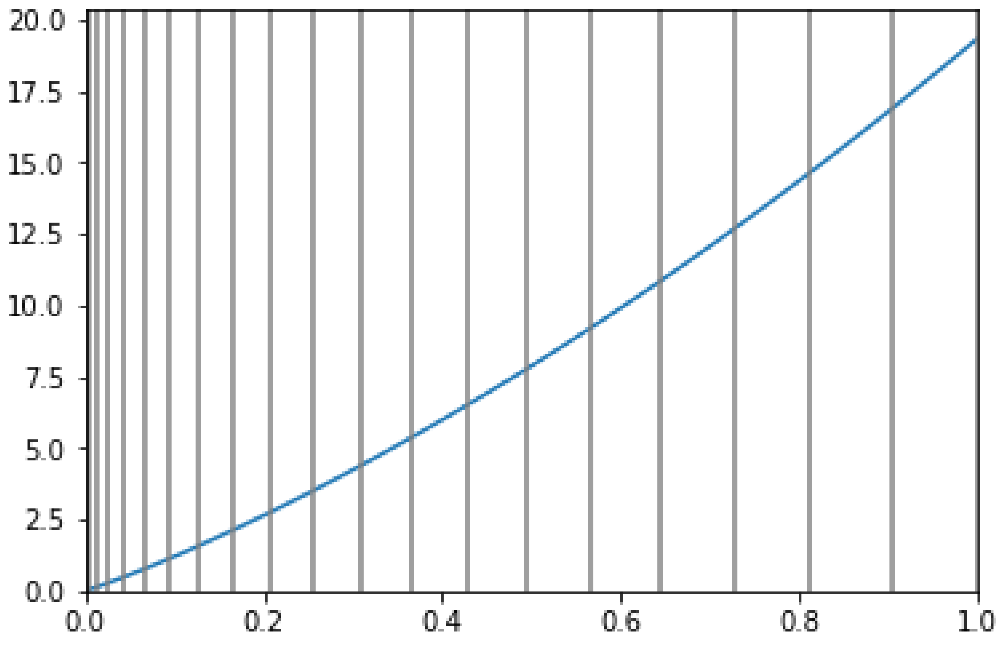
```
def compute_stages(pc, p_function):
    s = 1.0
    transaction_stages = [s]
    while s > 0:
        s, ell = optimal_choices(pc, p_function, s)
        transaction_stages.append(s)
    return np.array(transaction_stages)
```

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

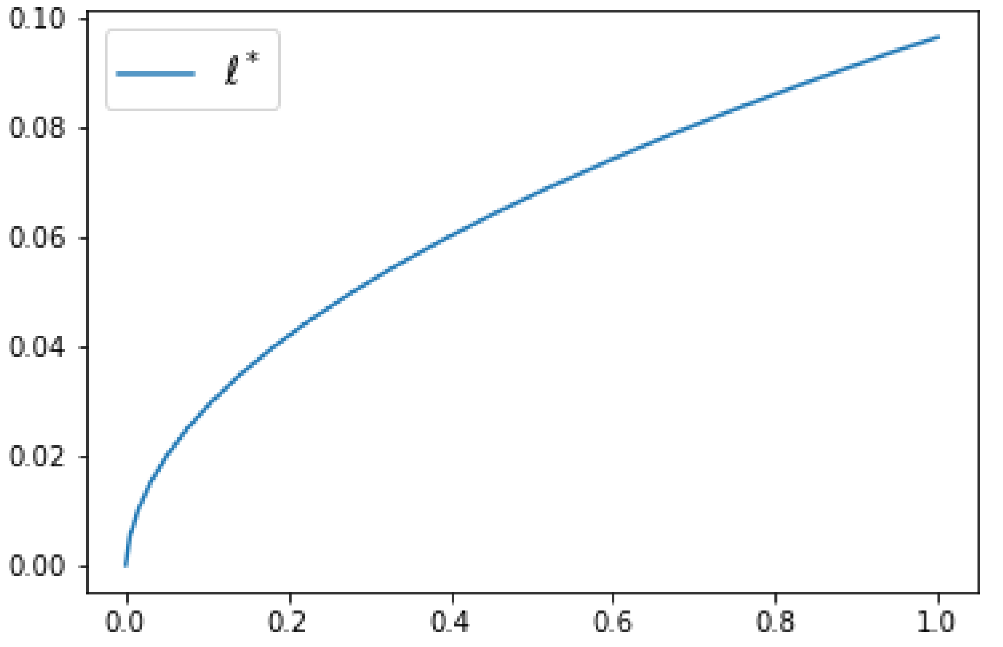
```
pc = ProductionChain()
p_star = compute_prices(pc)

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.

17.6 Exercises

17.6.1 Exercise 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).

17.6.2 Exercise 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.

17.7 Solutions

17.7.1 Exercise 1

```python
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
```

17.7.2 Exercise 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.

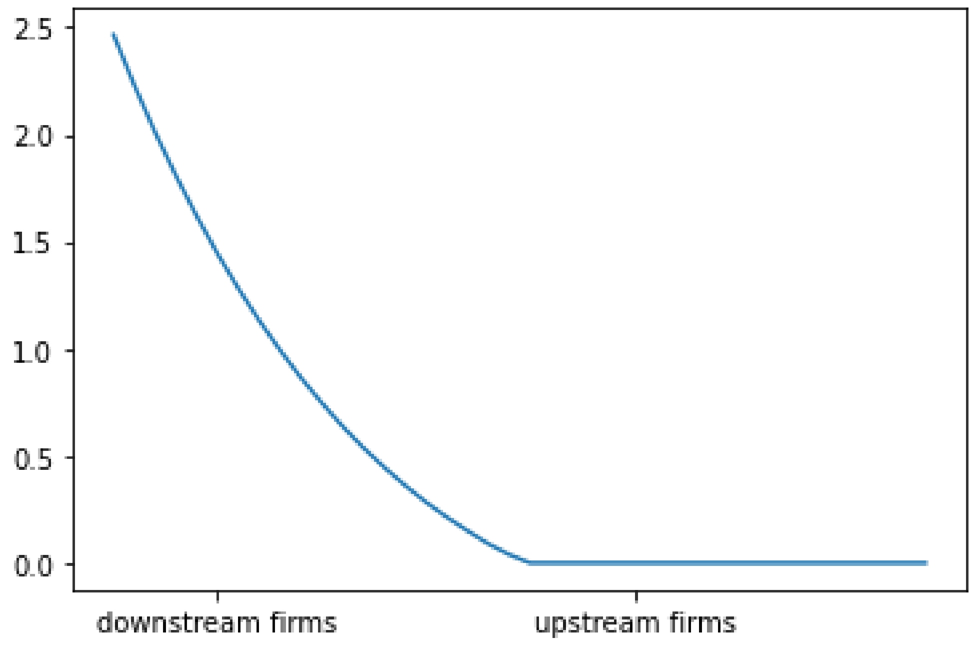
The equilibrium price function is strictly convex due to both transaction costs and diminishing returns to management.

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
```


downstream firms
upstream firms