Introduction to Economic Dynamics

Thomas J. Sargent and John Stachurski

Feb 19, 2024

CONTENTS

I Introduction to Dynamics **3**

1 Modeling COVID 19 **5**
1.1 Overview . . . 5
1.2 The SIR Model . . . 6
1.3 Implementation . . . 7
1.4 Experiments . . . 8
1.5 Ending Lockdown . . . 12

2 Inventory Dynamics **15**
2.1 Overview . . . 15
2.2 Sample Paths . . . 16
2.3 Marginal Distributions . . . 18
2.4 Exercises . . . 21

3 Samuelson Multiplier-Accelerator **25**
3.1 Overview . . . 25
3.2 Details . . . 27
3.3 Implementation . . . 30
3.4 Stochastic Shocks . . . 40
3.5 Government Spending . . . 42
3.6 Wrapping Everything Into a Class . . . 45
3.7 Using the LinearStateSpace Class . . . 50
3.8 Pure Multiplier Model . . . 57
3.9 Summary . . . 62

4 Kesten Processes and Firm Dynamics **63**
4.1 Overview . . . 63
4.2 Kesten Processes . . . 64
4.3 Heavy Tails . . . 67
4.4 Application: Firm Dynamics . . . 69
4.5 Exercises . . . 70

5 Wealth Distribution Dynamics **75**
5.1 Overview . . . 75
5.2 Lorenz Curves and the Gini Coefficient . . . 76
5.3 A Model of Wealth Dynamics . . . 79
5.4 Implementation . . . 80
5.5 Applications . . . 83
5.6 Exercises . . . 86

II Asset Pricing & Finance 89

6 Asset Pricing: Finite State Models 91
6.1 Overview . . . 91
6.2 Pricing Models . . . 92
6.3 Prices in the Risk-Neutral Case . . . 93
6.4 Risk Aversion and Asset Prices . . . 97
6.5 Exercises . . . 106

7 Competitive Equilibria with Arrow Securities 111
7.1 Introduction . . . 111
7.2 The setting . . . 112
7.3 Recursive Formulation . . . 113
7.4 State Variable Degeneracy . . . 114
7.5 Markov Asset Prices . . . 114
7.6 General Equilibrium . . . 116
7.7 Python Code . . . 120
7.8 Finite Horizon . . . 131

8 Heterogeneous Beliefs and Bubbles 137
8.1 Overview . . . 137
8.2 Structure of the Model . . . 138
8.3 Solving the Model . . . 140
8.4 Exercises . . . 145

9 Orthogonal Projections and Their Applications 149
9.1 Overview . . . 149
9.2 Key Definitions . . . 150
9.3 The Orthogonal Projection Theorem . . . 154
9.4 Orthonormal Basis . . . 157
9.5 Projection Via Matrix Algebra . . . 159
9.6 Least Squares Regression . . . 161
9.7 Orthogonalization and Decomposition . . . 163
9.8 Exercises . . . 164

10 Asset Pricing II: The Lucas Asset Pricing Model 167
10.1 Overview . . . 167
10.2 The Lucas Model . . . 168
10.3 Exercises . . . 175

11 Elementary Asset Pricing Theory 179
11.1 Overview . . . 179
11.2 Key Equation . . . 180
11.3 Implications of Key Equation . . . 180
11.4 Expected Return - Beta Representation . . . 181
11.5 Mean-Variance Frontier . . . 183
11.6 Sharpe Ratios and the Price of Risk . . . 186
11.7 Mathematical Structure of Frontier . . . 186
11.8 Multi-factor Models . . . 186
11.9 Empirical Implementations . . . 187
11.10 Exercises . . . 188

12 Two Modifications of Mean-Variance Portfolio Theory 193
12.1 Overview . . . 193
12.2 Mean-Variance Portfolio Choice . . . 194

12.3 Estimating Mean and Variance . . . 195
12.4 Black-Litterman Starting Point . . . 195
12.5 Details . . . 197
12.6 Adding Views . . . 199
12.7 Bayesian Interpretation . . . 200
12.8 Curve Decolletage . . . 201
12.9 Black-Litterman Recommendation as Regularization . . . 204
12.10 A Robust Control Operator . . . 206
12.11 A Robust Mean-Variance Portfolio Model . . . 208
12.12 Appendix . . . 208
12.13 Special Case – IID Sample . . . 209
12.14 Dependence and Sampling Frequency . . . 210
12.15 Frequency and the Mean Estimator . . . 211

13 Irrelevance of Capital Structures with Complete Markets 215
13.1 Introduction . . . 215
13.2 Competitive equilibrium . . . 219
13.3 Code . . . 225

14 Equilibrium Capital Structures with Incomplete Markets 235
14.1 Introduction . . . 235
14.2 Asset Markets . . . 239
14.3 Equilibrium verification . . . 242
14.4 Pseudo Code . . . 242
14.5 Code . . . 244
14.6 Examples . . . 254
14.7 A picture worth a thousand words . . . 271

III Search 273

15 Job Search I: The McCall Search Model 275
15.1 Overview . . . 275
15.2 The McCall Model . . . 276
15.3 Computing the Optimal Policy: Take 1 . . . 278
15.4 Computing an Optimal Policy: Take 2 . . . 284
15.5 Exercises . . . 285

16 Job Search II: Search and Separation 291
16.1 Overview . . . 291
16.2 The Model . . . 292
16.3 Solving the Model . . . 293
16.4 Implementation . . . 295
16.5 Impact of Parameters . . . 298
16.6 Exercises . . . 299

17 Job Search III: Fitted Value Function Iteration 303
17.1 Overview . . . 303
17.2 The Algorithm . . . 304
17.3 Implementation . . . 306
17.4 Exercises . . . 308

18 Job Search IV: Correlated Wage Offers 311
18.1 Overview . . . 311
18.2 The Model . . . 312

18.3 Implementation . . . 313
18.4 Unemployment Duration . . . 317
18.5 Exercises . . . 319

19 Job Search V: Modeling Career Choice **321**
19.1 Overview . . . 321
19.2 Model . . . 322
19.3 Implementation . . . 324
19.4 Exercises . . . 329

20 Job Search VI: On-the-Job Search **335**
20.1 Overview . . . 335
20.2 Model . . . 336
20.3 Implementation . . . 337
20.4 Solving for Policies . . . 340
20.5 Exercises . . . 343

21 Job Search VII: Search with Learning **347**
21.1 Overview . . . 347
21.2 Model . . . 348
21.3 Take 1: Solution by VFI . . . 351
21.4 Take 2: A More Efficient Method . . . 356
21.5 Another Functional Equation . . . 357
21.6 Solving the RWFE . . . 357
21.7 Implementation . . . 358
21.8 Exercises . . . 359
21.9 Solutions . . . 359
21.10 Appendix A . . . 361
21.11 Appendix B . . . 363
21.12 Examples . . . 367

22 Job Search VIII: A McCall Worker Q-Learns **379**
22.1 Overview . . . 379
22.2 Review of McCall Model . . . 380
22.3 Implied Quality Function Q . . . 384
22.4 From Probabilities to Samples . . . 385
22.5 Q-Learning . . . 385
22.6 Employed Worker Can't Quit . . . 393
22.7 Possible Extensions . . . 395

23 A Lake Model of Employment and Unemployment **397**
23.1 Overview . . . 397
23.2 The Model . . . 398
23.3 Implementation . . . 400
23.4 Dynamics of an Individual Worker . . . 405
23.5 Endogenous Job Finding Rate . . . 407
23.6 Exercises . . . 415

IV Optimal Savings **425**

24 Cake Eating I: Introduction to Optimal Saving **427**
24.1 Overview . . . 427
24.2 The Model . . . 428
24.3 The Value Function . . . 429

24.4 The Optimal Policy . . . 431
24.5 The Euler Equation . . . 432
24.6 Exercises . . . 434

25 Cake Eating II: Numerical Methods 437
25.1 Overview . . . 437
25.2 Reviewing the Model . . . 438
25.3 Value Function Iteration . . . 438
25.4 Time Iteration . . . 446
25.5 Exercises . . . 447

26 Optimal Growth I: The Stochastic Optimal Growth Model 453
26.1 Overview . . . 453
26.2 The Model . . . 454
26.3 Computation . . . 458
26.4 Exercises . . . 466

27 Optimal Growth II: Accelerating the Code with Numba 469
27.1 Overview . . . 469
27.2 The Model . . . 470
27.3 Computation . . . 471
27.4 Exercises . . . 475

28 Optimal Growth III: Time Iteration 481
28.1 Overview . . . 481
28.2 The Euler Equation . . . 482
28.3 Implementation . . . 484
28.4 Exercises . . . 489

29 Optimal Growth IV: The Endogenous Grid Method 493
29.1 Overview . . . 493
29.2 Key Idea . . . 494
29.3 Implementation . . . 495

30 The Income Fluctuation Problem I: Basic Model 501
30.1 Overview . . . 501
30.2 The Optimal Savings Problem . . . 502
30.3 Computation . . . 504
30.4 Implementation . . . 505
30.5 Exercises . . . 510

31 The Income Fluctuation Problem II: Stochastic Returns on Assets 517
31.1 Overview . . . 517
31.2 The Savings Problem . . . 518
31.3 Solution Algorithm . . . 519
31.4 Implementation . . . 521
31.5 Exercises . . . 526

V Other 529

32 Troubleshooting 531
32.1 Fixing Your Local Environment . . . 531
32.2 Reporting an Issue . . . 532

33 References 533

34 Execution Statistics 535

Bibliography 537

Index 541

This website presents an introductory set of lectures on economic dynamics.

- Introduction to Dynamics
  - *Modeling COVID 19*
  - *Inventory Dynamics*
  - *Samuelson Multiplier-Accelerator*
  - *Kesten Processes and Firm Dynamics*
  - *Wealth Distribution Dynamics*
- Asset Pricing & Finance
  - *Asset Pricing: Finite State Models*
  - *Competitive Equilibria with Arrow Securities*
  - *Heterogeneous Beliefs and Bubbles*
  - *Orthogonal Projections and Their Applications*
  - *Asset Pricing II: The Lucas Asset Pricing Model*
  - *Elementary Asset Pricing Theory*
  - *Two Modifications of Mean-Variance Portfolio Theory*
  - *Irrelevance of Capital Structures with Complete Markets*
  - *Equilibrium Capital Structures with Incomplete Markets*
- Search
  - *Job Search I: The McCall Search Model*
  - *Job Search II: Search and Separation*
  - *Job Search III: Fitted Value Function Iteration*
  - *Job Search IV: Correlated Wage Offers*
  - *Job Search V: Modeling Career Choice*
  - *Job Search VI: On-the-Job Search*
  - *Job Search VII: Search with Learning*
  - *Job Search VIII: A McCall Worker Q-Learns*
  - *A Lake Model of Employment and Unemployment*
- Optimal Savings
  - *Cake Eating I: Introduction to Optimal Saving*
  - *Cake Eating II: Numerical Methods*
  - *Optimal Growth I: The Stochastic Optimal Growth Model*
  - *Optimal Growth II: Accelerating the Code with Numba*
  - *Optimal Growth III: Time Iteration*
  - *Optimal Growth IV: The Endogenous Grid Method*
  - *The Income Fluctuation Problem I: Basic Model*
  - *The Income Fluctuation Problem II: Stochastic Returns on Assets*

- Other
  - *Troubleshooting*
  - *References*
  - *Execution Statistics*

Part I

Introduction to Dynamics

CHAPTER

ONE

MODELING COVID 19

Contents

- *Modeling COVID 19*
  - *Overview*
  - *The SIR Model*
  - *Implementation*
  - *Experiments*
  - *Ending Lockdown*

1.1 Overview

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

See, in particular

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

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

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

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

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

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

We will use the following standard imports:

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

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

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

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

1.2 The SIR Model

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

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

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

Comments:

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

1.2.1 Time Path

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

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

The interest is primarily in

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

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

$$\begin{aligned} \dot{s}(t) &= -\beta(t)\, s(t)\, i(t) \\ \dot{e}(t) &= \beta(t)\, s(t)\, i(t) - \sigma e(t) \\ \dot{i}(t) &= \sigma e(t) - \gamma i(t) \end{aligned} \tag{1.1}$$

In these equations,

- $\beta(t)$ is called the *transmission rate* (the rate at which individuals bump into others and expose them to the virus).
- σ is called the *infection rate* (the rate at which those who are exposed become infected)
- γ is called the *recovery rate* (the rate at which infected people recover or die).
- the dot symbol $\dot{y}$ represents the time derivative dy/dt.

We do not need to model the fraction r of the population in state R separately because the states form a partition.

In particular, the "removed" fraction of the population is $r = 1 - s - e - i$.

We will also track $c = i + r$, which is the cumulative caseload (i.e., all those who have or have had the infection).

The system (1.1) can be written in vector form as

$$\dot{x} = F(x, t), \qquad x := (s, e, i) \tag{1.2}$$

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

1.2.2 Parameters

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

As in Atkeson's note, we set

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

The transmission rate is modeled as

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

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

1.3 Implementation

First we set the population size to match the US.

```
pop_size = 3.3e8
```

Next we fix parameters as described above.

```
γ = 1 / 18
σ = 1 / 5.2
```

Now we construct a function that represents F in (1.2)

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

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

    """
    s, e, i = x

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

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

    return ds, de, di
```

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

The initial conditions are set to

```
# initial conditions of s, e, i
i_0 = 1e-7
e_0 = 4 * i_0
s_0 = 1 - i_0 - e_0
```

In vector form the initial condition is

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

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

```
def solve_path(R0, t_vec, x_init=x_0):
    """
    Solve for i(t) and c(t) via numerical integration,
    given the time path for R0.

    """
    G = lambda x, t: F(x, t, R0)
    s_path, e_path, i_path = odeint(G, x_init, t_vec).transpose()

    c_path = 1 - s_path - e_path       # cumulative cases
    return i_path, c_path
```

1.4 Experiments

Let's run some experiments using this code.

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

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

1.4.1 Experiment 1: Constant R0 Case

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

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

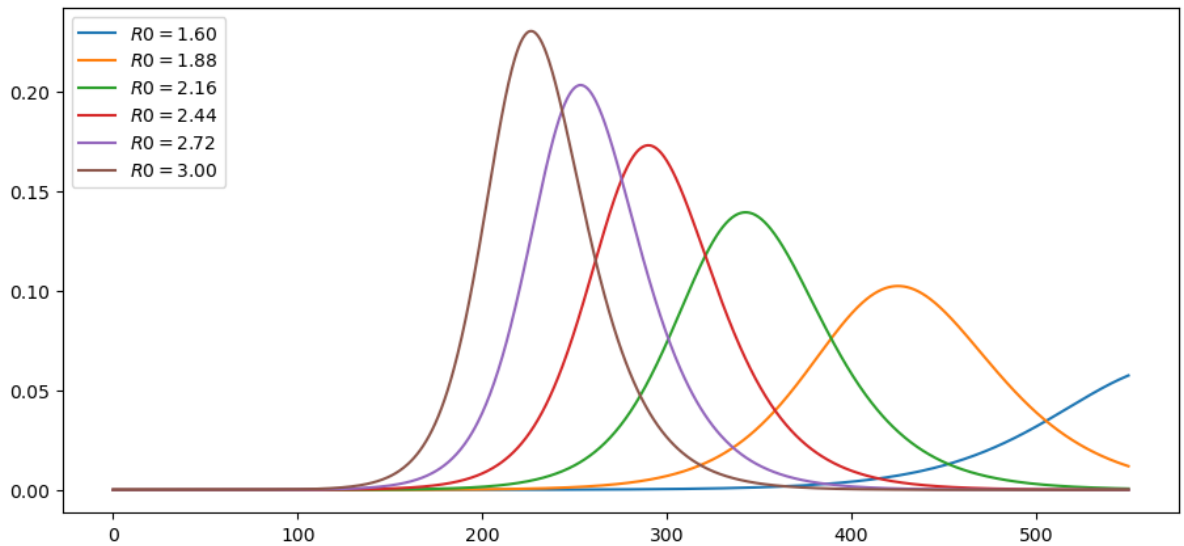
```
R0_vals = np.linspace(1.6, 3.0, 6)
labels = [f'$R0 = {r:.2f}$' for r in R0_vals]
i_paths, c_paths = [], []

for r in R0_vals:
    i_path, c_path = solve_path(r, t_vec)
    i_paths.append(i_path)
    c_paths.append(c_path)
```

Here's some code to plot the time paths.

```
def plot_paths(paths, labels, times=t_vec):

    fig, ax = plt.subplots()

    for path, label in zip(paths, labels):
        ax.plot(times, path, label=label)

    ax.legend(loc='upper left')

    plt.show()
```

Let's plot current cases as a fraction of the population.

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

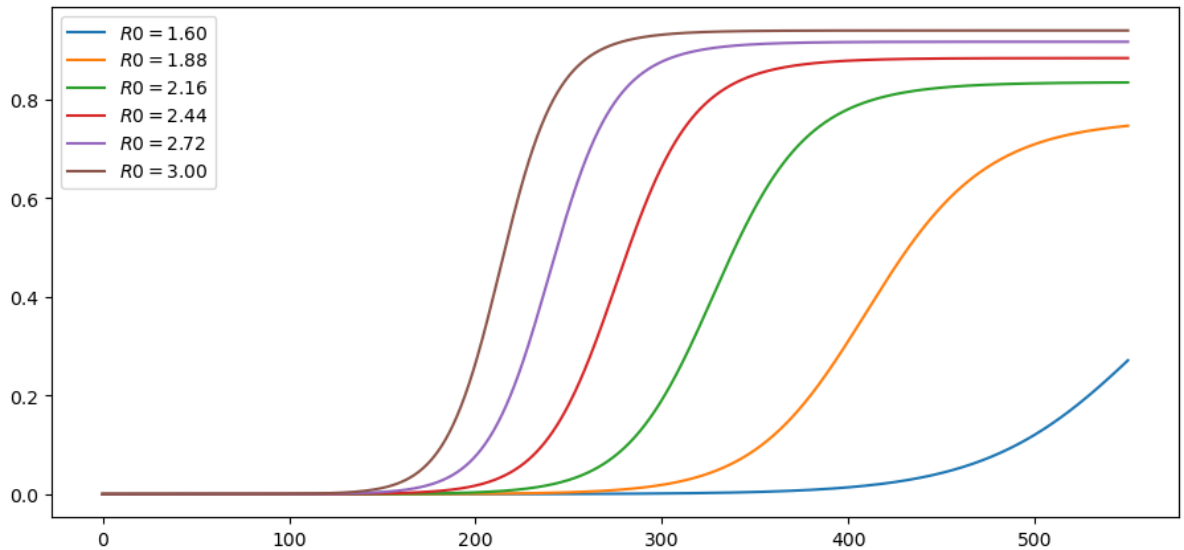


As expected, lower effective transmission rates defer the peak of infections.

They also lead to a lower peak in current cases.

Here are cumulative cases, as a fraction of population:

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



1.4.2 Experiment 2: Changing Mitigation

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

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

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

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

This is due to progressive adoption of stricter mitigation measures.

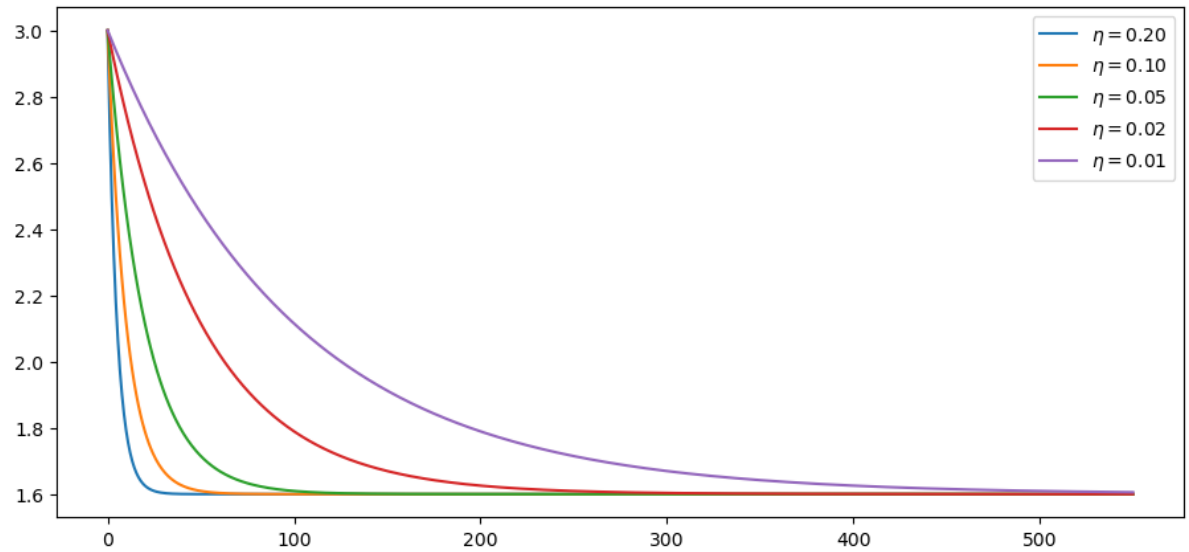
The parameter `η` controls the rate, or the speed at which restrictions are imposed.

We consider several different rates:

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

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

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

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

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

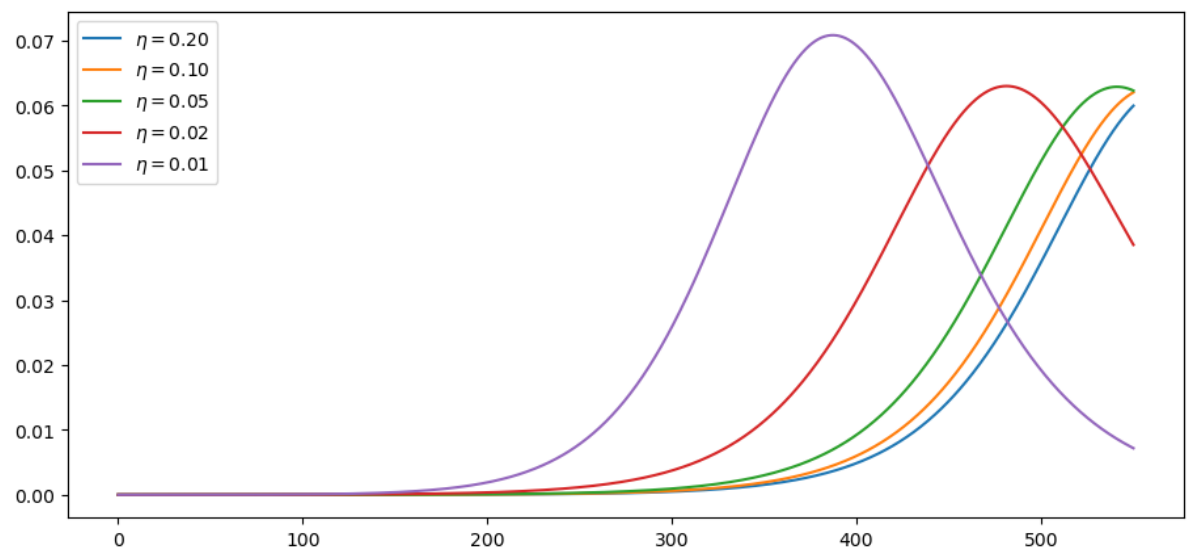


Let's calculate the time path of infected people:

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

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

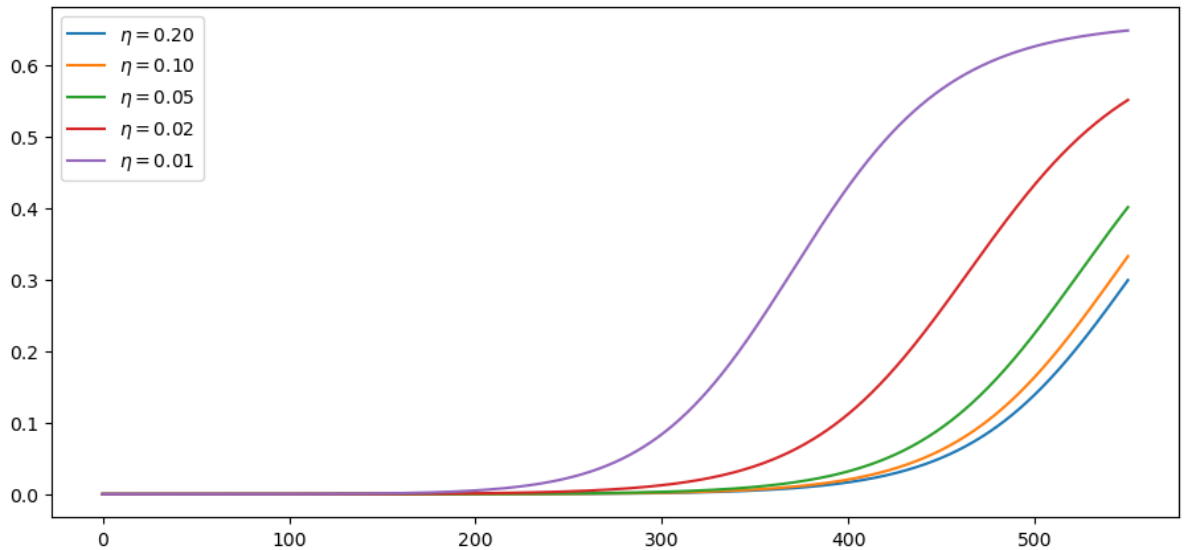
These are current cases under the different scenarios:

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



Here are cumulative cases, as a fraction of population:

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



1.5 Ending Lockdown

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

Consider these two mitigation scenarios:

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

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

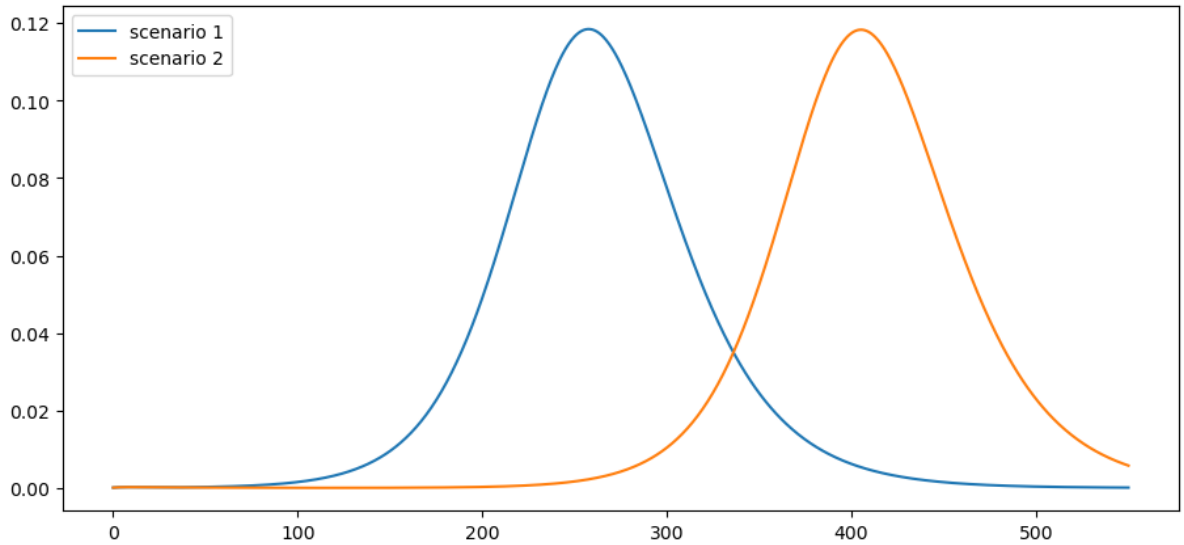
```
# initial conditions
i_0 = 25_000 / pop_size
e_0 = 75_000 / pop_size
s_0 = 1 - i_0 - e_0
x_0 = s_0, e_0, i_0
```

Let's calculate the paths:

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

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

i_paths, c_paths = [], []

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

Here is the number of active infections:

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



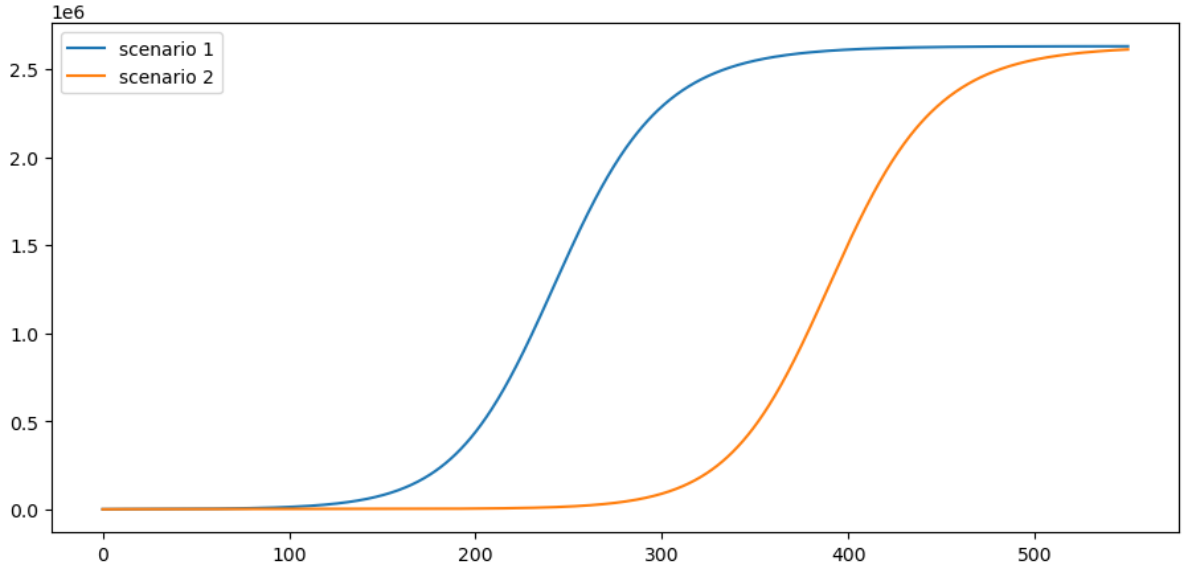
What kind of mortality can we expect under these scenarios?

Suppose that 1% of cases result in death

```
v = 0.01
```

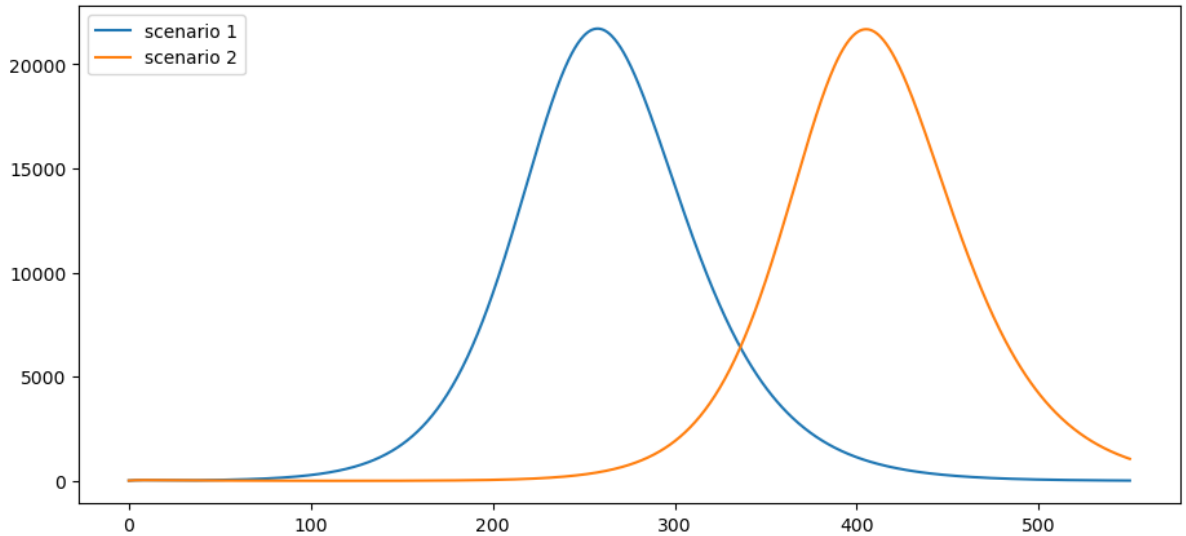
This is the cumulative number of deaths:

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



This is the daily death rate:

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



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

CHAPTER

TWO

INVENTORY DYNAMICS

Contents

- *Inventory Dynamics*
  - *Overview*
  - *Sample Paths*
  - *Marginal Distributions*
  - *Exercises*

2.1 Overview

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

Such firms

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

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

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

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

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

Let's start with some imports

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from numba import njit, float64, prange
from numba.experimental import jitclass
```

2.2 Sample Paths

Consider a firm with inventory X_t.

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

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

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

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

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

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

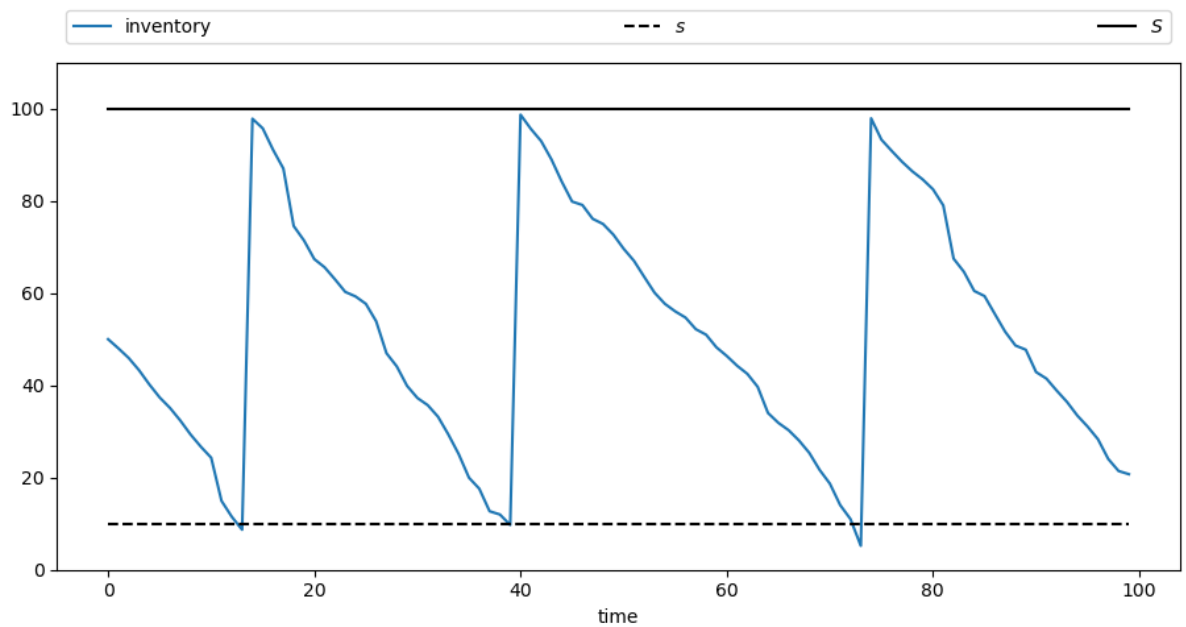
where μ and σ are parameters and $\{Z_t\}$ is IID and standard normal.

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

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


@jitclass(firm_data)
class Firm:

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

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

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

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

    def sim_inventory_path(self, x_init, sim_length):

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

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

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

```
firm = Firm()

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

X = firm.sim_inventory_path(x_init, sim_length)

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

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

plt.show()
```



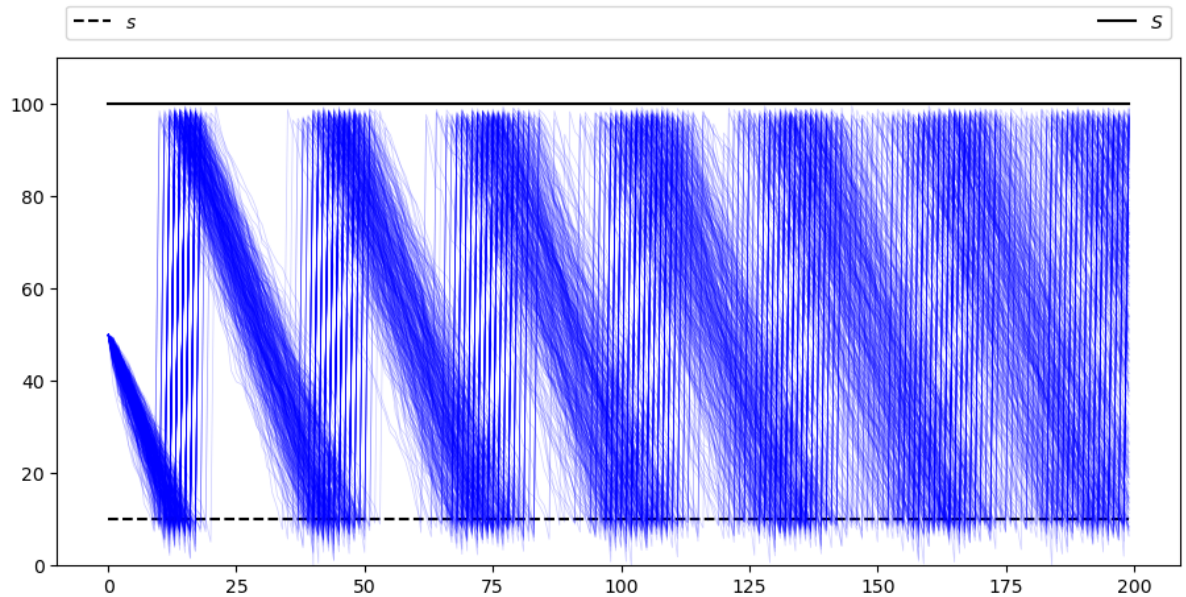
Now let's simulate multiple paths in order to build a more complete picture of the probabilities of different outcomes:

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

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

(continues on next page)

(continued from previous page)

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

plt.show()
```



2.3 Marginal Distributions

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

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

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

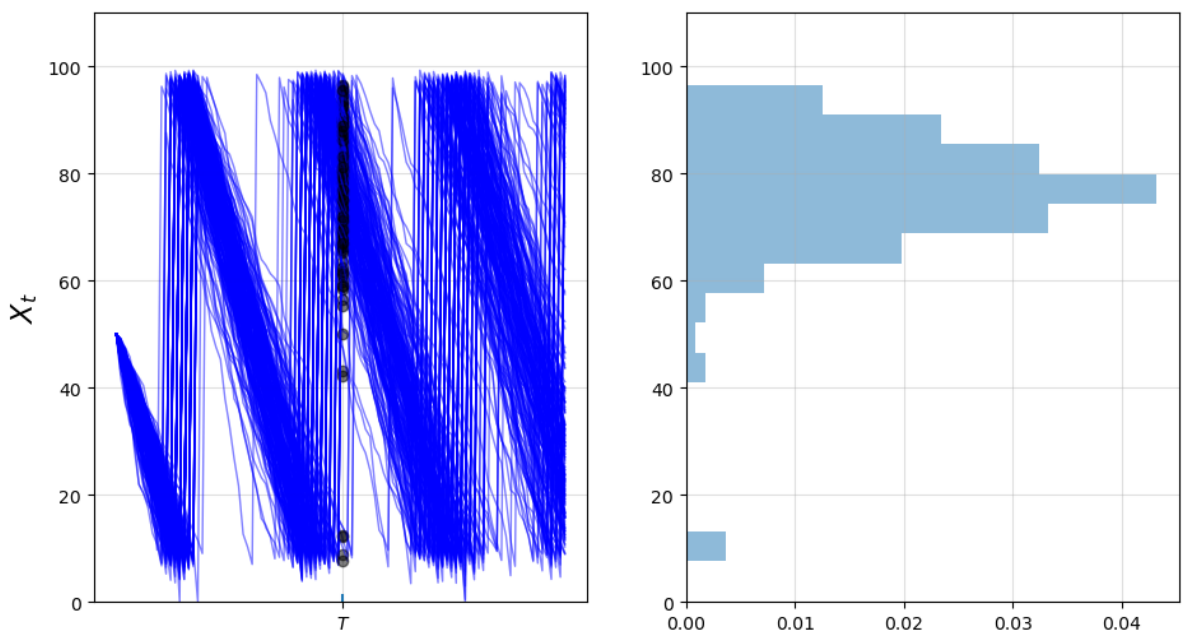
Here's one visualization, with $T = 50$.

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

ymin, ymax = 0, S + 10

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

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

ax = axes[0]

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

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

(continues on next page)

(continued from previous page)

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

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

axes[1].set_ylim(ymin, ymax)

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

plt.show()
```



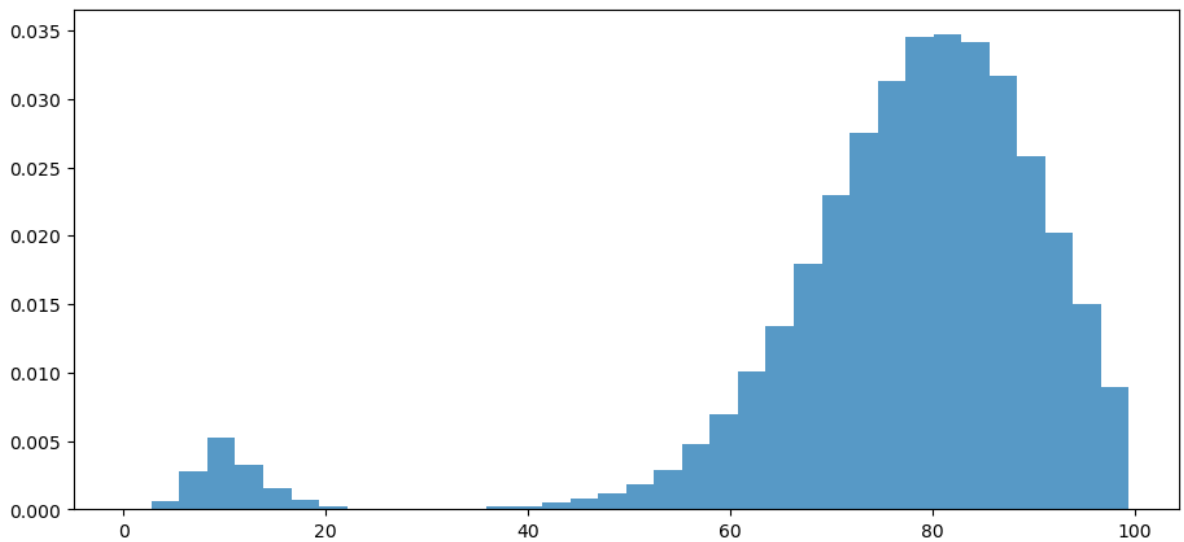
We can build up a clearer picture by drawing more samples

```
T = 50
M = 50_000

fig, ax = plt.subplots()

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

ax.hist(sample,
```

(continues on next page)

(continued from previous page)

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

plt.show()
```



Note that the distribution is bimodal

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

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

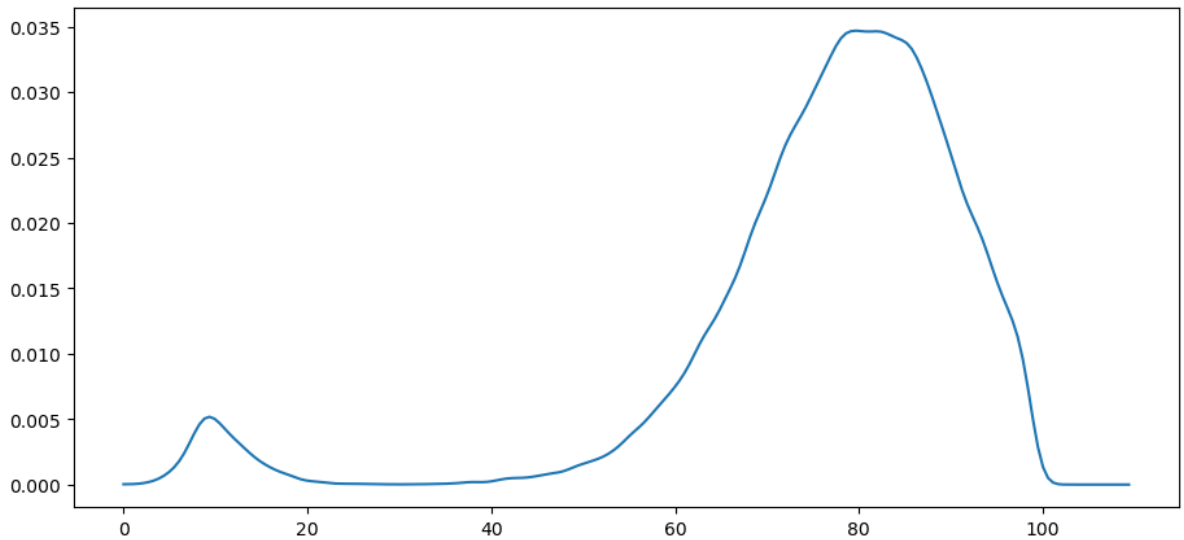
Kernel density estimators can be thought of as smoothed histograms.

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

We will use a kernel density estimator from scikit-learn

```
from sklearn.neighbors import KernelDensity

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

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

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

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



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

2.4 Exercises

Exercise 2.4.1

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

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

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

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

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

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

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

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

Solution to Exercise 2.4.1

Below is one possible solution:

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

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

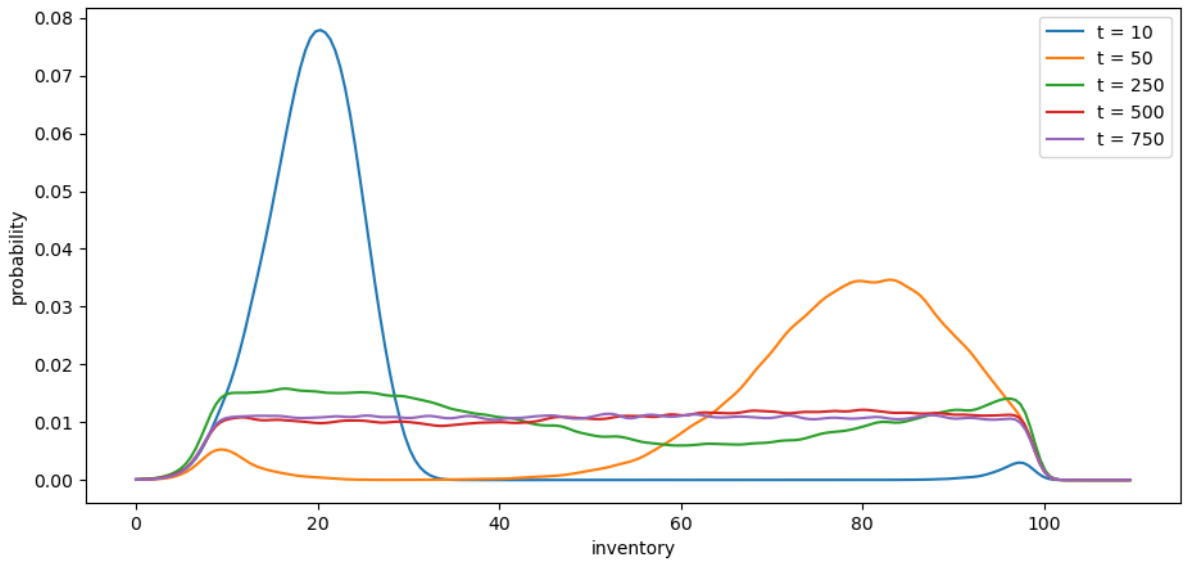
```
s, S, mu, sigma = firm.s, firm.S, firm.mu, firm.sigma

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

(continues on next page)

(continued from previous page)

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

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

    return new_inventory_levels
```

```
x_init = 50
num_firms = 50_000

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

first_diffs = np.diff(sample_dates)

fig, ax = plt.subplots()

X = np.full(num_firms, x_init)

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

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



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

We have reached a reasonable approximation of the stationary density.

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

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

Exercise 2.4.2

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

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

Solution to Exercise 2.4.2

Here is one solution.

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

We will also use parallelization across firms.

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

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

        for t in range(sim_length):
            Z = np.random.randn()
            D = np.exp(mu + sigma * Z)
            if x <= s:
```

(continues on next page)

(continued from previous page)

```
                x = max(S - D, 0)
                restock_counter += 1
            else:
                x = max(x - D, 0)

        if restock_counter > 1:
            firm_counter += 1

    return firm_counter / num_firms
```

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

```
%%time

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

```
Frequency of at least two stock outs = 0.44803
CPU times: user 2.57 s, sys: 4.31 ms, total: 2.58 s
Wall time: 914 ms
```

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

Depending on your system, the difference can be substantial.

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

CHAPTER

THREE

SAMUELSON MULTIPLIER-ACCELERATOR

Contents

- *Samuelson Multiplier-Accelerator*
  - *Overview*
  - *Details*
  - *Implementation*
  - *Stochastic Shocks*
  - *Government Spending*
  - *Wrapping Everything Into a Class*
  - *Using the LinearStateSpace Class*
  - *Pure Multiplier Model*
  - *Summary*

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

```
!pip install quantecon
```

3.1 Overview

This lecture creates non-stochastic and stochastic versions of Paul Samuelson's celebrated multiplier accelerator model [Sam39].

In doing so, we extend the example of the Solow model class in our second OOP lecture.

Our objectives are to

- provide a more detailed example of OOP and classes
- review a famous model
- review linear difference equations, both deterministic and stochastic

Let's start with some standard imports:

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

We'll also use the following for various tasks described below:

```
from quantecon import LinearStateSpace
import cmath
import math
import sympy
from sympy import Symbol, init_printing
from cmath import sqrt
```

3.1.1 Samuelson's Model

Samuelson used a *second-order linear difference equation* to represent a model of national output based on three components:

- a *national output identity* asserting that national output or national income is the sum of consumption plus investment plus government purchases.
- a Keynesian *consumption function* asserting that consumption at time t is equal to a constant times national output at time $t-1$.
- an investment *accelerator* asserting that investment at time t equals a constant called the *accelerator coefficient* times the difference in output between period $t-1$ and $t-2$.

Consumption plus investment plus government purchases constitute *aggregate demand,* which automatically calls forth an equal amount of *aggregate supply*.

(To read about linear difference equations see here or chapter IX of [Sar87].)

Samuelson used the model to analyze how particular values of the marginal propensity to consume and the accelerator coefficient might give rise to transient *business cycles* in national output.

Possible dynamic properties include

- smooth convergence to a constant level of output
- damped business cycles that eventually converge to a constant level of output
- persistent business cycles that neither dampen nor explode

Later we present an extension that adds a random shock to the right side of the national income identity representing random fluctuations in aggregate demand.

This modification makes national output become governed by a second-order *stochastic linear difference equation* that, with appropriate parameter values, gives rise to recurrent irregular business cycles.

(To read about stochastic linear difference equations see chapter XI of [Sar87].)

3.2 Details

Let's assume that

- $\{G_t\}$ is a sequence of levels of government expenditures – we'll start by setting $G_t = G$ for all t.
- $\{C_t\}$ is a sequence of levels of aggregate consumption expenditures, a key endogenous variable in the model.
- $\{I_t\}$ is a sequence of rates of investment, another key endogenous variable.
- $\{Y_t\}$ is a sequence of levels of national income, yet another endogenous variable.
- a is the marginal propensity to consume in the Keynesian consumption function $C_t = aY_{t-1} + \gamma$.
- b is the "accelerator coefficient" in the "investment accelerator" $I_t = b(Y_{t-1} - Y_{t-2})$.
- $\{\epsilon_t\}$ is an IID sequence standard normal random variables.
- $\sigma \geq 0$ is a "volatility" parameter — setting $\sigma = 0$ recovers the non-stochastic case that we'll start with.

The model combines the consumption function

$$C_t = aY_{t-1} + \gamma \tag{3.1}$$

with the investment accelerator

$$I_t = b(Y_{t-1} - Y_{t-2}) \tag{3.2}$$

and the national income identity

$$Y_t = C_t + I_t + G_t \tag{3.3}$$

- The parameter a is peoples' *marginal propensity to consume* out of income - equation (3.1) asserts that people consume a fraction of $a \in (0, 1)$ of each additional dollar of income.
- The parameter $b > 0$ is the investment accelerator coefficient - equation (3.2) asserts that people invest in physical capital when income is increasing and disinvest when it is decreasing.

Equations (3.1), (3.2), and (3.3) imply the following second-order linear difference equation for national income:

$$Y_t = (a + b)Y_{t-1} - bY_{t-2} + (\gamma + G_t)$$

or

$$Y_t = \rho_1 Y_{t-1} + \rho_2 Y_{t-2} + (\gamma + G_t) \tag{3.4}$$

where $\rho_1 = (a + b)$ and $\rho_2 = -b$.

To complete the model, we require two **initial conditions**.

If the model is to generate time series for $t = 0, \dots, T$, we require initial values

$$Y_{-1} = \bar{Y}_{-1}, \quad Y_{-2} = \bar{Y}_{-2}$$

We'll ordinarily set the parameters (a, b) so that starting from an arbitrary pair of initial conditions $(\bar{Y}_{-1}, \bar{Y}_{-2})$, national income Y_t converges to a constant value as t becomes large.

We are interested in studying

- the transient fluctuations in Y_t as it converges to its **steady state** level
- the **rate** at which it converges to a steady state level

The deterministic version of the model described so far — meaning that no random shocks hit aggregate demand — has only transient fluctuations.

We can convert the model to one that has persistent irregular fluctuations by adding a random shock to aggregate demand.

3.2.1 Stochastic Version of the Model

We create a **random** or **stochastic** version of the model by adding a random process of **shocks** or **disturbances** $\{\sigma\epsilon_t\}$ to the right side of equation (3.4), leading to the **second-order scalar linear stochastic difference equation**:

$$Y_t = G_t + a(1-b)Y_{t-1} - abY_{t-2} + \sigma\epsilon_t \tag{3.5}$$

3.2.2 Mathematical Analysis of the Model

To get started, let's set $G_t \equiv 0$, $\sigma = 0$, and $\gamma = 0$.

Then we can write equation (3.5) as

$$Y_t = \rho_1 Y_{t-1} + \rho_2 Y_{t-2}$$

or

$$Y_{t+2} - \rho_1 Y_{t+1} - \rho_2 Y_t = 0 \tag{3.6}$$

To discover the properties of the solution of (3.6), it is useful first to form the **characteristic polynomial** for (3.6):

$$z^2 - \rho_1 z - \rho_2 \tag{3.7}$$

where z is possibly a complex number.

We want to find the two **zeros** (a.k.a. **roots**) – namely λ_1, λ_2 – of the characteristic polynomial.

These are two special values of z, say $z = \lambda_1$ and $z = \lambda_2$, such that if we set z equal to one of these values in expression (3.7), the characteristic polynomial (3.7) equals zero:

$$z^2 - \rho_1 z - \rho_2 = (z - \lambda_1)(z - \lambda_2) = 0 \tag{3.8}$$

Equation (3.8) is said to **factor** the characteristic polynomial.

When the roots are complex, they will occur as a complex conjugate pair.

When the roots are complex, it is convenient to represent them in the polar form

$$\lambda_1 = re^{i\omega},\ \lambda_2 = re^{-i\omega}$$

where r is the *amplitude* of the complex number and ω is its *angle* or *phase*.

These can also be represented as

$$\lambda_1 = r(cos(\omega) + i\sin(\omega))$$

$$\lambda_2 = r(cos(\omega) - i\sin(\omega))$$

(To read about the polar form, see here)

Given **initial conditions** Y_{-1}, Y_{-2}, we want to generate a **solution** of the difference equation (3.6).

It can be represented as

$$Y_t = \lambda_1^t c_1 + \lambda_2^t c_2$$

where c_1 and c_2 are constants that depend on the two initial conditions and on ρ_1, ρ_2.

When the roots are complex, it is useful to pursue the following calculations.

Notice that

$$\begin{aligned} Y_t = & \quad c_1(re^{i\omega})^t + c_2(re^{-i\omega})^t \\ = & \quad c_1 r^t e^{i\omega t} + c_2 r^t e^{-i\omega t} \\ = & \quad c_1 r^t[\cos(\omega t) + i\sin(\omega t)] + c_2 r^t[\cos(\omega t) - i\sin(\omega t)] \\ = & \quad (c_1 + c_2) r^t \cos(\omega t) + i(c_1 - c_2) r^t \sin(\omega t) \end{aligned}$$

The only way that Y_t can be a real number for each t is if $c_1 + c_2$ is a real number and $c_1 - c_2$ is an imaginary number.

This happens only when c_1 and c_2 are complex conjugates, in which case they can be written in the polar forms

$$c_1 = ve^{i\theta}, \;\; c_2 = ve^{-i\theta}$$

So we can write

$$\begin{aligned} Y_t = & \quad ve^{i\theta} r^t e^{i\omega t} + ve^{-i\theta} r^t e^{-i\omega t} \\ = & \quad vr^t[e^{i(\omega t+\theta)} + e^{-i(\omega t+\theta)}] \\ = & \quad 2vr^t \cos(\omega t + \theta) \end{aligned}$$

where v and θ are constants that must be chosen to satisfy initial conditions for Y_{-1}, Y_{-2}.

This formula shows that when the roots are complex, Y_t displays oscillations with **period** $\check{p} = \frac{2\pi}{\omega}$ and **damping factor** r.

We say that $\check{p}$ is the **period** because in that amount of time the cosine wave $\cos(\omega t + \theta)$ goes through exactly one complete cycles.

(Draw a cosine function to convince yourself of this please)

Remark: Following [Sam39], we want to choose the parameters a, b of the model so that the absolute values (of the possibly complex) roots λ_1, λ_2 of the characteristic polynomial are both strictly less than one:

$$|\lambda_j| < 1 \qquad \text{for } j = 1, 2$$

Remark: When both roots λ_1, λ_2 of the characteristic polynomial have absolute values strictly less than one, the absolute value of the larger one governs the rate of convergence to the steady state of the non stochastic version of the model.

3.2.3 Things This Lecture Does

We write a function to generate simulations of a $\{Y_t\}$ sequence as a function of time.

The function requires that we put in initial conditions for Y_{-1}, Y_{-2}.

The function checks that a, b are set so that λ_1, λ_2 are less than unity in absolute value (also called "modulus").

The function also tells us whether the roots are complex, and, if they are complex, returns both their real and complex parts.

If the roots are both real, the function returns their values.

We use our function written to simulate paths that are stochastic (when $\sigma > 0$).

We have written the function in a way that allows us to input $\{G_t\}$ paths of a few simple forms, e.g.,

- one time jumps in G at some time
- a permanent jump in G that occurs at some time

We proceed to use the Samuelson multiplier-accelerator model as a laboratory to make a simple OOP example.

The "state" that determines next period's Y_{t+1} is now not just the current value Y_t but also the once lagged value Y_{t-1}.

This involves a little more bookkeeping than is required in the Solow model class definition.

We use the Samuelson multiplier-accelerator model as a vehicle for teaching how we can gradually add more features to the class.

We want to have a method in the class that automatically generates a simulation, either non-stochastic ($\sigma = 0$) or stochastic ($\sigma > 0$).

We also show how to map the Samuelson model into a simple instance of the `LinearStateSpace` class described here.

We can use a `LinearStateSpace` instance to do various things that we did above with our homemade function and class.

Among other things, we show by example that the eigenvalues of the matrix A that we use to form the instance of the `LinearStateSpace` class for the Samuelson model equal the roots of the characteristic polynomial (3.7) for the Samuelson multiplier accelerator model.

Here is the formula for the matrix A in the linear state space system in the case that government expenditures are a constant G:

$$A = \begin{bmatrix} 1 & 0 & 0 \\ \gamma + G & \rho_1 & \rho_2 \\ 0 & 1 & 0 \end{bmatrix}$$

3.3 Implementation

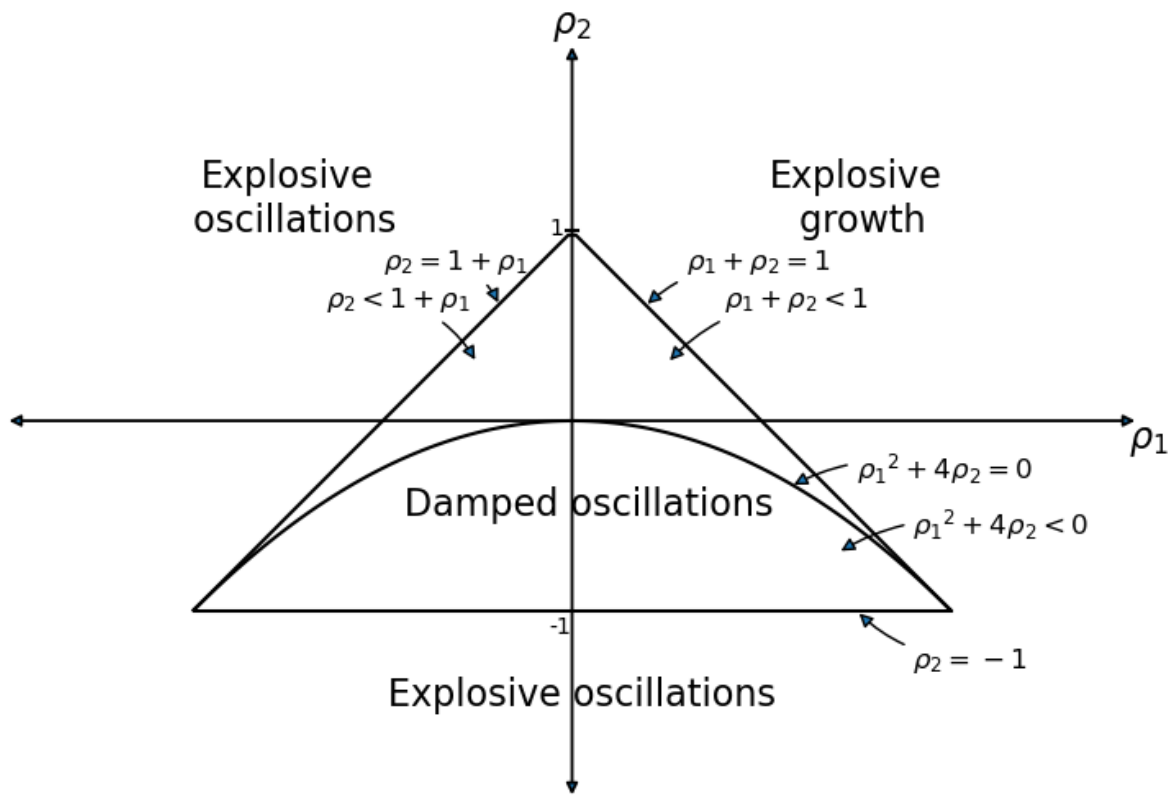
We'll start by drawing an informative graph from page 189 of [Sar87]

```
def param_plot():

    """This function creates the graph on page 189 of
    Sargent Macroeconomic Theory, second edition, 1987.
    """

    fig, ax = plt.subplots(figsize=(10, 6))
    ax.set_aspect('equal')

    # Set axis
    xmin, ymin = -3, -2
    xmax, ymax = -xmin, -ymin
    plt.axis([xmin, xmax, ymin, ymax])

    # Set axis labels
    ax.set(xticks=[], yticks=[])
    ax.set_xlabel(r'$\rho_2$', fontsize=16)
    ax.xaxis.set_label_position('top')
    ax.set_ylabel(r'$\rho_1$', rotation=0, fontsize=16)
    ax.yaxis.set_label_position('right')

    # Draw (t1, t2) points
    ρ1 = np.linspace(-2, 2, 100)
    ax.plot(ρ1, -abs(ρ1) + 1, c='black')
    ax.plot(ρ1, np.full_like(ρ1, -1), c='black')
    ax.plot(ρ1, -(ρ1**2 / 4), c='black')

    # Turn normal axes off
    for spine in ['left', 'bottom', 'top', 'right']:
        ax.spines[spine].set_visible(False)
```

(continues on next page)

(continued from previous page)

```
    # Add arrows to represent axes
    axes_arrows = {'arrowstyle': '<|-|>', 'lw': 1.3}
    ax.annotate('', xy=(xmin, 0), xytext=(xmax, 0), arrowprops=axes_arrows)
    ax.annotate('', xy=(0, ymin), xytext=(0, ymax), arrowprops=axes_arrows)

    # Annotate the plot with equations
    plot_arrowsl = {'arrowstyle': '-|>', 'connectionstyle': "arc3, rad=-0.2"}
    plot_arrowsr = {'arrowstyle': '-|>', 'connectionstyle': "arc3, rad=0.2"}
    ax.annotate(r'$\rho_1 + \rho_2 < 1$', xy=(0.5, 0.3), xytext=(0.8, 0.6),
                arrowprops=plot_arrowsr, fontsize='12')
    ax.annotate(r'$\rho_1 + \rho_2 = 1$', xy=(0.38, 0.6), xytext=(0.6, 0.8),
                arrowprops=plot_arrowsr, fontsize='12')
    ax.annotate(r'$\rho_2 < 1 + \rho_1$', xy=(-0.5, 0.3), xytext=(-1.3, 0.6),
                arrowprops=plot_arrowsl, fontsize='12')
    ax.annotate(r'$\rho_2 = 1 + \rho_1$', xy=(-0.38, 0.6), xytext=(-1, 0.8),
                arrowprops=plot_arrowsl, fontsize='12')
    ax.annotate(r'$\rho_2 = -1$', xy=(1.5, -1), xytext=(1.8, -1.3),
                arrowprops=plot_arrowsl, fontsize='12')
    ax.annotate(r'${\rho_1}^2 + 4\rho_2 = 0$', xy=(1.15, -0.35),
                xytext=(1.5, -0.3), arrowprops=plot_arrowsr, fontsize='12')
    ax.annotate(r'${\rho_1}^2 + 4\rho_2 < 0$', xy=(1.4, -0.7),
                xytext=(1.8, -0.6), arrowprops=plot_arrowsr, fontsize='12')

    # Label categories of solutions
    ax.text(1.5, 1, 'Explosive\n growth', ha='center', fontsize=16)
    ax.text(-1.5, 1, 'Explosive\n oscillations', ha='center', fontsize=16)
    ax.text(0.05, -1.5, 'Explosive oscillations', ha='center', fontsize=16)
    ax.text(0.09, -0.5, 'Damped oscillations', ha='center', fontsize=16)

    # Add small marker to y-axis
    ax.axhline(y=1.005, xmin=0.495, xmax=0.505, c='black')
    ax.text(-0.12, -1.12, '-1', fontsize=10)
    ax.text(-0.12, 0.98, '1', fontsize=10)

    return fig

param_plot()
plt.show()
```



The graph portrays regions in which the (λ_1, λ_2) root pairs implied by the $(\rho_1 = (a+b), \rho_2 = -b)$ difference equation parameter pairs in the Samuelson model are such that:

- (λ_1, λ_2) are complex with modulus less than 1 - in this case, the $\{Y_t\}$ sequence displays damped oscillations.
- (λ_1, λ_2) are both real, but one is strictly greater than 1 - this leads to explosive growth.
- (λ_1, λ_2) are both real, but one is strictly less than -1 - this leads to explosive oscillations.
- (λ_1, λ_2) are both real and both are less than 1 in absolute value - in this case, there is smooth convergence to the steady state without damped cycles.

Later we'll present the graph with a red mark showing the particular point implied by the setting of (a, b).

3.3.1 Function to Describe Implications of Characteristic Polynomial

```
def categorize_solution(ρ1, ρ2):

    """This function takes values of ρ1 and ρ2 and uses them
    to classify the type of solution
    """

    discriminant = ρ1 ** 2 + 4 * ρ2
    if ρ2 > 1 + ρ1 or ρ2 < -1:
        print('Explosive oscillations')
    elif ρ1 + ρ2 > 1:
        print('Explosive growth')
    elif discriminant < 0:
        print('Roots are complex with modulus less than one; \
```

(continues on next page)

(continued from previous page)

```
therefore damped oscillations')
    else:
        print('Roots are real and absolute values are less than one; \
therefore get smooth convergence to a steady state')
```

```
### Test the categorize_solution function

categorize_solution(1.3, -.4)
```

```
Roots are real and absolute values are less than one; therefore get smooth␣
↪convergence to a steady state
```

3.3.2 Function for Plotting Paths

A useful function for our work below is

```
def plot_y(function=None):

    """Function plots path of Y_t"""

    plt.subplots(figsize=(10, 6))
    plt.plot(function)
    plt.xlabel('Time $t$')
    plt.ylabel('$Y_t$', rotation=0)
    plt.grid()
    plt.show()
```

3.3.3 Manual or "by hand" Root Calculations

The following function calculates roots of the characteristic polynomial using high school algebra.

(We'll calculate the roots in other ways later)

The function also plots a Y_t starting from initial conditions that we set

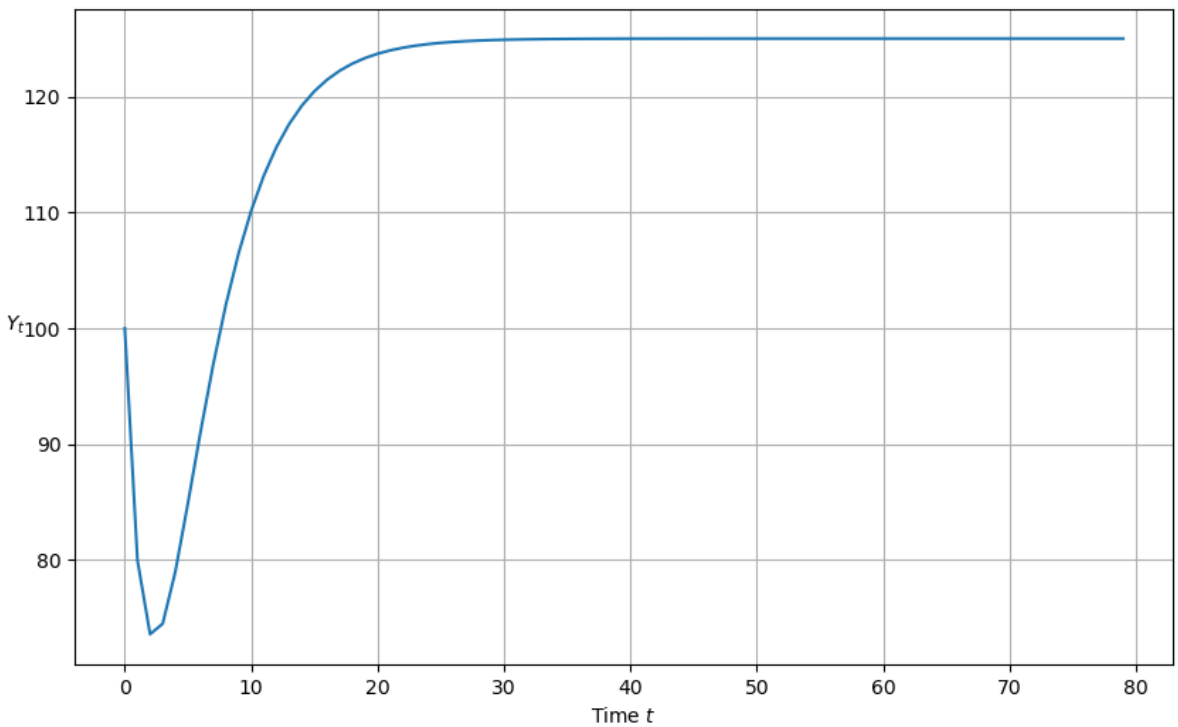
```
# This is a 'manual' method

def y_nonstochastic(y_0=100, y_1=80, α=.92, β=.5, γ=10, n=80):

    """Takes values of parameters and computes the roots of characteristic
    polynomial. It tells whether they are real or complex and whether they
    are less than unity in absolute value.It also computes a simulation of
    length n starting from the two given initial conditions for national
    income
    """

    roots = []

    ρ1 = α + β
    ρ2 = -β
```

(continues on next page)

(continued from previous page)

```
    print(f'ρ_1 is {ρ1}')
    print(f'ρ_2 is {ρ2}')

    discriminant = ρ1 ** 2 + 4 * ρ2

    if discriminant == 0:
        roots.append(-ρ1 / 2)
        print('Single real root: ')
        print(''.join(str(roots)))
    elif discriminant > 0:
        roots.append((-ρ1 + sqrt(discriminant).real) / 2)
        roots.append((-ρ1 - sqrt(discriminant).real) / 2)
        print('Two real roots: ')
        print(''.join(str(roots)))
    else:
        roots.append((-ρ1 + sqrt(discriminant)) / 2)
        roots.append((-ρ1 - sqrt(discriminant)) / 2)
        print('Two complex roots: ')
        print(''.join(str(roots)))

    if all(abs(root) < 1 for root in roots):
        print('Absolute values of roots are less than one')
    else:
        print('Absolute values of roots are not less than one')

    def transition(x, t): return ρ1 * x[t - 1] + ρ2 * x[t - 2] + y

    y_t = [y_0, y_1]

    for t in range(2, n):
        y_t.append(transition(y_t, t))

    return y_t

plot_y(y_nonstochastic())
```

```
ρ_1 is 1.42
ρ_2 is -0.5
Two real roots:
[-0.6459687576256715, -0.7740312423743284]
Absolute values of roots are less than one
```



3.3.4 Reverse-Engineering Parameters to Generate Damped Cycles

The next cell writes code that takes as inputs the modulus r and phase ϕ of a conjugate pair of complex numbers in polar form

$$\lambda_1 = r\exp(i\phi), \quad \lambda_2 = r\exp(-i\phi)$$

- The code assumes that these two complex numbers are the roots of the characteristic polynomial
- It then reverse-engineers (a, b) and (ρ_1, ρ_2), pairs that would generate those roots

```
### code to reverse-engineer a cycle
### y_t = r^t (c_1 cos(ϕ t) + c2 sin(ϕ t))
###

def f(r, ϕ):
    """
    Takes modulus r and angle ϕ of complex number r exp(j ϕ)
    and creates ρ1 and ρ2 of characteristic polynomial for which
    r exp(j ϕ) and r exp(- j ϕ) are complex roots.

    Returns the multiplier coefficient a and the accelerator coefficient b
    that verifies those roots.
    """
    g1 = cmath.rect(r, ϕ)  # Generate two complex roots
    g2 = cmath.rect(r, -ϕ)
    ρ1 = g1 + g2           # Implied ρ1, ρ2
    ρ2 = -g1 * g2
    b = -ρ2                # Reverse-engineer a and b that validate these
    a = ρ1 - b
```

(continues on next page)

(continued from previous page)

```
    return ρ1, ρ2, a, b

## Now let's use the function in an example
## Here are the example parameters

r = .95
period = 10                # Length of cycle in units of time
ϕ = 2 * math.pi/period

## Apply the function

ρ1, ρ2, a, b = f(r, ϕ)

print(f"a, b = {a}, {b}")
print(f"ρ1, ρ2 = {ρ1}, {ρ2}")
```

```
a, b = (0.6346322893124001+0j), (0.9024999999999999-0j)
ρ1, ρ2 = (1.5371322893124+0j), (-0.9024999999999999+0j)
```

```
## Print the real components of ρ1 and ρ2

ρ1 = ρ1.real
ρ2 = ρ2.real

ρ1, ρ2
```

```
(1.5371322893124, -0.9024999999999999)
```

3.3.5 Root Finding Using Numpy

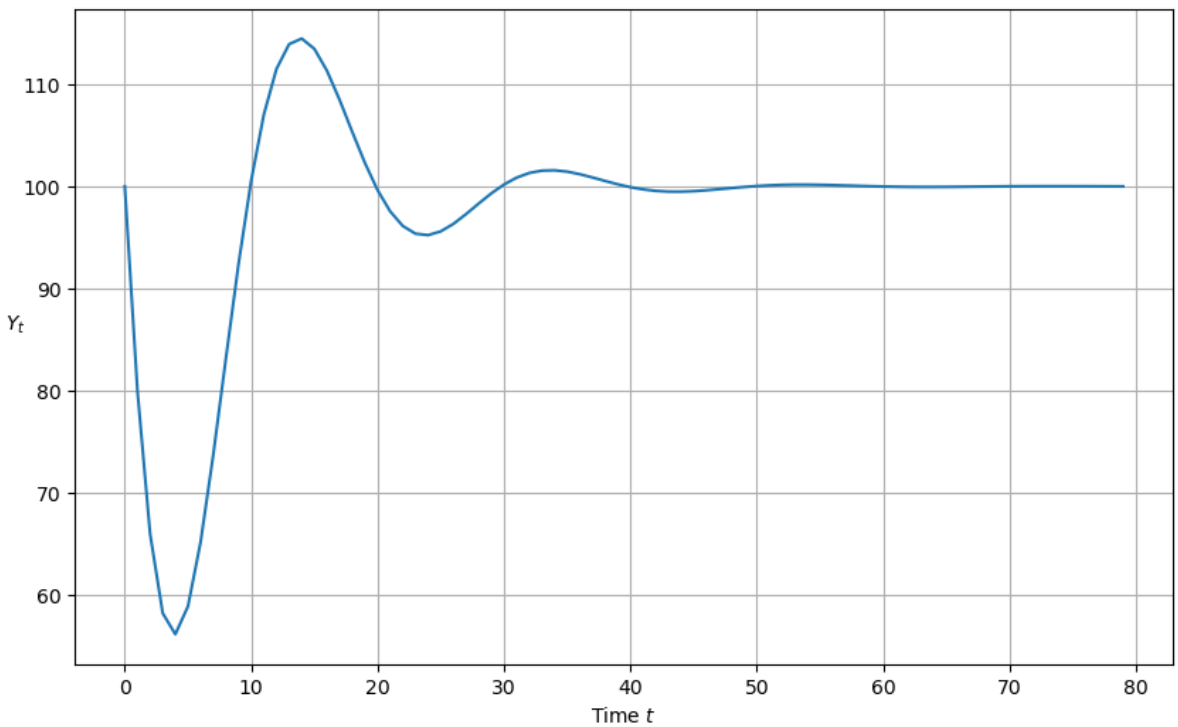
Here we'll use numpy to compute the roots of the characteristic polynomial

```
r1, r2 = np.roots([1, -ρ1, -ρ2])

p1 = cmath.polar(r1)
p2 = cmath.polar(r2)

print(f"r, ϕ = {r}, {ϕ}")
print(f"p1, p2 = {p1}, {p2}")
# print(f"g1, g2 = {g1}, {g2}")

print(f"a, b = {a}, {b}")
print(f"ρ1, ρ2 = {ρ1}, {ρ2}")
```

```
r, ϕ = 0.95, 0.6283185307179586
p1, p2 = (0.95, 0.6283185307179586), (0.95, -0.6283185307179586)
a, b = (0.6346322893124001+0j), (0.9024999999999999-0j)
ρ1, ρ2 = 1.5371322893124, -0.9024999999999999
```

```python
##=== This method uses numpy to calculate roots ===#


def y_nonstochastic(y_0=100, y_1=80, α=.9, β=.8, γ=10, n=80):

    """ Rather than computing the roots of the characteristic
    polynomial by hand as we did earlier, this function
    enlists numpy to do the work for us
    """

    # Useful constants
    ρ1 = α + β
    ρ2 = -β

    categorize_solution(ρ1, ρ2)

    # Find roots of polynomial
    roots = np.roots([1, -ρ1, -ρ2])
    print(f'Roots are {roots}')

    # Check if real or complex
    if all(isinstance(root, complex) for root in roots):
        print('Roots are complex')
    else:
        print('Roots are real')

    # Check if roots are less than one
    if all(abs(root) < 1 for root in roots):
        print('Roots are less than one')
    else:
        print('Roots are not less than one')

    # Define transition equation
    def transition(x, t): return ρ1 * x[t - 1] + ρ2 * x[t - 2] + γ

    # Set initial conditions
    y_t = [y_0, y_1]

    # Generate y_t series
    for t in range(2, n):
        y_t.append(transition(y_t, t))

    return y_t

plot_y(y_nonstochastic())
```

```
Roots are complex with modulus less than one; therefore damped oscillations
Roots are [0.85+0.27838822j 0.85-0.27838822j]
Roots are complex
Roots are less than one
```



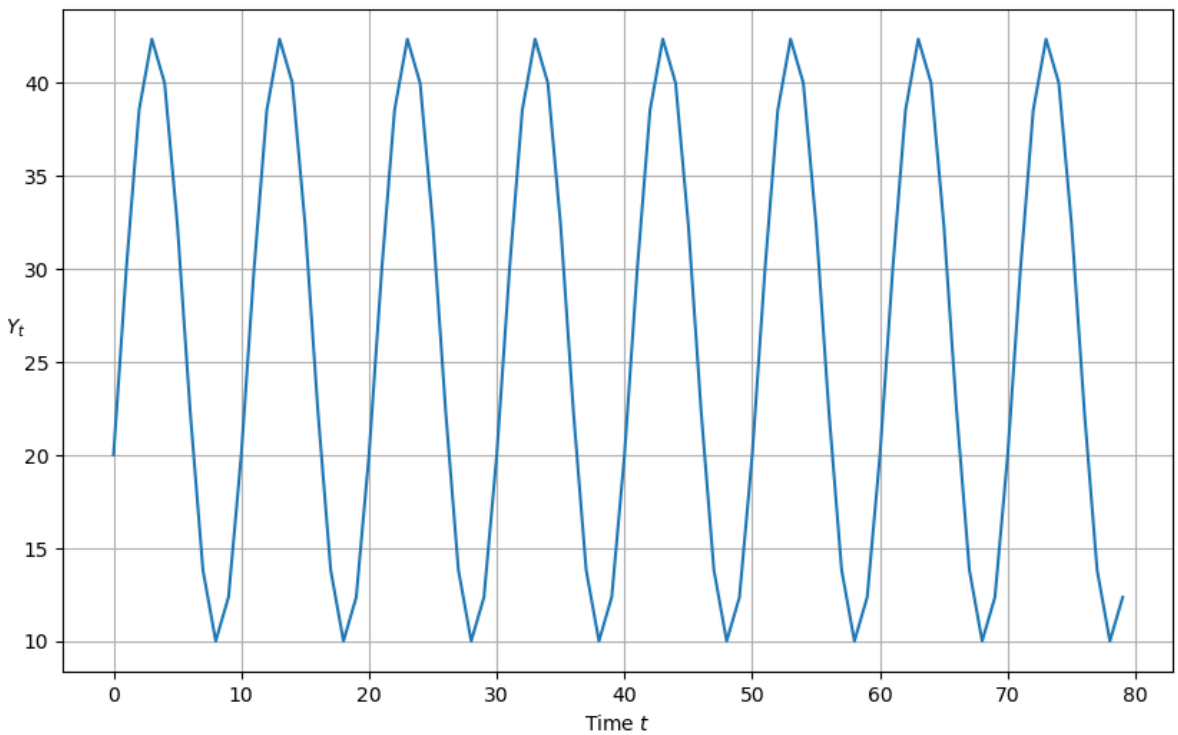
3.3.6 Reverse-Engineered Complex Roots: Example

The next cell studies the implications of reverse-engineered complex roots.

We'll generate an **undamped** cycle of period 10

```
r = 1   # Generates undamped, nonexplosive cycles

period = 10   # Length of cycle in units of time
ϕ = 2 * math.pi/period

## Apply the reverse-engineering function f

ρ1, ρ2, a, b = f(r, ϕ)

# Drop the imaginary part so that it is a valid input into y_nonstochastic
a = a.real
b = b.real

print(f"a, b = {a}, {b}")

ytemp = y_nonstochastic(α=a, β=b, y_0=20, y_1=30)
plot_y(ytemp)
```

```
a, b = 0.6180339887498949, 1.0
Roots are complex with modulus less than one; therefore damped oscillations
Roots are [0.80901699+0.58778525j 0.80901699-0.58778525j]
Roots are complex
Roots are less than one
```



3.3.7 Digression: Using Sympy to Find Roots

We can also use sympy to compute analytic formulas for the roots

```
init_printing()

r1 = Symbol("ρ_1")
r2 = Symbol("ρ_2")
z = Symbol("z")

sympy.solve(z**2 - r1*z - r2, z)
```

$$\left[\frac{\rho_1}{2} - \frac{\sqrt{\rho_1^2 + 4\rho_2}}{2}, \ \frac{\rho_1}{2} + \frac{\sqrt{\rho_1^2 + 4\rho_2}}{2} \right]$$

```
a = Symbol("α")
b = Symbol("β")
r1 = a + b
r2 = -b

sympy.solve(z**2 - r1*z - r2, z)
```

$$\left[\frac{\alpha}{2} + \frac{\beta}{2} - \frac{\sqrt{\alpha^2 + 2\alpha\beta + \beta^2 - 4\beta}}{2}, \ \frac{\alpha}{2} + \frac{\beta}{2} + \frac{\sqrt{\alpha^2 + 2\alpha\beta + \beta^2 - 4\beta}}{2} \right]$$

3.4 Stochastic Shocks

Now we'll construct some code to simulate the stochastic version of the model that emerges when we add a random shock process to aggregate demand

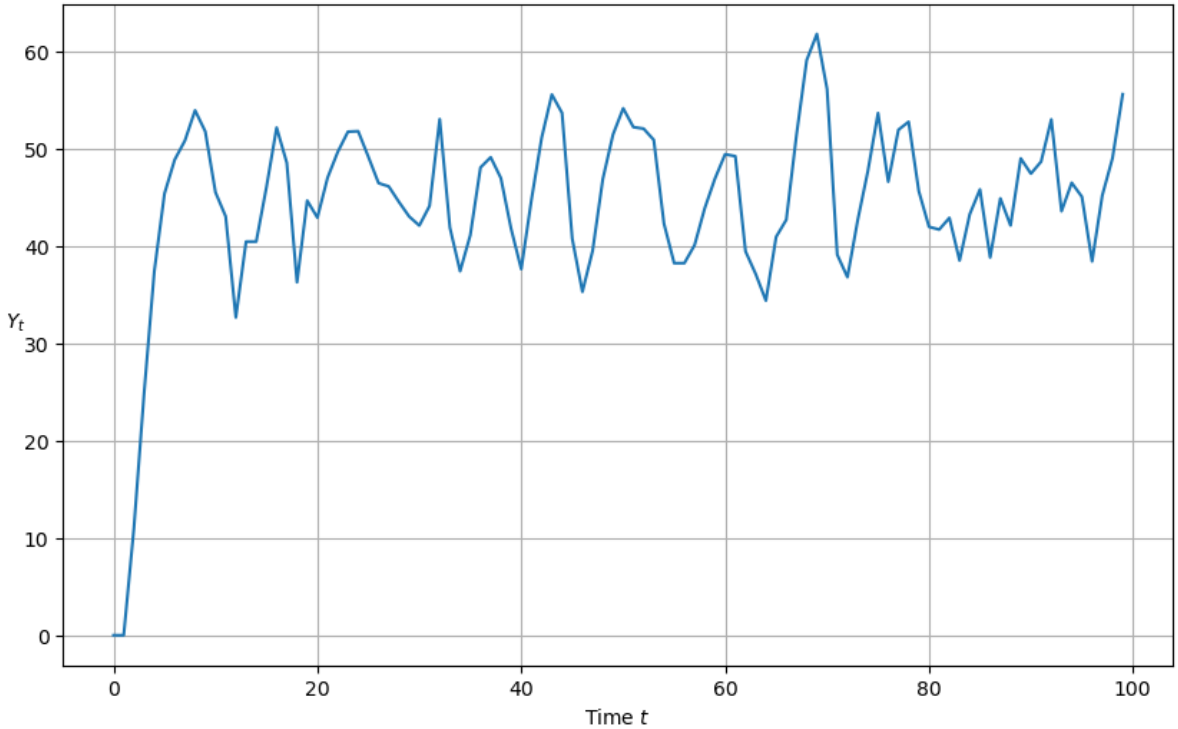
```
def y_stochastic(y_0=0, y_1=0, α=0.8, β=0.2, γ=10, n=100, σ=5):

    """This function takes parameters of a stochastic version of
    the model and proceeds to analyze the roots of the characteristic
    polynomial and also generate a simulation.
    """

    # Useful constants
    ρ1 = α + β
    ρ2 = -β

    # Categorize solution
    categorize_solution(ρ1, ρ2)

    # Find roots of polynomial
    roots = np.roots([1, -ρ1, -ρ2])
    print(roots)

    # Check if real or complex
    if all(isinstance(root, complex) for root in roots):
        print('Roots are complex')
    else:
        print('Roots are real')

    # Check if roots are less than one
    if all(abs(root) < 1 for root in roots):
        print('Roots are less than one')
    else:
        print('Roots are not less than one')

    # Generate shocks
    ϵ = np.random.normal(0, 1, n)

    # Define transition equation
    def transition(x, t): return ρ1 * \
        x[t - 1] + ρ2 * x[t - 2] + γ + σ * ϵ[t]

    # Set initial conditions
    y_t = [y_0, y_1]

    # Generate y_t series
    for t in range(2, n):
        y_t.append(transition(y_t, t))

    return y_t

plot_y(y_stochastic())
```

```
Roots are real and absolute values are less than one; therefore get smooth␣
↪convergence to a steady state
[0.7236068 0.2763932]
```

(continues on next page)

(continued from previous page)

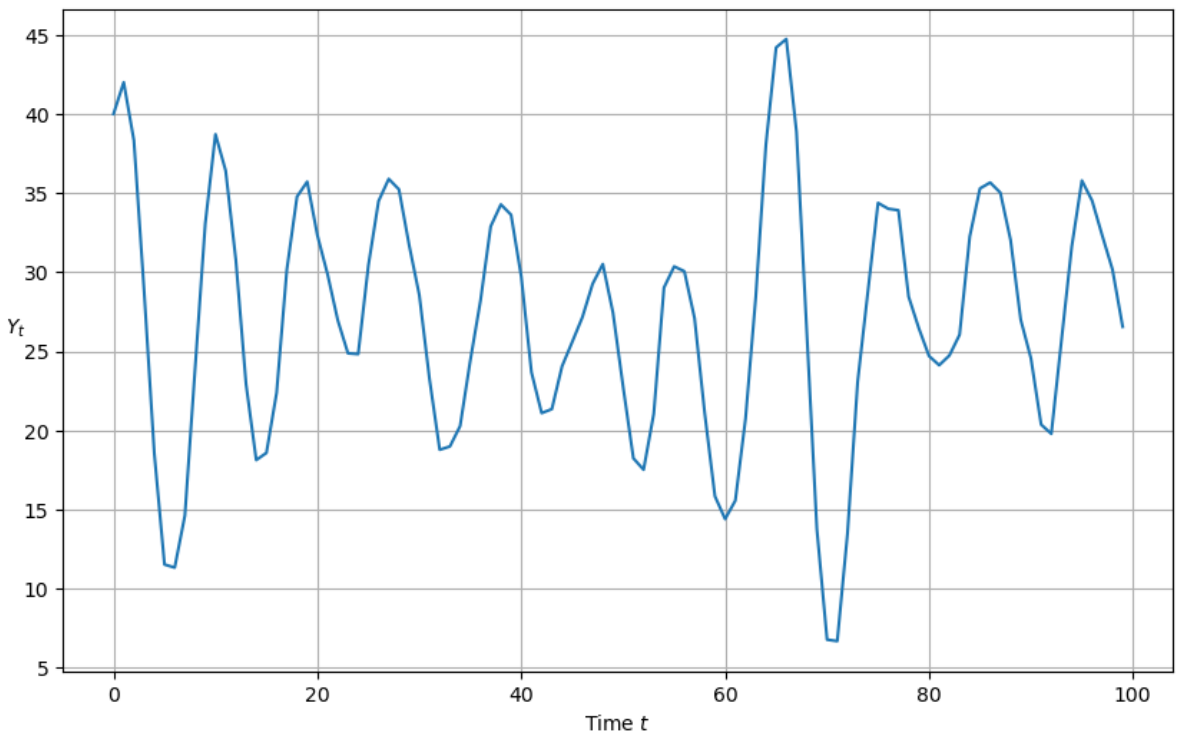
```
Roots are real
Roots are less than one
```



Let's do a simulation in which there are shocks and the characteristic polynomial has complex roots

```
r = .97

period = 10   #  Length of cycle in units of time
ϕ = 2 * math.pi/period

### Apply the  reverse-engineering function f

ρ1, ρ2, a, b = f(r, ϕ)

# Drop the imaginary part so that it is a valid input into y_nonstochastic
a = a.real
b = b.real

print(f"a, b = {a}, {b}")
plot_y(y_stochastic(y_0=40, y_1 = 42, α=a, β=b, σ=2, n=100))
```

```
a, b = 0.6285929690873979, 0.9409000000000001
Roots are complex with modulus less than one; therefore damped oscillations
[0.78474648+0.57015169j 0.78474648-0.57015169j]
Roots are complex
Roots are less than one
```



3.5 Government Spending

This function computes a response to either a permanent or one-off increase in government expenditures

```
def y_stochastic_g(y_0=20,
                   y_1=20,
                   α=0.8,
                   β=0.2,
                   γ=10,
                   n=100,
                   σ=2,
                   g=0,
                   g_t=0,
                   duration='permanent'):

    """This program computes a response to a permanent increase
    in government expenditures that occurs at time 20
    """

    # Useful constants
    ρ1 = α + β
    ρ2 = -β

    # Categorize solution
    categorize_solution(ρ1, ρ2)

    # Find roots of polynomial
    roots = np.roots([1, -ρ1, -ρ2])
    print(roots)
```

(continues on next page)

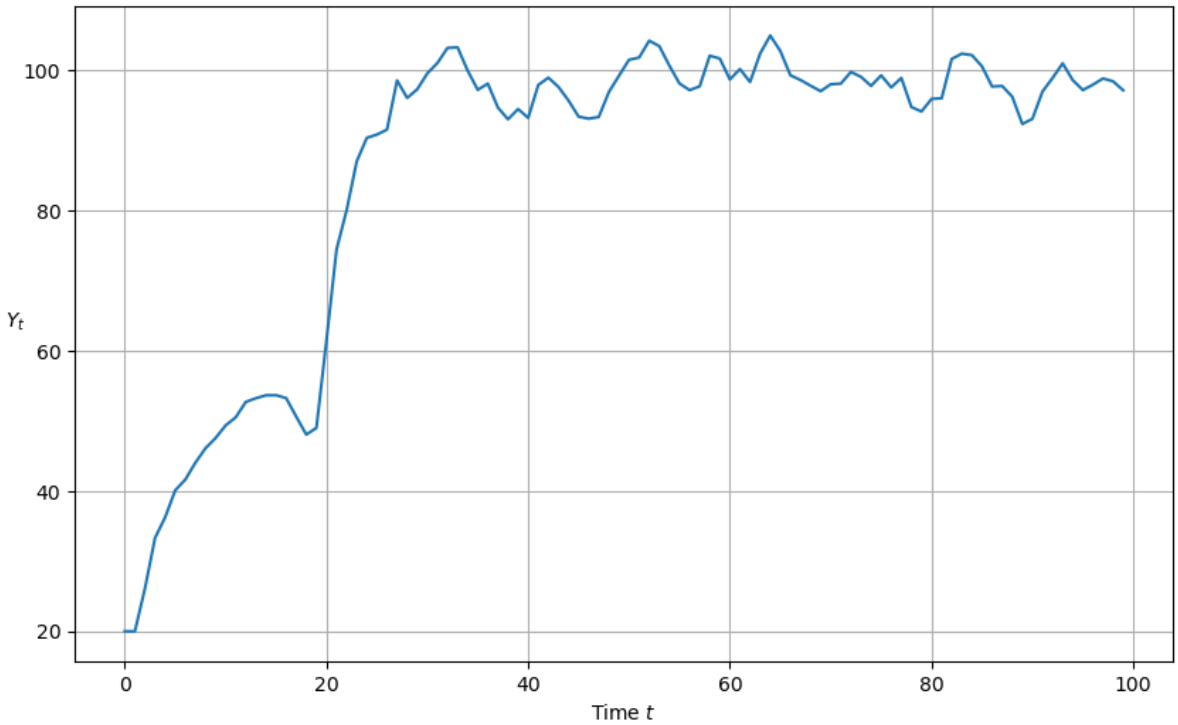
(continued from previous page)

```
    # Check if real or complex
    if all(isinstance(root, complex) for root in roots):
        print('Roots are complex')
    else:
        print('Roots are real')

    # Check if roots are less than one
    if all(abs(root) < 1 for root in roots):
        print('Roots are less than one')
    else:
        print('Roots are not less than one')

    # Generate shocks
    ϵ = np.random.normal(0, 1, n)

    def transition(x, t, g):

        # Non-stochastic - separated to avoid generating random series
        # when not needed
        if σ == 0:
            return ρ1 * x[t - 1] + ρ2 * x[t - 2] + γ + g

        # Stochastic
        else:
            ϵ = np.random.normal(0, 1, n)
            return ρ1 * x[t - 1] + ρ2 * x[t - 2] + γ + g + σ * ϵ[t]

    # Create list and set initial conditions
    y_t = [y_0, y_1]

    # Generate y_t series
    for t in range(2, n):

        # No government spending
        if g == 0:
            y_t.append(transition(y_t, t))

        # Government spending (no shock)
        elif g != 0 and duration == None:
            y_t.append(transition(y_t, t))

        # Permanent government spending shock
        elif duration == 'permanent':
            if t < g_t:
                y_t.append(transition(y_t, t, g=0))
            else:
                y_t.append(transition(y_t, t, g=g))

        # One-off government spending shock
        elif duration == 'one-off':
            if t == g_t:
                y_t.append(transition(y_t, t, g=g))
            else:
                y_t.append(transition(y_t, t, g=0))
    return y_t
```

A permanent government spending shock can be simulated as follows

```
plot_y(y_stochastic_g(g=10, g_t=20, duration='permanent'))
```

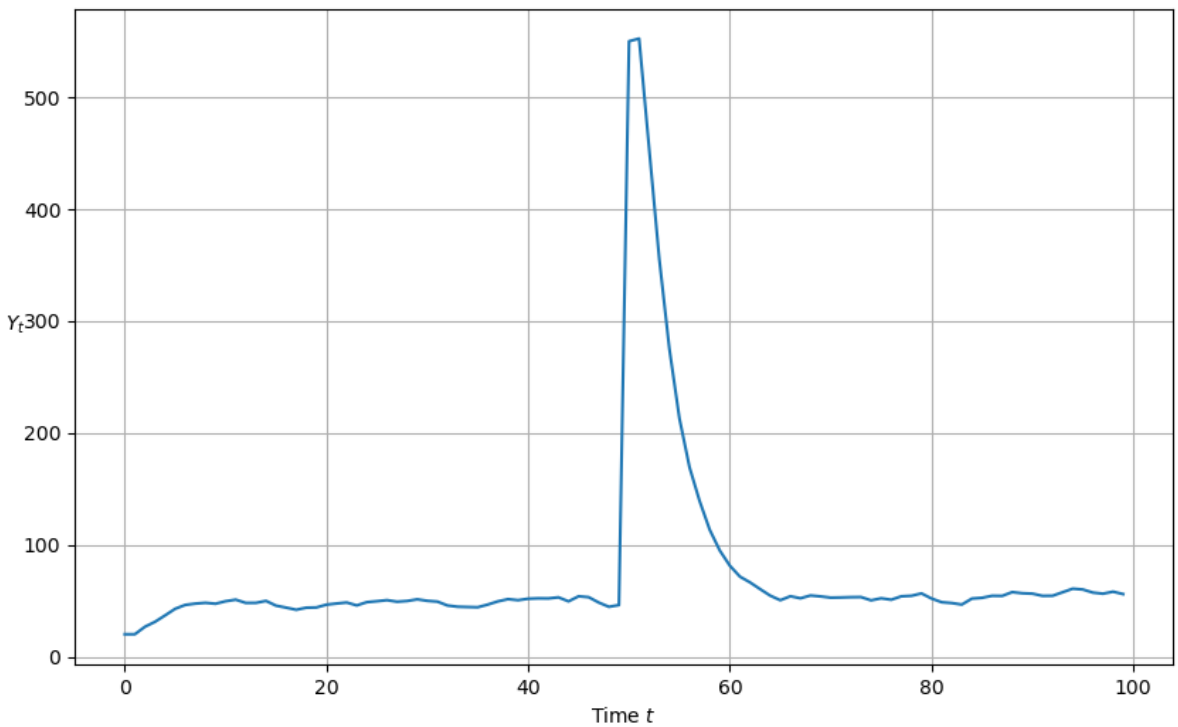
```
Roots are real and absolute values are less than one; therefore get smooth␣
↪convergence to a steady state
[0.7236068 0.2763932]
Roots are real
Roots are less than one
```



We can also see the response to a one time jump in government expenditures

```
plot_y(y_stochastic_g(g=500, g_t=50, duration='one-off'))
```

```
Roots are real and absolute values are less than one; therefore get smooth␣
↪convergence to a steady state
[0.7236068 0.2763932]
Roots are real
Roots are less than one
```



3.6 Wrapping Everything Into a Class

Up to now, we have written functions to do the work.

Now we'll roll up our sleeves and write a Python class called `Samuelson` for the Samuelson model

```
class Samuelson():

    """This class represents the Samuelson model, otherwise known as the
    multiple-accelerator model. The model combines the Keynesian multiplier
    with the accelerator theory of investment.

    The path of output is governed by a linear second-order difference equation

    .. math::

        Y_t =  + \alpha (1 + \beta) Y_{t-1} - \alpha \beta Y_{t-2}

    Parameters
    ----------
    y_0 : scalar
        Initial condition for Y_0
    y_1 : scalar
        Initial condition for Y_1
    α : scalar
        Marginal propensity to consume
    β : scalar
        Accelerator coefficient
    n : int
```

(continues on next page)

(continued from previous page)

```
        Number of iterations
    σ : scalar
        Volatility parameter. It must be greater than or equal to 0. Set
        equal to 0 for a non-stochastic model.
    g : scalar
        Government spending shock
    g_t : int
        Time at which government spending shock occurs. Must be specified
        when duration != None.
    duration : {None, 'permanent', 'one-off'}
        Specifies type of government spending shock. If none, government
        spending equal to g for all t.

    """

    def __init__(self,
                 y_0=100,
                 y_1=50,
                 α=1.3,
                 β=0.2,
                 γ=10,
                 n=100,
                 σ=0,
                 g=0,
                 g_t=0,
                 duration=None):

        self.y_0, self.y_1, self.α, self.β = y_0, y_1, α, β
        self.n, self.g, self.g_t, self.duration = n, g, g_t, duration
        self.γ, self.σ = γ, σ
        self.ρ1 = α + β
        self.ρ2 = -β
        self.roots = np.roots([1, -self.ρ1, -self.ρ2])

    def root_type(self):
        if all(isinstance(root, complex) for root in self.roots):
            return 'Complex conjugate'
        elif len(self.roots) > 1:
            return 'Double real'
        else:
            return 'Single real'

    def root_less_than_one(self):
        if all(abs(root) < 1 for root in self.roots):
            return True

    def solution_type(self):
        ρ1, ρ2 = self.ρ1, self.ρ2
        discriminant = ρ1 ** 2 + 4 * ρ2
        if ρ2 >= 1 + ρ1 or ρ2 <= -1:
            return 'Explosive oscillations'
        elif ρ1 + ρ2 >= 1:
            return 'Explosive growth'
        elif discriminant < 0:
            return 'Damped oscillations'
        else:
```

(continues on next page)

(continued from previous page)

```
            return 'Steady state'

    def _transition(self, x, t, g):

        # Non-stochastic - separated to avoid generating random series
        # when not needed
        if self.σ == 0:
            return self.ρ1 * x[t - 1] + self.ρ2 * x[t - 2] + self.γ + g

        # Stochastic
        else:
            ϵ = np.random.normal(0, 1, self.n)
            return self.ρ1 * x[t - 1] + self.ρ2 * x[t - 2] + self.γ + g \
                + self.σ * ϵ[t]

    def generate_series(self):

        # Create list and set initial conditions
        y_t = [self.y_0, self.y_1]

        # Generate y_t series
        for t in range(2, self.n):

            # No government spending
            if self.g == 0:
                y_t.append(self._transition(y_t, t))

            # Government spending (no shock)
            elif self.g != 0 and self.duration == None:
                y_t.append(self._transition(y_t, t))

            # Permanent government spending shock
            elif self.duration == 'permanent':
                if t < self.g_t:
                    y_t.append(self._transition(y_t, t, g=0))
                else:
                    y_t.append(self._transition(y_t, t, g=self.g))

            # One-off government spending shock
            elif self.duration == 'one-off':
                if t == self.g_t:
                    y_t.append(self._transition(y_t, t, g=self.g))
                else:
                    y_t.append(self._transition(y_t, t, g=0))
        return y_t

    def summary(self):
        print('Summary\n' + '-' * 50)
        print(f'Root type: {self.root_type()}')
        print(f'Solution type: {self.solution_type()}')
        print(f'Roots: {str(self.roots)}')

        if self.root_less_than_one() == True:
            print('Absolute value of roots is less than one')
        else:
            print('Absolute value of roots is not less than one')
```

(continues on next page)

(continued from previous page)

```
        if self.σ > 0:
            print('Stochastic series with σ = ' + str(self.σ))
        else:
            print('Non-stochastic series')

        if self.g != 0:
            print('Government spending equal to ' + str(self.g))

        if self.duration != None:
            print(self.duration.capitalize() +
                  ' government spending shock at t = ' + str(self.g_t))

    def plot(self):
        fig, ax = plt.subplots(figsize=(10, 6))
        ax.plot(self.generate_series())
        ax.set(xlabel='Iteration', xlim=(0, self.n))
        ax.set_ylabel('$Y_t$', rotation=0)
        ax.grid()

        # Add parameter values to plot
        paramstr = f'$\\alpha={self.α:.2f}$ \n $\\beta={self.β:.2f}$ \n \
        $\\gamma={self.γ:.2f}$ \n $\\sigma={self.σ:.2f}$ \n \
        $\\rho_1={self.ρ1:.2f}$ \n $\\rho_2={self.ρ2:.2f}$'
        props = dict(fc='white', pad=10, alpha=0.5)
        ax.text(0.87, 0.05, paramstr, transform=ax.transAxes,
                fontsize=12, bbox=props, va='bottom')

        return fig

    def param_plot(self):

        # Uses the param_plot() function defined earlier (it is then able
        # to be used standalone or as part of the model)

        fig = param_plot()
        ax = fig.gca()

        # Add λ values to legend
        for i, root in enumerate(self.roots):
            if isinstance(root, complex):
                # Need to fill operator for positive as string is split apart
                operator = ['+', '']
                label = rf'$\lambda_{i+1} = {sam.roots[i].real:.2f} {operator[i]}
↪{sam.roots[i].imag:.2f}i$'
            else:
                label = rf'$\lambda_{i+1} = {sam.roots[i].real:.2f}$'
            ax.scatter(0, 0, 0, label=label) # dummy to add to legend

        # Add ρ pair to plot
        ax.scatter(self.ρ1, self.ρ2, 100, 'red', '+',
            label=r'$(\ \rho_1, \ \rho_2 \ )$', zorder=5)

        plt.legend(fontsize=12, loc=3)

        return fig
```

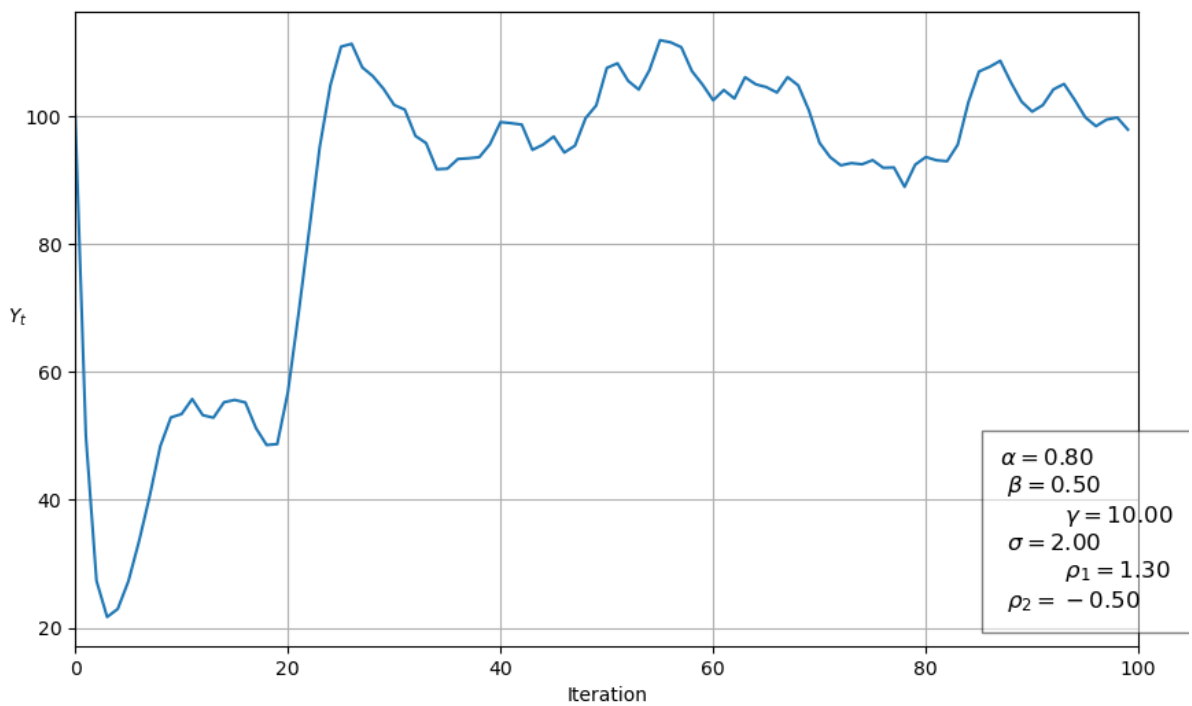
3.6.1 Illustration of Samuelson Class

Now we'll put our Samuelson class to work on an example

```
sam = Samuelson(α=0.8, β=0.5, σ=2, g=10, g_t=20, duration='permanent')
sam.summary()
```

```
Summary
--------------------------------------------------
Root type: Complex conjugate
Solution type: Damped oscillations
Roots: [0.65+0.27838822j 0.65-0.27838822j]
Absolute value of roots is less than one
Stochastic series with σ = 2
Government spending equal to 10
Permanent government spending shock at t = 20
```

```
sam.plot()
plt.show()
```

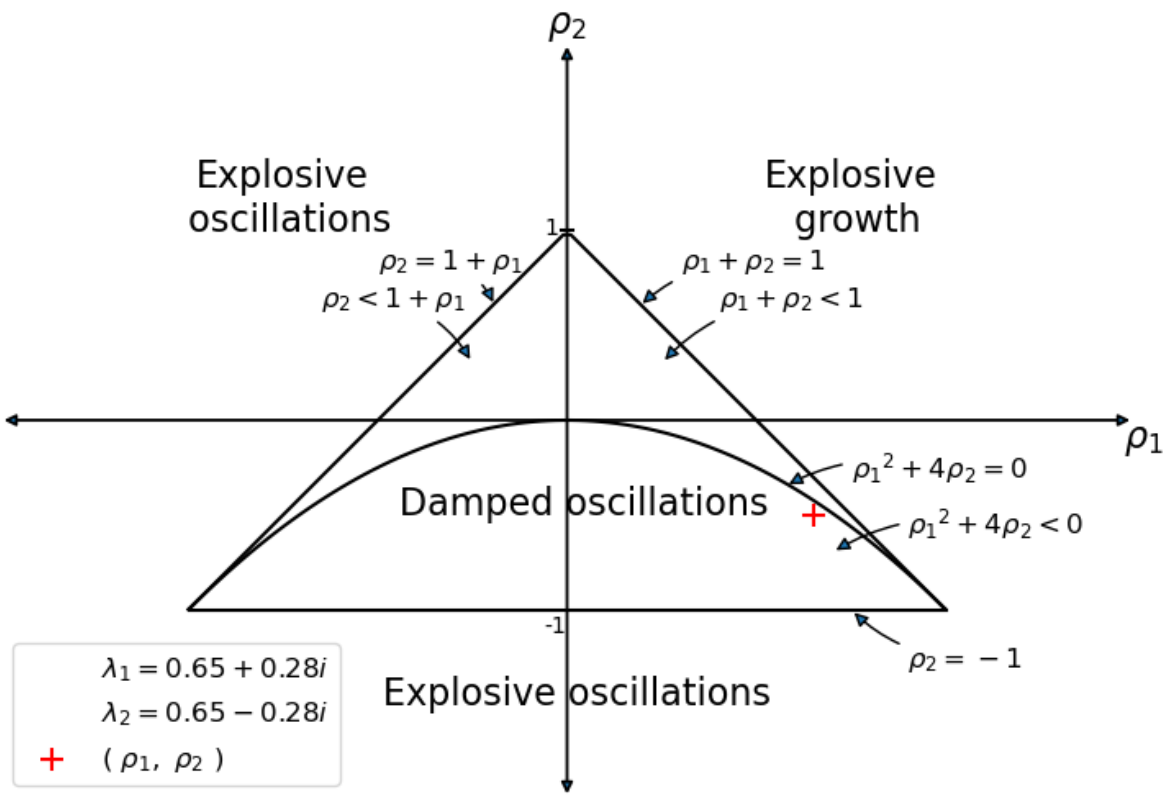


3.6.2 Using the Graph

We'll use our graph to show where the roots lie and how their location is consistent with the behavior of the path just graphed.

The red + sign shows the location of the roots

```
sam.param_plot()
plt.show()
```



3.7 Using the LinearStateSpace Class

It turns out that we can use the QuantEcon.py LinearStateSpace class to do much of the work that we have done from scratch above.

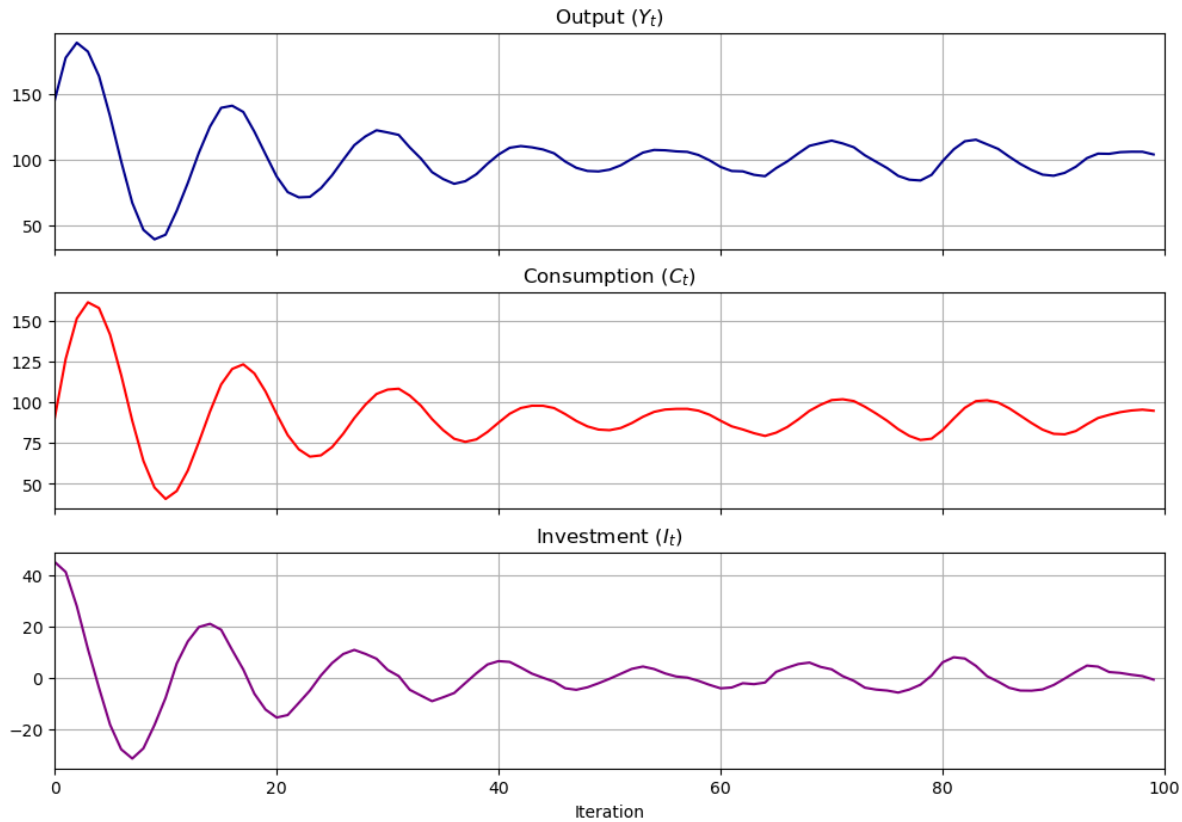
Here is how we map the Samuelson model into an instance of a `LinearStateSpace` class

```
"""This script maps the Samuelson model in the the
``LinearStateSpace`` class
"""
α = 0.8
β = 0.9
ρ1 = α + β
ρ2 = -β
γ = 10
σ = 1
```

(continues on next page)

(continued from previous page)

```
g = 10
n = 100

A = [[1,        0,      0],
     [γ + g,   ρ1,     ρ2],
     [0,        1,      0]]

G = [[γ + g, ρ1,   ρ2],         # this is Y_{t+1}
     [γ,      α,    0],         # this is C_{t+1}
     [0,      β,   -β]]         # this is I_{t+1}

μ_0 = [1, 100, 50]
C = np.zeros((3,1))
C[1] = σ # stochastic

sam_t = LinearStateSpace(A, C, G, mu_0=μ_0)

x, y = sam_t.simulate(ts_length=n)

fig, axes = plt.subplots(3, 1, sharex=True, figsize=(12, 8))
titles = ['Output ($Y_t$)', 'Consumption ($C_t$)', 'Investment ($I_t$)']
colors = ['darkblue', 'red', 'purple']
for ax, series, title, color in zip(axes, y, titles, colors):
    ax.plot(series, color=color)
    ax.set(title=title, xlim=(0, n))
    ax.grid()

axes[-1].set_xlabel('Iteration')

plt.show()
```



3.7.1 Other Methods in the `LinearStateSpace` Class

Let's plot **impulse response functions** for the instance of the Samuelson model using a method in the `LinearStateSpace` class

```
imres = sam_t.impulse_response()
imres = np.asarray(imres)
y1 = imres[:, :, 0]
y2 = imres[:, :, 1]
y1.shape
```

$$(2,\ 6,\ 1)$$

Now let's compute the zeros of the characteristic polynomial by simply calculating the eigenvalues of A

```
A = np.asarray(A)
w, v = np.linalg.eig(A)
print(w)
```

```
[0.85+0.42130749j 0.85-0.42130749j 1.  +0.j        ]
```

3.7.2 Inheriting Methods from `LinearStateSpace`

We could also create a subclass of `LinearStateSpace` (inheriting all its methods and attributes) to add more functions to use

```
class SamuelsonLSS(LinearStateSpace):

    """
    This subclass creates a Samuelson multiplier-accelerator model
    as a linear state space system.
    """
    def __init__(self,
                 y_0=100,
                 y_1=50,
                 α=0.8,
                 β=0.9,
                 γ=10,
                 σ=1,
                 g=10):

        self.α, self.β = α, β
        self.y_0, self.y_1, self.g = y_0, y_1, g
        self.γ, self.σ = γ, σ

        # Define intial conditions
        self.μ_0 = [1, y_0, y_1]

        self.ρ1 = α + β
        self.ρ2 = -β

        # Define transition matrix
        self.A = [[1,                  0,          0],
                  [γ + g,        self.ρ1,    self.ρ2],
                  [0,                  1,          0]]

        # Define output matrix
        self.G = [[γ + g, self.ρ1, self.ρ2],          # this is Y_{t+1}
                  [γ,            α,        0],          # this is C_{t+1}
                  [0,            β,       -β]]          # this is I_{t+1}

        self.C = np.zeros((3, 1))
        self.C[1] = σ  # stochastic

        # Initialize LSS with parameters from Samuelson model
        LinearStateSpace.__init__(self, self.A, self.C, self.G, mu_0=self.μ_0)

    def plot_simulation(self, ts_length=100, stationary=True):

        # Temporarily store original parameters
        temp_mu = self.mu_0
        temp_Sigma = self.Sigma_0

        # Set distribution parameters equal to their stationary
        # values for simulation
        if stationary == True:
            try:
                self.mu_x, self.mu_y, self.Sigma_x, self.Sigma_y, self.Sigma_yx = \
```

(continues on next page)

(continued from previous page)

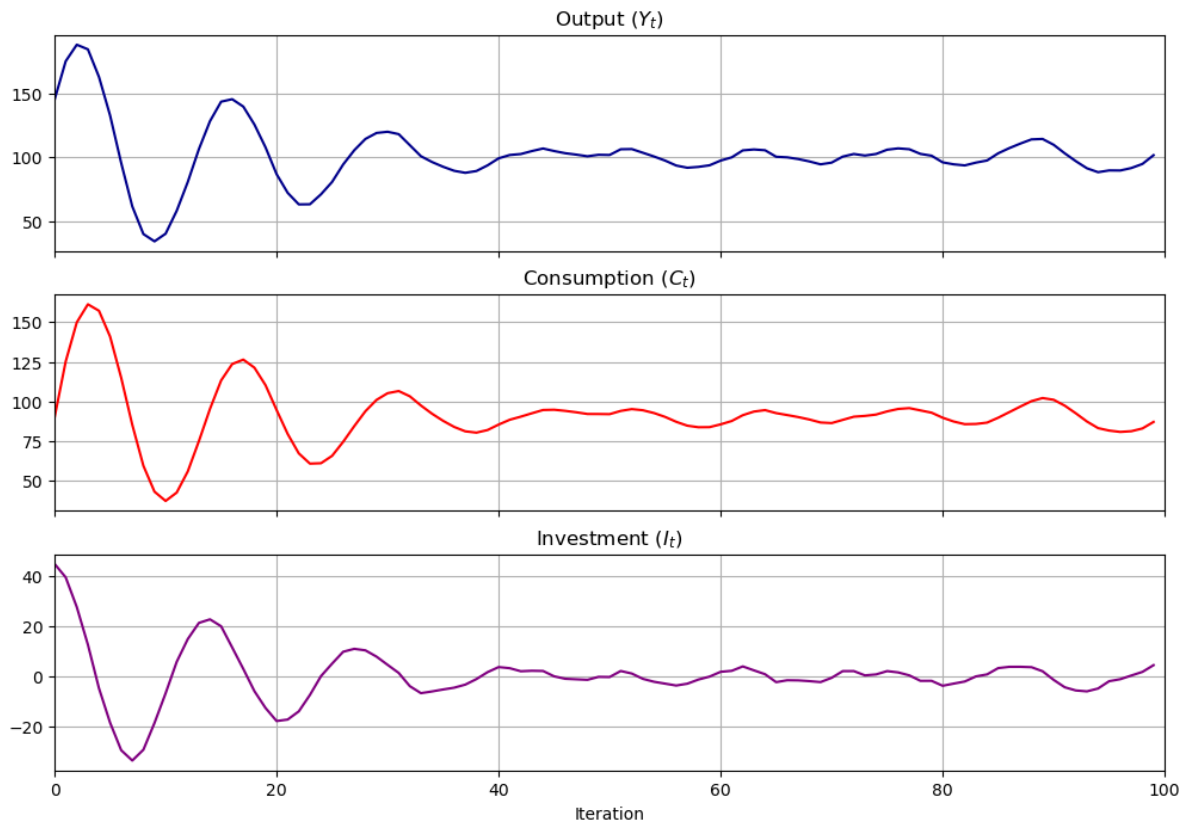
```
                    self.stationary_distributions()
                self.mu_0 = self.mu_x
                self.Sigma_0 = self.Sigma_x
            # Exception where no convergence achieved when
            #calculating stationary distributions
            except ValueError:
                print('Stationary distribution does not exist')

        x, y = self.simulate(ts_length)

        fig, axes = plt.subplots(3, 1, sharex=True, figsize=(12, 8))
        titles = ['Output ($Y_t$)', 'Consumption ($C_t$)', 'Investment ($I_t$)']
        colors = ['darkblue', 'red', 'purple']
        for ax, series, title, color in zip(axes, y, titles, colors):
            ax.plot(series, color=color)
            ax.set(title=title, xlim=(0, n))
            ax.grid()

        axes[-1].set_xlabel('Iteration')

        # Reset distribution parameters to their initial values
        self.mu_0 = temp_mu
        self.Sigma_0 = temp_Sigma

        return fig

    def plot_irf(self, j=5):

        x, y = self.impulse_response(j)

        # Reshape into 3 x j matrix for plotting purposes
        yimf = np.array(y).flatten().reshape(j+1, 3).T

        fig, axes = plt.subplots(3, 1, sharex=True, figsize=(12, 8))
        labels = ['$Y_t$', '$C_t$', '$I_t$']
        colors = ['darkblue', 'red', 'purple']
        for ax, series, label, color in zip(axes, yimf, labels, colors):
            ax.plot(series, color=color)
            ax.set(xlim=(0, j))
            ax.set_ylabel(label, rotation=0, fontsize=14, labelpad=10)
            ax.grid()

        axes[0].set_title('Impulse Response Functions')
        axes[-1].set_xlabel('Iteration')

        return fig

    def multipliers(self, j=5):
        x, y = self.impulse_response(j)
        return np.sum(np.array(y).flatten().reshape(j+1, 3), axis=0)
```

3.7.3 Illustrations

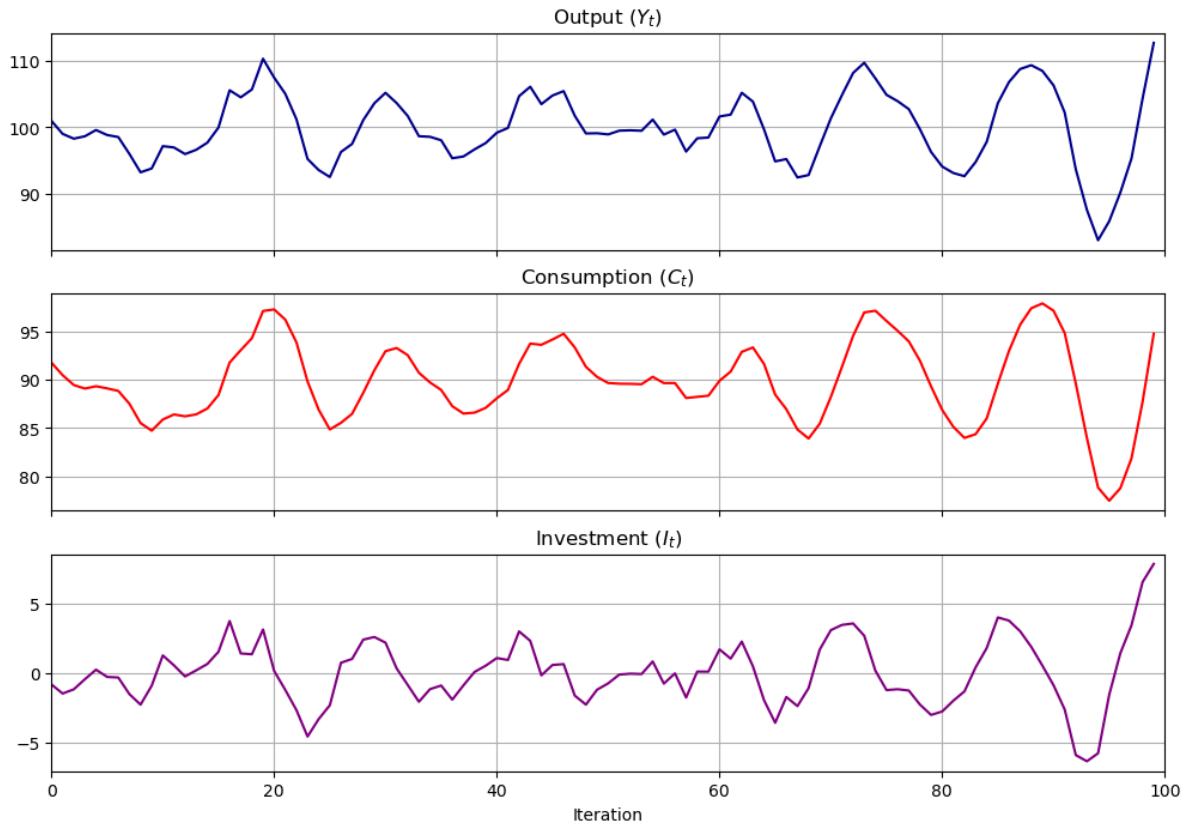
Let's show how we can use the `SamuelsonLSS`

```
samlss = SamuelsonLSS()
```

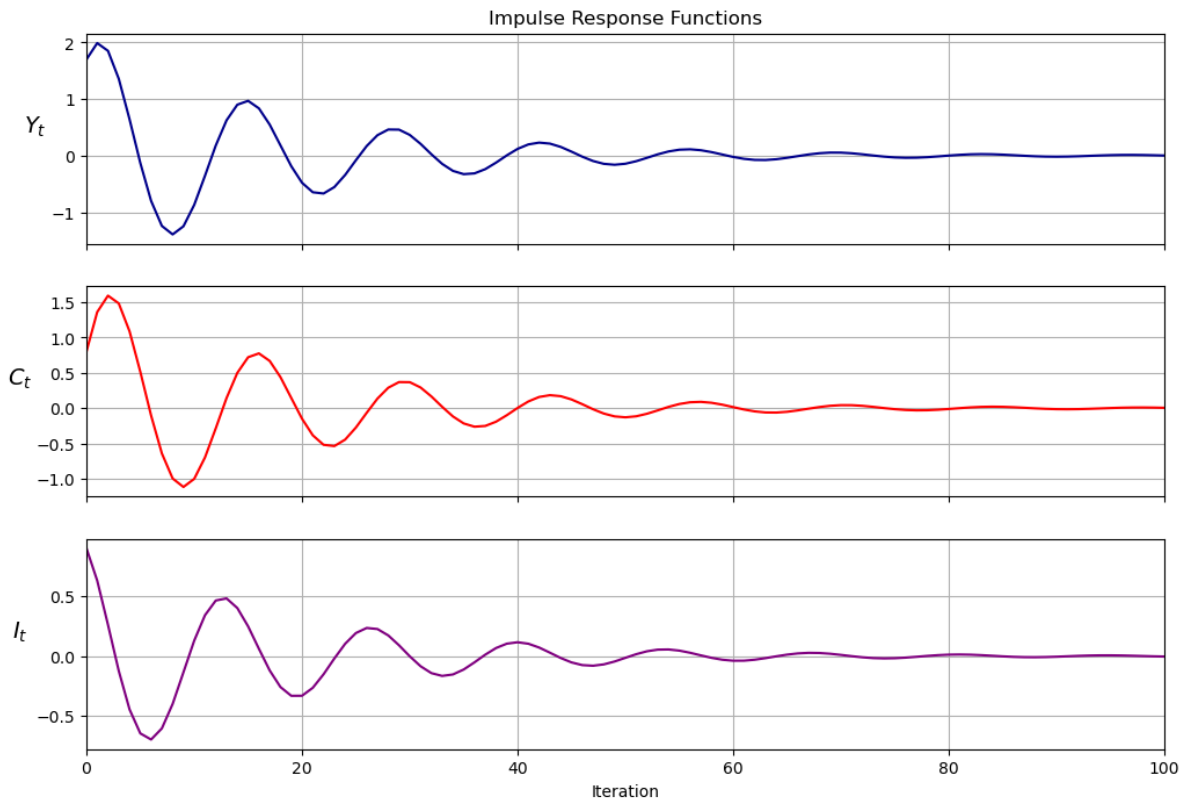
```
samlss.plot_simulation(100, stationary=False)
plt.show()
```




```
samlss.plot_simulation(100, stationary=True)
plt.show()
```




```
samlss.plot_irf(100)
plt.show()
```




```
samlss.multipliers()
```

```
array([7.414389, 6.835896, 0.578493])
```

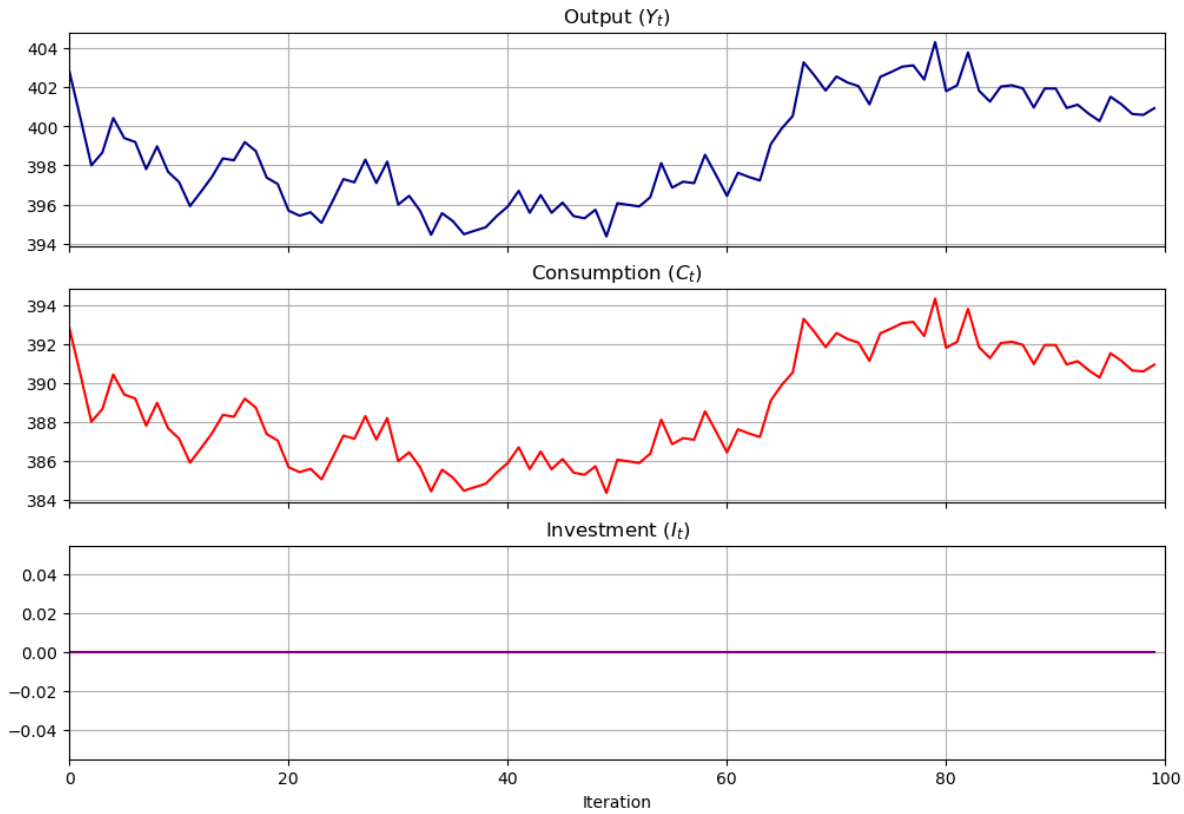
3.8 Pure Multiplier Model

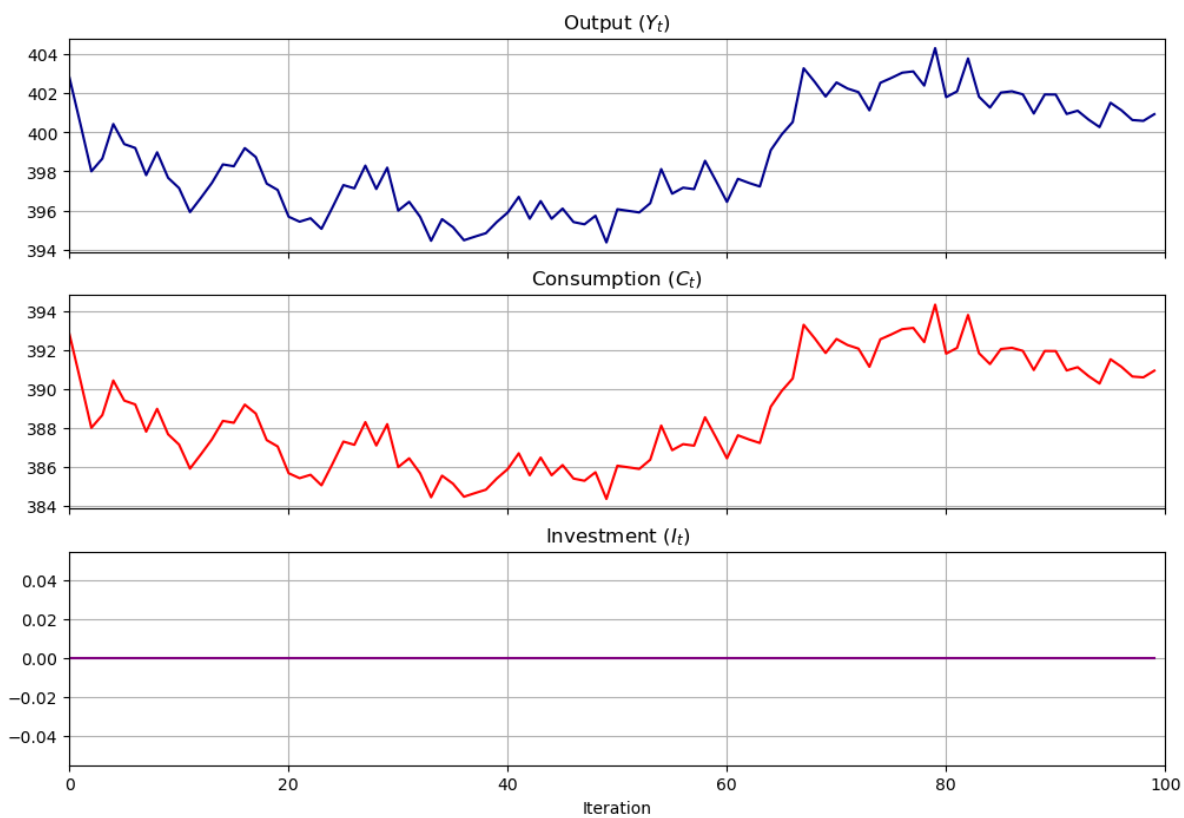
Let's shut down the accelerator by setting $b = 0$ to get a pure multiplier model

- the absence of cycles gives an idea about why Samuelson included the accelerator

```
pure_multiplier = SamuelsonLSS(α=0.95, β=0)
```

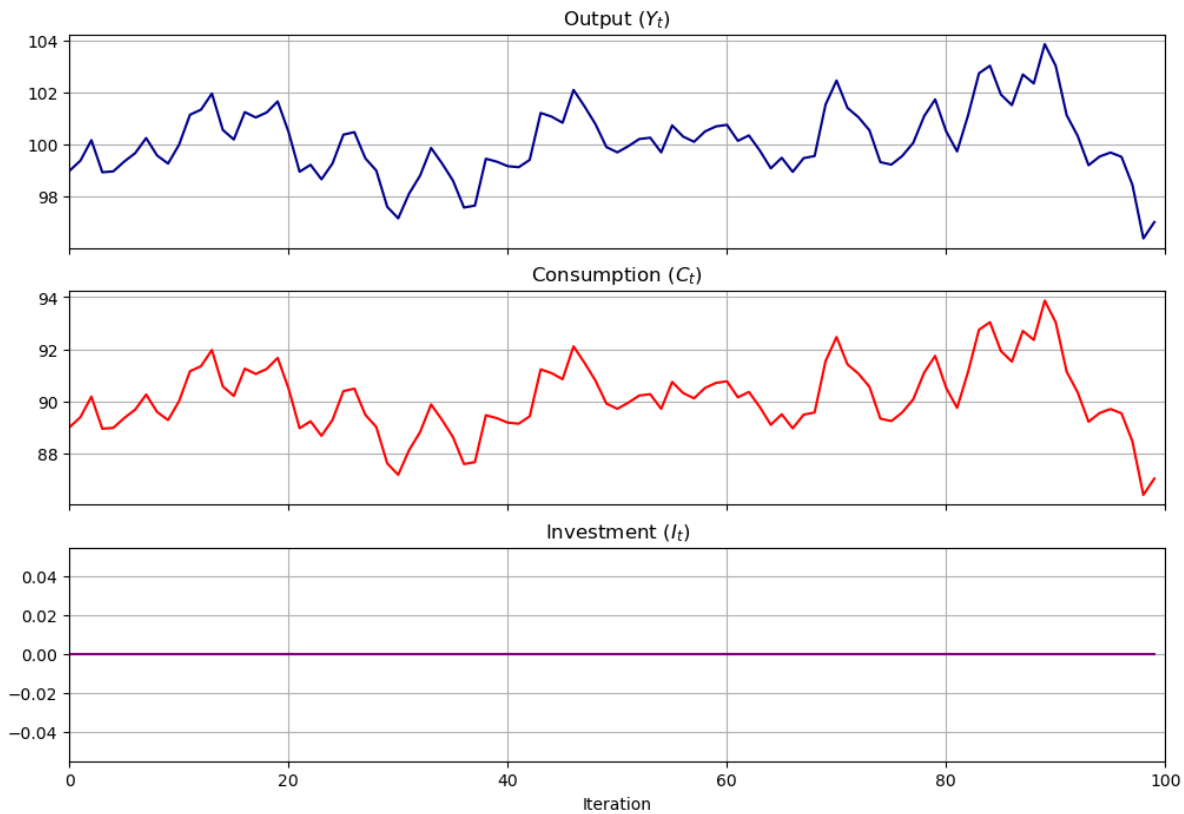
```
pure_multiplier.plot_simulation()
```

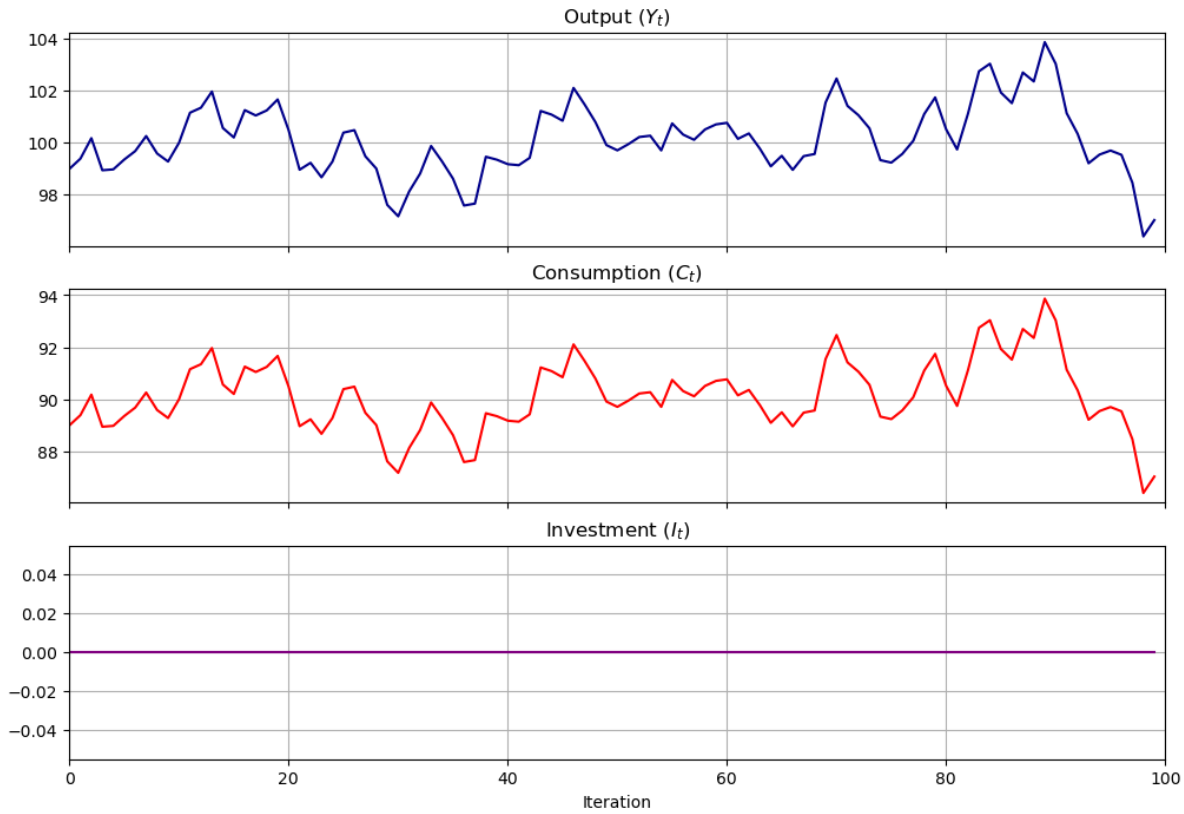

Output (Y_t)
404
402
400
398
396
394
Consumption (C_t)
394
392
390
388
386
384
Investment (I_t)
0.04
0.02
0.00
−0.02
−0.04
0
20
40
60
80
100
Iteration


Output (Y_t)
404
402
400
398
396
394
Consumption (C_t)
394
392
390
388
386
384
Investment (I_t)
0.04
0.02
0.00
−0.02
−0.04
0
20
40
60
80
100
Iteration

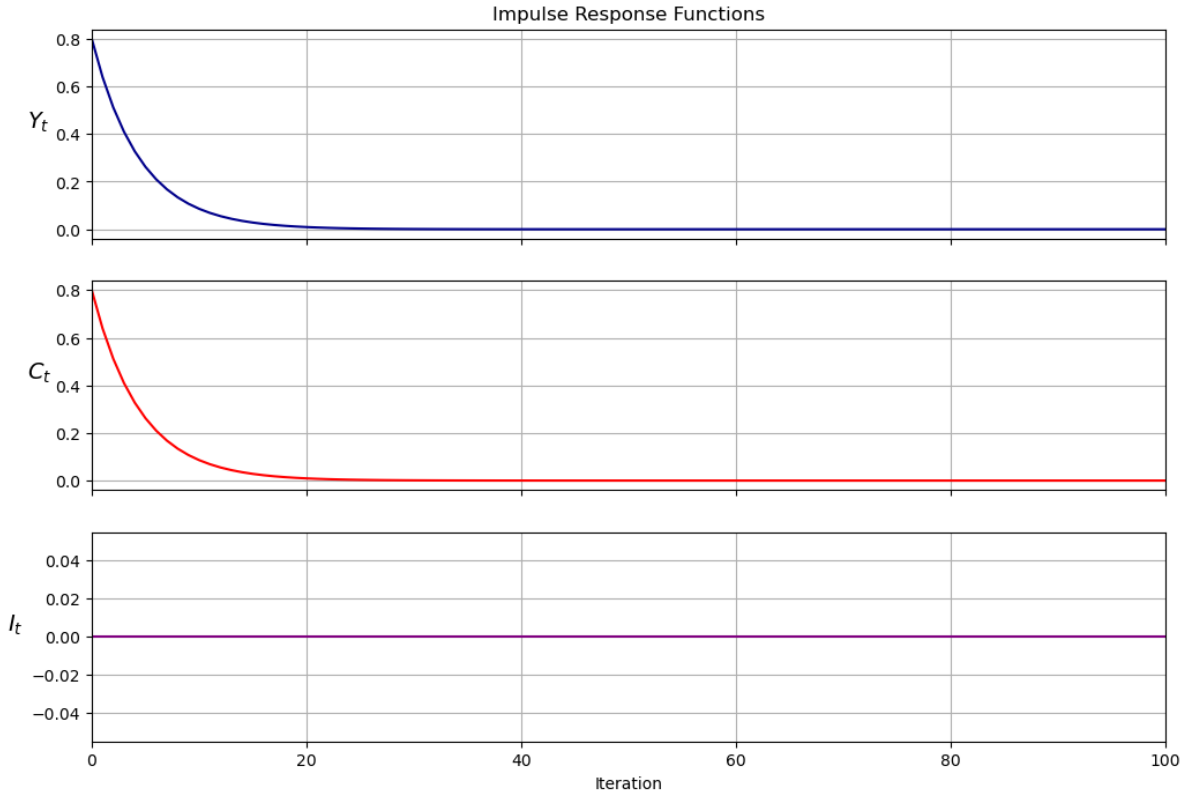
```
pure_multiplier = SamuelsonLSS(α=0.8, β=0)
```

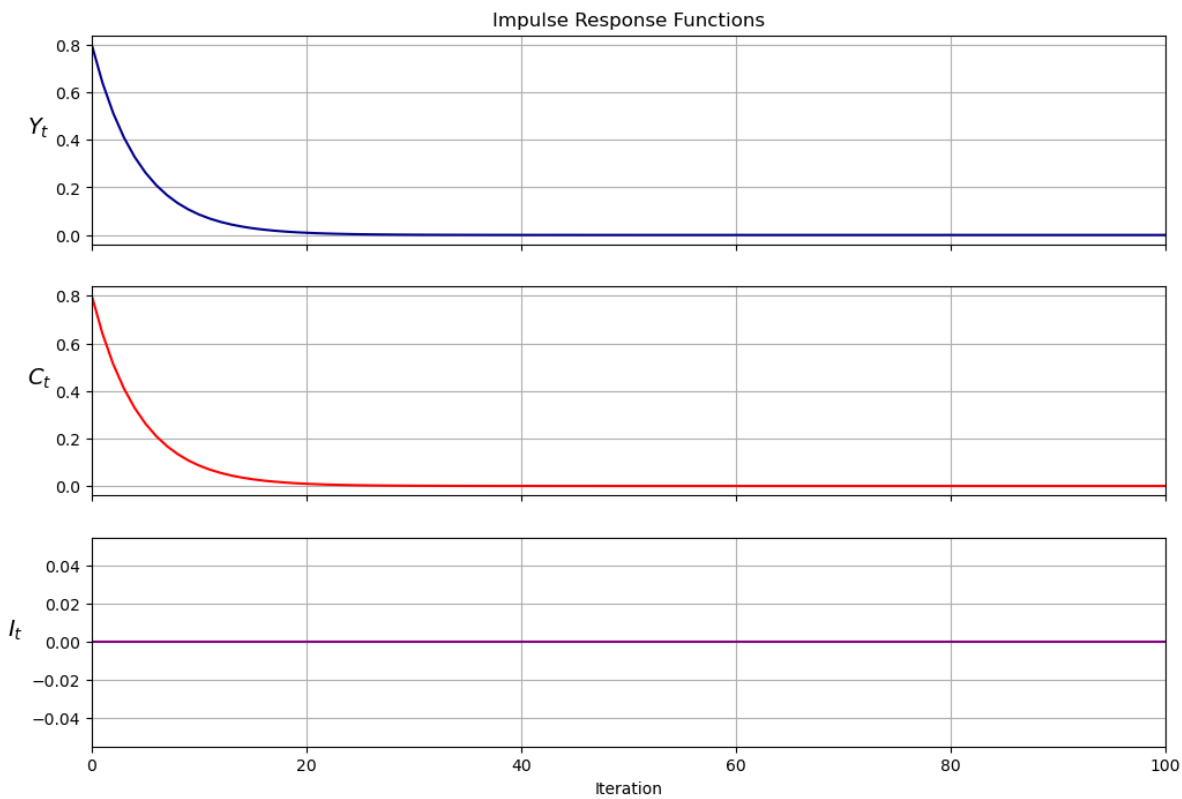
```
pure_multiplier.plot_simulation()
```






```
pure_multiplier.plot_irf(100)
```


Impulse Response Functions
Y_t
C_t
I_t
Iteration


Impulse Response Functions
Y_t
C_t
I_t
Iteration

3.9 Summary

In this lecture, we wrote functions and classes to represent non-stochastic and stochastic versions of the Samuelson (1939) multiplier-accelerator model, described in [Sam39].

We saw that different parameter values led to different output paths, which could either be stationary, explosive, or oscillating.

We also were able to represent the model using the QuantEcon.py LinearStateSpace class.

CHAPTER

FOUR

KESTEN PROCESSES AND FIRM DYNAMICS

Contents

- *Kesten Processes and Firm Dynamics*
  - *Overview*
  - *Kesten Processes*
  - *Heavy Tails*
  - *Application: Firm Dynamics*
  - *Exercises*

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

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

4.1 Overview

Previously we learned about linear scalar-valued stochastic processes (AR(1) models).

Now we generalize these linear models slightly by allowing the multiplicative coefficient to be stochastic.

Such processes are known as Kesten processes after German–American mathematician Harry Kesten (1931–2019)

Although simple to write down, Kesten processes are interesting for at least two reasons:

1. A number of significant economic processes are or can be described as Kesten processes.
2. Kesten processes generate interesting dynamics, including, in some cases, heavy-tailed cross-sectional distributions.

We will discuss these issues as we go along.

Let's start with some imports:

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

The following two lines are only added to avoid a `FutureWarning` caused by compatibility issues between pandas and matplotlib.

```
from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters()
```

Additional technical background related to this lecture can be found in the monograph of [BDM+16].

4.2 Kesten Processes

A **Kesten process** is a stochastic process of the form

$$X_{t+1} = a_{t+1} X_t + \eta_{t+1} \tag{4.1}$$

where $\{a_t\}_{t \geq 1}$ and $\{\eta_t\}_{t \geq 1}$ are IID sequences.

We are interested in the dynamics of $\{X_t\}_{t \geq 0}$ when X_0 is given.

We will focus on the nonnegative scalar case, where X_t takes values in $\mathbb{R}_+$.

In particular, we will assume that

- the initial condition X_0 is nonnegative,
- $\{a_t\}_{t \geq 1}$ is a nonnegative IID stochastic process and
- $\{\eta_t\}_{t \geq 1}$ is another nonnegative IID stochastic process, independent of the first.

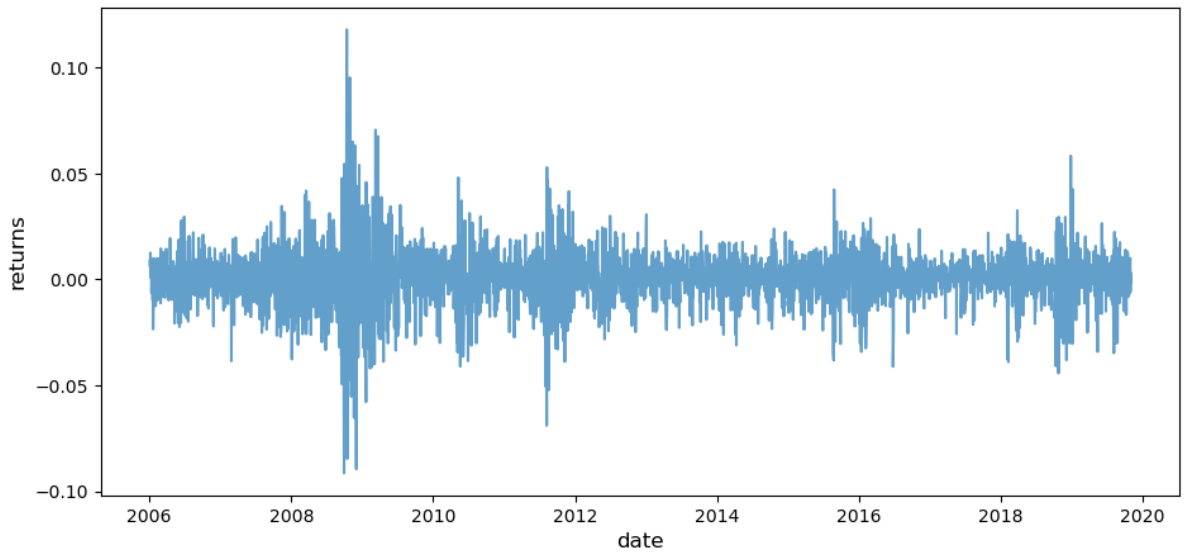
4.2.1 Example: GARCH Volatility

The GARCH model is common in financial applications, where time series such as asset returns exhibit time varying volatility.

For example, consider the following plot of daily returns on the Nasdaq Composite Index for the period 1st January 2006 to 1st November 2019.

```
import yfinance as yf

s = yf.download('^IXIC', '2006-1-1', '2019-11-1')['Adj Close']

r = s.pct_change()

fig, ax = plt.subplots()

ax.plot(r, alpha=0.7)

ax.set_ylabel('returns', fontsize=12)
ax.set_xlabel('date', fontsize=12)

plt.show()
```

```
[*********************100%%**********************]  1 of 1 completed
```



Notice how the series exhibits bursts of volatility (high variance) and then settles down again.

GARCH models can replicate this feature.

The GARCH(1, 1) volatility process takes the form

$$\sigma_{t+1}^2 = \alpha_0 + \sigma_t^2(\alpha_1 \xi_{t+1}^2 + \beta) \tag{4.2}$$

where $\{\xi_t\}$ is IID with $\mathbb{E}\xi_t^2 = 1$ and all parameters are positive.

Returns on a given asset are then modeled as

$$r_t = \sigma_t \zeta_t \tag{4.3}$$

where $\{\zeta_t\}$ is again IID and independent of $\{\xi_t\}$.

The volatility sequence $\{\sigma_t^2\}$, which drives the dynamics of returns, is a Kesten process.

4.2.2 Example: Wealth Dynamics

Suppose that a given household saves a fixed fraction s of its current wealth in every period.

The household earns labor income y_t at the start of time t.

Wealth then evolves according to

$$w_{t+1} = R_{t+1} s w_t + y_{t+1} \tag{4.4}$$

where $\{R_t\}$ is the gross rate of return on assets.

If $\{R_t\}$ and $\{y_t\}$ are both IID, then (4.4) is a Kesten process.

4.2.3 Stationarity

In earlier lectures, such as the one on AR(1) processes, we introduced the notion of a stationary distribution.

In the present context, we can define a stationary distribution as follows:

The distribution F^* on $\mathbb{R}$ is called **stationary** for the Kesten process (4.1) if

$$X_t \sim F^* \quad \Longrightarrow \quad a_{t+1} X_t + \eta_{t+1} \sim F^* \tag{4.5}$$

In other words, if the current state X_t has distribution F^*, then so does the next period state X_{t+1}.

We can write this alternatively as

$$F^*(y) = \int \mathbb{P}\{a_{t+1} x + \eta_{t+1} \leq y\} F^*(dx) \quad \text{for all } y \geq 0. \tag{4.6}$$

The left hand side is the distribution of the next period state when the current state is drawn from F^*.

The equality in (4.6) states that this distribution is unchanged.

4.2.4 Cross-Sectional Interpretation

There is an important cross-sectional interpretation of stationary distributions, discussed previously but worth repeating here.

Suppose, for example, that we are interested in the wealth distribution — that is, the current distribution of wealth across households in a given country.

Suppose further that

- the wealth of each household evolves independently according to (4.4),
- F^* is a stationary distribution for this stochastic process and
- there are many households.

Then F^* is a steady state for the cross-sectional wealth distribution in this country.

In other words, if F^* is the current wealth distribution then it will remain so in subsequent periods, *ceteris paribus*.

To see this, suppose that F^* is the current wealth distribution.

What is the fraction of households with wealth less than y next period?

To obtain this, we sum the probability that wealth is less than y tomorrow, given that current wealth is w, weighted by the fraction of households with wealth w.

Noting that the fraction of households with wealth in interval dw is $F^*(dw)$, we get

$$\int \mathbb{P}\{R_{t+1} s w + y_{t+1} \leq y\} F^*(dw)$$

By the definition of stationarity and the assumption that F^* is stationary for the wealth process, this is just $F^*(y)$.

Hence the fraction of households with wealth in $[0, y]$ is the same next period as it is this period.

Since y was chosen arbitrarily, the distribution is unchanged.

4.2.5 Conditions for Stationarity

The Kesten process $X_{t+1} = a_{t+1} X_t + \eta_{t+1}$ does not always have a stationary distribution.

For example, if $a_t \equiv \eta_t \equiv 1$ for all t, then $X_t = X_0 + t$, which diverges to infinity.

To prevent this kind of divergence, we require that $\{a_t\}$ is strictly less than 1 most of the time.

In particular, if

$$\mathbb{E} \ln a_t < 0 \quad \text{and} \quad \mathbb{E}\eta_t < \infty \tag{4.7}$$

then a unique stationary distribution exists on $\mathbb{R}_+$.

- See, for example, theorem 2.1.3 of [BDM+16], which provides slightly weaker conditions.

As one application of this result, we see that the wealth process (4.4) will have a unique stationary distribution whenever labor income has finite mean and $\mathbb{E} \ln R_t + \ln s < 0$.

4.3 Heavy Tails

Under certain conditions, the stationary distribution of a Kesten process has a Pareto tail.

(See our earlier lecture on heavy-tailed distributions for background.)

This fact is significant for economics because of the prevalence of Pareto-tailed distributions.

4.3.1 The Kesten–Goldie Theorem

To state the conditions under which the stationary distribution of a Kesten process has a Pareto tail, we first recall that a random variable is called **nonarithmetic** if its distribution is not concentrated on $\{\dots, -2t, -t, 0, t, 2t, \dots\}$ for any $t \geq 0$.

For example, any random variable with a density is nonarithmetic.

The famous Kesten–Goldie Theorem (see, e.g., [BDM+16], theorem 2.4.4) states that if

1. the stationarity conditions in (4.7) hold,
2. the random variable a_t is positive with probability one and nonarithmetic,
3. $\mathbb{P}\{a_t x + \eta_t = x\} < 1$ for all $x \in \mathbb{R}_+$ and
4. there exists a positive constant α such that

$$\mathbb{E} a_t^\alpha = 1, \quad \mathbb{E}\eta_t^\alpha < \infty, \quad \text{and} \quad \mathbb{E}[a_t^{\alpha+1}] < \infty$$

then the stationary distribution of the Kesten process has a Pareto tail with tail index α.

More precisely, if F^* is the unique stationary distribution and $X^* \sim F^*$, then

$$\lim_{x \to \infty} x^\alpha \mathbb{P}\{X^* > x\} = c$$

for some positive constant c.

4.3.2 Intuition

Later we will illustrate the Kesten–Goldie Theorem using rank-size plots.

Prior to doing so, we can give the following intuition for the conditions.

Two important conditions are that $\mathbb{E} \ln a_t < 0$, so the model is stationary, and $\mathbb{E} a_t^\alpha = 1$ for some $\alpha > 0$.

The first condition implies that the distribution of a_t has a large amount of probability mass below 1.

The second condition implies that the distribution of a_t has at least some probability mass at or above 1.

The first condition gives us existence of the stationary condition.

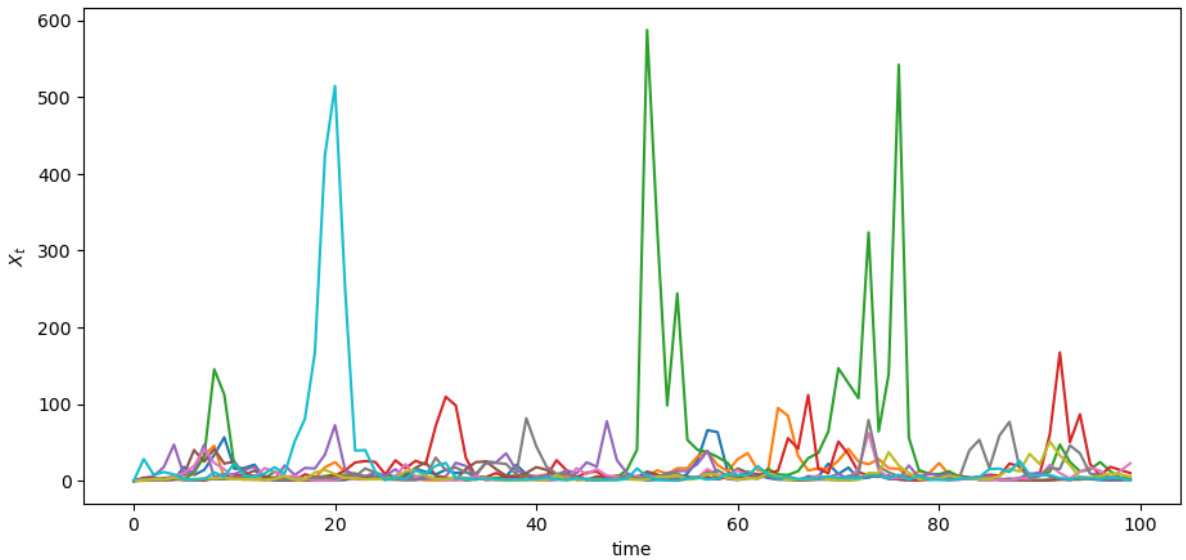
The second condition means that the current state can be expanded by a_t.

If this occurs for several concurrent periods, the effects compound each other, since a_t is multiplicative.

This leads to spikes in the time series, which fill out the extreme right hand tail of the distribution.

The spikes in the time series are visible in the following simulation, which generates of 10 paths when a_t and b_t are lognormal.

```
μ = -0.5
σ = 1.0

def kesten_ts(ts_length=100):
    x = np.zeros(ts_length)
    for t in range(ts_length-1):
        a = np.exp(μ + σ * np.random.randn())
        b = np.exp(np.random.randn())
        x[t+1] = a * x[t] + b
    return x

fig, ax = plt.subplots()

num_paths = 10
np.random.seed(12)

for i in range(num_paths):
    ax.plot(kesten_ts())

ax.set(xlabel='time', ylabel='$X_t$')
plt.show()
```



4.4 Application: Firm Dynamics

As noted in our lecture on heavy tails, for common measures of firm size such as revenue or employment, the US firm size distribution exhibits a Pareto tail (see, e.g., [Axt01], [Gab16]).

Let us try to explain this rather striking fact using the Kesten–Goldie Theorem.

4.4.1 Gibrat's Law

It was postulated many years ago by Robert Gibrat [Gib31] that firm size evolves according to a simple rule whereby size next period is proportional to current size.

This is now known as Gibrat's law of proportional growth.

We can express this idea by stating that a suitably defined measure s_t of firm size obeys

$$\frac{s_{t+1}}{s_t} = a_{t+1} \tag{4.8}$$

for some positive IID sequence $\{a_t\}$.

One implication of Gibrat's law is that the growth rate of individual firms does not depend on their size.

However, over the last few decades, research contradicting Gibrat's law has accumulated in the literature.

For example, it is commonly found that, on average,

1. small firms grow faster than large firms (see, e.g., [Eva87] and [Hal87]) and
2. the growth rate of small firms is more volatile than that of large firms [DRS89].

On the other hand, Gibrat's law is generally found to be a reasonable approximation for large firms [Eva87].

We can accommodate these empirical findings by modifying (4.8) to

$$s_{t+1} = a_{t+1} s_t + b_{t+1} \tag{4.9}$$

where $\{a_t\}$ and $\{b_t\}$ are both IID and independent of each other.

In the exercises you are asked to show that (4.9) is more consistent with the empirical findings presented above than Gibrat's law in (4.8).

4.4.2 Heavy Tails

So what has this to do with Pareto tails?

The answer is that (4.9) is a Kesten process.

If the conditions of the Kesten–Goldie Theorem are satisfied, then the firm size distribution is predicted to have heavy tails — which is exactly what we see in the data.

In the exercises below we explore this idea further, generalizing the firm size dynamics and examining the corresponding rank-size plots.

We also try to illustrate why the Pareto tail finding is significant for quantitative analysis.

4.5 Exercises

Exercise 4.5.1

Simulate and plot 15 years of daily returns (consider each year as having 250 working days) using the GARCH(1, 1) process in (4.2)–(4.3).

Take ξ_t and ζ_t to be independent and standard normal.

Set $\alpha_0 = 0.00001, \alpha_1 = 0.1, \beta = 0.9$ and $\sigma_0 = 0$.

Compare visually with the Nasdaq Composite Index returns *shown above*.

While the time path differs, you should see bursts of high volatility.

Solution to Exercise 4.5.1

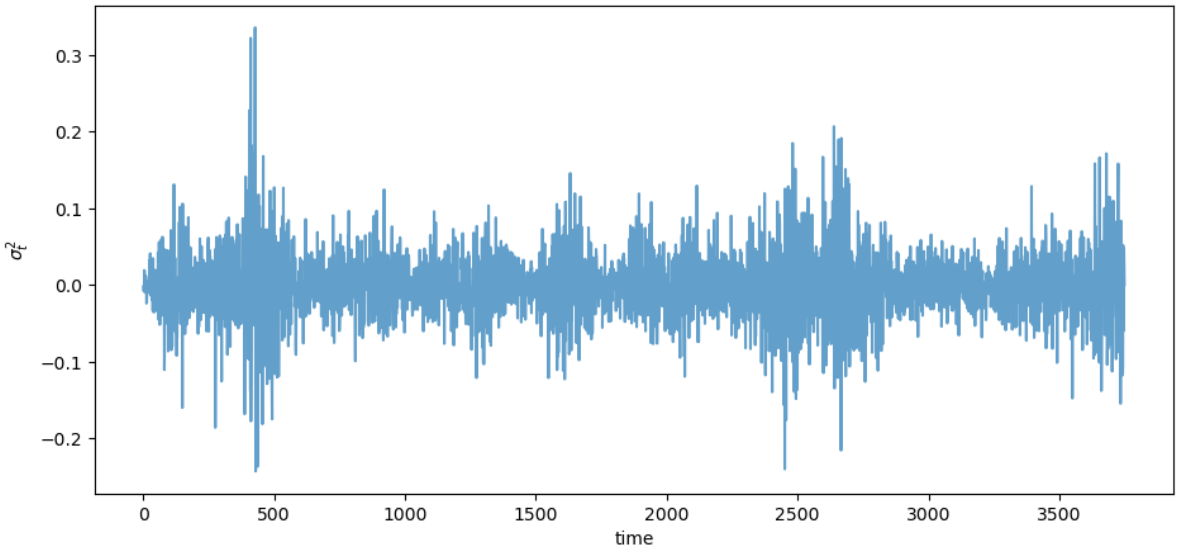
Here is one solution:

```
α_0 = 1e-5
α_1 = 0.1
β = 0.9

years = 15
days = years * 250

def garch_ts(ts_length=days):
    σ2 = 0
    r = np.zeros(ts_length)
    for t in range(ts_length-1):
        ξ = np.random.randn()
        σ2 = α_0 + σ2 * (α_1 * ξ**2 + β)
        r[t] = np.sqrt(σ2) * np.random.randn()
    return r

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

(continues on next page)

(continued from previous page)

```
np.random.seed(12)

ax.plot(garch_ts(), alpha=0.7)

ax.set(xlabel='time', ylabel='$\\sigma_t^2$')
plt.show()
```



Exercise 4.5.2

In our discussion of firm dynamics, it was claimed that (4.9) is more consistent with the empirical literature than Gibrat's law in (4.8).

(The empirical literature was reviewed immediately above (4.9).)

In what sense is this true (or false)?

Solution to Exercise 4.5.2

The empirical findings are that

1. small firms grow faster than large firms and
2. the growth rate of small firms is more volatile than that of large firms.

Also, Gibrat's law is generally found to be a reasonable approximation for large firms than for small firms

The claim is that the dynamics in (4.9) are more consistent with points 1-2 than Gibrat's law.

To see why, we rewrite (4.9) in terms of growth dynamics:

$$\frac{s_{t+1}}{s_t} = a_{t+1} + \frac{b_{t+1}}{s_t} \tag{4.10}$$

Taking $s_t = s$ as given, the mean and variance of firm growth are

$$\mathbb{E}a + \frac{\mathbb{E}b}{s} \quad \text{and} \quad \mathbb{V}a + \frac{\mathbb{V}b}{s^2}$$

Both of these decline with firm size s, consistent with the data.

Moreover, the law of motion (4.10) clearly approaches Gibrat's law (4.8) as s_t gets large.

Exercise 4.5.3

Consider an arbitrary Kesten process as given in (4.1).

Suppose that $\{a_t\}$ is lognormal with parameters (μ, σ).

In other words, each a_t has the same distribution as $\exp(\mu + \sigma Z)$ when Z is standard normal.

Suppose further that $\mathbb{E}\eta_t^r < \infty$ for every $r > 0$, as would be the case if, say, η_t is also lognormal.

Show that the conditions of the Kesten–Goldie theorem are satisfied if and only if $\mu < 0$.

Obtain the value of α that makes the Kesten–Goldie conditions hold.

Solution to Exercise 4.5.3

Since a_t has a density it is nonarithmetic.

Since a_t has the same density as $a = \exp(\mu + \sigma Z)$ when Z is standard normal, we have

$$\mathbb{E} \ln a_t = \mathbb{E}(\mu + \sigma Z) = \mu,$$

and since η_t has finite moments of all orders, the stationarity condition holds if and only if $\mu < 0$.

Given the properties of the lognormal distribution (which has finite moments of all orders), the only other condition in doubt is existence of a positive constant α such that $\mathbb{E}a_t^\alpha = 1$.

This is equivalent to the statement

$$\exp\left(\alpha\mu + \frac{\alpha^2\sigma^2}{2}\right) = 1.$$

Solving for α gives $\alpha = -2\mu/\sigma^2$.

Exercise 4.5.4

One unrealistic aspect of the firm dynamics specified in (4.9) is that it ignores entry and exit.

In any given period and in any given market, we observe significant numbers of firms entering and exiting the market.

Empirical discussion of this can be found in a famous paper by Hugo Hopenhayn [Hop92].

In the same paper, Hopenhayn builds a model of entry and exit that incorporates profit maximization by firms and market clearing quantities, wages and prices.

In his model, a stationary equilibrium occurs when the number of entrants equals the number of exiting firms.

In this setting, firm dynamics can be expressed as

$$s_{t+1} = e_{t+1}\mathbb{1}\{s_t < \bar{s}\} + (a_{t+1}s_t + b_{t+1})\mathbb{1}\{s_t \geq \bar{s}\} \tag{4.11}$$

Here

- the state variable s_t represents productivity (which is a proxy for output and hence firm size),
- the IID sequence $\{e_t\}$ is thought of as a productivity draw for a new entrant and

- the variable $\bar{s}$ is a threshold value that we take as given, although it is determined endogenously in Hopenhayn's model.

The idea behind (4.11) is that firms stay in the market as long as their productivity s_t remains at or above $\bar{s}$.

- In this case, their productivity updates according to (4.9).

Firms choose to exit when their productivity s_t falls below $\bar{s}$.

- In this case, they are replaced by a new firm with productivity e_{t+1}.

What can we say about dynamics?

Although (4.11) is not a Kesten process, it does update in the same way as a Kesten process when s_t is large.

So perhaps its stationary distribution still has Pareto tails?

Your task is to investigate this question via simulation and rank-size plots.

The approach will be to

1. generate M draws of s_T when M and T are large and
2. plot the largest 1,000 of the resulting draws in a rank-size plot.

(The distribution of s_T will be close to the stationary distribution when T is large.)

In the simulation, assume that

- each of a_t, b_t and e_t is lognormal,
- the parameters are

```
μ_a = -0.5          # location parameter for a
σ_a = 0.1           # scale parameter for a
μ_b = 0.0           # location parameter for b
σ_b = 0.5           # scale parameter for b
μ_e = 0.0           # location parameter for e
σ_e = 0.5           # scale parameter for e
s_bar = 1.0         # threshold
T = 500             # sampling date
M = 1_000_000       # number of firms
s_init = 1.0        # initial condition for each firm
```

Solution to Exercise 4.5.4

Here's one solution. First we generate the observations:

```
from numba import njit, prange
from numpy.random import randn


@njit(parallel=True)
def generate_draws(μ_a=-0.5,
                   σ_a=0.1,
                   μ_b=0.0,
                   σ_b=0.5,
                   μ_e=0.0,
                   σ_e=0.5,
                   s_bar=1.0,
                   T=500,
```

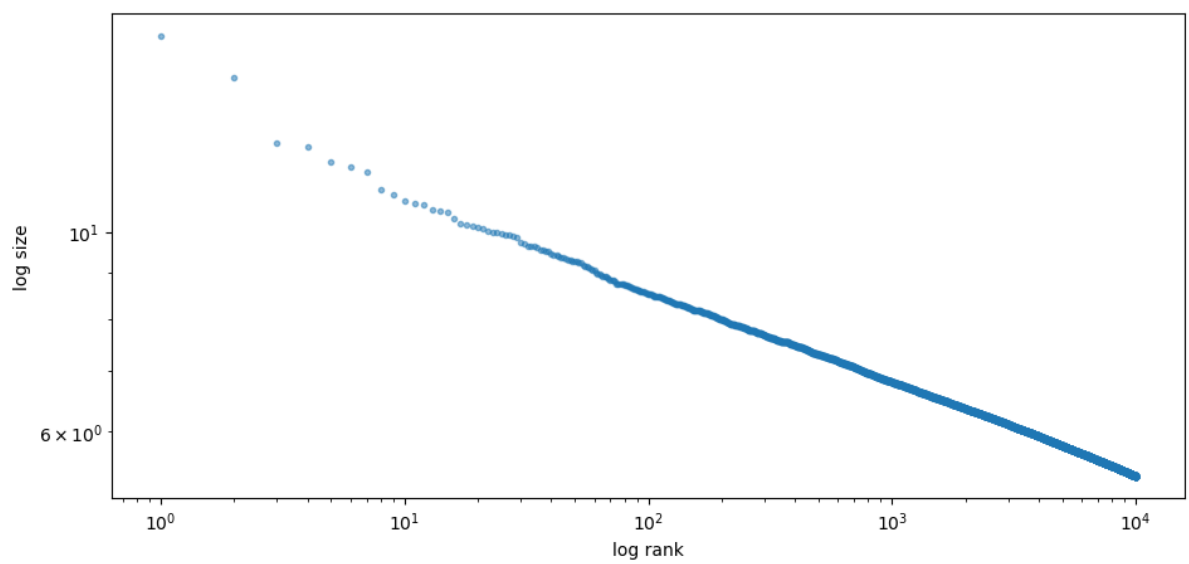
(continues on next page)

(continued from previous page)

```
                    M=1_000_000,
                    s_init=1.0):

    draws = np.empty(M)
    for m in prange(M):
        s = s_init
        for t in range(T):
            if s < s_bar:
                new_s = np.exp(μ_e + σ_e *  randn())
            else:
                a = np.exp(μ_a + σ_a * randn())
                b = np.exp(μ_b + σ_b * randn())
                new_s = a * s + b
            s = new_s
        draws[m] = s

    return draws

data = generate_draws()
```

Now we produce the rank-size plot:

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

rank_data, size_data = qe.rank_size(data, c=0.01)
ax.loglog(rank_data, size_data, 'o', markersize=3.0, alpha=0.5)
ax.set_xlabel("log rank")
ax.set_ylabel("log size")

plt.show()
```



The plot produces a straight line, consistent with a Pareto tail.

CHAPTER

FIVE

WEALTH DISTRIBUTION DYNAMICS

Contents

- *Wealth Distribution Dynamics*
  - *Overview*
  - *Lorenz Curves and the Gini Coefficient*
  - *A Model of Wealth Dynamics*
  - *Implementation*
  - *Applications*
  - *Exercises*

See also:

A version of this lecture using a `GPU` is available here

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

```
!pip install quantecon
```

5.1 Overview

This notebook gives an introduction to wealth distribution dynamics, with a focus on

- modeling and computing the wealth distribution via simulation,
- measures of inequality such as the Lorenz curve and Gini coefficient, and
- how inequality is affected by the properties of wage income and returns on assets.

One interesting property of the wealth distribution we discuss is Pareto tails.

The wealth distribution in many countries exhibits a Pareto tail

- See this lecture for a definition.
- For a review of the empirical evidence, see, for example, [BB18].

This is consistent with high concentration of wealth amongst the richest households.

It also gives us a way to quantify such concentration, in terms of the tail index.

One question of interest is whether or not we can replicate Pareto tails from a relatively simple model.

5.1.1 A Note on Assumptions

The evolution of wealth for any given household depends on their savings behavior.

Modeling such behavior will form an important part of this lecture series.

However, in this particular lecture, we will be content with rather ad hoc (but plausible) savings rules.

We do this to more easily explore the implications of different specifications of income dynamics and investment returns.

At the same time, all of the techniques discussed here can be plugged into models that use optimization to obtain savings rules.

We will use the following imports.

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
import quantecon as qe
from numba import njit, float64, prange
from numba.experimental import jitclass
```

5.2 Lorenz Curves and the Gini Coefficient

Before we investigate wealth dynamics, we briefly review some measures of inequality.

5.2.1 Lorenz Curves

One popular graphical measure of inequality is the Lorenz curve.

The package QuantEcon.py, already imported above, contains a function to compute Lorenz curves.

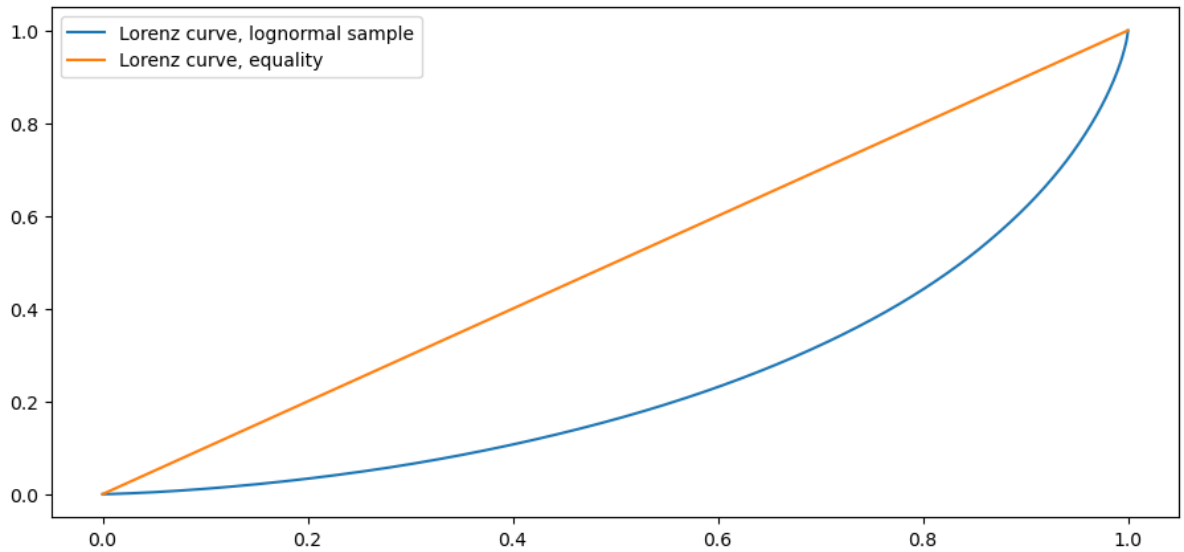
To illustrate, suppose that

```
n = 10_000                      # size of sample
w = np.exp(np.random.randn(n))  # lognormal draws
```

is data representing the wealth of 10,000 households.

We can compute and plot the Lorenz curve as follows:

```
f_vals, l_vals = qe.lorenz_curve(w)

fig, ax = plt.subplots()
ax.plot(f_vals, l_vals, label='Lorenz curve, lognormal sample')
ax.plot(f_vals, f_vals, label='Lorenz curve, equality')
ax.legend()
plt.show()
```



This curve can be understood as follows: if point (x, y) lies on the curve, it means that, collectively, the bottom $(100x)\%$ of the population holds $(100y)\%$ of the wealth.

The "equality" line is the 45 degree line (which might not be exactly 45 degrees in the figure, depending on the aspect ratio).

A sample that produces this line exhibits perfect equality.

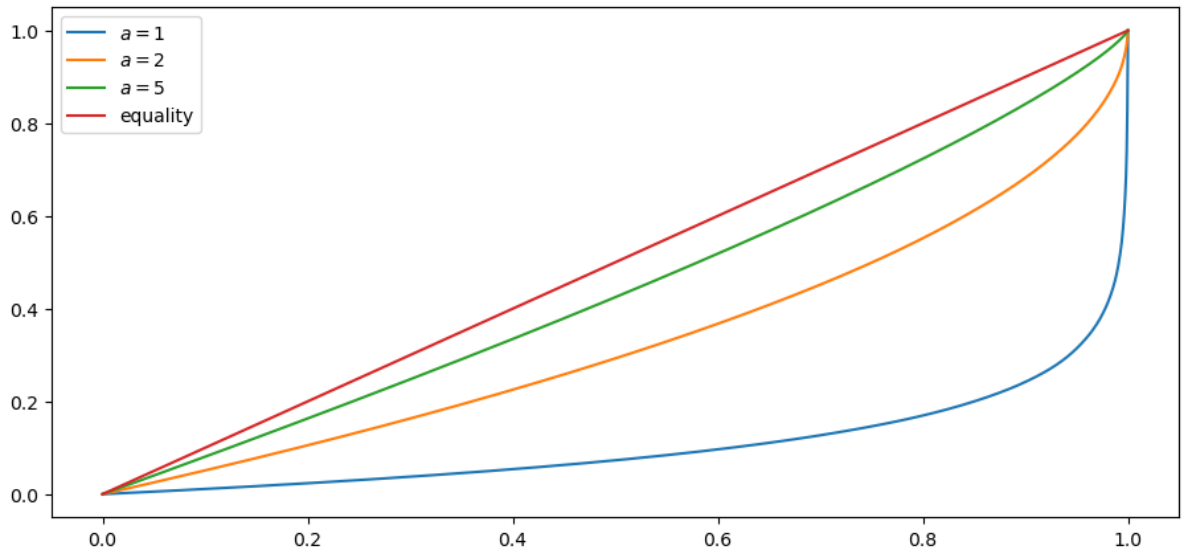
The other line in the figure is the Lorenz curve for the lognormal sample, which deviates significantly from perfect equality.

For example, the bottom 80% of the population holds around 40% of total wealth.

Here is another example, which shows how the Lorenz curve shifts as the underlying distribution changes.

We generate 10,000 observations using the Pareto distribution with a range of parameters, and then compute the Lorenz curve corresponding to each set of observations.

```
a_vals = (1, 2, 5)              # Pareto tail index
n = 10_000                      # size of each sample
fig, ax = plt.subplots()
for a in a_vals:
    u = np.random.uniform(size=n)
    y = u**(-1/a)               # distributed as Pareto with tail index a
    f_vals, l_vals = qe.lorenz_curve(y)
    ax.plot(f_vals, l_vals, label=f'$a = {a}$')
ax.plot(f_vals, f_vals, label='equality')
ax.legend()
plt.show()
```



You can see that, as the tail parameter of the Pareto distribution increases, inequality decreases.

This is to be expected, because a higher tail index implies less weight in the tail of the Pareto distribution.

5.2.2 The Gini Coefficient

The definition and interpretation of the Gini coefficient can be found on the corresponding Wikipedia page.

A value of 0 indicates perfect equality (corresponding the case where the Lorenz curve matches the 45 degree line) and a value of 1 indicates complete inequality (all wealth held by the richest household).

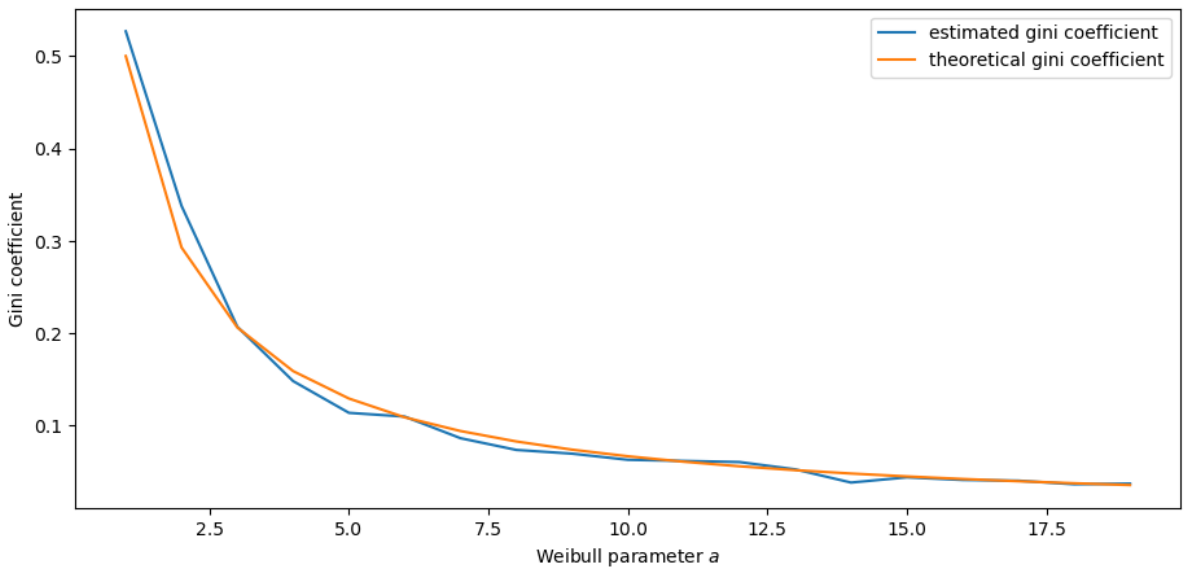
The QuantEcon.py library contains a function to calculate the Gini coefficient.

We can test it on the Weibull distribution with parameter a, where the Gini coefficient is known to be

$$G = 1 - 2^{-1/a}$$

Let's see if the Gini coefficient computed from a simulated sample matches this at each fixed value of a.

```
a_vals = range(1, 20)
ginis = []
ginis_theoretical = []
n = 100

fig, ax = plt.subplots()
for a in a_vals:
    y = np.random.weibull(a, size=n)
    ginis.append(qe.gini_coefficient(y))
    ginis_theoretical.append(1 - 2**(-1/a))
ax.plot(a_vals, ginis, label='estimated gini coefficient')
ax.plot(a_vals, ginis_theoretical, label='theoretical gini coefficient')
ax.legend()
ax.set_xlabel("Weibull parameter $a$")
ax.set_ylabel("Gini coefficient")
plt.show()
```



The simulation shows that the fit is good.

5.3 A Model of Wealth Dynamics

Having discussed inequality measures, let us now turn to wealth dynamics.

The model we will study is

$$w_{t+1} = (1 + r_{t+1})s(w_t) + y_{t+1} \tag{5.1}$$

where

- w_t is wealth at time t for a given household,
- r_t is the rate of return of financial assets,
- y_t is current non-financial (e.g., labor) income and
- $s(w_t)$ is current wealth net of consumption

Letting $\{z_t\}$ be a correlated state process of the form

$$z_{t+1} = az_t + b + \sigma_z \epsilon_{t+1}$$

we'll assume that

$$R_t := 1 + r_t = c_r \exp(z_t) + \exp(\mu_r + \sigma_r \xi_t)$$

and

$$y_t = c_y \exp(z_t) + \exp(\mu_y + \sigma_y \zeta_t)$$

Here $\{(\epsilon_t, \xi_t, \zeta_t)\}$ is IID and standard normal in $\mathbb{R}^3$.

The value of c_r should be close to zero, since rates of return on assets do not exhibit large trends.

When we simulate a population of households, we will assume all shocks are idiosyncratic (i.e., specific to individual households and independent across them).

Regarding the savings function s, our default model will be

$$s(w) = s_0 w \cdot \mathbb{1}\{w \geq \hat{w}\} \tag{5.2}$$

where s_0 is a positive constant.

Thus, for $w < \hat{w}$, the household saves nothing. For $w \geq \bar{w}$, the household saves a fraction s_0 of their wealth.

We are using something akin to a fixed savings rate model, while acknowledging that low wealth households tend to save very little.

5.4 Implementation

Here's some type information to help Numba.

```
wealth_dynamics_data = [
    ('w_hat',  float64),    # savings parameter
    ('s_0',    float64),    # savings parameter
    ('c_y',    float64),    # labor income parameter
    ('μ_y',    float64),    # labor income paraemter
    ('σ_y',    float64),    # labor income parameter
    ('c_r',    float64),    # rate of return parameter
    ('μ_r',    float64),    # rate of return parameter
    ('σ_r',    float64),    # rate of return parameter
    ('a',      float64),    # aggregate shock parameter
    ('b',      float64),    # aggregate shock parameter
    ('σ_z',    float64),    # aggregate shock parameter
    ('z_mean', float64),    # mean of z process
    ('z_var', float64),     # variance of z process
    ('y_mean', float64),    # mean of y process
    ('R_mean', float64)     # mean of R process
]
```

Here's a class that stores instance data and implements methods that update the aggregate state and household wealth.

```
@jitclass(wealth_dynamics_data)
class WealthDynamics:

    def __init__(self,
                 w_hat=1.0,
                 s_0=0.75,
                 c_y=1.0,
                 μ_y=1.0,
                 σ_y=0.2,
                 c_r=0.05,
                 μ_r=0.1,
                 σ_r=0.5,
                 a=0.5,
                 b=0.0,
                 σ_z=0.1):

        self.w_hat, self.s_0 = w_hat, s_0
        self.c_y, self.μ_y, self.σ_y = c_y, μ_y, σ_y
        self.c_r, self.μ_r, self.σ_r = c_r, μ_r, σ_r
        self.a, self.b, self.σ_z = a, b, σ_z
```

(continues on next page)

(continued from previous page)

```
        # Record stationary moments
        self.z_mean = b / (1 - a)
        self.z_var = σ_z**2 / (1 - a**2)
        exp_z_mean = np.exp(self.z_mean + self.z_var / 2)
        self.R_mean = c_r * exp_z_mean + np.exp(μ_r + σ_r**2 / 2)
        self.y_mean = c_y * exp_z_mean + np.exp(μ_y + σ_y**2 / 2)

        # Test a stability condition that ensures wealth does not diverge
        # to infinity.
        α = self.R_mean * self.s_0
        if α >= 1:
            raise ValueError("Stability condition failed.")

    def parameters(self):
        """
        Collect and return parameters.
        """
        parameters = (self.w_hat, self.s_0,
                      self.c_y, self.μ_y, self.σ_y,
                      self.c_r, self.μ_r, self.σ_r,
                      self.a, self.b, self.σ_z)
        return parameters

    def update_states(self, w, z):
        """
        Update one period, given current wealth w and persistent
        state z.
        """

        # Simplify names
        params = self.parameters()
        w_hat, s_0, c_y, μ_y, σ_y, c_r, μ_r, σ_r, a, b, σ_z = params
        zp = a * z + b + σ_z * np.random.randn()

        # Update wealth
        y = c_y * np.exp(zp) + np.exp(μ_y + σ_y * np.random.randn())
        wp = y
        if w >= w_hat:
            R = c_r * np.exp(zp) + np.exp(μ_r + σ_r * np.random.randn())
            wp += R * s_0 * w
        return wp, zp
```

Here's function to simulate the time series of wealth for in individual households.

```
@njit
def wealth_time_series(wdy, w_0, n):
    """
    Generate a single time series of length n for wealth given
    initial value w_0.

    The initial persistent state z_0 for each household is drawn from
    the stationary distribution of the AR(1) process.

        * wdy: an instance of WealthDynamics
        * w_0: scalar
        * n: int
```

(continues on next page)

(continued from previous page)

```
    """
    z = wdy.z_mean + np.sqrt(wdy.z_var) * np.random.randn()
    w = np.empty(n)
    w[0] = w_0
    for t in range(n-1):
        w[t+1], z = wdy.update_states(w[t], z)
    return w
```

Now here's function to simulate a cross section of households forward in time.

Note the use of parallelization to speed up computation.

```
@njit(parallel=True)
def update_cross_section(wdy, w_distribution, shift_length=500):
    """
    Shifts a cross-section of household forward in time

    * wdy: an instance of WealthDynamics
    * w_distribution: array_like, represents current cross-section

    Takes a current distribution of wealth values as w_distribution
    and updates each w_t in w_distribution to w_{t+j}, where
    j = shift_length.

    Returns the new distribution.

    """
    new_distribution = np.empty_like(w_distribution)

    # Update each household
    for i in prange(len(new_distribution)):
        z = wdy.z_mean + np.sqrt(wdy.z_var) * np.random.randn()
        w = w_distribution[i]
        for t in range(shift_length-1):
            w, z = wdy.update_states(w, z)
        new_distribution[i] = w
    return new_distribution
```

Parallelization is very effective in the function above because the time path of each household can be calculated independently once the path for the aggregate state is known.

5.5 Applications

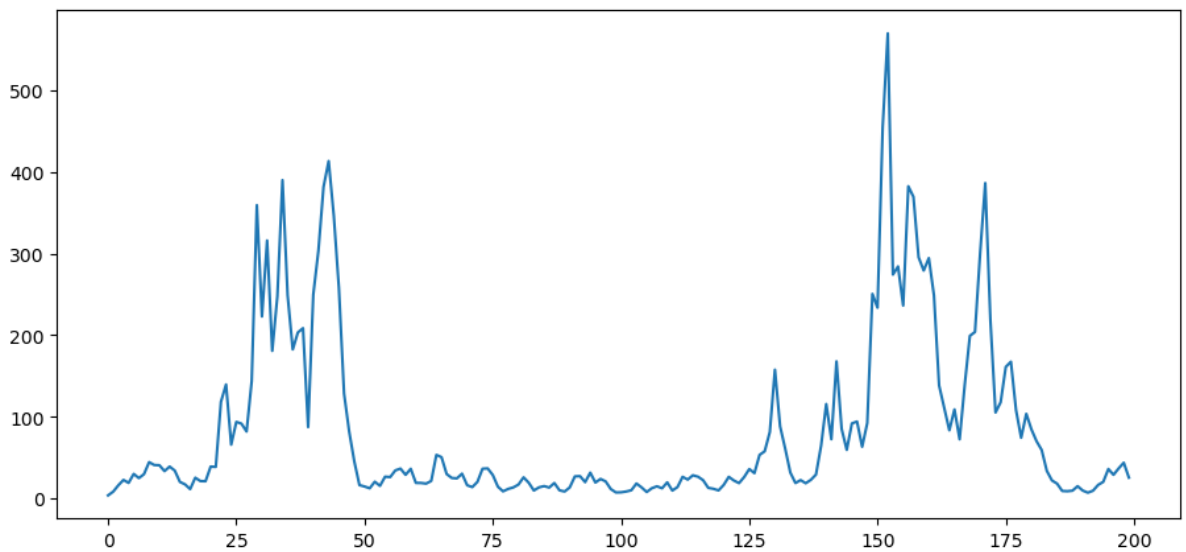
Let's try simulating the model at different parameter values and investigate the implications for the wealth distribution.

5.5.1 Time Series

Let's look at the wealth dynamics of an individual household.

```
wdy = WealthDynamics()
ts_length = 200
w = wealth_time_series(wdy, wdy.y_mean, ts_length)
```

```
fig, ax = plt.subplots()
ax.plot(w)
plt.show()
```



Notice the large spikes in wealth over time.

Such spikes are similar to what we observed in time series when *we studied Kesten processes*.

5.5.2 Inequality Measures

Let's look at how inequality varies with returns on financial assets.

The next function generates a cross section and then computes the Lorenz curve and Gini coefficient.

```
def generate_lorenz_and_gini(wdy, num_households=100_000, T=500):
    """
    Generate the Lorenz curve data and gini coefficient corresponding to a
    WealthDynamics mode by simulating num_households forward to time T.
    """
    ψ_0 = np.full(num_households, wdy.y_mean)
    z_0 = wdy.z_mean
```

(continues on next page)

(continued from previous page)

```
    ψ_star = update_cross_section(wdy, ψ_0, shift_length=T)
    return qe.gini_coefficient(ψ_star), qe.lorenz_curve(ψ_star)
```

Now we investigate how the Lorenz curves associated with the wealth distribution change as return to savings varies.

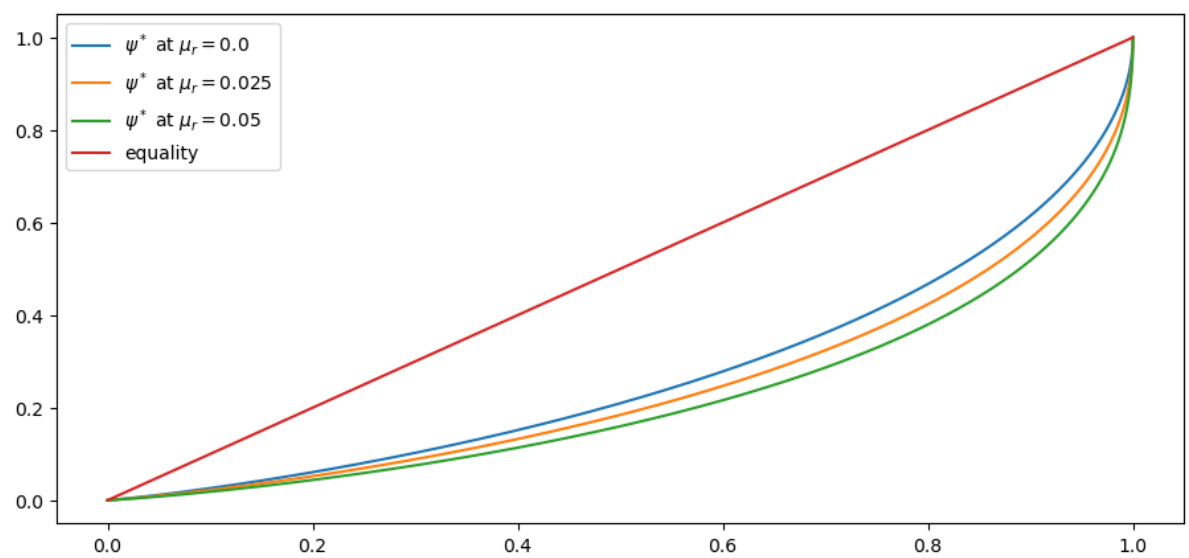
The code below plots Lorenz curves for three different values of μ_r.

If you are running this yourself, note that it will take one or two minutes to execute.

This is unavoidable because we are executing a CPU intensive task.

In fact the code, which is JIT compiled and parallelized, runs extremely fast relative to the number of computations.

```
%%time

fig, ax = plt.subplots()
μ_r_vals = (0.0, 0.025, 0.05)
gini_vals = []

for μ_r in μ_r_vals:
    wdy = WealthDynamics(μ_r=μ_r)
    gv, (f_vals, l_vals) = generate_lorenz_and_gini(wdy)
    ax.plot(f_vals, l_vals, label=f'$\psi^*$ at $\mu_r = {μ_r:0.2}$')
    gini_vals.append(gv)

ax.plot(f_vals, f_vals, label='equality')
ax.legend(loc="upper left")
plt.show()
```




```
CPU times: user 56.7 s, sys: 76.8 ms, total: 56.7 s
Wall time: 14.9 s
```

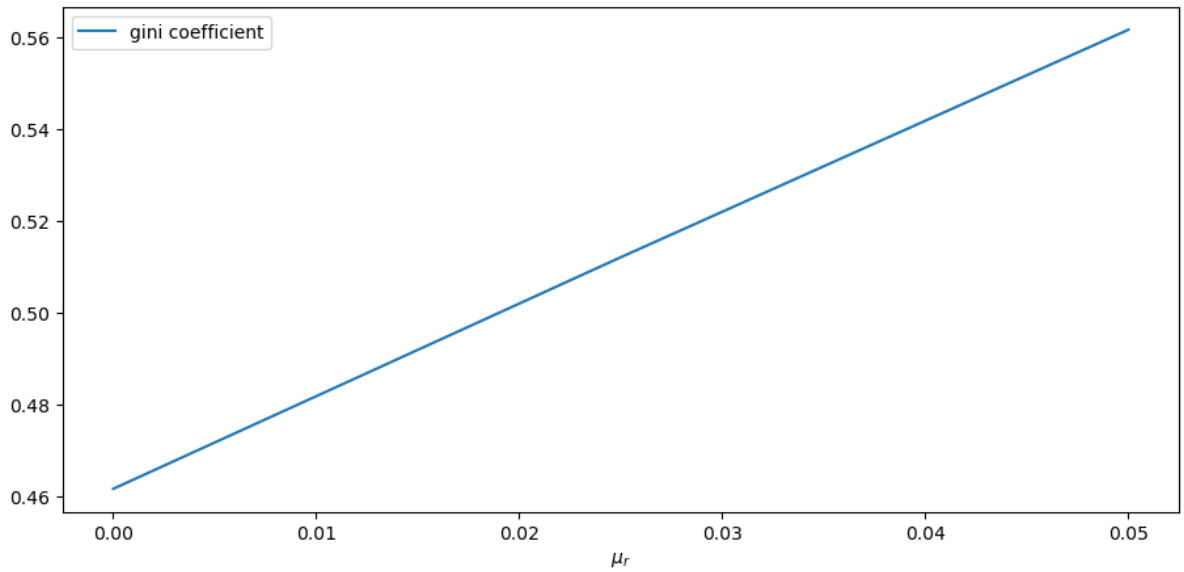
The Lorenz curve shifts downwards as returns on financial income rise, indicating a rise in inequality.

We will look at this again via the Gini coefficient immediately below, but first consider the following image of our system resources when the code above is executing:

Since the code is both efficiently JIT compiled and fully parallelized, it's close to impossible to make this sequence of tasks run faster without changing hardware.

Now let's check the Gini coefficient.

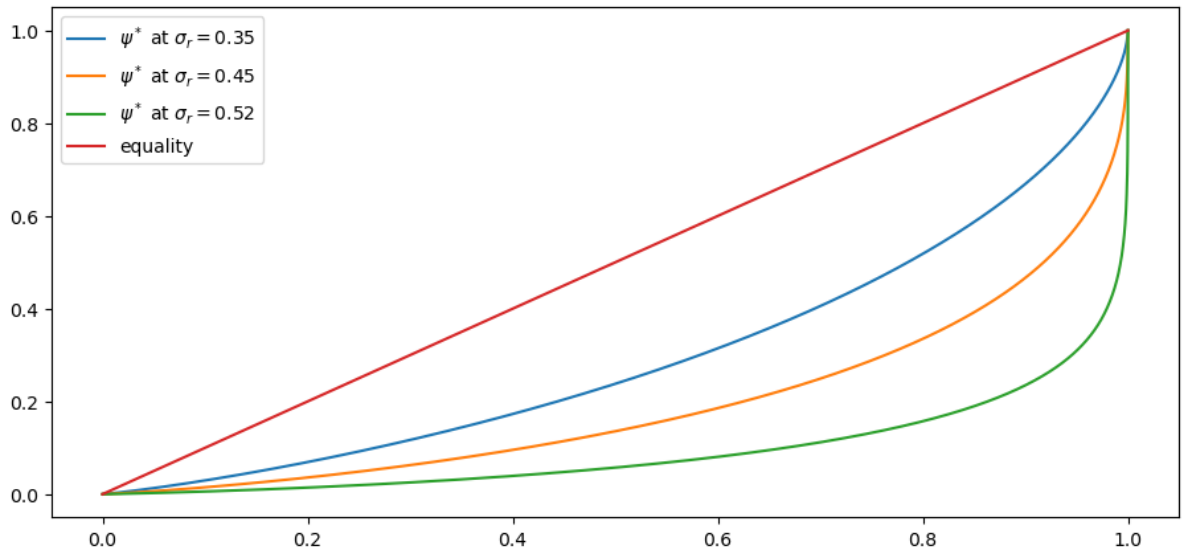
```
fig, ax = plt.subplots()
ax.plot(μ_r_vals, gini_vals, label='gini coefficient')
ax.set_xlabel("$\mu_r$")
ax.legend()
plt.show()
```



Once again, we see that inequality increases as returns on financial income rise.

Let's finish this section by investigating what happens when we change the volatility term σ_r in financial returns.

```
%%time

fig, ax = plt.subplots()
σ_r_vals = (0.35, 0.45, 0.52)
gini_vals = []

for σ_r in σ_r_vals:
    wdy = WealthDynamics(σ_r=σ_r)
    gv, (f_vals, l_vals) = generate_lorenz_and_gini(wdy)
    ax.plot(f_vals, l_vals, label=f'$\psi^*$ at $\sigma_r = {σ_r:0.2}$')
    gini_vals.append(gv)

ax.plot(f_vals, f_vals, label='equality')
ax.legend(loc="upper left")
plt.show()
```




```
CPU times: user 56.1 s, sys: 24.1 ms, total: 56.2 s
Wall time: 14.3 s
```

We see that greater volatility has the effect of increasing inequality in this model.

5.6 Exercises

Exercise 5.6.1

For a wealth or income distribution with Pareto tail, a higher tail index suggests lower inequality.

Indeed, it is possible to prove that the Gini coefficient of the Pareto distribution with tail index a is $1/(2a-1)$.

To the extent that you can, confirm this by simulation.

In particular, generate a plot of the Gini coefficient against the tail index using both the theoretical value just given and the value computed from a sample via `qe.gini_coefficient`.

For the values of the tail index, use `a_vals = np.linspace(1, 10, 25)`.

Use sample of size 1,000 for each a and the sampling method for generating Pareto draws employed in the discussion of Lorenz curves for the Pareto distribution.

To the extent that you can, interpret the monotone relationship between the Gini index and a.

Solution to Exercise 5.6.1

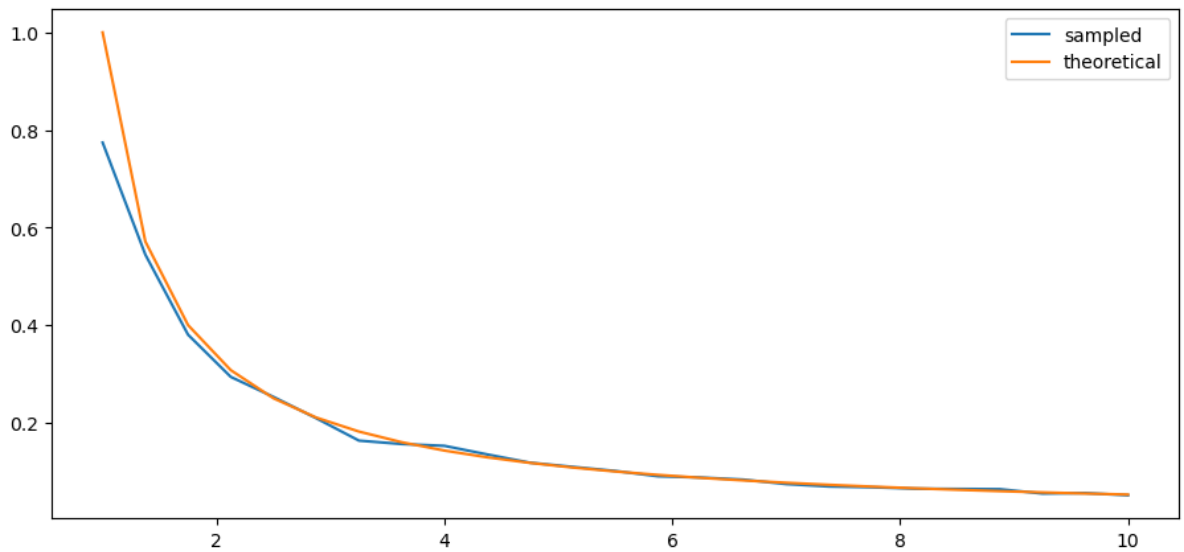
Here is one solution, which produces a good match between theory and simulation.

```
a_vals = np.linspace(1, 10, 25)  # Pareto tail index
ginis = np.empty_like(a_vals)

n = 1000                         # size of each sample
fig, ax = plt.subplots()
```

(continues on next page)

(continued from previous page)

```
for i, a in enumerate(a_vals):
    y = np.random.uniform(size=n)**(-1/a)
    ginis[i] = qe.gini_coefficient(y)
ax.plot(a_vals, ginis, label='sampled')
ax.plot(a_vals, 1/(2*a_vals - 1), label='theoretical')
ax.legend()
plt.show()
```



In general, for a Pareto distribution, a higher tail index implies less weight in the right hand tail.

This means less extreme values for wealth and hence more equality.

More equality translates to a lower Gini index.

Exercise 5.6.2

The wealth process (5.1) is similar to a *Kesten process*.

This is because, according to (5.2), savings is constant for all wealth levels above $\hat{w}$.

When savings is constant, the wealth process has the same quasi-linear structure as a Kesten process, with multiplicative and additive shocks.

The Kesten–Goldie theorem tells us that Kesten processes have Pareto tails under a range of parameterizations.

The theorem does not directly apply here, since savings is not always constant and since the multiplicative and additive terms in (5.1) are not IID.

At the same time, given the similarities, perhaps Pareto tails will arise.

To test this, run a simulation that generates a cross-section of wealth and generate a rank-size plot.

If you like, you can use the function `rank_size` from the `quantecon` library (documentation here).

In viewing the plot, remember that Pareto tails generate a straight line. Is this what you see?

For sample size and initial conditions, use

```
num_households = 250_000
T = 500                                          # shift forward T periods
ψ_0 = np.full(num_households, wdy.y_mean)    # initial distribution
z_0 = wdy.z_mean
```

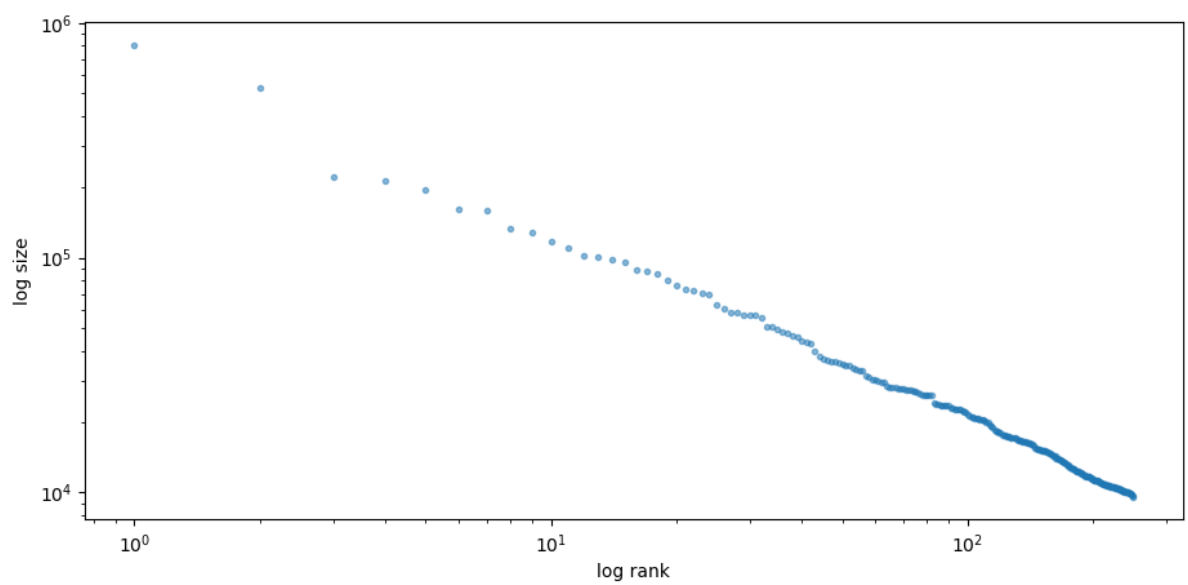
Solution to Exercise 5.6.2

First let's generate the distribution:

```
num_households = 250_000
T = 500  # how far to shift forward in time
wdy = WealthDynamics()
ψ_0 = np.full(num_households, wdy.y_mean)
z_0 = wdy.z_mean

ψ_star = update_cross_section(wdy, ψ_0, shift_length=T)
```

Now let's see the rank-size plot:

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

rank_data, size_data = qe.rank_size(ψ_star, c=0.001)
ax.loglog(rank_data, size_data, 'o', markersize=3.0, alpha=0.5)
ax.set_xlabel("log rank")
ax.set_ylabel("log size")

plt.show()
```



Part II

Asset Pricing & Finance

CHAPTER

SIX

ASSET PRICING: FINITE STATE MODELS

Contents

- *Asset Pricing: Finite State Models*
  - *Overview*
  - *Pricing Models*
  - *Prices in the Risk-Neutral Case*
  - *Risk Aversion and Asset Prices*
  - *Exercises*

> "A little knowledge of geometric series goes a long way" – Robert E. Lucas, Jr.
>
> "Asset pricing is all about covariances" – Lars Peter Hansen

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

```
!pip install quantecon
```

6.1 Overview

An asset is a claim on one or more future payoffs.

The spot price of an asset depends primarily on

- the anticipated income stream
- attitudes about risk
- rates of time preference

In this lecture, we consider some standard pricing models and dividend stream specifications.

We study how prices and dividend-price ratios respond in these different scenarios.

We also look at creating and pricing *derivative* assets that repackage income streams.

Key tools for the lecture are

- Markov processses
- formulas for predicting future values of functions of a Markov state

- a formula for predicting the discounted sum of future values of a Markov state

Let's start with some imports:

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

6.2 Pricing Models

Let $\{d_t\}_{t \geq 0}$ be a stream of dividends

- A time-t **cum-dividend** asset is a claim to the stream $d_t, d_{t+1}, \ldots$.
- A time-t **ex-dividend** asset is a claim to the stream $d_{t+1}, d_{t+2}, \ldots$.

Let's look at some equations that we expect to hold for prices of assets under ex-dividend contracts (we will consider cum-dividend pricing in the exercises).

6.2.1 Risk-Neutral Pricing

Our first scenario is risk-neutral pricing.

Let $\beta = 1/(1+\rho)$ be an intertemporal discount **factor**, where ρ is the **rate** at which agents discount the future.

The basic risk-neutral asset pricing equation for pricing one unit of an ex-dividend asset is

$$p_t = \beta \mathbb{E}_t [d_{t+1} + p_{t+1}] \tag{6.1}$$

This is a simple "cost equals expected benefit" relationship.

Here $\mathbb{E}_t[y]$ denotes the best forecast of y, conditioned on information available at time t.

More precisely, $\mathbb{E}_t[y]$ is the mathematical expectation of y conditional on information available at time t.

6.2.2 Pricing with Random Discount Factor

What happens if for some reason traders discount payouts differently depending on the state of the world?

Michael Harrison and David Kreps [HK79a] and Lars Peter Hansen and Scott Richard [HR87] showed that in quite general settings the price of an ex-dividend asset obeys

$$p_t = \mathbb{E}_t \left[m_{t+1} (d_{t+1} + p_{t+1}) \right] \tag{6.2}$$

for some **stochastic discount factor** m_{t+1}.

Here the fixed discount factor β in (6.1) has been replaced by the random variable m_{t+1}.

How anticipated future payoffs are evaluated now depends on statistical properties of m_{t+1}.

The stochastic discount factor can be specified to capture the idea that assets that tend to have good payoffs in bad states of the world are valued more highly than other assets whose payoffs don't behave that way.

This is because such assets pay well when funds are more urgently wanted.

We give examples of how the stochastic discount factor has been modeled below.

6.2.3 Asset Pricing and Covariances

Recall that, from the definition of a conditional covariance $\text{cov}_t(x_{t+1}, y_{t+1})$, we have

$$\mathbb{E}_t(x_{t+1}y_{t+1}) = \text{cov}_t(x_{t+1}, y_{t+1}) + \mathbb{E}_t x_{t+1}\mathbb{E}_t y_{t+1} \tag{6.3}$$

If we apply this definition to the asset pricing equation (6.2) we obtain

$$p_t = \mathbb{E}_t m_{t+1}\mathbb{E}_t(d_{t+1} + p_{t+1}) + \text{cov}_t(m_{t+1}, d_{t+1} + p_{t+1}) \tag{6.4}$$

It is useful to regard equation (6.4) as a generalization of equation (6.1)

- In equation (6.1), the stochastic discount factor $m_{t+1} = \beta$, a constant.
- In equation (6.1), the covariance term $\text{cov}_t(m_{t+1}, d_{t+1} + p_{t+1})$ is zero because $m_{t+1} = \beta$.
- In equation (6.1), $\mathbb{E}_t m_{t+1}$ can be interpreted as the reciprocal of the one-period risk-free gross interest rate.
- When m_{t+1} covaries more negatively with the payout $p_{t+1} + d_{t+1}$, the price of the asset is lower.

Equation (6.4) asserts that the covariance of the stochastic discount factor with the one period payout $d_{t+1} + p_{t+1}$ is an important determinant of the price p_t.

We give examples of some models of stochastic discount factors that have been proposed later in this lecture and also in a later lecture.

6.2.4 The Price-Dividend Ratio

Aside from prices, another quantity of interest is the **price-dividend ratio** $v_t := p_t/d_t$.

Let's write down an expression that this ratio should satisfy.

We can divide both sides of (6.2) by d_t to get

$$v_t = \mathbb{E}_t \left[m_{t+1} \frac{d_{t+1}}{d_t}(1 + v_{t+1}) \right] \tag{6.5}$$

Below we'll discuss the implication of this equation.

6.3 Prices in the Risk-Neutral Case

What can we say about price dynamics on the basis of the models described above?

The answer to this question depends on

1. the process we specify for dividends
2. the stochastic discount factor and how it correlates with dividends

For now we'll study the risk-neutral case in which the stochastic discount factor is constant.

We'll focus on how an asset price depends on a dividend process.

6.3.1 Example 1: Constant Dividends

The simplest case is risk-neutral price of a constant, non-random dividend stream $d_t = d > 0$.

Removing the expectation from (6.1) and iterating forward gives

$$\begin{aligned} p_t &= \beta(d + p_{t+1}) \\ &= \beta(d + \beta(d + p_{t+2})) \\ &\quad \vdots \\ &= \beta(d + \beta d + \beta^2 d + \cdots + \beta^{k-2} d + \beta^{k-1} p_{t+k}) \end{aligned}$$

If $\lim_{k \to +\infty} \beta^{k-1} p_{t+k} = 0$, this sequence converges to

$$\bar{p} := \frac{\beta d}{1 - \beta} \tag{6.6}$$

This is the equilibrium price in the constant dividend case.

Indeed, simple algebra shows that setting $p_t = \bar{p}$ for all t satisfies the difference equation $p_t = \beta(d + p_{t+1})$.

6.3.2 Example 2: Dividends with Deterministic Growth Paths

Consider a growing, non-random dividend process $d_{t+1} = g d_t$ where $0 < g\beta < 1$.

While prices are not usually constant when dividends grow over time, a price dividend-ratio can be.

If we guess this, substituting $v_t = v$ into (6.5) as well as our other assumptions, we get $v = \beta g(1 + v)$.

Since $\beta g < 1$, we have a unique positive solution:

$$v = \frac{\beta g}{1 - \beta g}$$

The price is then

$$p_t = \frac{\beta g}{1 - \beta g} d_t$$

If, in this example, we take $g = 1 + \kappa$ and let $\rho := 1/\beta - 1$, then the price becomes

$$p_t = \frac{1 + \kappa}{\rho - \kappa} d_t$$

This is called the *Gordon formula*.

6.3.3 Example 3: Markov Growth, Risk-Neutral Pricing

Next, we consider a dividend process

$$d_{t+1} = g_{t+1} d_t \tag{6.7}$$

The stochastic growth factor $\{g_t\}$ is given by

$$g_t = g(X_t), \quad t = 1, 2, \ldots$$

where

1. $\{X_t\}$ is a finite Markov chain with state space S and transition probabilities

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

2. g is a given function on S taking nonnegative values

You can think of

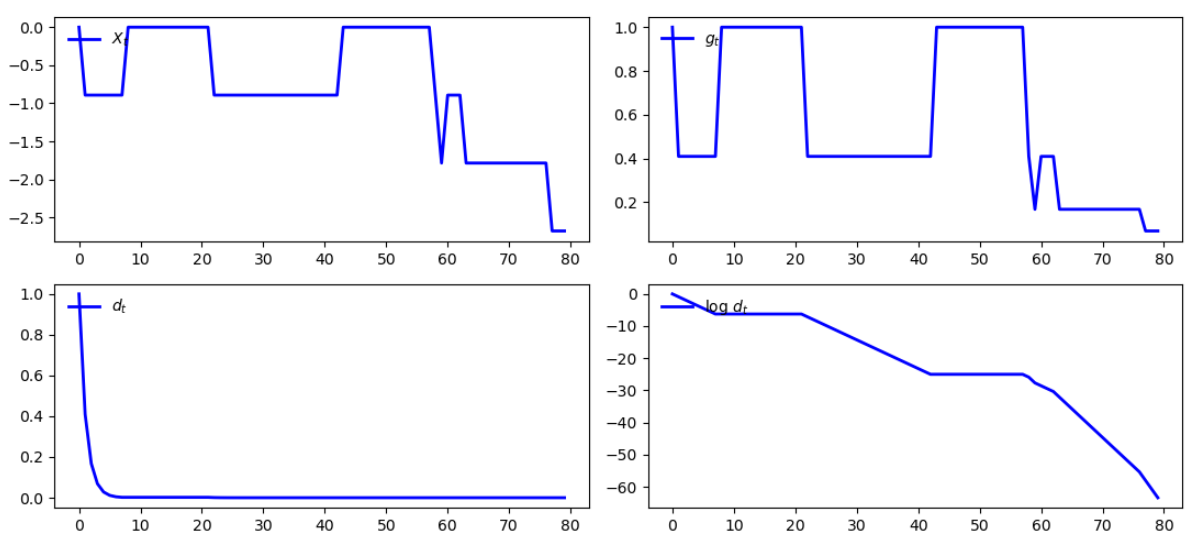
- S as n possible "states of the world" and X_t as the current state.
- g as a function that maps a given state X_t into a growth of dividends factor $g_t = g(X_t)$.
- $\ln g_t = \ln(d_{t+1}/d_t)$ is the growth rate of dividends.

(For a refresher on notation and theory for finite Markov chains see this lecture)

The next figure shows a simulation, where

- $\{X_t\}$ evolves as a discretized AR1 process produced using Tauchen's method.
- $g_t = \exp(X_t)$, so that $\ln g_t = X_t$ is the growth rate.

```
n = 7
mc = qe.tauchen(n, 0.96, 0.25)
sim_length = 80

x_series = mc.simulate(sim_length, init=np.median(mc.state_values))
g_series = np.exp(x_series)
d_series = np.cumprod(g_series) # Assumes d_0 = 1

series = [x_series, g_series, d_series, np.log(d_series)]
labels = ['$X_t$', '$g_t$', '$d_t$', r'$\log \, d_t$']

fig, axes = plt.subplots(2, 2)
for ax, s, label in zip(axes.flatten(), series, labels):
    ax.plot(s, 'b-', lw=2, label=label)
    ax.legend(loc='upper left', frameon=False)
plt.tight_layout()
plt.show()
```



Pricing Formula

To obtain asset prices in this setting, let's adapt our analysis from the case of deterministic growth.

In that case, we found that v is constant.

This encourages us to guess that, in the current case, v_t is a fixed function of the state X_t.

We seek a function v such that the price-dividend ratio satisfies $v_t = v(X_t)$.

We can substitute this guess into (6.5) to get

$$v(X_t) = \beta \mathbb{E}_t[g(X_{t+1})(1 + v(X_{t+1}))]$$

If we condition on $X_t = x$, this becomes

$$v(x) = \beta \sum_{y \in S} g(y)(1 + v(y))P(x, y)$$

or

$$v(x) = \beta \sum_{y \in S} K(x, y)(1 + v(y)) \quad \text{where} \quad K(x, y) := g(y)P(x, y) \tag{6.8}$$

Suppose that there are n possible states $x_1, \dots, x_n$.

We can then think of (6.8) as n stacked equations, one for each state, and write it in matrix form as

$$v = \beta K(\mathbb{1} + v) \tag{6.9}$$

Here

- v is understood to be the column vector $(v(x_1), \dots, v(x_n))'$.
- K is the matrix $(K(x_i, x_j))_{1 \leq i,j \leq n}$.
- $\mathbb{1}$ is a column vector of ones.

When does equation (6.9) have a unique solution?

From the Neumann series lemma and Gelfand's formula, equation (6.9) has a unique solution when βK has spectral radius strictly less than one.

Thus, we require that the eigenvalues of K be strictly less than β^{-1} in modulus.

The solution is then

$$v = (I - \beta K)^{-1} \beta K \mathbb{1} \tag{6.10}$$

6.3.4 Code

Let's calculate and plot the price-dividend ratio at some parameters.

As before, we'll generate $\{X_t\}$ as a discretized AR1 process and set $g_t = \exp(X_t)$.

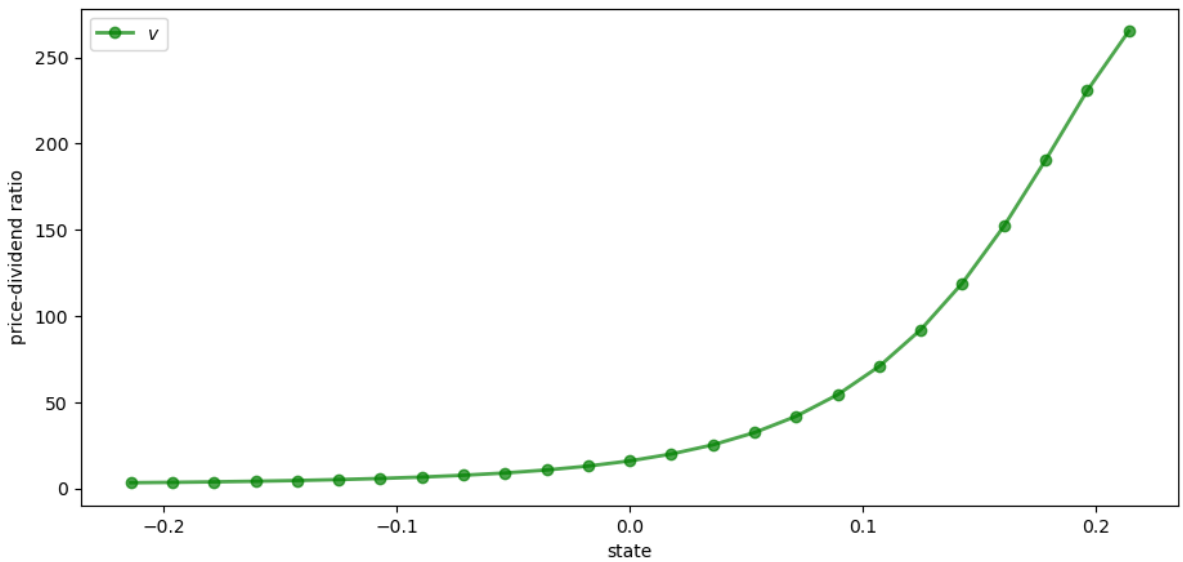
Here's the code, including a test of the spectral radius condition

```
n = 25  # Size of state space
β = 0.9
mc = qe.tauchen(n, 0.96, 0.02)
```

(continues on next page)

(continued from previous page)

```
K = mc.P * np.exp(mc.state_values)

warning_message = "Spectral radius condition fails"
assert np.max(np.abs(eigvals(K))) < 1 / β,  warning_message

I = np.identity(n)
v = solve(I - β * K, β * K @ np.ones(n))

fig, ax = plt.subplots()
ax.plot(mc.state_values, v, 'g-o', lw=2, alpha=0.7, label='$v$')
ax.set_ylabel("price-dividend ratio")
ax.set_xlabel("state")
ax.legend(loc='upper left')
plt.show()
```



Why does the price-dividend ratio increase with the state?

The reason is that this Markov process is positively correlated, so high current states suggest high future states.

Moreover, dividend growth is increasing in the state.

The anticipation of high future dividend growth leads to a high price-dividend ratio.

6.4 Risk Aversion and Asset Prices

Now let's turn to the case where agents are risk averse.

We'll price several distinct assets, including

- An endowment stream
- A consol (a type of bond issued by the UK government in the 19th century)
- Call options on a consol

6.4.1 Pricing a Lucas Tree

Let's start with a version of the celebrated asset pricing model of Robert E. Lucas, Jr. [Luc78].

Lucas considered an abstract pure exchange economy with these features:

- a single non-storable consumption good
- a Markov process that governs the total amount of the consumption good available each period
- a single *tree* that each period yields *fruit* that equals the total amount of consumption available to the economy
- a competitive market in *shares* in the tree that entitles their owners to corresponding shares of the *dividend* stream, i.e., the *fruit* stream, yielded by the tree
- a representative consumer who in a competitive equilibrium
  - consumes the economy's entire endowment each period
  - owns 100 percent of the shares in the tree

As in [Luc78], we suppose that the stochastic discount factor takes the form

$$m_{t+1} = \beta \frac{u'(c_{t+1})}{u'(c_t)} \tag{6.11}$$

where u is a concave utility function and c_t is time t consumption of a representative consumer.

(A derivation of this expression is given in a later lecture)

Assume the existence of an endowment that follows growth process (6.7).

The asset being priced is a claim on the endowment process, i.e., the *Lucas tree* described above.

Following [Luc78], we suppose that in equilibrium the representative consumer's consumption equals the aggregate endowment, so that $d_t = c_t$ for all t.

For utility, we'll assume the **constant relative risk aversion** (CRRA) specification

$$u(c) = \frac{c^{1-\gamma}}{1-\gamma} \text{ with } \gamma > 0 \tag{6.12}$$

When $\gamma = 1$ we let $u(c) = \ln c$.

Inserting the CRRA specification into (6.11) and using $c_t = d_t$ gives

$$m_{t+1} = \beta \left(\frac{c_{t+1}}{c_t} \right)^{-\gamma} = \beta g_{t+1}^{-\gamma} \tag{6.13}$$

Substituting this into (6.5) gives the price-dividend ratio formula

$$v(X_t) = \beta \mathbb{E}_t \left[g(X_{t+1})^{1-\gamma} (1 + v(X_{t+1})) \right] \tag{6.14}$$

Conditioning on $X_t = x$, we can write this as

$$v(x) = \beta \sum_{y \in S} g(y)^{1-\gamma} (1 + v(y)) P(x, y)$$

If we let

$$J(x, y) := g(y)^{1-\gamma} P(x, y)$$

then we can rewrite equation (6.14) in vector form as

$$v = \beta J (\mathbb{1} + v)$$

Assuming that the spectral radius of J is strictly less than β^{-1}, this equation has the unique solution

$$v = (I - \beta J)^{-1} \beta J \mathbb{1} \tag{6.15}$$

We will define a function tree_price to compute v given parameters stored in the class AssetPriceModel

```
class AssetPriceModel:
    """
    A class that stores the primitives of the asset pricing model.

    Parameters
    ----------
    β : scalar, float
        Discount factor
    mc : MarkovChain
        Contains the transition matrix and set of state values for the state
        process
    γ : scalar(float)
        Coefficient of risk aversion
    g : callable
        The function mapping states to growth rates

    """
    def __init__(self, β=0.96, mc=None, γ=2.0, g=np.exp):
        self.β, self.γ = β, γ
        self.g = g

        # A default process for the Markov chain
        if mc is None:
            self.ρ = 0.9
            self.σ = 0.02
            self.mc = qe.tauchen(n, self.ρ, self.σ)
        else:
            self.mc = mc

        self.n = self.mc.P.shape[0]

    def test_stability(self, Q):
        """
        Stability test for a given matrix Q.
        """
        sr = np.max(np.abs(eigvals(Q)))
        if not sr < 1 / self.β:
            msg = f"Spectral radius condition failed with radius = {sr}"
            raise ValueError(msg)


def tree_price(ap):
    """
    Computes the price-dividend ratio of the Lucas tree.

    Parameters
    ----------
    ap: AssetPriceModel
        An instance of AssetPriceModel containing primitives

    Returns
    -------
```

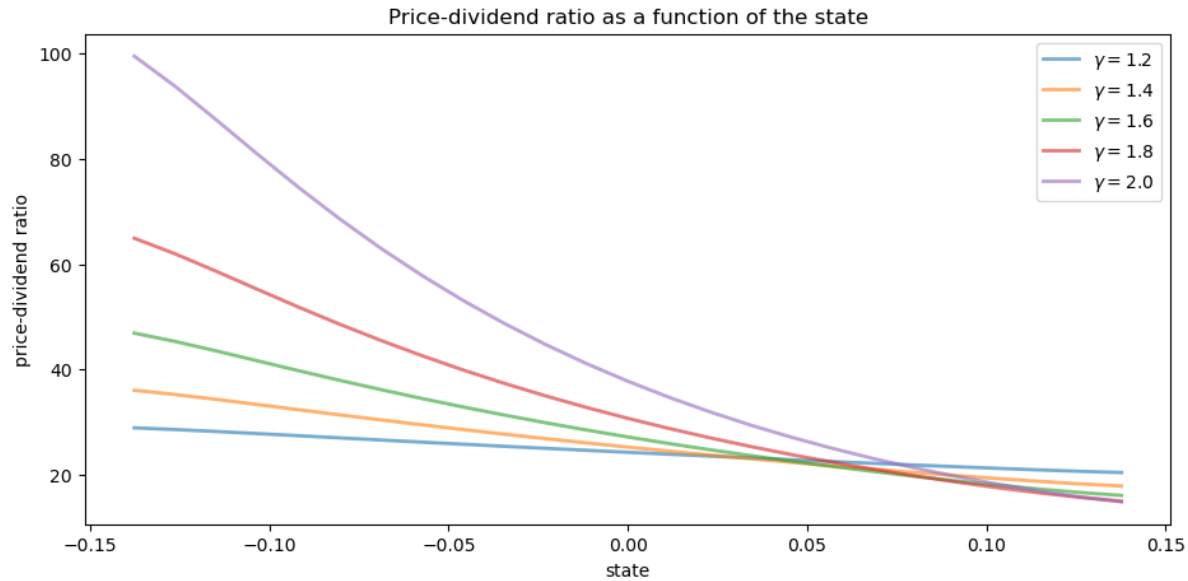
(continues on next page)

(continued from previous page)

```
    v : array_like(float)
        Lucas tree price-dividend ratio

    """
    # Simplify names, set up matrices
    β, γ, P, y = ap.β, ap.γ, ap.mc.P, ap.mc.state_values
    J = P * ap.g(y)**(1 - γ)

    # Make sure that a unique solution exists
    ap.test_stability(J)

    # Compute v
    I = np.identity(ap.n)
    Ones = np.ones(ap.n)
    v = solve(I - β * J, β * J @ Ones)

    return v
```

Here's a plot of v as a function of the state for several values of γ, with a positively correlated Markov process and $g(x) = \exp(x)$

```
γs = [1.2, 1.4, 1.6, 1.8, 2.0]
ap = AssetPriceModel()
states = ap.mc.state_values

fig, ax = plt.subplots()

for γ in γs:
    ap.γ = γ
    v = tree_price(ap)
    ax.plot(states, v, lw=2, alpha=0.6, label=rf"$\gamma = {γ}$")

ax.set_title('Price-dividend ratio as a function of the state')
ax.set_ylabel("price-dividend ratio")
ax.set_xlabel("state")
ax.legend(loc='upper right')
plt.show()
```



Notice that v is decreasing in each case.

This is because, with a positively correlated state process, higher states indicate higher future consumption growth.

With the stochastic discount factor (6.13), higher growth decreases the discount factor, lowering the weight placed on future dividends.

Special Cases

In the special case $\gamma = 1$, we have $J = P$.

Recalling that $P^i \mathbb{1} = \mathbb{1}$ for all i and applying Neumann's geometric series lemma, we are led to

$$v = \beta(I - \beta P)^{-1}\mathbb{1} = \beta \sum_{i=0}^{\infty} \beta^i P^i \mathbb{1} = \beta \frac{1}{1-\beta}\mathbb{1}$$

Thus, with log preferences, the price-dividend ratio for a Lucas tree is constant.

Alternatively, if $\gamma = 0$, then $J = K$ and we recover the risk-neutral solution (6.10).

This is as expected, since $\gamma = 0$ implies $u(c) = c$ (and hence agents are risk-neutral).

6.4.2 A Risk-Free Consol

Consider the same pure exchange representative agent economy.

A risk-free consol promises to pay a constant amount $\zeta > 0$ each period.

Recycling notation, let p_t now be the price of an ex-coupon claim to the consol.

An ex-coupon claim to the consol entitles an owner at the end of period t to

- ζ in period $t+1$, plus
- the right to sell the claim for p_{t+1} next period

The price satisfies (6.2) with $d_t = \zeta$, or

$$p_t = \mathbb{E}_t \left[m_{t+1}(\zeta + p_{t+1}) \right]$$

With the stochastic discount factor (6.13), this becomes

$$p_t = \mathbb{E}_t \left[\beta g_{t+1}^{-\gamma} (\zeta + p_{t+1}) \right] \tag{6.16}$$

Guessing a solution of the form $p_t = p(X_t)$ and conditioning on $X_t = x$, we get

$$p(x) = \beta \sum_{y \in S} g(y)^{-\gamma} (\zeta + p(y)) P(x, y)$$

Letting $M(x, y) = P(x, y) g(y)^{-\gamma}$ and rewriting in vector notation yields the solution

$$p = (I - \beta M)^{-1} \beta M \zeta \mathbb{1} \tag{6.17}$$

The above is implemented in the function consol_price.

```
def consol_price(ap, ζ):
    """
    Computes price of a consol bond with payoff ζ

    Parameters
    ----------
    ap: AssetPriceModel
        An instance of AssetPriceModel containing primitives

    ζ : scalar(float)
        Coupon of the console

    Returns
    -------
    p : array_like(float)
        Console bond prices

    """
    # Simplify names, set up matrices
    β, γ, P, y = ap.β, ap.γ, ap.mc.P, ap.mc.state_values
    M = P * ap.g(y)**(- γ)

    # Make sure that a unique solution exists
    ap.test_stability(M)

    # Compute price
    I = np.identity(ap.n)
    Ones = np.ones(ap.n)
    p = solve(I - β * M, β * ζ * M @ Ones)

    return p
```

6.4.3 Pricing an Option to Purchase the Consol

Let's now price options of various maturities.

We'll study an option that gives the owner the right to purchase a consol at a price p_S.

An Infinite Horizon Call Option

We want to price an *infinite horizon* option to purchase a consol at a price p_S.

The option entitles the owner at the beginning of a period either

1. to purchase the bond at price p_S now, or
2. not to exercise the option to purchase the asset now but to retain the right to exercise it later

Thus, the owner either *exercises* the option now or chooses *not to exercise* and wait until next period.

This is termed an infinite-horizon *call option* with *strike price* p_S.

The owner of the option is entitled to purchase the consol at price p_S at the beginning of any period, after the coupon has been paid to the previous owner of the bond.

The fundamentals of the economy are identical with the one above, including the stochastic discount factor and the process for consumption.

Let $w(X_t, p_S)$ be the value of the option when the time t growth state is known to be X_t but *before* the owner has decided whether to exercise the option at time t (i.e., today).

Recalling that $p(X_t)$ is the value of the consol when the initial growth state is X_t, the value of the option satisfies

$$w(X_t, p_S) = \max \left\{ \beta \, \mathbb{E}_t \frac{u'(c_{t+1})}{u'(c_t)} w(X_{t+1}, p_S), \; p(X_t) - p_S \right\}$$

The first term on the right is the value of waiting, while the second is the value of exercising now.

We can also write this as

$$w(x, p_S) = \max \left\{ \beta \sum_{y \in S} P(x,y) g(y)^{-\gamma} w(y, p_S), \; p(x) - p_S \right\} \tag{6.18}$$

With $M(x,y) = P(x,y)g(y)^{-\gamma}$ and w as the vector of values $(w(x_i), p_S)_{i=1}^n$, we can express (6.18) as the nonlinear vector equation

$$w = \max\{\beta M w, \; p - p_S \mathbb{1}\} \tag{6.19}$$

To solve (6.19), form an operator T that maps vector w into vector Tw via

$$Tw = \max\{\beta M w, \; p - p_S \mathbb{1}\}$$

Start at some initial w and iterate with T to convergence .

We can find the solution with the following function call_option

```
def call_option(ap, ζ, p_s, ϵ=1e-7):
    """
    Computes price of a call option on a consol bond.

    Parameters
    ----------
    ap: AssetPriceModel
        An instance of AssetPriceModel containing primitives

    ζ : scalar(float)
        Coupon of the console
```

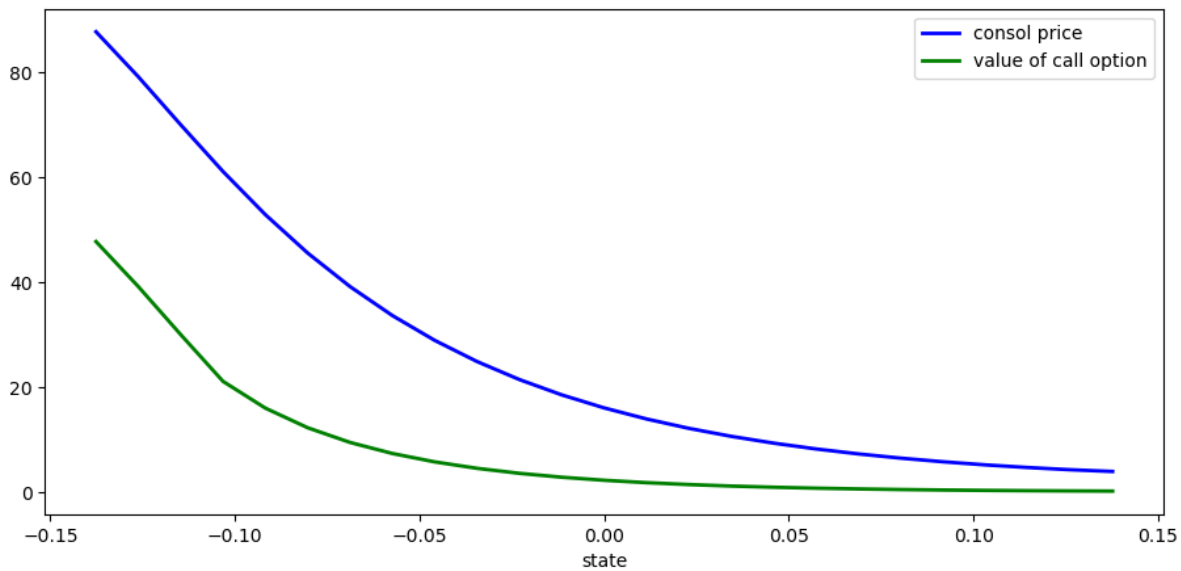
(continues on next page)

(continued from previous page)

```
    p_s : scalar(float)
        Strike price

    ϵ : scalar(float), optional(default=1e-8)
        Tolerance for infinite horizon problem

    Returns
    -------
    w : array_like(float)
        Infinite horizon call option prices

    """
    # Simplify names, set up matrices
    β, γ, P, y = ap.β, ap.γ, ap.mc.P, ap.mc.state_values
    M = P * ap.g(y)**(- γ)

    # Make sure that a unique consol price exists
    ap.test_stability(M)

    # Compute option price
    p = consol_price(ap, ζ)
    w = np.zeros(ap.n)
    error = ϵ + 1
    while error > ϵ:
        # Maximize across columns
        w_new = np.maximum(β * M @ w, p - p_s)
        # Find maximal difference of each component and update
        error = np.amax(np.abs(w - w_new))
        w = w_new

    return w
```

Here's a plot of w compared to the consol price when $P_S = 40$

```
ap = AssetPriceModel(β=0.9)
ζ = 1.0
strike_price = 40

x = ap.mc.state_values
p = consol_price(ap, ζ)
w = call_option(ap, ζ, strike_price)

fig, ax = plt.subplots()
ax.plot(x, p, 'b-', lw=2, label='consol price')
ax.plot(x, w, 'g-', lw=2, label='value of call option')
ax.set_xlabel("state")
ax.legend(loc='upper right')
plt.show()
```



In high values of the Markov growth state, the value of the option is close to zero.

This is despite the facts that the Markov chain is irreducible and that low states — where the consol prices are high — will be visited recurrently.

The reason for low valuations in high Markov growth states is that $\beta = 0.9$, so future payoffs are discounted substantially.

6.4.4 Risk-Free Rates

Let's look at risk-free interest rates over different periods.

The One-period Risk-free Interest Rate

As before, the stochastic discount factor is $m_{t+1} = \beta g_{t+1}^{-\gamma}$.

It follows that the reciprocal R_t^{-1} of the gross risk-free interest rate R_t in state x is

$$\mathbb{E}_t m_{t+1} = \beta \sum_{y \in S} P(x, y) g(y)^{-\gamma}$$

We can write this as

$$m_1 = \beta M \mathbb{1}$$

where the i-th element of m_1 is the reciprocal of the one-period gross risk-free interest rate in state x_i.

Other Terms

Let m_j be an $n \times 1$ vector whose i th component is the reciprocal of the j -period gross risk-free interest rate in state x_i.

Then $m_1 = \beta M$, and $m_{j+1} = M m_j$ for $j \geq 1$.

6.5 Exercises

Exercise 6.5.1

In the lecture, we considered **ex-dividend assets**.

A **cum-dividend** asset is a claim to the stream $d_t, d_{t+1}, \ldots$.

Following (6.1), find the risk-neutral asset pricing equation for one unit of a cum-dividend asset.

With a constant, non-random dividend stream $d_t = d > 0$, what is the equilibrium price of a cum-dividend asset?

With a growing, non-random dividend process $d_t = gd_t$ where $0 < g\beta < 1$, what is the equilibrium price of a cum-dividend asset?

Solution to Exercise 6.5.1

For a cum-dividend asset, the basic risk-neutral asset pricing equation is

$$p_t = d_t + \beta \mathbb{E}_t[p_{t+1}]$$

With constant dividends, the equilibrium price is

$$p_t = \frac{1}{1-\beta} d_t$$

With a growing, non-random dividend process, the equilibrium price is

$$p_t = \frac{1}{1-\beta g} d_t$$

Exercise 6.5.2

Consider the following primitives

```
n = 5  # Size of State Space
P = np.full((n, n), 0.0125)
P[range(n), range(n)] += 1 - P.sum(1)
# State values of the Markov chain
s = np.array([0.95, 0.975, 1.0, 1.025, 1.05])
γ = 2.0
β = 0.94
```

Let g be defined by $g(x) = x$ (that is, g is the identity map).

Compute the price of the Lucas tree.

Do the same for

- the price of the risk-free consol when $\zeta = 1$
- the call option on the consol when $\zeta = 1$ and $p_S = 150.0$

Solution to Exercise 6.5.2

First, let's enter the parameters:

```
n = 5
P = np.full((n, n), 0.0125)
P[range(n), range(n)] += 1 - P.sum(1)
s = np.array([0.95, 0.975, 1.0, 1.025, 1.05])  # State values
mc = qe.MarkovChain(P, state_values=s)

γ = 2.0
β = 0.94
ζ = 1.0
p_s = 150.0
```

Next, we'll create an instance of `AssetPriceModel` to feed into the functions

```
apm = AssetPriceModel(β=β, mc=mc, γ=γ, g=lambda x: x)
```

Now we just need to call the relevant functions on the data:

```
tree_price(apm)
```

```
array([29.47401578, 21.93570661, 17.57142236, 14.72515002, 12.72221763])
```

```
consol_price(apm, ζ)
```

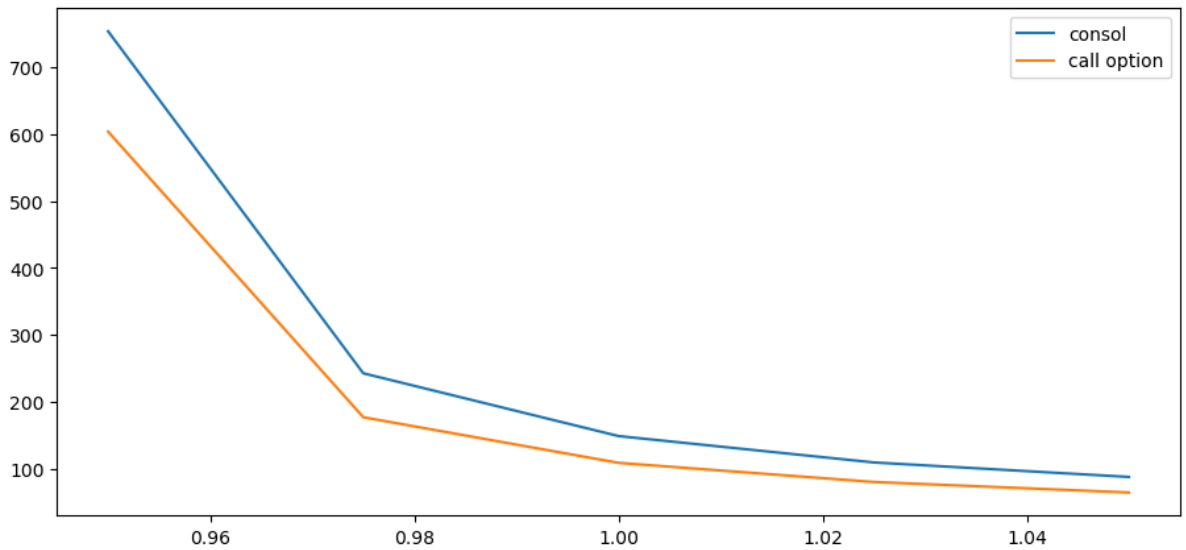
```
array([753.87100476, 242.55144082, 148.67554548, 109.25108965,
        87.56860139])
```

```
call_option(apm, ζ, p_s)
```

```
array([603.87100476, 176.8393343 , 108.67734499,  80.05179254,
        64.30843748])
```

Let's show the last two functions as a plot

```
fig, ax = plt.subplots()
ax.plot(s, consol_price(apm, ζ), label='consol')
ax.plot(s, call_option(apm, ζ, p_s), label='call option')
ax.legend()
plt.show()
```



Exercise 6.5.3

Let's consider finite horizon call options, which are more common than infinite horizon ones.

Finite horizon options obey functional equations closely related to (6.18).

A k period option expires after k periods.

If we view today as date zero, a k period option gives the owner the right to exercise the option to purchase the risk-free consol at the strike price p_S at dates $0, 1, \ldots, k-1$.

The option expires at time k.

Thus, for $k = 1, 2, \ldots$, let $w(x, k)$ be the value of a k-period option.

It obeys

$$w(x,k) = \max\left\{\beta \sum_{y \in S} P(x,y) g(y)^{-\gamma} w(y, k-1),\ p(x) - p_S\right\}$$

where $w(x, 0) = 0$ for all x.

We can express this as a sequence of nonlinear vector equations

$$w_k = \max\{\beta M w_{k-1},\ p - p_S \mathbb{1}\} \quad k = 1, 2, \ldots \quad \text{with } w_0 = 0$$

Write a function that computes w_k for any given k.

Compute the value of the option with `k = 5` and `k = 25` using parameter values as in Exercise 6.5.1.

Is one higher than the other? Can you give intuition?

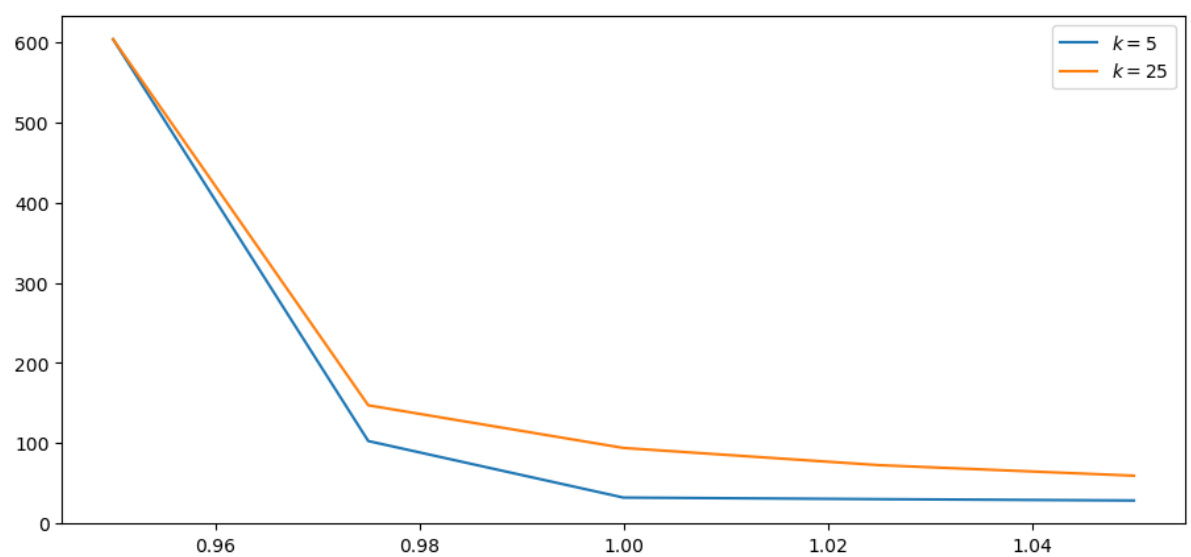
Solution to Exercise 6.5.3

Here's a suitable function:

```
def finite_horizon_call_option(ap, ζ, p_s, k):
    """
    Computes k period option value.
    """
    # Simplify names, set up matrices
    β, γ, P, y = ap.β, ap.γ, ap.mc.P, ap.mc.state_values
    M = P * ap.g(y)**(- γ)

    # Make sure that a unique solution exists
    ap.test_stability(M)


    # Compute option price
    p = consol_price(ap, ζ)
    w = np.zeros(ap.n)
    for i in range(k):
        # Maximize across columns
        w = np.maximum(β * M @ w, p - p_s)

    return w
```

Now let's compute the option values at `k=5` and `k=25`

```
fig, ax = plt.subplots()
for k in [5, 25]:
    w = finite_horizon_call_option(apm, ζ, p_s, k)
    ax.plot(s, w, label=rf'$k = {k}$')
ax.legend()
plt.show()
```



Not surprisingly, options with larger k are worth more.

This is because an owner has a longer horizon over which the option can be exercised.

CHAPTER

SEVEN

COMPETITIVE EQUILIBRIA WITH ARROW SECURITIES

7.1 Introduction

This lecture presents Python code for experimenting with competitive equilibria of an infinite-horizon pure exchange economy with

- Heterogeneous agents
- Endowments of a single consumption that are person-specific functions of a common Markov state
- Complete markets in one-period Arrow state-contingent securities
- Discounted expected utility preferences of a kind often used in macroeconomics and finance
- Common expected utility preferences across agents
- Common beliefs across agents
- A constant relative risk aversion (CRRA) one-period utility function that implies the existence of a representative consumer whose consumption process can be plugged into a formula for the pricing kernel for one-step Arrow securities and thereby determine equilbrium prices before determining an equilibrium distribution of wealth

Diverse endowments across agents provide motivations for individuals to want to reallocate consumption goods across time and Markov states

We impose restrictions that allow us to **Bellmanize** competitive equilibrium prices and quantities

We use Bellman equations to describe

- asset prices
- continuation wealth levels for each person
- state-by-state natural debt limits for each person

In the course of presenting the model we shall describe these important ideas

- a **resolvent operator** widely used in this class of models
- absence of **borrowing limits** in finite horizon economies
- state-by-state **borrowing limits** required in infinite horizon economies
- a counterpart of the **law of iterated expectations** known as a **law of iterated values**
- a **state-variable degeneracy** that prevails within a competitive equilibrium and that opens the way to various appearances of resolvent operators

7.2 The setting

In effect, this lecture implements a Python version of the model presented in section 9.3.3 of Ljungqvist and Sargent [LS18].

7.2.1 Preferences and endowments

In each period $t \geq 0$, a stochastic event $s_t \in \mathbf{S}$ is realized.

Let the history of events up until time t be denoted $s^t = [s_0, s_1, \dots, s_{t-1}, s_t]$.

(Sometimes we inadvertently reverse the recording order and denote a history as $s^t = [s_t, s_{t-1}, \dots, s_1, s_0]$.)

The unconditional probability of observing a particular sequence of events s^t is given by a probability measure $\pi_t(s^t)$.

For $t > \tau$, we write the probability of observing s^t conditional on the realization of s^τas $\pi_t(s^t|s^\tau)$.

We assume that trading occurs after observing s_0, which we capture by setting $\pi_0(s_0) = 1$ for the initially given value of s_0.

In this lecture we shall follow much macroeconomics and econometrics and assume that $\pi_t(s^t)$ is induced by a Markov process.

There are K consumers named $k = 1, \dots, K$.

Consumer k owns a stochastic endowment of one good $y_t^k(s^t)$ that depends on the history s^t.

The history s^t is publicly observable.

Consumer i purchases a history-dependent consumption plan $c^k = \{c_t^k(s^t)\}_{t=0}^\infty$

Consumer i orders consumption plans by

$$U_k(c^k) = \sum_{t=0}^\infty \sum_{s^t} \beta^t u_k[c_t^k(s^t)]\pi_t(s^t),$$

where $0 < \beta < 1$.

The right side is equal to $E_0 \sum_{t=0}^\infty \beta^t u_k(c_t^k)$, where E_0 is the mathematical expectation operator, conditioned on s_0.

Here $u_k(c)$ is an increasing, twice continuously differentiable, strictly concave function of consumption $c \geq 0$ of one good.

The utility function pf person k satisfies the Inada condition

$$\lim_{c \downarrow 0} u_k'(c) = +\infty.$$

This condition implies that each agent chooses strictly positive consumption for every date-history pair (t, s^t).

Those interior solutions enable us to confine our analysis to Euler equations that hold with equality and also guarantee that **natural debt limits** don't bind in economies like ours with sequential trading of Arrow securities.

We adopt the assumption, routinely employed in much of macroeconomics, that consumers share probabilities $\pi_t(s^t)$ for all t and s^t.

A **feasible allocation** satisfies

$$\sum_i c_t^k(s^t) \leq \sum_i y_t^k(s^t)$$

for all t and for all s^t.

7.3 Recursive Formulation

Following descriptions in section 9.3.3 of Ljungqvist and Sargent [LS18] chapter 9, we set up a competitive equilibrium of a pure exchange economy with complete markets in one-period Arrow securities.

When endowments $y^k(s)$ are all functions of a common Markov state s, the pricing kernel takes the form $Q(s'|s)$, where $Q(s'|s)$ is the price of one unit of consumption in state s' at date $t+1$ when the Markov state at date t is s.

These enable us to provide a recursive formulation of a consumer's optimization problem.

Consumer i's state at time t is its financial wealth a_t^k and Markov state s_t.

Let $v^k(a,s)$ be the optimal value of consumer i's problem starting from state (a,s).

- $v^k(a,s)$ is the maximum expected discounted utility that consumer i with current financial wealth a can attain in Markov state s.

The optimal value function satisfies the Bellman equation

$$v^k(a,s) = \max_{c,\hat{a}(s')} \left\{ u_k(c) + \beta \sum_{s'} v^k[\hat{a}(s'), s']\pi(s'|s) \right\}$$

where maximization is subject to the budget constraint

$$c + \sum_{s'} \hat{a}(s')Q(s'|s) \leq y^k(s) + a$$

and also the constraints

$$\begin{aligned} c &\geq 0, \\ -\hat{a}(s') &\leq \bar{A}^k(s'), \qquad \forall s' \in \mathbf{S} \end{aligned}$$

with the second constraint evidently being a set of state-by-state debt limits.

Note that the value function and decision rule that solve the Bellman equation implicitly depend on the pricing kernel $Q(\cdot|\cdot)$ because it appears in the agent's budget constraint.

Use the first-order conditions for the problem on the right of the Bellman equation and a Benveniste-Scheinkman formula and rearrange to get

$$Q(s_{t+1}|s_t) = \frac{\beta u_k'(c_{t+1}^k)\pi(s_{t+1}|s_t)}{u_k'(c_t^k)},$$

where it is understood that $c_t^k = c^k(s_t)$ and $c_{t+1}^k = c^k(s_{t+1})$.

A **recursive competitive equilibrium** is an initial distribution of wealth $\vec{a}_0$, a set of borrowing limits $\{\bar{A}^k(s)\}_{k=1}^K$, a pricing kernel $Q(s'|s)$, sets of value functions $\{v^k(a,s)\}_{i=1}^K$, and decision rules $\{c^k(s), a^k(s)\}_{i=1}^K$ such that

- The state-by-state borrowing constraints satisfy the recursion

$$\bar{A}^k(s) = y^k(s) + \sum_{s'} Q(s'|s)\bar{A}^k(s')$$

- For all i, given a_0^k, $\bar{A}^k(s)$, and the pricing kernel, the value functions and decision rules solve the consumers' problems;
- For all realizations of $\{s_t\}_{t=0}^\infty$, the consumption and asset portfolios $\{\{c_t^k, \{\hat{a}_{t+1}^k(s')\}_{s'}\}_i\}_t$ satisfy $\sum_i c_t^k = \sum_i y^k(s_t)$ and $\sum_i \hat{a}_{t+1}^k(s') = 0$ for all t and s'.
- The initial financial wealth vector $\vec{a}_0$ satisfies $\sum_{i=1}^K a_0^k = 0$.

The third condition asserts that there are zero net aggregate claims in all Markov states.

The fourth condition asserts that the economy is closed and starts from a situation in which there are zero net aggregate claims.

7.4 State Variable Degeneracy

Please see Ljungqvist and Sargent [LS18] for a description of timing protocol for trades consistent with an Arrow-Debreu vision in which

- at time 0 there are complete markets in a complete menu of history s^t-contingent claims on consumption at all dates that all trades occur at time zero
- all trades occur once and for all at time 0

If an allocation and pricing kernel Q in a recursive competitive equilibrium are to be consistent with the equilibrium allocation and price system that prevail in a corresponding complete markets economy with such history-contingent commodities and all trades occurring at time 0, we must impose that $a_0^k = 0$ for $k = 1, \dots, K$.

That is what assures that at time 0 the present value of each agent's consumption equals the present value of his endowment stream, the single budget constraint in arrangement with all trades occurring at time 0.

Starting the system with $a_0^k = 0$ for all i has a striking implication that we can call **state variable degeneracy**.

Here is what we mean by **state variable degeneracy**:

Although two state variables a, s appear in the value function $v^k(a, s)$, within a recursive competitive equilibrium starting from $a_0^k = 0 \ \forall i$ at initial Markov state s_0, two outcomes prevail:

- $a_0^k = 0$ for all i whenever the Markov state s_t returns to s_0.
- Financial wealth a is an exact function of the Markov state s.

The first finding asserts that each household recurrently visits the zero financial wealth state with which it began life.

The second finding asserts that within a competitive equilibrium the exogenous Markov state is all we require to track an individual.

Financial wealth turns out to be redundant because it is an exact function of the Markov state for each individual.

This outcome depends critically on there being complete markets in Arrow securities.

For example, it does not prevail in the incomplete markets setting of this lecture The Aiyagari Model

7.5 Markov Asset Prices

Let's start with a brief summary of formulas for computing asset prices in a Markov setting.

The setup assumes the following infrastructure

- Markov states: $s \in S = [\bar{s}_1, \dots, \bar{s}_n]$ governed by an n-state Markov chain with transition probability

$$P_{ij} = \text{Pr}\left\{s_{t+1} = \bar{s}_j \mid s_t = \bar{s}_k\right\}$$

- A collection $h = 1, \dots, H$ of $n \times 1$ vectors of H assets that pay off $d^h(s)$ in state s
- An $n \times n$ matrix pricing kernel Q for one-period Arrow securities, where Q_{ij} = price at time t in state $s_t = \bar{s}_i$ of one unit of consumption when $s_{t+1} = \bar{s}_j$ at time $t+1$:

$$Q_{ij} = \text{Price}\left\{s_{t+1} = \bar{s}_j \mid s_t = \bar{s}_i\right\}$$

- The price of risk-free one-period bond in state i is $R_i^{-1} = \sum_j Q_{i,j}$
- The gross rate of return on a one-period risk-free bond Markov state $\bar{s}_i$ is $R_i = (\sum_j Q_{i,j})^{-1}$

7.5.1 Exogenous Pricing Kernel

At this point, we'll take the pricing kernel Q as exogenous, i.e., determined outside the model

Two examples would be

- $Q = \beta P$ where $\beta \in (0, 1)$
- $Q = SP$ where S is an $n \times n$ matrix of *stochastic discount factors*

We'll write down implications of Markov asset pricing in a nutshell for two types of assets

- the price in Markov state s at time t of a **cum dividend** stock that entitles the owner at the beginning of time t to the time t dividend and the option to sell the asset at time $t+1$. The price evidently satisfies $p^h(\bar{s}_i) = d^h(\bar{s}_i) + \sum_j Q_{ij} p^h(\bar{s}_j)$, which implies that the vector p^h satisfies $p^h = d^h + Qp^h$ which implies the formula

$$p^h = (I - Q)^{-1} d^h$$

- the price in Markov state s at time t of an **ex dividend** stock that entitles the owner at the end of time t to the time $t+1$ dividend and the option to sell the stock at time $t+1$. The price is

$$p^h = (I - Q)^{-1} Q d^h$$

Below, we describe an equilibrium model with trading of one-period Arrow securities in which the pricing kernel is endogenous.

In constructing our model, we'll repeatedly encounter formulas that remind us of our asset pricing formulas.

7.5.2 Multi-Step-Forward Transition Probabilities and Pricing Kernels

The (i, j) component of the k-step ahead transition probability P^ℓ is

$$Prob(s_{t+\ell} = \bar{s}_j | s_t = \bar{s}_i) = P^\ell_{i,j}$$

The (i, j) component of the ℓ-step ahead pricing kernel Q^ℓ is

$$Q^{(\ell)}(s_{t+\ell} = \bar{s}_j | s_t = \bar{s}_i) = Q^\ell_{i,j}$$

We'll use these objects to state a useful property in asset pricing theory.

7.5.3 Laws of Iterated Expectations and Iterated Values

A **law of iterated values** has a mathematical structure that parallels a **law of iterated expectations**

We can describe its structure readily in the Markov setting of this lecture

Recall the following recursion satisfied by j step ahead transition probabilites for our finite state Markov chain:

$$P_j(s_{t+j} | s_t) = \sum_{s_{t+1}} P_{j-1}(s_{t+j} | s_{t+1}) P(s_{t+1} | s_t)$$

We can use this recursion to verify the law of iterated expectations applied to computing the conditional expectation of a

random variable $d(s_{t+j})$ conditioned on s_t via the following string of equalities

$$\begin{aligned}
E\left[Ed(s_{t+j})|s_{t+1}\right]|s_t &= \sum_{s_{t+1}}\left[\sum_{s_{t+j}} d(s_{t+j})P_{j-1}(s_{t+j}|s_{t+1})\right]P(s_{t+1}|s_t) \\
&= \sum_{s_{t+j}} d(s_{t+j})\left[\sum_{s_{t+1}} P_{j-1}(s_{t+j}|s_{t+1})P(s_{t+1}|s_t)\right] \\
&= \sum_{s_{t+j}} d(s_{t+j})P_j(s_{t+j}|s_t) \\
&= Ed(s_{t+j})|s_t
\end{aligned}$$

The pricing kernel for j step ahead Arrow securities satisfies the recursion

$$Q_j(s_{t+j}|s_t) = \sum_{s_{t+1}} Q_{j-1}(s_{t+j}|s_{t+1})Q(s_{t+1}|s_t)$$

The time t **value** in Markov state s_t of a time $t+j$ payout $d(s_{t+j})$ is

$$V(d(s_{t+j})|s_t) = \sum_{s_{t+j}} d(s_{t+j})Q_j(s_{t+j}|s_t)$$

The **law of iterated values** states

$$V\left[V(d(s_{t+j})|s_{t+1})\right]|s_t = V(d(s_{t+j}))|s_t$$

We verify it by pursuing the following a string of inequalities that are counterparts to those we used to verify the law of iterated expectations:

$$\begin{aligned}
V\left[V(d(s_{t+j})|s_{t+1})\right]|s_t &= \sum_{s_{t+1}}\left[\sum_{s_{t+j}} d(s_{t+j})Q_{j-1}(s_{t+j}|s_{t+1})\right]Q(s_{t+1}|s_t) \\
&= \sum_{s_{t+j}} d(s_{t+j})\left[\sum_{s_{t+1}} Q_{j-1}(s_{t+j}|s_{t+1})Q(s_{t+1}|s_t)\right] \\
&= \sum_{s_{t+j}} d(s_{t+j})Q_j(s_{t+j}|s_t) \\
&= EV(d(s_{t+j}))|s_t
\end{aligned}$$

7.6 General Equilibrium

Now we are ready to do some fun calculations.

We find it interesting to think in terms of analytical **inputs** into and **outputs** from our general equilibrium theorizing.

7.6.1 Inputs

- Markov states: $s \in S = [\bar{s}_1, \dots, \bar{s}_n]$ governed by an n-state Markov chain with transition probability

$$P_{ij} = \Pr\left\{s_{t+1} = \bar{s}_j \mid s_t = \bar{s}_i\right\}$$

- A collection of $K \times 1$ vectors of individual k endowments: $y^k(s), k = 1, \dots, K$
- An $n \times 1$ vector of aggregate endowment: $y(s) \equiv \sum_{k=1}^{K} y^k(s)$

- A collection of $K \times 1$ vectors of individual k consumptions: $c^k(s), k = 1, \ldots, K$
- A collection of restrictions on feasible consumption allocations for $s \in S$:

$$c(s) = \sum_{k=1}^{K} c^k(s) \leq y(s)$$

- Preferences: a common utility functional across agents $E_0 \sum_{t=0}^{\infty} \beta^t u(c_t^k)$ with CRRA one-period utility function $u(c)$ and discount factor $\beta \in (0,1)$

The one-period utility function is

$$u(c) = \frac{c^{1-\gamma}}{1-\gamma}$$

so that

$$u'(c) = c^{-\gamma}$$

7.6.2 Outputs

- An $n \times n$ matrix pricing kernel Q for one-period Arrow securities, where Q_{ij} = price at time t in state $s_t = \bar{s}_i$ of one unit of consumption when $s_{t+1} = \bar{s}_j$ at time $t+1$
- pure exchange so that $c(s) = y(s)$
- a $K \times 1$ vector distribution of wealth vector α, $\alpha_k \geq 0, \sum_{k=1}^{K} \alpha_k = 1$
- A collection of $n \times 1$ vectors of individual k consumptions: $c^k(s), k = 1, \ldots, K$

7.6.3 Q is the Pricing Kernel

For any agent $k \in [1, \ldots, K]$, at the equilibrium allocation, the one-period Arrow securities pricing kernel satisfies

$$Q_{ij} = \beta \left(\frac{c^k(\bar{s}_j)}{c^k(\bar{s}_i)} \right)^{-\gamma} P_{ij}$$

where Q is an $n \times n$ matrix

This follows from agent k's first-order necessary conditions.

But with the CRRA preferences that we have assumed, individual consumptions vary proportionately with aggregate consumption and therefore with the aggregate endowment.

- This is a consequence of our preference specification implying that **Engle curves** affine in wealth and therefore satisfy conditions for **Gorman aggregation**

Thus,

$$c^k(s) = \alpha_k c(s) = \alpha_k y(s)$$

for an arbitrary **distribution of wealth** in the form of an $K \times 1$ vector α that satisfies

$$\alpha_k \in (0,1), \quad \sum_{k=1}^{K} \alpha_k = 1$$

This means that we can compute the pricing kernel from

$$Q_{ij} = \beta \left(\frac{y_j}{y_i} \right)^{-\gamma} P_{ij} \tag{7.1}$$

Note that Q_{ij} is independent of vector α.

Key finding: We can compute competitive equilibrium **prices** prior to computing a **distribution of wealth**.

7.6.4 Values

Having computed an equilibrium pricing kernel Q, we can compute several **values** that are required to pose or represent the solution of an individual household's optimum problem.

We denote an $K \times 1$ vector of state-dependent values of agents' endowments in Markov state s as

$$A(s) = \begin{bmatrix} A^1(s) \\ \vdots \\ A^K(s) \end{bmatrix}, \quad s \in [\bar{s}_1, \dots, \bar{s}_n]$$

and an $n \times 1$ vector of continuation endowment values for each individual k as

$$A^k = \begin{bmatrix} A^k(\bar{s}_1) \\ \vdots \\ A^k(\bar{s}_n) \end{bmatrix}, \quad k \in [1, \dots, K]$$

A^k of consumer k satisfies

$$A^k = [I - Q]^{-1} [y^k]$$

where

$$y^k = \begin{bmatrix} y^k(\bar{s}_1) \\ \vdots \\ y^k(\bar{s}_n) \end{bmatrix} \equiv \begin{bmatrix} y_1^k \\ \vdots \\ v_n^k \end{bmatrix}$$

In a competitive equilibrium of an **infinite horizon** economy with sequential trading of one-period Arrow securities, $A^k(s)$ serves as a state-by-state vector of **debt limits** on the quantities of one-period Arrow securities paying off in state s at time $t+1$ that individual k can issue at time t.

These are often called **natural debt limits**.

Evidently, they equal the maximum amount that it is feasible for individual k to repay even if he consumes zero goods forevermore.

Remark: If we have an Inada condition at zero consumption or just impose that consumption be nonnegative, then in a **finite horizon** economy with sequential trading of one-period Arrow securities there is no need to impose natural debt limits. See the section below on a Finite Horizon Economy.

7.6.5 Continuation Wealth

Continuation wealth plays an important role in Bellmanizing a competitive equilibrium with sequential trading of a complete set of one-period Arrow securities.

We denote an $K \times 1$ vector of state-dependent continuation wealths in Markov state s as

$$\psi(s) = \begin{bmatrix} \psi^1(s) \\ \vdots \\ \psi^K(s) \end{bmatrix}, \quad s \in [\bar{s}_1, \dots, \bar{s}_n]$$

and an $n \times 1$ vector of continuation wealths for each individual k as

$$\psi^k = \begin{bmatrix} \psi^k(\bar{s}_1) \\ \vdots \\ \psi^k(\bar{s}_n) \end{bmatrix}, \quad k \in [1, \dots, K]$$

Continuation wealth ψ^k of consumer k satisfies

$$\psi^k = [I - Q]^{-1} [\alpha_k y - y^k] \tag{7.2}$$

where

$$y^k = \begin{bmatrix} y^k(\bar{s}_1) \\ \vdots \\ y^k(\bar{s}_n) \end{bmatrix}, \quad y = \begin{bmatrix} y(\bar{s}_1) \\ \vdots \\ y(\bar{s}_n) \end{bmatrix}$$

Note that $\sum_{k=1}^K \psi^k = 0_{n \times 1}$.

Remark: At the initial state $s_0 \in [\bar{s}_1, \dots, \bar{s}_n]$, the continuation wealth $\psi^k(s_0) = 0$ for all agents $k = 1, \dots, K$. This indicates that the economy begins with all agents being debt-free and financial-asset-free at time 0, state s_0.

Remark: Note that all agents' continuation wealths recurrently return to zero when the Markov state returns to whatever value s_0 it had at time 0.

7.6.6 Optimal Portfolios

A nifty feature of the model is that an optimal portfolio of a type k agent equals the continuation wealth that we just computed.

Thus, agent k's state-by-state purchases of Arrow securities next period depend only on next period's Markov state and equal

$$a_k(s) = \psi^k(s), \quad s \in [\bar{s}_1, \dots, \bar{s}_n] \tag{7.3}$$

7.6.7 Equilibrium Wealth Distribution α

With the initial state being a particular state $s_0 \in [\bar{s}_1, \dots, \bar{s}_n]$, we must have

$$\psi^k(s_0) = 0, \quad k = 1, \dots, K$$

which means the equilibrium distribution of wealth satisfies

$$\alpha_k = \frac{V_z y^k}{V_z y} \tag{7.4}$$

where $V \equiv [I - Q]^{-1}$ and z is the row index corresponding to the initial state s_0.

Since $\sum_{k=1}^K V_z y^k = V_z y$, $\sum_{k=1}^K \alpha_k = 1$.

In summary, here is the logical flow of an algorithm to compute a competitive equilibrium:

- compute Q from the aggregate allocation and formula (7.1)
- compute the distribution of wealth α from the formula (7.4)
- Using α assign each consumer k the share α_k of the aggregate endowment at each state
- return to the α-dependent formula (7.2) and compute continuation wealths

- via formula (7.3) equate agent k's portfolio to its continuation wealth state by state

We can also add formulas for optimal value functions in a competitive equilibrium with trades in a complete set of one-period state-contingent Arrow securities.

Call the optimal value functions J^k for consumer k.

For the infinite horizon economy now under study, the formula is

$$J^k = (I - \beta P)^{-1} u(\alpha_k y), \quad u(c) = \frac{c^{1-\gamma}}{1-\gamma}$$

where it is understood that $u(\alpha_k y)$ is a vector.

7.7 Python Code

We are ready to dive into some Python code.

As usual, we start with Python imports.

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

```
np.set_printoptions(suppress=True)
```

First, we create a Python class to compute the objects that comprise a competitive equilibrium with sequential trading of one-period Arrow securities.

In addition to handly infinite-horizon economies, the code is set up to handle finite-horizon economies indexed by horizon T.

We'll study some finite horizon economies after we look at some infinite-horizon economies.

```
class RecurCompetitive:
    """
    A class that represents a recursive competitive economy
    with one-period Arrow securities.
    """

    def __init__(self,
                 s,         # state vector
                 P,         # transition matrix
                 ys,        # endowments ys = [y1, y2, .., yI]
                 γ=0.5,     # risk aversion
                 β=0.98,    # discount rate
                 T=None):   # time horizon, none if infinite

        # preference parameters
        self.γ = γ
        self.β = β

        # variables dependent on state
        self.s = s
        self.P = P
        self.ys = ys
        self.y = np.sum(ys, 1)
```

(continues on next page)

(continued from previous page)

```
        # dimensions
        self.n, self.K = ys.shape

        # compute pricing kernel
        self.Q = self.pricing_kernel()

        # compute price of risk-free one-period bond
        self.PRF = self.price_risk_free_bond()

        # compute risk-free rate
        self.R = self.risk_free_rate()

        # V = [I - Q]^{-1} (infinite case)
        if T is None:
            self.T = None
            self.V = np.empty((1, n, n))
            self.V[0] = np.linalg.inv(np.eye(n) - self.Q)
        # V = [I + Q + Q^2 + ... + Q^T] (finite case)
        else:
            self.T = T
            self.V = np.empty((T+1, n, n))
            self.V[0] = np.eye(n)

            Qt = np.eye(n)
            for t in range(1, T+1):
                Qt = Qt.dot(self.Q)
                self.V[t] = self.V[t-1] + Qt

        # natural debt limit
        self.A = self.V[-1] @ ys

    def u(self, c):
        "The CRRA utility"

        return c ** (1 - self.γ) / (1 - self.γ)

    def u_prime(self, c):
        "The first derivative of CRRA utility"

        return c ** (-self.γ)

    def pricing_kernel(self):
        "Compute the pricing kernel matrix Q"

        c = self.y

        n = self.n
        Q = np.empty((n, n))

        for i in range(n):
            for j in range(n):
                ratio = self.u_prime(c[j]) / self.u_prime(c[i])
                Q[i, j] = self.β * ratio * P[i, j]

        self.Q = Q
```

(continues on next page)

(continued from previous page)

```
        return Q

    def wealth_distribution(self, s0_idx):
        "Solve for wealth distribution α"

        # set initial state
        self.s0_idx = s0_idx

        # simplify notations
        n = self.n
        Q = self.Q
        y, ys = self.y, self.ys

        # row of V corresponding to s0
        Vs0 = self.V[-1, s0_idx, :]
        α = Vs0 @ self.ys / (Vs0 @ self.y)

        self.α = α

        return α

    def continuation_wealths(self):
        "Given α, compute the continuation wealths ψ"

        diff = np.empty((n, K))
        for k in range(K):
            diff[:, k] = self.α[k] * self.y - self.ys[:, k]

        ψ = self.V @ diff
        self.ψ = ψ

        return ψ

    def price_risk_free_bond(self):
        "Give Q, compute price of one-period risk free bond"

        PRF = np.sum(self.Q, 0)
        self.PRF = PRF

        return PRF

    def risk_free_rate(self):
        "Given Q, compute one-period gross risk-free interest rate R"

        R = np.sum(self.Q, 0)
        R = np.reciprocal(R)
        self.R = R

        return R

    def value_functionss(self):
        "Given α, compute the optimal value functions J in equilibrium"

        n, T = self.n, self.T
        β = self.β
```

(continues on next page)

(continued from previous page)

```
        P = self.P

        # compute (I - βP)^(-1) in infinite case
        if T is None:
            P_seq = np.empty((1, n, n))
            P_seq[0] = np.linalg.inv(np.eye(n) - β * P)
        # and (I + βP + ... + β^T P^T) in finite case
        else:
            P_seq = np.empty((T+1, n, n))
            P_seq[0] = np.eye(n)

            Pt = np.eye(n)
            for t in range(1, T+1):
                Pt = Pt.dot(P)
                P_seq[t] = P_seq[t-1] + Pt * β ** t

        # compute the matrix [u(α_1 y), ..., u(α_K, y)]
        flow = np.empty((n, K))
        for k in range(K):
            flow[:, k] = self.u(self.α[k] * self.y)

        J = P_seq @ flow

        self.J = J

        return J
```

7.7.1 Example 1

Please read the preceding class for default parameter values and the following Python code for the fundamentals of the economy.

Here goes.

```
# dimensions
K, n = 2, 2

# states
s = np.array([0, 1])

# transition
P = np.array([[.5, .5], [.5, .5]])

# endowments
ys = np.empty((n, K))
ys[:, 0] = 1 - s        # y1
ys[:, 1] = s            # y2
```

```
ex1 = RecurCompetitive(s, P, ys)
```

```
# endowments
ex1.ys
```

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

```
# pricing kernal
ex1.Q
```

```
array([[0.49, 0.49],
       [0.49, 0.49]])
```

```
# Risk free rate R
ex1.R
```

```
array([1.02040816, 1.02040816])
```

```
# natural debt limit, A = [A1, A2, ..., AI]
ex1.A
```

```
array([[25.5, 24.5],
       [24.5, 25.5]])
```

```
# when the initial state is state 1
print(f'α = {ex1.wealth_distribution(s0_idx=0)}')
print(f'ψ = \n{ex1.continuation_wealths()}')
print(f'J = \n{ex1.value_functionss()}')
```

```
α = [0.51 0.49]
ψ =
[[-0.  0.]
  [ 1. -1.]]]
J =
[[[71.41428429 70.        ]
  [71.41428429 70.        ]]]
```

```
# when the initial state is state 2
print(f'α = {ex1.wealth_distribution(s0_idx=1)}')
print(f'ψ = \n{ex1.continuation_wealths()}')
print(f'J = \n{ex1.value_functionss()}')
```

```
α = [0.49 0.51]
ψ =
[[-1.  1.]
  [ 0. -0.]]]
J =
[[[70.         71.41428429]
  [70.         71.41428429]]]
```

7.7.2 Example 2

```
# dimensions
K, n = 2, 2

# states
s = np.array([1, 2])

# transition
P = np.array([[.5, .5], [.5, .5]])

# endowments
ys = np.empty((n, K))
ys[:, 0] = 1.5          # y1
ys[:, 1] = s            # y2
```

```
ex2 = RecurCompetitive(s, P, ys)
```

```
# endowments

print("ys = \n", ex2.ys)

# pricing kernal
print ("Q = \n", ex2.Q)

# Risk free rate R
print("R = ", ex2.R)
```

```
ys =
 [[1.5 1. ]
 [1.5 2. ]]
Q =
 [[0.49       0.41412558]
 [0.57977582 0.49      ]]
R =  [0.93477529 1.10604104]
```

```
# pricing kernal
ex2.Q
```

```
array([[0.49      , 0.41412558],
       [0.57977582, 0.49      ]])
```

```
# Risk free rate R
ex2.R
```

```
array([0.93477529, 1.10604104])
```

```
# natural debt limit, A = [A1, A2, ..., AI]
ex2.A
```

```
array([[69.30941886, 66.91255848],
       [81.73318641, 79.98879094]])
```

```
# when the initial state is state 1
print(f'α = {ex2.wealth_distribution(s0_idx=0)}')
print(f'ψ = \n{ex2.continuation_wealths()}')
print(f'J = \n{ex2.value_functionss()}')
```

```
α = [0.50879763 0.49120237]
ψ =
[[[-0.          0.        ]
  [ 0.55057195 -0.55057195]]]
J =
[[[122.907875   120.76397493]
  [123.32114686 121.17003803]]]
```

```
# when the initial state is state 1
print(f'α = {ex2.wealth_distribution(s0_idx=1)}')
print(f'ψ = \n{ex2.continuation_wealths()}')
print(f'J = \n{ex2.value_functionss()}')
```

```
α = [0.50539319 0.49460681]
ψ =
[[[-0.46375886  0.46375886]
  [ 0.         -0.        ]]]
J =
[[[122.49598809 121.18174895]
  [122.907875   121.58921679]]]
```

7.7.3 Example 3

```
# dimensions
K, n = 2, 2

# states
s = np.array([1, 2])

# transition
λ = 0.9
P = np.array([[1-λ, λ], [0, 1]])

# endowments
ys = np.empty((n, K))
ys[:, 0] = [1, 0]          # y1
ys[:, 1] = [0, 1]          # y2
```

```
ex3 = RecurCompetitive(s, P, ys)
```

```
# endowments
```

(continues on next page)

(continued from previous page)

```
print("ys = ", ex3.ys)

# pricing kernel
print ("Q = ", ex3.Q)

# Risk free rate R
print("R = ", ex3.R)
```

```
ys =  [[1. 0.]
 [0. 1.]]
Q =  [[0.098 0.882]
 [0.    0.98 ]]
R =  [10.20408163  0.53705693]
```

```
# pricing kernel
ex3.Q
```

```
array([[0.098, 0.882],
       [0.   , 0.98 ]])
```

```
# natural debt limit, A = [A1, A2, ..., AI]
ex3.A
```

```
array([[ 1.10864745, 48.89135255],
       [ 0.        , 50.        ]])
```

Note that the natural debt limit for agent 1 in state 2 is 0.

```
# when the initial state is state 1
print(f'α = {ex3.wealth_distribution(s0_idx=0)}')
print(f'ψ = \n{ex3.continuation_wealths()}')
print(f'J = \n{ex3.value_functionss()}')
```

```
α = [0.02217295 0.97782705]
ψ =
[[[ 0.         -0.        ]
  [ 1.10864745 -1.10864745]]]
J =
[[[14.89058394 98.88513796]
  [14.89058394 98.88513796]]]
```

```
# when the initial state is state 1
print(f'α = {ex3.wealth_distribution(s0_idx=1)}')
print(f'ψ = \n{ex3.continuation_wealths()}')
print(f'J = \n{ex3.value_functionss()}')
```

```
α = [0. 1.]
ψ =
[[[-1.10864745  1.10864745]
  [ 0.          0.        ]]]
```

(continues on next page)

(continued from previous page)

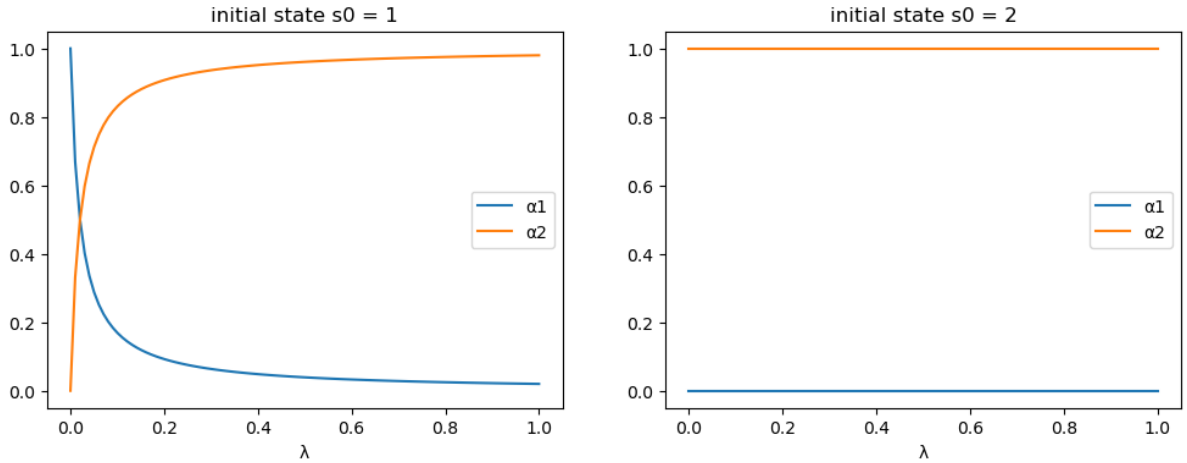
```
J =
[[[  0. 100.]
  [  0. 100.]]]
```

For the specification of the Markov chain in example 3, let's take a look at how the equilibrium allocation changes as a function of transition probability λ.

```
λ_seq = np.linspace(0, 1, 100)

# prepare containers
αs0_seq = np.empty((len(λ_seq), 2))
αs1_seq = np.empty((len(λ_seq), 2))

for i, λ in enumerate(λ_seq):
    P = np.array([[1-λ, λ], [0, 1]])
    ex3 = RecurCompetitive(s, P, ys)

    # initial state s0 = 1
    α = ex3.wealth_distribution(s0_idx=0)
    αs0_seq[i, :] = α

    # initial state s0 = 2
    α = ex3.wealth_distribution(s0_idx=1)
    αs1_seq[i, :] = α
```

```
/tmp/ipykernel_6453/2194301487.py:126: RuntimeWarning: divide by zero encountered␣
↪in reciprocal
  R = np.reciprocal(R)
```

```
fig, axs = plt.subplots(1, 2, figsize=(12, 4))

for i, αs_seq in enumerate([αs0_seq, αs1_seq]):
    for j in range(2):
        axs[i].plot(λ_seq, αs_seq[:, j], label=f'α{j+1}')
        axs[i].set_xlabel('λ')
        axs[i].set_title(f'initial state s0 = {s[i]}')
        axs[i].legend()

plt.show()
```



7.7.4 Example 4

```
# dimensions
K, n = 2, 3

# states
s = np.array([1, 2, 3])

# transition
λ = .9
μ = .9
δ = .05

P = np.array([[1-λ, λ, 0], [μ/2, μ, μ/2], [(1-δ)/2, (1-δ)/2, δ]])

# endowments
ys = np.empty((n, K))
ys[:, 0] = [.25, .75, .2]        # y1
ys[:, 1] = [1.25, .25, .2]       # y2
```

```
ex4 = RecurCompetitive(s, P, ys)
```

```
# endowments
print("ys = \n", ex4.ys)

# pricing kernal
print ("Q = \n", ex4.Q)

# Risk free rate R
print("R = ", ex4.R)

# natural debt limit, A = [A1, A2, ..., AI]
print("A = \n", ex4.A)

print('')

for i in range(1, 4):
```

(continues on next page)

(continued from previous page)

```
    # when the initial state is state i
    print(f"when the initial state is state {i}")
    print(f'α = {ex4.wealth_distribution(s0_idx=i-1)}')
    print(f'ψ = \n{ex4.continuation_wealths()}')
    print(f'J = \n{ex4.value_functionss()}\n')
```

```
ys =
 [[0.25 1.25]
 [0.75 0.25]
 [0.2  0.2 ]]
Q =
 [[0.098      1.08022498 0.        ]
 [0.36007499 0.882      0.69728222]
 [0.24038317 0.29440805 0.049     ]]
R =  [1.43172499 0.44313807 1.33997564]
A =
 [[-1.4141307  -0.45854174]
 [-1.4122483  -1.54005386]
 [-0.58434331 -0.3823659 ]]

when the initial state is state 1
α = [0.75514045 0.24485955]
ψ =
[[[ 0.          0.        ]
  [-0.81715447  0.81715447]
  [-0.14565791  0.14565791]]]
J =
[[[-2.65741909 -1.51322919]
  [-5.13103133 -2.92179221]
  [-2.65649938 -1.51270548]]]

when the initial state is state 2
α = [0.47835493 0.52164507]
ψ =
[[[ 0.5183286  -0.5183286 ]
  [ 0.         -0.        ]
  [ 0.12191319 -0.12191319]]]
J =
[[[-2.11505328 -2.20868477]
  [-4.08381377 -4.26460049]
  [-2.11432128 -2.20792037]]]

when the initial state is state 3
α = [0.60446648 0.39553352]
ψ =
[[[ 0.28216299 -0.28216299]
  [-0.37231938  0.37231938]
  [ 0.         -0.        ]]]
J =
[[[-2.37756442 -1.92325926]
  [-4.59067883 -3.71349163]
  [-2.37674158 -1.92259365]]]
```

7.8 Finite Horizon

The Python class **RecurCompetitive** provided above also can be used to compute competitive equilibrium allocations and Arrow securities prices for finite horizon economies.

The setting is a finite-horizon version of the one above except that time now runs for $T+1$ periods $t \in \mathbf{T} = \{0, 1, \dots, T\}$.

Consequently, we want $T+1$ counterparts to objects described above, with one important exception: we won't need **borrowing limits**.

- borrowing limits aren't required for a finite horizon economy in which a one-period utility function $u(c)$ satisfies an Inada condition that sets the marginal utility of consumption at zero consumption to zero.
- Nonnegativity of consumption choices at all $t \in \mathbf{T}$ automatically limits borrowing.

7.8.1 Continuation Wealths

We denote a $K \times 1$ vector of state-dependent continuation wealths in Markov state s at time t as

$$\psi_t(s) = \begin{bmatrix} \psi^1(s) \\ \vdots \\ \psi^K(s) \end{bmatrix}, \quad s \in [\bar{s}_1, \dots, \bar{s}_n]$$

and an $n \times 1$ vector of continuation wealths for each individual k as

$$\psi_t^k = \begin{bmatrix} \psi_t^k(\bar{s}_1) \\ \vdots \\ \psi_t^k(\bar{s}_n) \end{bmatrix}, \quad k \in [1, \dots, K]$$

Continuation wealths ψ^k of consumer k satisfy

$$\begin{aligned} \psi_T^k &= [\alpha_k y - y^k] \\ \psi_{T-1}^k &= [I + Q][\alpha_k y - y^k] \\ \vdots \quad & \quad \vdots \\ \psi_0^k &= [I + Q + Q^2 + \cdots + Q^T][\alpha_k y - y^k] \end{aligned} \tag{7.5}$$

where

$$y^k = \begin{bmatrix} y^k(\bar{s}_1) \\ \vdots \\ y^k(\bar{s}_n) \end{bmatrix}, \quad y = \begin{bmatrix} y(\bar{s}_1) \\ \vdots \\ y(\bar{s}_n) \end{bmatrix}$$

Note that $\sum_{k=1}^K \psi_t^k = 0_{n \times 1}$ for all $t \in \mathbf{T}$.

Remark: At the initial state $s_0 \in [\bar{s}_1, \dots, \bar{s}_n]$, for all agents $k = 1, \dots, K$, continuation wealth $\psi_0^k(s_0) = 0$. This indicates that the economy begins with all agents being debt-free and financial-asset-free at time 0, state s_0.

Remark: Note that all agents' continuation wealths return to zero when the Markov state returns to whatever value s_0 it had at time 0. This will recur if the Markov chain makes the initial state s_0 recurrent.

With the initial state being a particular state $s_0 \in [\bar{s}_1, \dots, \bar{s}_n]$, we must have

$$\psi_0^k(s_0) = 0, \quad k = 1, \dots, K$$

which means the equilibrium distribution of wealth satisfies

$$\alpha_k = \frac{V_z y^k}{V_z y} \tag{7.6}$$

where now in our finite-horizon economy

$$V = [I + Q + Q^2 + \cdots + Q^T] \tag{7.7}$$

and z is the row index corresponding to the initial state s_0.

Since $\sum_{k=1}^K V_z y^k = V_z y$, $\sum_{k=1}^K \alpha_k = 1$.

In summary, here is the logical flow of an algorithm to compute a competitive equilibrium with Arrow securities in our finite-horizon Markov economy:

- compute Q from the aggregate allocation and formula (7.1)
- compute the distribution of wealth α from formulas (7.6) and (7.7)
- using α, assign each consumer k the share α_k of the aggregate endowment at each state
- return to the α-dependent formula (7.5) for continuation wealths and compute continuation wealths
- equate agent k's portfolio to its continuation wealth state by state

While for the infinite horizon economy, the formula for value functions is

$$J^k = (I - \beta P)^{-1} u(\alpha_k y), \quad u(c) = \frac{c^{1-\gamma}}{1-\gamma}$$

for the finite horizon economy the formula is

$$J_0^k = (I + \beta P + \cdots + \beta^T P^T) u(\alpha_k y),$$

where it is understood that $u(\alpha_k y)$ is a vector.

7.8.2 Finite Horizon Example

Below we revisit the economy defined in example 1, but set the time horizon to be $T = 10$.

```
# dimensions
K, n = 2, 2

# states
s = np.array([0, 1])

# transition
P = np.array([[.5, .5], [.5, .5]])

# endowments
ys = np.empty((n, K))
ys[:, 0] = 1 - s        # y1
ys[:, 1] = s            # y2
```

```
ex1_finite = RecurCompetitive(s, P, ys, T=10)
```

```
# (I + Q + Q^2 + ... + Q^T)
ex1_finite.V[-1]
```

```
array([[5.48171623, 4.48171623],
       [4.48171623, 5.48171623]])
```

```
# endowments
ex1_finite.ys
```

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

```
# pricing kernal
ex1_finite.Q
```

```
array([[0.49, 0.49],
       [0.49, 0.49]])
```

```
# Risk free rate R
ex1_finite.R
```

```
array([1.02040816, 1.02040816])
```

In the finite time horizon case, ψ and J are returned as sequences.

Components are ordered from $t = T$ to $t = 0$.

```
# when the initial state is state 2
print(f'α = {ex1_finite.wealth_distribution(s0_idx=0)}')
print(f'ψ = \n{ex1_finite.continuation_wealths()}\n')
print(f'J = \n{ex1_finite.value_functionss()}')
```

```
α = [0.55018351 0.44981649]
ψ =
[[[-0.44981649  0.44981649]
  [ 0.55018351 -0.55018351]]

 [[-0.40063665  0.40063665]
  [ 0.59936335 -0.59936335]]

 [[-0.35244041  0.35244041]
  [ 0.64755959 -0.64755959]]

 [[-0.30520809  0.30520809]
  [ 0.69479191 -0.69479191]]

 [[-0.25892042  0.25892042]
  [ 0.74107958 -0.74107958]]

 [[-0.21355851  0.21355851]
  [ 0.78644149 -0.78644149]]

 [[-0.16910383  0.16910383]
  [ 0.83089617 -0.83089617]]

 [[-0.12553824  0.12553824]
  [ 0.87446176 -0.87446176]]
```

(continues on next page)

(continued from previous page)

```
 [[-0.08284397  0.08284397]
  [ 0.91715603 -0.91715603]]

 [[-0.04100358  0.04100358]
  [ 0.95899642 -0.95899642]]

 [[-0.          0.        ]
  [ 1.         -1.        ]]]

J =
[[[ 1.48348712  1.3413672 ]
  [ 1.48348712  1.3413672 ]]

 [[ 2.9373045   2.65590706]
  [ 2.9373045   2.65590706]]

 [[ 4.36204553  3.94415611]
  [ 4.36204553  3.94415611]]

 [[ 5.75829174  5.20664019]
  [ 5.75829174  5.20664019]]

 [[ 7.12661302  6.44387459]
  [ 7.12661302  6.44387459]]

 [[ 8.46756788  7.6563643 ]
  [ 8.46756788  7.6563643 ]]

 [[ 9.78170364  8.84460421]
  [ 9.78170364  8.84460421]]

 [[11.06955669 10.00907933]
  [11.06955669 10.00907933]]

 [[12.33165268 11.15026494]
  [12.33165268 11.15026494]]

 [[13.56850674 12.26862684]
  [13.56850674 12.26862684]]

 [[14.78062373 13.3646215 ]
  [14.78062373 13.3646215 ]]]
```

```
# when the initial state is state 2
print(f'α = {ex1_finite.wealth_distribution(s0_idx=1)}')
print(f'ψ = \n{ex1_finite.continuation_wealths()}\n')
print(f'J = \n{ex1_finite.value_functionss()}')
```

```
α = [0.44981649 0.55018351]
ψ =
[[[-0.55018351  0.55018351]
  [ 0.44981649 -0.44981649]]

 [[-0.59936335  0.59936335]
  [ 0.40063665 -0.40063665]]
```

(continues on next page)

(continued from previous page)

```
 [[-0.64755959  0.64755959]
  [ 0.35244041 -0.35244041]]

 [[-0.69479191  0.69479191]
  [ 0.30520809 -0.30520809]]

 [[-0.74107958  0.74107958]
  [ 0.25892042 -0.25892042]]

 [[-0.78644149  0.78644149]
  [ 0.21355851 -0.21355851]]

 [[-0.83089617  0.83089617]
  [ 0.16910383 -0.16910383]]

 [[-0.87446176  0.87446176]
  [ 0.12553824 -0.12553824]]

 [[-0.91715603  0.91715603]
  [ 0.08284397 -0.08284397]]

 [[-0.95899642  0.95899642]
  [ 0.04100358 -0.04100358]]

 [[-1.          1.        ]
  [ 0.         -0.        ]]]

J =
[[[ 1.3413672   1.48348712]
  [ 1.3413672   1.48348712]]

 [[ 2.65590706  2.9373045 ]
  [ 2.65590706  2.9373045 ]]

 [[ 3.94415611  4.36204553]
  [ 3.94415611  4.36204553]]

 [[ 5.20664019  5.75829174]
  [ 5.20664019  5.75829174]]

 [[ 6.44387459  7.12661302]
  [ 6.44387459  7.12661302]]

 [[ 7.6563643   8.46756788]
  [ 7.6563643   8.46756788]]

 [[ 8.84460421  9.78170364]
  [ 8.84460421  9.78170364]]

 [[10.00907933 11.06955669]
  [10.00907933 11.06955669]]

 [[11.15026494 12.33165268]
  [11.15026494 12.33165268]]
```

(continues on next page)

(continued from previous page)

```
 [[12.26862684 13.56850674]
  [12.26862684 13.56850674]]

 [[13.3646215  14.78062373]
  [13.3646215  14.78062373]]]
```

We can check the results with finite horizon converges to the ones with infinite horizon as $T \to \infty$.

```
ex1_large = RecurCompetitive(s, P, ys, T=10000)
ex1_large.wealth_distribution(s0_idx=1)
```

```
array([0.49, 0.51])
```

```
ex1.V, ex1_large.V[-1]
```

```
(array([[[25.5, 24.5],
         [24.5, 25.5]]]),
 array([[25.5, 24.5],
        [24.5, 25.5]]))
```

```
ex1_large.continuation_wealths()
ex1.ψ, ex1_large.ψ[-1]
```

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

```
ex1_large.value_functionss()
ex1.J, ex1_large.J[-1]
```

```
(array([[[70.        , 71.41428429],
         [70.        , 71.41428429]]]),
 array([[70.        , 71.41428429],
        [70.        , 71.41428429]]))
```

CHAPTER

EIGHT

HETEROGENEOUS BELIEFS AND BUBBLES

Contents

- *Heterogeneous Beliefs and Bubbles*
  - *Overview*
  - *Structure of the Model*
  - *Solving the Model*
  - *Exercises*

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

```
!pip install quantecon
```

8.1 Overview

This lecture describes a version of a model of Harrison and Kreps [HK78].

The model determines the price of a dividend-yielding asset that is traded by two types of self-interested investors.

The model features

- heterogeneous beliefs
- incomplete markets
- short sales constraints, and possibly …
- (leverage) limits on an investor's ability to borrow in order to finance purchases of a risky asset

Let's start with some standard imports:

```
import numpy as np
import quantecon as qe
import scipy.linalg as la
```

8.1.1 References

Prior to reading the following, you might like to review our lectures on

- Markov chains
- *Asset pricing with finite state space*

8.1.2 Bubbles

Economists differ in how they define a *bubble*.

The Harrison-Kreps model illustrates the following notion of a bubble that attracts many economists:

> *A component of an asset price can be interpreted as a bubble when all investors agree that the current price of the asset exceeds what they believe the asset's underlying dividend stream justifies.*

8.2 Structure of the Model

The model simplifies things by ignoring alterations in the distribution of wealth among investors who have hard-wired different beliefs about the fundamentals that determine asset payouts.

There is a fixed number A of shares of an asset.

Each share entitles its owner to a stream of dividends $\{d_t\}$ governed by a Markov chain defined on a state space $S \in \{0, 1\}$.

The dividend obeys

$$d_t = \begin{cases} 0 & \text{if } s_t = 0 \\ 1 & \text{if } s_t = 1 \end{cases}$$

An owner of a share at the end of time t and the beginning of time $t+1$ is entitled to the dividend paid at time $t+1$.

Thus, the stock is traded **ex dividend**.

An owner of a share at the beginning of time $t+1$ is also entitled to sell the share to another investor during time $t+1$.

Two types $h = a, b$ of investors differ only in their beliefs about a Markov transition matrix P with typical element

$$P(i,j) = \mathbb{P}\{s_{t+1} = j \mid s_t = i\}$$

Investors of type a believe the transition matrix

$$P_a = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{2}{3} & \frac{1}{3} \end{bmatrix}$$

Investors of type b think the transition matrix is

$$P_b = \begin{bmatrix} \frac{2}{3} & \frac{1}{3} \\ \frac{1}{4} & \frac{3}{4} \end{bmatrix}$$

Thus, in state 0, a type a investor is more optimistic about next period's dividend than is investor b.

But in state 1, a type a investor is more pessimistic about next period's dividend than is investor b.

The stationary (i.e., invariant) distributions of these two matrices can be calculated as follows:

```
qa = np.array([[1/2, 1/2], [2/3, 1/3]])
qb = np.array([[2/3, 1/3], [1/4, 3/4]])
mca = qe.MarkovChain(qa)
mcb = qe.MarkovChain(qb)
mca.stationary_distributions
```

```
array([[0.57142857, 0.42857143]])
```

```
mcb.stationary_distributions
```

```
array([[0.42857143, 0.57142857]])
```

The stationary distribution of P_a is approximately $\pi_a = \begin{bmatrix} .57 & .43 \end{bmatrix}$.

The stationary distribution of P_b is approximately $\pi_b = \begin{bmatrix} .43 & .57 \end{bmatrix}$.

Thus, a type a investor is more pessimistic on average.

8.2.1 Ownership Rights

An owner of the asset at the end of time t is entitled to the dividend at time $t+1$ and also has the right to sell the asset at time $t+1$.

Both types of investors are risk-neutral and both have the same fixed discount factor $\beta \in (0,1)$.

In our numerical example, we'll set $\beta = .75$, just as Harrison and Kreps [HK78] did.

We'll eventually study the consequences of two alternative assumptions about the number of shares A relative to the resources that our two types of investors can invest in the stock.

1. Both types of investors have enough resources (either wealth or the capacity to borrow) so that they can purchase the entire available stock of the asset[1].
2. No single type of investor has sufficient resources to purchase the entire stock.

Case 1 is the case studied in Harrison and Kreps.

In case 2, both types of investors always hold at least some of the asset.

8.2.2 Short Sales Prohibited

No short sales are allowed.

This matters because it limits how pessimists can express their opinions.

- They **can** express themselves by selling their shares.
- They **cannot** express themsevles more loudly by artificially "manufacturing shares" – that is, they cannot borrow shares from more optimistic investors and then immediately sell them.

[1] By assuming that both types of agents always have "deep enough pockets" to purchase all of the asset, the model takes wealth dynamics off the table. The Harrison-Kreps model generates high trading volume when the state changes either from 0 to 1 or from 1 to 0.

8.2.3 Optimism and Pessimism

The above specifications of the perceived transition matrices P_a and P_b, taken directly from Harrison and Kreps, build in stochastically alternating temporary optimism and pessimism.

Remember that state 1 is the high dividend state.

- In state 0, a type a agent is more optimistic about next period's dividend than a type b agent.
- In state 1, a type b agent is more optimistic about next period's dividend than a type a agaub is.

However, the stationary distributions $\pi_a = \begin{bmatrix} .57 & .43 \end{bmatrix}$ and $\pi_b = \begin{bmatrix} .43 & .57 \end{bmatrix}$ tell us that a type b person is more optimistic about the dividend process in the long run than is a type a person.

8.2.4 Information

Investors know a price function mapping the state s_t at t into the equilibrium price $p(s_t)$ that prevails in that state.

This price function is endogenous and to be determined below.

When investors choose whether to purchase or sell the asset at t, they also know s_t.

8.3 Solving the Model

Now let's turn to solving the model.

We'll determine equilibrium prices under a particular specification of beliefs and constraints on trading selected from one of the specifications described above.

We shall compare equilibrium price functions under the following alternative assumptions about beliefs:

1. There is only one type of agent, either a or b.
2. There are two types of agents differentiated only by their beliefs. Each type of agent has sufficient resources to purchase all of the asset (Harrison and Kreps's setting).
3. There are two types of agents with different beliefs, but because of limited wealth and/or limited leverage, both types of investors hold the asset each period.

8.3.1 Summary Table

The following table gives a summary of the findings obtained in the remainder of the lecture (in an exercise you will be asked to recreate the table and also reinterpret parts of it).

The table reports implications of Harrison and Kreps's specifications of P_a, P_b, β.

s_t	0	1
p_a	1.33	1.22
p_b	1.45	1.91
p_o	1.85	2.08
p_p	1	1
$\hat{p}_a$	1.85	1.69
$\hat{p}_b$	1.69	2.08

Here

- p_a is the equilibrium price function under homogeneous beliefs P_a
- p_b is the equilibrium price function under homogeneous beliefs P_b
- p_o is the equilibrium price function under heterogeneous beliefs with optimistic marginal investors
- p_p is the equilibrium price function under heterogeneous beliefs with pessimistic marginal investors
- $\hat{p}_a$ is the amount type a investors are willing to pay for the asset
- $\hat{p}_b$ is the amount type b investors are willing to pay for the asset

We'll explain these values and how they are calculated one row at a time.

The row corresponding to p_o applies when both types of investor have enough resources to purchse the entire stock of the asset and strict short sales constraints prevail so that temporarily optimistic investors always price the asset.

The row corresponding to p_p would apply if neither type of investor has enough resources to purchase the entire stock of the asset and both types must hold the asset.

The row corresponding to p_p would also apply if both types have enough resources to buy the entire stock of the asset but short sales are also possible so that temporarily pessimistic investors price the asset.

8.3.2 Single Belief Prices

We'll start by pricing the asset under homogeneous beliefs.

(This is the case treated in *the lecture* on asset pricing with finite Markov states)

Suppose that there is only one type of investor, either of type a or b, and that this investor always "prices the asset".

Let $p_h = \begin{bmatrix} p_h(0) \\ p_h(1) \end{bmatrix}$ be the equilibrium price vector when all investors are of type h.

The price today equals the expected discounted value of tomorrow's dividend and tomorrow's price of the asset:

$$p_h(s) = \beta \left(P_h(s,0)(0 + p_h(0)) + P_h(s,1)(1 + p_h(1)) \right), \quad s = 0, 1 \tag{8.1}$$

These equations imply that the equilibrium price vector is

$$\begin{bmatrix} p_h(0) \\ p_h(1) \end{bmatrix} = \beta [I - \beta P_h]^{-1} P_h \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{8.2}$$

The first two rows of the table report $p_a(s)$ and $p_b(s)$.

Here's a function that can be used to compute these values

```
def price_single_beliefs(transition, dividend_payoff, β=.75):
    """
    Function to Solve Single Beliefs
    """
    # First compute inverse piece
    imbq_inv = la.inv(np.eye(transition.shape[0]) - β * transition)

    # Next compute prices
    prices = β * imbq_inv @ transition @ dividend_payoff

    return prices
```

Single Belief Prices as Benchmarks

These equilibrium prices under homogeneous beliefs are important benchmarks for the subsequent analysis.

- $p_h(s)$ tells what a type h investor thinks is the "fundamental value" of the asset.
- Here "fundamental value" means the expected discounted present value of future dividends.

We will compare these fundamental values of the asset with equilibrium values when traders have different beliefs.

8.3.3 Pricing under Heterogeneous Beliefs

There are several cases to consider.

The first is when both types of agents have sufficient wealth to purchase all of the asset themselves.

In this case, the marginal investor who prices the asset is the more optimistic type so that the equilibrium price $\bar{p}$ satisfies Harrison and Kreps's key equation:

$$\bar{p}(s) = \beta \max \left\{ P_a(s,0)\bar{p}(0) + P_a(s,1)(1+\bar{p}(1)),\ P_b(s,0)\bar{p}(0) + P_b(s,1)(1+\bar{p}(1)) \right\} \tag{8.3}$$

for $s = 0, 1$.

In the above equation, the max on the right side is over the two prospective values of next period's payout from owning the asset.

The marginal investor who prices the asset in state s is of type a if

$$P_a(s,0)\bar{p}(0) + P_a(s,1)(1+\bar{p}(1)) > P_b(s,0)\bar{p}(0) + P_b(s,1)(1+\bar{p}(1))$$

The marginal investor is of type b if

$$P_a(s,1)\bar{p}(0) + P_a(s,1)(1+\bar{p}(1)) < P_b(s,1)\bar{p}(0) + P_b(s,1)(1+\bar{p}(1))$$

Thus the marginal investor is the (temporarily) optimistic type.

Equation (8.3) is a functional equation that, like a Bellman equation, can be solved by

- starting with a guess for the price vector $\bar{p}$ and
- iterating to convergence on the operator that maps a guess $\bar{p}^j$ into an updated guess $\bar{p}^{j+1}$ defined by the right side of (8.3), namely

$$\bar{p}^{j+1}(s) = \beta \max \left\{ P_a(s,0)\bar{p}^j(0) + P_a(s,1)(1+\bar{p}^j(1)),\ P_b(s,0)\bar{p}^j(0) + P_b(s,1)(1+\bar{p}^j(1)) \right\} \tag{8.4}$$

for $s = 0, 1$.

The third row of the table labeled p_o reports equilibrium prices that solve the functional equation when $\beta = .75$.

Here the type that is optimistic about s_{t+1} prices the asset in state s_t.

It is instructive to compare these prices with the equilibrium prices for the homogeneous belief economies that solve under beliefs P_a and P_b reported in the rows labeled p_a and p_b, respectively.

Equilibrium prices p_o in the heterogeneous beliefs economy evidently exceed what any prospective investor regards as the fundamental value of the asset in each possible state.

Nevertheless, the economy recurrently visits a state that makes each investor want to purchase the asset for more than he believes its future dividends are worth.

An investor is willing to pay more than what he believes is warranted by fundamental value of the prospective dividend stream because he expects to have the option later to sell the asset to another investor who will value the asset more highly than he will then.

- Investors of type a are willing to pay the following price for the asset

$$\hat{p}_a(s) = \begin{cases} \bar{p}(0) & \text{if } s_t = 0 \\ \beta(P_a(1,0)\bar{p}(0) + P_a(1,1)(1+\bar{p}(1))) & \text{if } s_t = 1 \end{cases}$$

- Investors of type b are willing to pay the following price for the asset

$$\hat{p}_b(s) = \begin{cases} \beta(P_b(0,0)\bar{p}(0) + P_b(0,1)(1+\bar{p}(1))) & \text{if } s_t = 0 \\ \bar{p}(1) & \text{if } s_t = 1 \end{cases}$$

Evidently, $\hat{p}_a(1) < \bar{p}(1)$ and $\hat{p}_b(0) < \bar{p}(0)$.

Investors of type a want to sell the asset in state 1 while investors of type b want to sell it in state 0.

- The asset changes hands whenever the state changes from 0 to 1 or from 1 to 0.
- The valuations $\hat{p}_a(s)$ and $\hat{p}_b(s)$ are displayed in the fourth and fifth rows of the table.
- Even pessimistic investors who don't buy the asset think that it is worth more than they think future dividends are worth.

Here's code to solve for $\bar{p}$, $\hat{p}_a$ and $\hat{p}_b$ using the iterative method described above

```
def price_optimistic_beliefs(transitions, dividend_payoff, β=.75,
                             max_iter=50000, tol=1e-16):
    """
    Function to Solve Optimistic Beliefs
    """
    # We will guess an initial price vector of [0, 0]
    p_new = np.array([[0], [0]])
    p_old = np.array([[10.], [10.]])

    # We know this is a contraction mapping, so we can iterate to conv
    for i in range(max_iter):
        p_old = p_new
        p_new = β * np.max([q @ p_old
                            + q @ dividend_payoff for q in transitions],
                            1)

        # If we succeed in converging, break out of for loop
        if np.max(np.sqrt((p_new - p_old)**2)) < tol:
            break

    ptwiddle = β * np.min([q @ p_old
                           + q @ dividend_payoff for q in transitions],
                           1)

    phat_a = np.array([p_new[0], ptwiddle[1]])
    phat_b = np.array([ptwiddle[0], p_new[1]])

    return p_new, phat_a, phat_b
```

8.3.4 Insufficient Funds

Outcomes differ when the more optimistic type of investor has insufficient wealth — or insufficient ability to borrow enough — to hold the entire stock of the asset.

In this case, the asset price must adjust to attract pessimistic investors.

Instead of equation (8.3), the equilibrium price satisfies

$$\check{p}(s) = \beta \min \{P_a(s,1)\check{p}(0) + P_a(s,1)(1+\check{p}(1)),\ P_b(s,1)\check{p}(0) + P_b(s,1)(1+\check{p}(1))\} \tag{8.5}$$

and the marginal investor who prices the asset is always the one that values it *less* highly than does the other type.

Now the marginal investor is always the (temporarily) pessimistic type.

Notice from the sixth row of that the pessimistic price p_o is lower than the homogeneous belief prices p_a and p_b in both states.

When pessimistic investors price the asset according to (8.5), optimistic investors think that the asset is underpriced.

If they could, optimistic investors would willingly borrow at a one-period risk-free gross interest rate β^{-1} to purchase more of the asset.

Implicit constraints on leverage prohibit them from doing so.

When optimistic investors price the asset as in equation (8.3), pessimistic investors think that the asset is overpriced and would like to sell the asset short.

Constraints on short sales prevent that.

Here's code to solve for $\check{p}$ using iteration

```
def price_pessimistic_beliefs(transitions, dividend_payoff, β=.75,
                              max_iter=50000, tol=1e-16):
    """
    Function to Solve Pessimistic Beliefs
    """
    # We will guess an initial price vector of [0, 0]
    p_new = np.array([[0], [0]])
    p_old = np.array([[10.], [10.]])

    # We know this is a contraction mapping, so we can iterate to conv
    for i in range(max_iter):
        p_old = p_new
        p_new = β * np.min([q @ p_old
                            + q @ dividend_payoff for q in transitions],
                           1)

        # If we succeed in converging, break out of for loop
        if np.max(np.sqrt((p_new - p_old)**2)) < tol:
            break

    return p_new
```

8.3.5 Further Interpretation

[Sch14] interprets the Harrison-Kreps model as a model of a bubble — a situation in which an asset price exceeds what every investor thinks is merited by his or her beliefs about the value of the asset's underlying dividend stream.

Scheinkman stresses these features of the Harrison-Kreps model:

- High volume occurs when the Harrison-Kreps pricing formula (8.3) prevails.
- Type a investors sell the entire stock of the asset to type b investors every time the state switches from $s_t = 0$ to $s_t = 1$.
- Type b investors sell the asset to type a investors every time the state switches from $s_t = 1$ to $s_t = 0$.

Scheinkman takes this as a strength of the model because he observes high volume during *famous bubbles*.

- If the *supply* of the asset is increased sufficiently either physically (more "houses" are built) or artificially (ways are invented to short sell "houses"), bubbles end when the asset supply has grown enough to outstrip optimistic investors' resources for purchasing the asset.
- If optimistic investors finance their purchases by borrowing, tightening leverage constraints can extinguish a bubble.

Scheinkman extracts insights about the effects of financial regulations on bubbles.

He emphasizes how limiting short sales and limiting leverage have opposite effects.

8.4 Exercises

Exercise 8.4.1

This exercise invites you to recreate the summary table using the functions we have built above.

s_t	0	1
p_a	1.33	1.22
p_b	1.45	1.91
p_o	1.85	2.08
p_p	1	1
$\hat{p}_a$	1.85	1.69
$\hat{p}_b$	1.69	2.08

You will want first to define the transition matrices and dividend payoff vector.

In addition, below we'll add an interpretation of the row corresponding to p_o by inventing two additional types of agents, one of whom is **permanently optimistic**, the other who is **permanently pessimistic**.

We construct subjective transition probability matrices for our permanently optimistic and permanently pessimistic investors as follows.

The permanently optimistic investors(i.e., the investor with the most optimistic beliefs in each state) believes the transition matrix

$$P_o = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{4} & \frac{3}{4} \end{bmatrix}$$

The permanently pessimistic investor believes the transition matrix

$$P_p = \begin{bmatrix} \frac{2}{3} & \frac{1}{3} \\ \frac{2}{3} & \frac{1}{3} \end{bmatrix}$$

We'll use these transition matrices when we present our solution of exercise 1 below.

Solution to Exercise 8.4.1

First, we will obtain equilibrium price vectors with homogeneous beliefs, including when all investors are optimistic or pessimistic.

```
qa = np.array([[1/2, 1/2], [2/3, 1/3]])     # Type a transition matrix
qb = np.array([[2/3, 1/3], [1/4, 3/4]])     # Type b transition matrix
# Optimistic investor transition matrix
qopt = np.array([[1/2, 1/2], [1/4, 3/4]])
# Pessimistic investor transition matrix
qpess = np.array([[2/3, 1/3], [2/3, 1/3]])

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

transitions = [qa, qb, qopt, qpess]
labels = ['p_a', 'p_b', 'p_optimistic', 'p_pessimistic']

for transition, label in zip(transitions, labels):
    print(label)
    print("=" * 20)
    s0, s1 = np.round(price_single_beliefs(transition, dividendreturn), 2)
    print(f"State 0: {s0}")
    print(f"State 1: {s1}")
    print("-" * 20)
```

```
p_a
====================
State 0: [1.33]
State 1: [1.22]
--------------------
p_b
====================
State 0: [1.45]
State 1: [1.91]
--------------------
p_optimistic
====================
State 0: [1.85]
State 1: [2.08]
--------------------
p_pessimistic
====================
State 0: [1.]
State 1: [1.]
--------------------
```

We will use the price_optimistic_beliefs function to find the price under heterogeneous beliefs.

```
opt_beliefs = price_optimistic_beliefs([qa, qb], dividendreturn)
labels = ['p_optimistic', 'p_hat_a', 'p_hat_b']

for p, label in zip(opt_beliefs, labels):
    print(label)
```

(continues on next page)

(continued from previous page)

```
    print("=" * 20)
    s0, s1 = np.round(p, 2)
    print(f"State 0: {s0}")
    print(f"State 1: {s1}")
    print("-" * 20)
```

```
p_optimistic
====================
State 0: [1.85]
State 1: [2.08]
--------------------
p_hat_a
====================
State 0: [1.85]
State 1: [1.69]
--------------------
p_hat_b
====================
State 0: [1.69]
State 1: [2.08]
--------------------
```

Notice that the equilibrium price with heterogeneous beliefs is equal to the price under single beliefs with **permanently optimistic** investors - this is due to the marginal investor in the heterogeneous beliefs equilibrium always being the type who is temporarily optimistic.

CHAPTER

NINE

ORTHOGONAL PROJECTIONS AND THEIR APPLICATIONS

Contents

- *Orthogonal Projections and Their Applications*
  - *Overview*
  - *Key Definitions*
  - *The Orthogonal Projection Theorem*
  - *Orthonormal Basis*
  - *Projection Via Matrix Algebra*
  - *Least Squares Regression*
  - *Orthogonalization and Decomposition*
  - *Exercises*

9.1 Overview

Orthogonal projection is a cornerstone of vector space methods, with many diverse applications.

These include

- Least squares projection, also known as linear regression
- Conditional expectations for multivariate normal (Gaussian) distributions
- Gram–Schmidt orthogonalization
- QR decomposition
- Orthogonal polynomials
- etc

In this lecture, we focus on

- key ideas
- least squares regression

We'll require the following imports:

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

9.1.1 Further Reading

For background and foundational concepts, see our lecture on linear algebra.

For more proofs and greater theoretical detail, see A Primer in Econometric Theory.

For a complete set of proofs in a general setting, see, for example, [Rom05].

For an advanced treatment of projection in the context of least squares prediction, see this book chapter.

9.2 Key Definitions

Assume $x, z \in \mathbb{R}^n$.

Define $\langle x, z \rangle = \sum_i x_i z_i$.

Recall $\|x\|^2 = \langle x, x \rangle$.

The **law of cosines** states that $\langle x, z \rangle = \|x\| \|z\| \cos(\theta)$ where θ is the angle between the vectors x and z.

When $\langle x, z \rangle = 0$, then $\cos(\theta) = 0$ and x and z are said to be **orthogonal** and we write $x \perp z$.

For a linear subspace $S \subset \mathbb{R}^n$, we call $x \in \mathbb{R}^n$ **orthogonal to** S if $x \perp z$ for all $z \in S$, and write $x \perp S$.

The **orthogonal complement** of linear subspace $S \subset \mathbb{R}^n$ is the set $S^\perp := \{x \in \mathbb{R}^n \ : \ x \perp S\}$.

$S^\perp$ is a linear subspace of $\mathbb{R}^n$

- To see this, fix $x, y \in S^\perp$ and $\alpha, \beta \in \mathbb{R}$.
- Observe that if $z \in S$, then

$$\langle \alpha x + \beta y, z \rangle = \alpha \langle x, z \rangle + \beta \langle y, z \rangle = \alpha \times 0 + \beta \times 0 = 0$$

- Hence $\alpha x + \beta y \in S^\perp$, as was to be shown

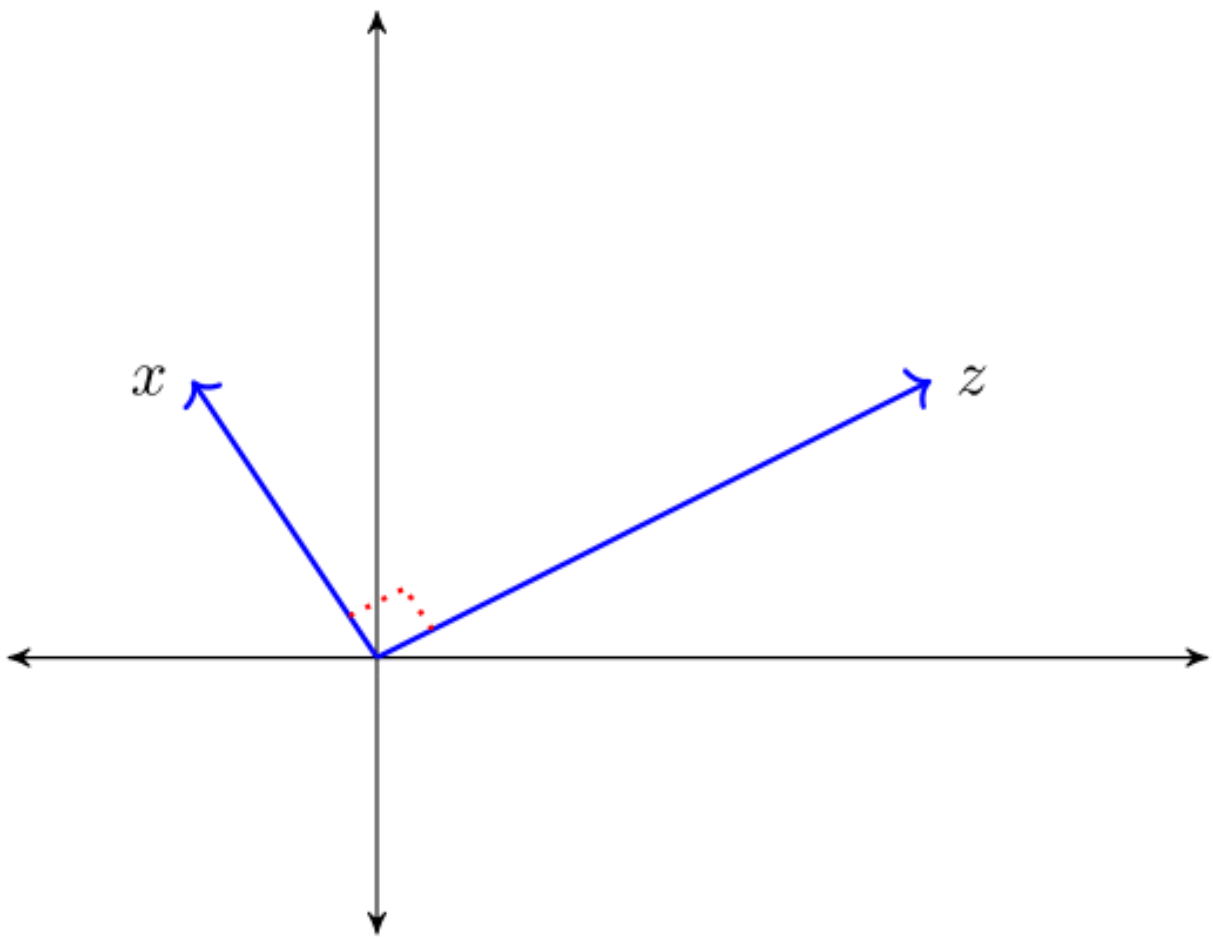
A set of vectors $\{x_1, \dots, x_k\} \subset \mathbb{R}^n$ is called an **orthogonal set** if $x_i \perp x_j$ whenever $i \neq j$.

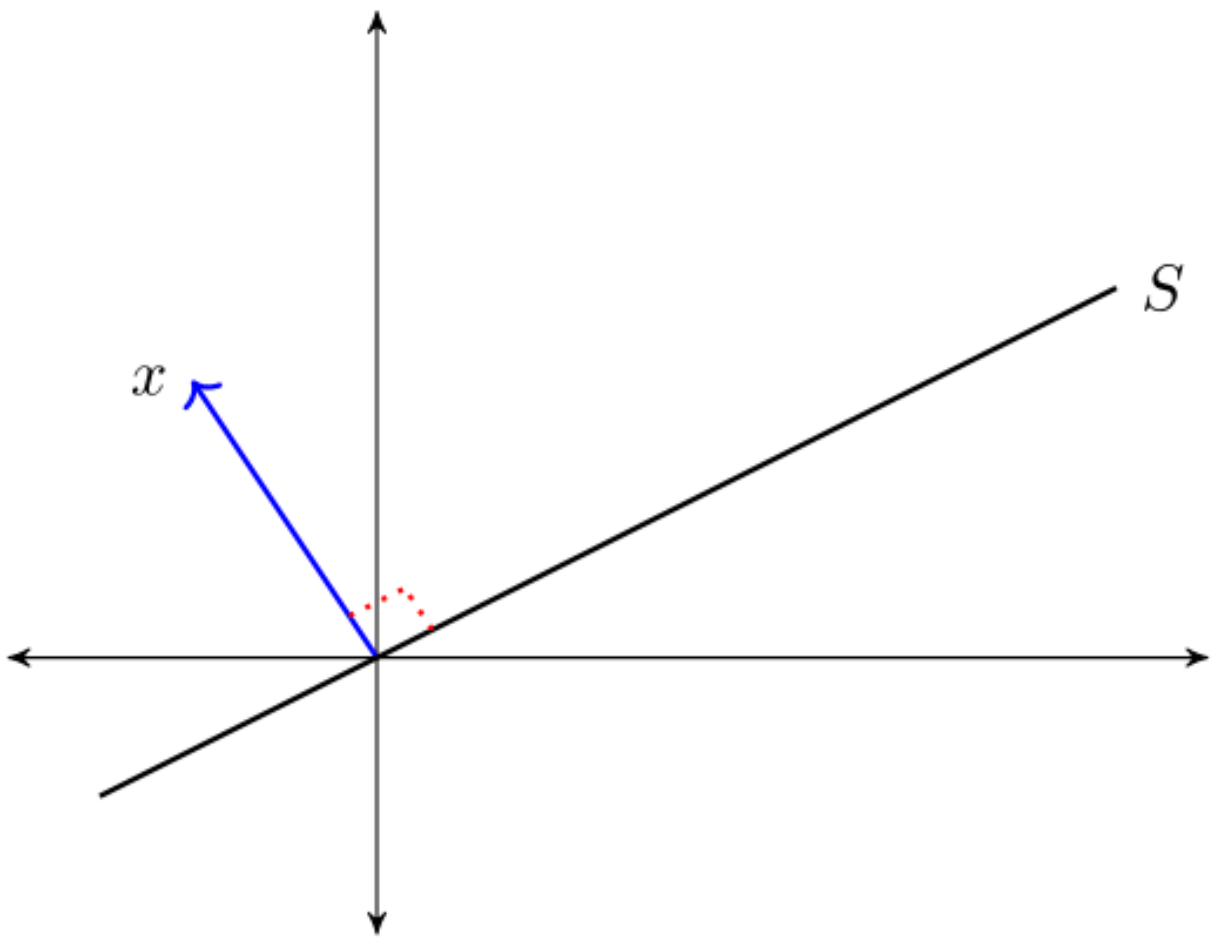
If $\{x_1, \dots, x_k\}$ is an orthogonal set, then the **Pythagorean Law** states that

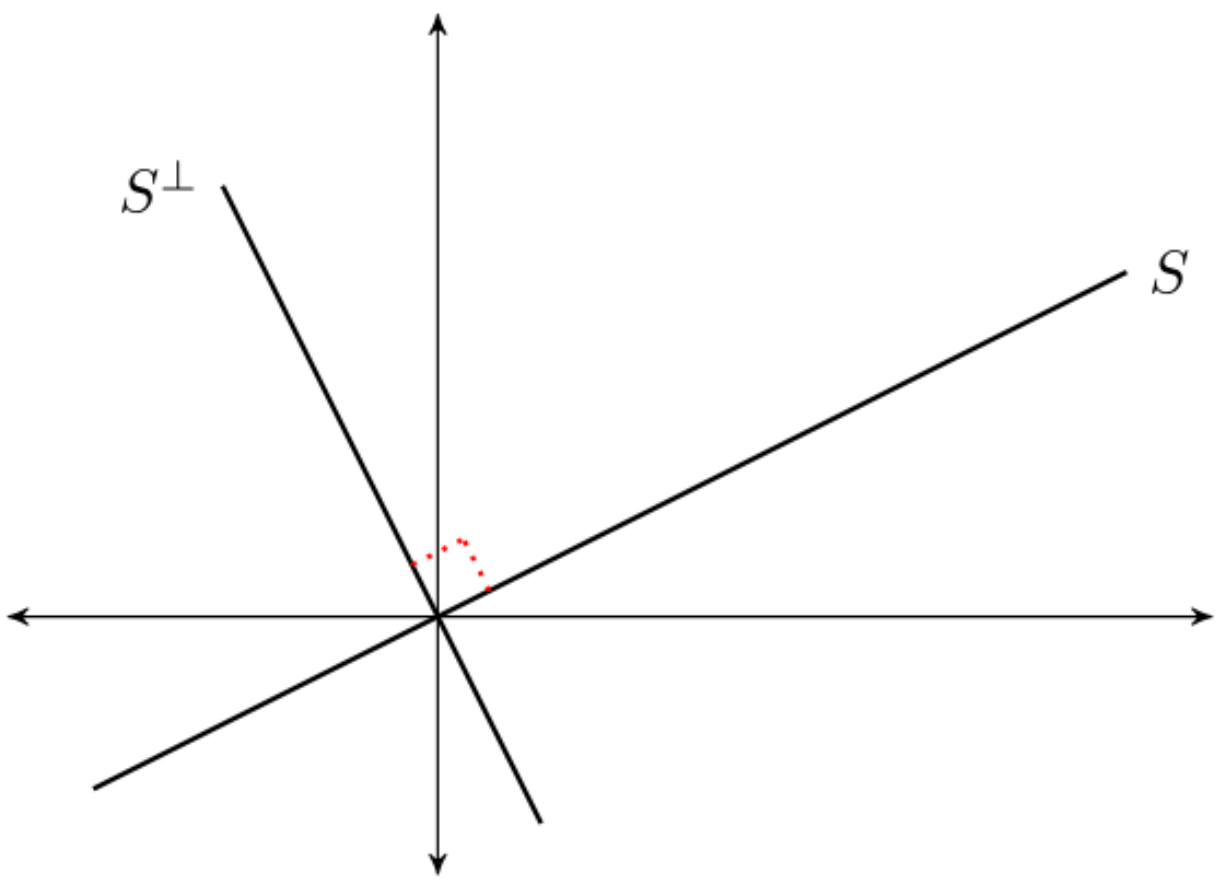
$$\|x_1 + \cdots + x_k\|^2 = \|x_1\|^2 + \cdots + \|x_k\|^2$$

For example, when $k = 2$, $x_1 \perp x_2$ implies

$$\|x_1 + x_2\|^2 = \langle x_1 + x_2, x_1 + x_2 \rangle = \langle x_1, x_1 \rangle + 2\langle x_2, x_1 \rangle + \langle x_2, x_2 \rangle = \|x_1\|^2 + \|x_2\|^2$$


x
z


S
x


$S^\perp$
S

9.2.1 Linear Independence vs Orthogonality

If $X \subset \mathbb{R}^n$ is an orthogonal set and $0 \notin X$, then X is linearly independent.

Proving this is a nice exercise.

While the converse is not true, a kind of partial converse holds, as we'll *see below*.

9.3 The Orthogonal Projection Theorem

What vector within a linear subspace of $\mathbb{R}^n$ best approximates a given vector in $\mathbb{R}^n$?

The next theorem answers this question.

Theorem (OPT) Given $y \in \mathbb{R}^n$ and linear subspace $S \subset \mathbb{R}^n$, there exists a unique solution to the minimization problem

$$\hat{y} := \underset{z \in S}{\arg\min} \, \|y - z\|$$

The minimizer $\hat{y}$ is the unique vector in $\mathbb{R}^n$ that satisfies

- $\hat{y} \in S$
- $y - \hat{y} \perp S$

The vector $\hat{y}$ is called the **orthogonal projection** of y onto S.

The next figure provides some intuition

9.3.1 Proof of Sufficiency

We'll omit the full proof.

But we will prove sufficiency of the asserted conditions.

To this end, let $y \in \mathbb{R}^n$ and let S be a linear subspace of $\mathbb{R}^n$.

Let $\hat{y}$ be a vector in $\mathbb{R}^n$ such that $\hat{y} \in S$ and $y - \hat{y} \perp S$.

Let z be any other point in S and use the fact that S is a linear subspace to deduce

$$\|y - z\|^2 = \|(y - \hat{y}) + (\hat{y} - z)\|^2 = \|y - \hat{y}\|^2 + \|\hat{y} - z\|^2$$

Hence $\|y - z\| \geq \|y - \hat{y}\|$, which completes the proof.

9.3.2 Orthogonal Projection as a Mapping

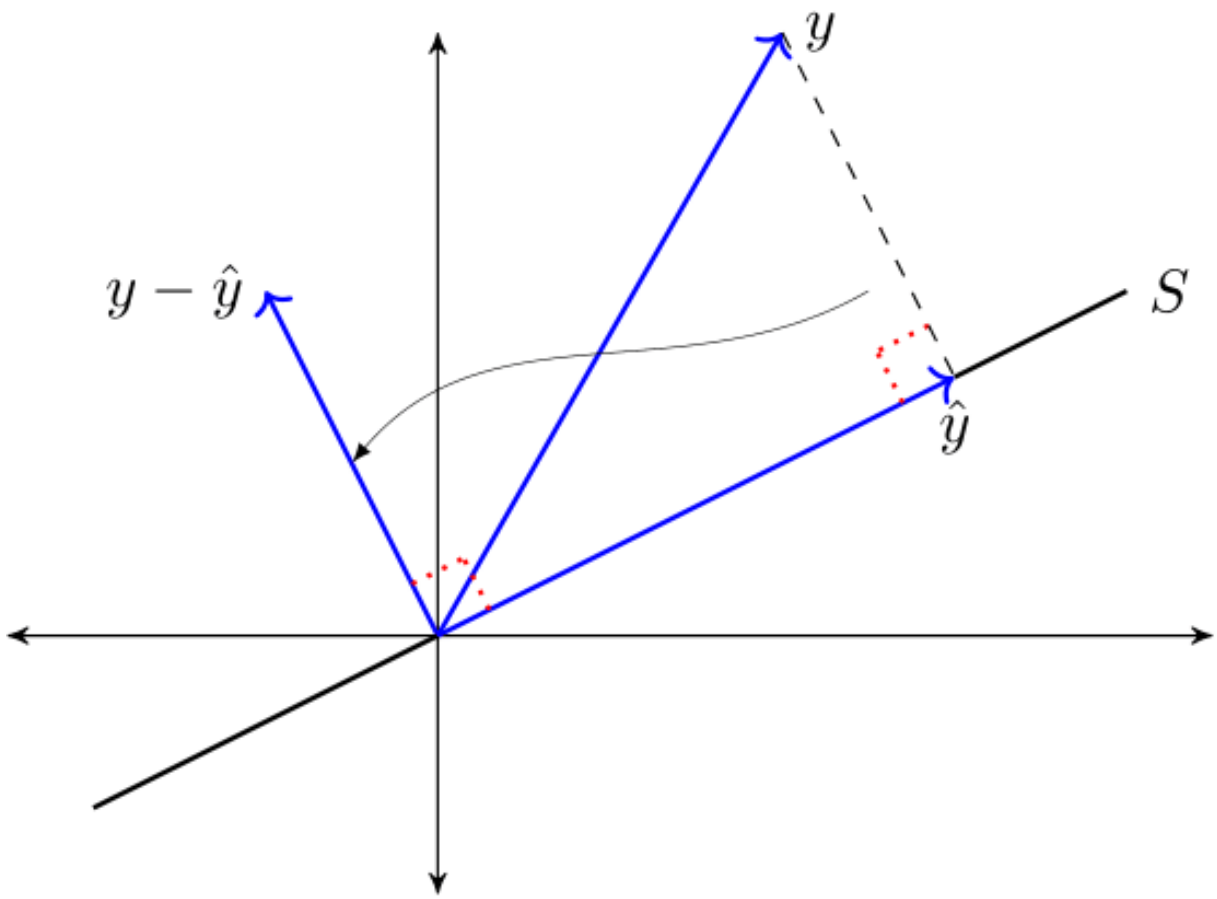
For a linear space Y and a fixed linear subspace S, we have a functional relationship

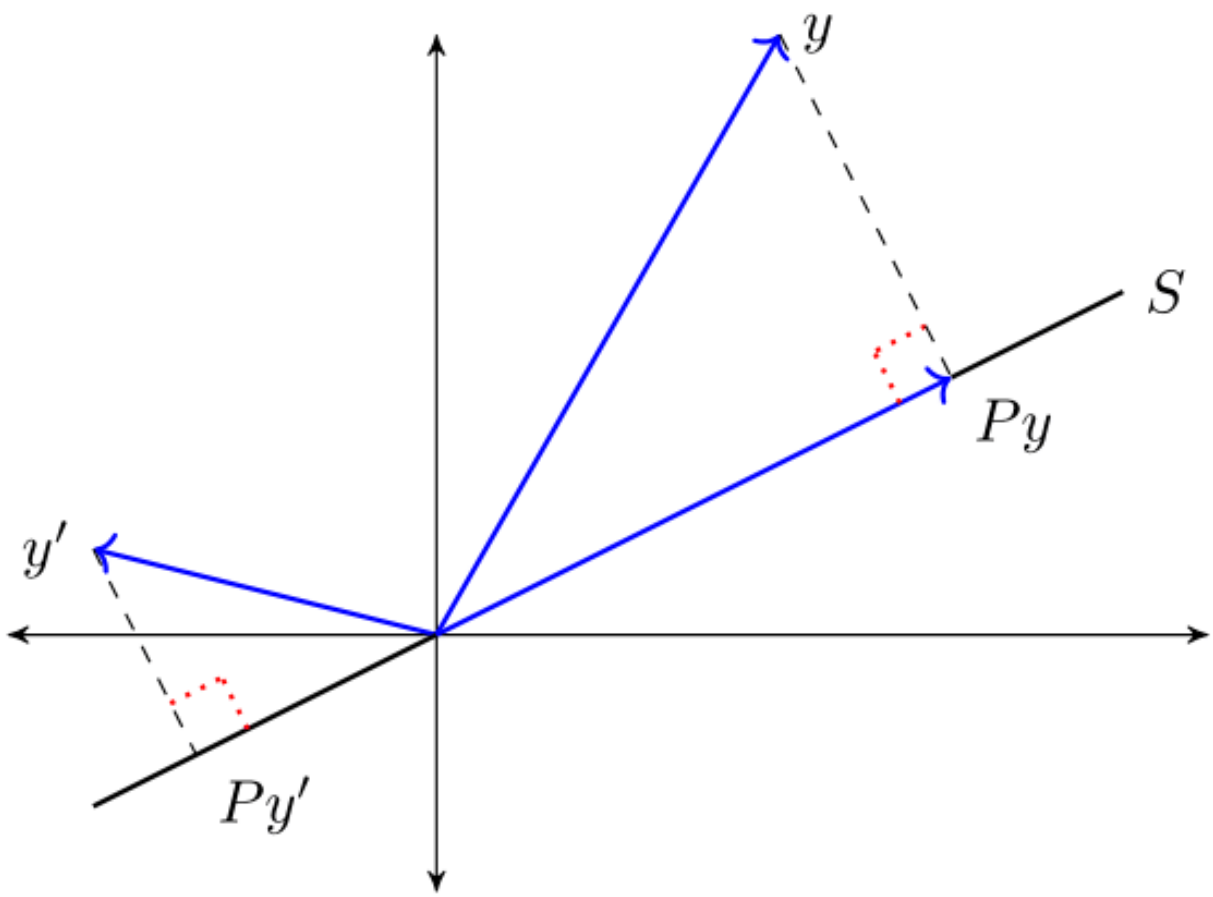
$$y \in Y \;\; \mapsto \;\; \text{its orthogonal projection } \hat{y} \in S$$

By the OPT, this is a well-defined mapping or *operator* from $\mathbb{R}^n$ to $\mathbb{R}^n$.

In what follows we denote this operator by a matrix P

- Py represents the projection $\hat{y}$.
- This is sometimes expressed as $\hat{E}_S y = Py$, where $\hat{E}$ denotes a **wide-sense expectations operator** and the subscript S indicates that we are projecting y onto the linear subspace S.


y
$y - \hat{y}$
S
$\hat{y}$


y
S
Py
y'
Py'

The operator P is called the **orthogonal projection mapping onto** S.

It is immediate from the OPT that for any $y \in \mathbb{R}^n$

1. $Py \in S$ and
2. $y - Py \perp S$

From this, we can deduce additional useful properties, such as

1. $\|y\|^2 = \|Py\|^2 + \|y - Py\|^2$ and
2. $\|Py\| \leq \|y\|$

For example, to prove 1, observe that $y = Py + y - Py$ and apply the Pythagorean law.

Orthogonal Complement

Let $S \subset \mathbb{R}^n$.

The **orthogonal complement** of S is the linear subspace $S^\perp$ that satisfies $x_1 \perp x_2$ for every $x_1 \in S$ and $x_2 \in S^\perp$.

Let Y be a linear space with linear subspace S and its orthogonal complement $S^\perp$.

We write

$$Y = S \oplus S^\perp$$

to indicate that for every $y \in Y$ there is unique $x_1 \in S$ and a unique $x_2 \in S^\perp$ such that $y = x_1 + x_2$.

Moreover, $x_1 = \hat{E}_S y$ and $x_2 = y - \hat{E}_S y$.

This amounts to another version of the OPT:

Theorem. If S is a linear subspace of $\mathbb{R}^n$, $\hat{E}_S y = Py$ and $\hat{E}_{S^\perp} y = My$, then

$$Py \perp My \quad \text{and} \quad y = Py + My \quad \text{for all } \ y \in \mathbb{R}^n$$

The next figure illustrates

9.4 Orthonormal Basis

An orthogonal set of vectors $O \subset \mathbb{R}^n$ is called an **orthonormal set** if $\|u\| = 1$ for all $u \in O$.

Let S be a linear subspace of $\mathbb{R}^n$ and let $O \subset S$.

If O is orthonormal and span $O = S$, then O is called an **orthonormal basis** of S.

O is necessarily a basis of S (being independent by orthogonality and the fact that no element is the zero vector).

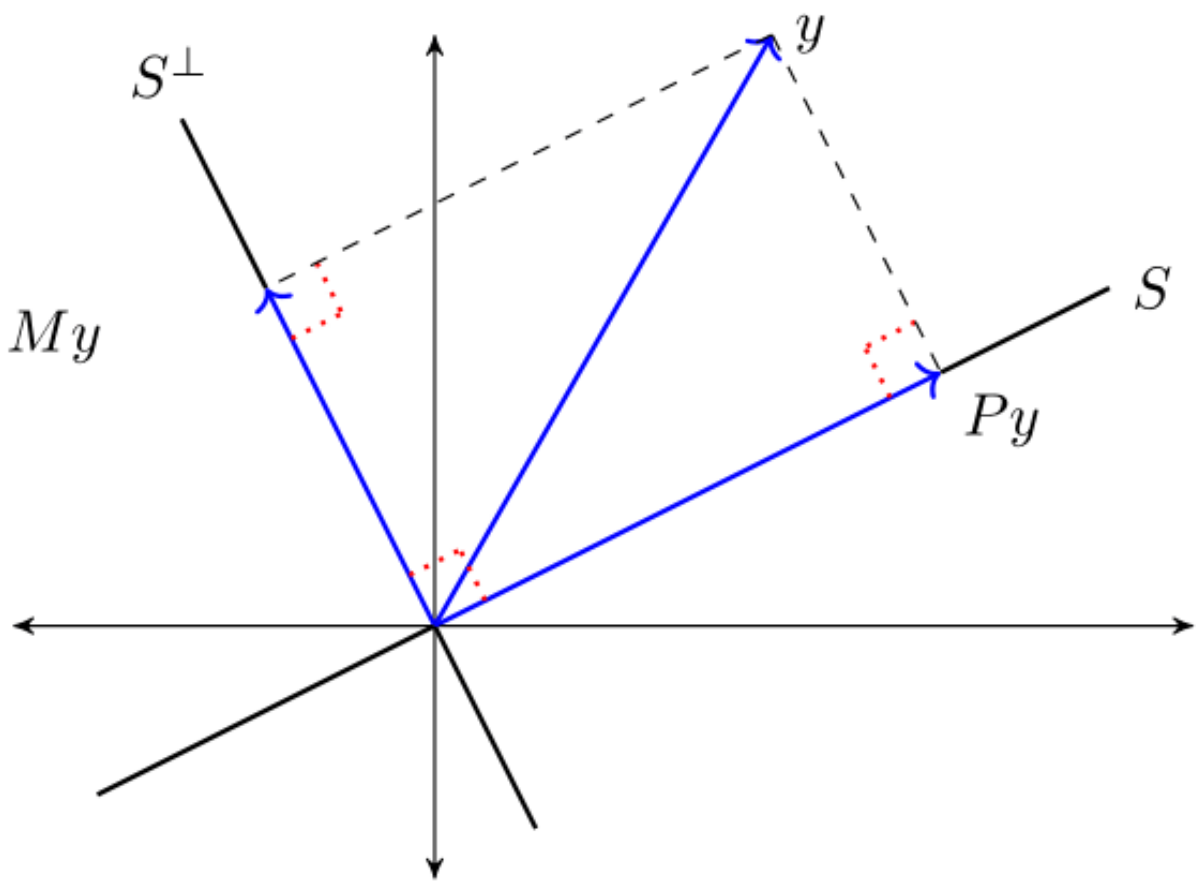
One example of an orthonormal set is the canonical basis $\{e_1, \dots, e_n\}$ that forms an orthonormal basis of $\mathbb{R}^n$, where e_i is the i th unit vector.

If $\{u_1, \dots, u_k\}$ is an orthonormal basis of linear subspace S, then

$$x = \sum_{i=1}^k \langle x, u_i \rangle u_i \quad \text{for all} \quad x \in S$$

To see this, observe that since $x \in \text{span}\{u_1, \dots, u_k\}$, we can find scalars $\alpha_1, \dots, \alpha_k$ that verify

$$x = \sum_{j=1}^k \alpha_j u_j \tag{9.1}$$


y
$S^\perp$
S
My
Py

Taking the inner product with respect to u_i gives

$$\langle x, u_i \rangle = \sum_{j=1}^{k} \alpha_j \langle u_j, u_i \rangle = \alpha_i$$

Combining this result with (9.1) verifies the claim.

9.4.1 Projection onto an Orthonormal Basis

When a subspace onto which we project is orthonormal, computing the projection simplifies:

Theorem If $\{u_1, \dots, u_k\}$ is an orthonormal basis for S, then

$$Py = \sum_{i=1}^{k} \langle y, u_i \rangle u_i, \quad \forall\, y \in \mathbb{R}^n \tag{9.2}$$

Proof: Fix $y \in \mathbb{R}^n$ and let Py be defined as in (9.2).

Clearly, $Py \in S$.

We claim that $y - Py \perp S$ also holds.

It sufficies to show that $y - Py \perp$ any basis vector u_i.

This is true because

$$\left\langle y - \sum_{i=1}^{k} \langle y, u_i \rangle u_i, u_j \right\rangle = \langle y, u_j \rangle - \sum_{i=1}^{k} \langle y, u_i \rangle \langle u_i, u_j \rangle = 0$$

(Why is this sufficient to establish the claim that $y - Py \perp S$?)

9.5 Projection Via Matrix Algebra

Let S be a linear subspace of $\mathbb{R}^n$ and let $y \in \mathbb{R}^n$.

We want to compute the matrix P that verifies

$$\hat{E}_S y = Py$$

Evidently Py is a linear function from $y \in \mathbb{R}^n$ to $Py \in \mathbb{R}^n$.

This reference is useful.

Theorem. Let the columns of $n \times k$ matrix X form a basis of S. Then

$$P = X(X'X)^{-1}X'$$

Proof: Given arbitrary $y \in \mathbb{R}^n$ and $P = X(X'X)^{-1}X'$, our claim is that

1. $Py \in S$, and
2. $y - Py \perp S$

Claim 1 is true because

$$Py = X(X'X)^{-1}X'y = Xa \quad \text{when} \quad a := (X'X)^{-1}X'y$$

An expression of the form Xa is precisely a linear combination of the columns of X and hence an element of S.

Claim 2 is equivalent to the statement

$$y - X(X'X)^{-1}X'y \perp Xb \quad \text{for all} \quad b \in \mathbb{R}^K$$

To verify this, notice that if $b \in \mathbb{R}^K$, then

$$(Xb)'[y - X(X'X)^{-1}X'y] = b'[X'y - X'y] = 0$$

The proof is now complete.

9.5.1 Starting with the Basis

It is common in applications to start with $n \times k$ matrix X with linearly independent columns and let

$$S := \operatorname{span} X := \operatorname{span}\{\operatorname{col}_1 X, \dots, \operatorname{col}_k X\}$$

Then the columns of X form a basis of S.

From the preceding theorem, $P = X(X'X)^{-1}X'y$ projects y onto S.

In this context, P is often called the **projection matrix**

- The matrix $M = I - P$ satisfies $My = \hat{E}_{S^\perp} y$ and is sometimes called the **annihilator matrix**.

9.5.2 The Orthonormal Case

Suppose that U is $n \times k$ with orthonormal columns.

Let $u_i := \operatorname{col} U_i$ for each i, let $S := \operatorname{span} U$ and let $y \in \mathbb{R}^n$.

We know that the projection of y onto S is

$$Py = U(U'U)^{-1}U'y$$

Since U has orthonormal columns, we have $U'U = I$.

Hence

$$Py = UU'y = \sum_{i=1}^{k} \langle u_i, y \rangle u_i$$

We have recovered our earlier result about projecting onto the span of an orthonormal basis.

9.5.3 Application: Overdetermined Systems of Equations

Let $y \in \mathbb{R}^n$ and let X be $n \times k$ with linearly independent columns.

Given X and y, we seek $b \in \mathbb{R}^k$ that satisfies the system of linear equations $Xb = y$.

If $n > k$ (more equations than unknowns), then b is said to be **overdetermined**.

Intuitively, we may not be able to find a b that satisfies all n equations.

The best approach here is to

- Accept that an exact solution may not exist.
- Look instead for an approximate solution.

By approximate solution, we mean a $b \in \mathbb{R}^k$ such that Xb is close to y.

The next theorem shows that a best approximation is well defined and unique.

The proof uses the OPT.

Theorem The unique minimizer of $\|y - Xb\|$ over $b \in \mathbb{R}^K$ is

$$\hat{\beta} := (X'X)^{-1}X'y$$

Proof: Note that

$$X\hat{\beta} = X(X'X)^{-1}X'y = Py$$

Since Py is the orthogonal projection onto $\text{span}(X)$ we have

$$\|y - Py\| \leq \|y - z\| \text{ for any } z \in \text{span}(X)$$

Because $Xb \in \text{span}(X)$

$$\|y - X\hat{\beta}\| \leq \|y - Xb\| \text{ for any } b \in \mathbb{R}^K$$

This is what we aimed to show.

9.6 Least Squares Regression

Let's apply the theory of orthogonal projection to least squares regression.

This approach provides insights about many geometric properties of linear regression.

We treat only some examples.

9.6.1 Squared Risk Measures

Given pairs $(x, y) \in \mathbb{R}^K \times \mathbb{R}$, consider choosing $f \colon \mathbb{R}^K \to \mathbb{R}$ to minimize the **risk**

$$R(f) := \mathbb{E}\left[(y - f(x))^2\right]$$

If probabilities and hence $\mathbb{E}$ are unknown, we cannot solve this problem directly.

However, if a sample is available, we can estimate the risk with the **empirical risk**:

$$\min_{f \in \mathcal{F}} \frac{1}{N} \sum_{n=1}^{N} (y_n - f(x_n))^2$$

Minimizing this expression is called **empirical risk minimization**.

The set $\mathcal{F}$ is sometimes called the hypothesis space.

The theory of statistical learning tells us that to prevent overfitting we should take the set $\mathcal{F}$ to be relatively simple.

If we let $\mathcal{F}$ be the class of linear functions, the problem is

$$\min_{b \in \mathbb{R}^K} \sum_{n=1}^{N} (y_n - b'x_n)^2$$

This is the sample **linear least squares problem**.

9.6.2 Solution

Define the matrices

$$y := \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{pmatrix}, \quad x_n := \begin{pmatrix} x_{n1} \\ x_{n2} \\ \vdots \\ x_{nK} \end{pmatrix} = n\text{-th obs on all regressors}$$

and

$$X := \begin{pmatrix} x_1' \\ x_2' \\ \vdots \\ x_N' \end{pmatrix} :=: \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1K} \\ x_{21} & x_{22} & \cdots & x_{2K} \\ \vdots & \vdots & & \vdots \\ x_{N1} & x_{N2} & \cdots & x_{NK} \end{pmatrix}$$

We assume throughout that $N > K$ and X is full column rank.

If you work through the algebra, you will be able to verify that $\|y - Xb\|^2 = \sum_{n=1}^N (y_n - b'x_n)^2$.

Since monotone transforms don't affect minimizers, we have

$$\operatorname*{arg\,min}_{b \in \mathbb{R}^K} \sum_{n=1}^N (y_n - b'x_n)^2 = \operatorname*{arg\,min}_{b \in \mathbb{R}^K} \|y - Xb\|$$

By our results about overdetermined linear systems of equations, the solution is

$$\hat{\beta} := (X'X)^{-1}X'y$$

Let P and M be the projection and annihilator associated with X:

$$P := X(X'X)^{-1}X' \quad \text{and} \quad M := I - P$$

The **vector of fitted values** is

$$\hat{y} := X\hat{\beta} = Py$$

The **vector of residuals** is

$$\hat{u} := y - \hat{y} = y - Py = My$$

Here are some more standard definitions:

- The **total sum of squares** is $:= \|y\|^2$.
- The **sum of squared residuals** is $:= \|\hat{u}\|^2$.
- The **explained sum of squares** is $:= \|\hat{y}\|^2$.

  TSS = ESS + SSR

We can prove this easily using the OPT.

From the OPT we have $y = \hat{y} + \hat{u}$ and $\hat{u} \perp \hat{y}$.

Applying the Pythagorean law completes the proof.

9.7 Orthogonalization and Decomposition

Let's return to the connection between linear independence and orthogonality touched on above.

A result of much interest is a famous algorithm for constructing orthonormal sets from linearly independent sets.

The next section gives details.

9.7.1 Gram-Schmidt Orthogonalization

Theorem For each linearly independent set $\{x_1, \dots, x_k\} \subset \mathbb{R}^n$, there exists an orthonormal set $\{u_1, \dots, u_k\}$ with

$$\text{span}\{x_1, \dots, x_i\} = \text{span}\{u_1, \dots, u_i\} \quad \text{for} \quad i = 1, \dots, k$$

The **Gram-Schmidt orthogonalization** procedure constructs an orthogonal set $\{u_1, u_2, \dots, u_n\}$.

One description of this procedure is as follows:

- For $i = 1, \dots, k$, form $S_i := \text{span}\{x_1, \dots, x_i\}$ and $S_i^\perp$
- Set $v_1 = x_1$
- For $i \geq 2$ set $v_i := \hat{E}_{S_{i-1}^\perp} x_i$ and $u_i := v_i / \|v_i\|$

The sequence $u_1, \dots, u_k$ has the stated properties.

A Gram-Schmidt orthogonalization construction is a key idea behind the Kalman filter described in A First Look at the Kalman filter.

In some exercises below, you are asked to implement this algorithm and test it using projection.

9.7.2 QR Decomposition

The following result uses the preceding algorithm to produce a useful decomposition.

Theorem If X is $n \times k$ with linearly independent columns, then there exists a factorization $X = QR$ where

- R is $k \times k$, upper triangular, and nonsingular
- Q is $n \times k$ with orthonormal columns

Proof sketch: Let

- $x_j := \text{col}_j(X)$
- $\{u_1, \dots, u_k\}$ be orthonormal with the same span as $\{x_1, \dots, x_k\}$ (to be constructed using Gram–Schmidt)
- Q be formed from cols u_i

Since $x_j \in \text{span}\{u_1, \dots, u_j\}$, we have

$$x_j = \sum_{i=1}^{j} \langle u_i, x_j \rangle u_i \quad \text{for } j = 1, \dots, k$$

Some rearranging gives $X = QR$.

9.7.3 Linear Regression via QR Decomposition

For matrices X and y that overdetermine β in the linear equation system $y = X\beta$, we found the least squares approximator $\hat{\beta} = (X'X)^{-1}X'y$.

Using the QR decomposition $X = QR$ gives

$$\begin{aligned}\hat{\beta} &= (R'Q'QR)^{-1}R'Q'y \\ &= (R'R)^{-1}R'Q'y \\ &= R^{-1}(R')^{-1}R'Q'y = R^{-1}Q'y\end{aligned}$$

Numerical routines would in this case use the alternative form $R\hat{\beta} = Q'y$ and back substitution.

9.8 Exercises

Exercise 9.8.1

Show that, for any linear subspace $S \subset \mathbb{R}^n$, $S \cap S^\perp = \{0\}$.

Solution to Exercise 9.8.1

If $x \in S$ and $x \in S^\perp$, then we have in particular that $\langle x, x\rangle = 0$, but then $x = 0$.

Exercise 9.8.2

Let $P = X(X'X)^{-1}X'$ and let $M = I - P$. Show that P and M are both idempotent and symmetric. Can you give any intuition as to why they should be idempotent?

Solution to Exercise 9.8.2

Symmetry and idempotence of M and P can be established using standard rules for matrix algebra. The intuition behind idempotence of M and P is that both are orthogonal projections. After a point is projected into a given subspace, applying the projection again makes no difference (A point inside the subspace is not shifted by orthogonal projection onto that space because it is already the closest point in the subspace to itself).

Exercise 9.8.3

Using Gram-Schmidt orthogonalization, produce a linear projection of y onto the column space of X and verify this using the projection matrix $P := X(X'X)^{-1}X'$ and also using QR decomposition, where:

$$y := \begin{pmatrix} 1 \\ 3 \\ -3 \end{pmatrix},$$

and

$$X := \begin{pmatrix} 1 & 0 \\ 0 & -6 \\ 2 & 2 \end{pmatrix}$$

Solution to Exercise 9.8.3

Here's a function that computes the orthonormal vectors using the GS algorithm given in the lecture

```
def gram_schmidt(X):
    """
    Implements Gram-Schmidt orthogonalization.

    Parameters
    ----------
    X : an n x k array with linearly independent columns

    Returns
    -------
    U : an n x k array with orthonormal columns

    """

    # Set up
    n, k = X.shape
    U = np.empty((n, k))
    I = np.eye(n)

    # The first col of U is just the normalized first col of X
    v1 = X[:,0]
    U[:, 0] = v1 / np.sqrt(np.sum(v1 * v1))

    for i in range(1, k):
        # Set up
        b = X[:, i]        # The vector we're going to project
        Z = X[:, 0:i]      # First i-1 columns of X

        # Project onto the orthogonal complement of the col span of Z
        M = I - Z @ np.linalg.inv(Z.T @ Z) @ Z.T
        u = M @ b

        # Normalize
        U[:, i] = u / np.sqrt(np.sum(u * u))

    return U
```

Here are the arrays we'll work with

```
y = [1, 3, -3]

X = [[1,  0],
     [0, -6],
     [2,  2]]

X, y = [np.asarray(z) for z in (X, y)]
```

First, let's try projection of y onto the column space of X using the ordinary matrix expression:

```
Py1 = X @ np.linalg.inv(X.T @ X) @ X.T @ y
Py1
```

```
array([-0.56521739,  3.26086957, -2.2173913 ])
```

Now let's do the same using an orthonormal basis created from our `gram_schmidt` function

```
U = gram_schmidt(X)
U
```

```
array([[ 0.4472136 , -0.13187609],
       [ 0.        , -0.98907071],
       [ 0.89442719,  0.06593805]])
```

```
Py2 = U @ U.T @ y
Py2
```

```
array([-0.56521739,  3.26086957, -2.2173913 ])
```

This is the same answer. So far so good. Finally, let's try the same thing but with the basis obtained via QR decomposition:

```
Q, R = qr(X, mode='economic')
Q
```

```
array([[-0.4472136 , -0.13187609],
       [-0.        , -0.98907071],
       [-0.89442719,  0.06593805]])
```

```
Py3 = Q @ Q.T @ y
Py3
```

```
array([-0.56521739,  3.26086957, -2.2173913 ])
```

Again, we obtain the same answer.

CHAPTER

TEN

ASSET PRICING II: THE LUCAS ASSET PRICING MODEL

Contents

- *Asset Pricing II: The Lucas Asset Pricing Model*
  - *Overview*
  - *The Lucas Model*
  - *Exercises*

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

```
!pip install interpolation
```

10.1 Overview

As stated in an earlier lecture, an asset is a claim on a stream of prospective payments.

What is the correct price to pay for such a claim?

The elegant asset pricing model of Lucas [Luc78] attempts to answer this question in an equilibrium setting with risk-averse agents.

While we mentioned some consequences of Lucas' model earlier, it is now time to work through the model more carefully and try to understand where the fundamental asset pricing equation comes from.

A side benefit of studying Lucas' model is that it provides a beautiful illustration of model building in general and equilibrium pricing in competitive models in particular.

Another difference to our first asset pricing lecture is that the state space and shock will be continuous rather than discrete.

Let's start with some imports:

```
import numpy as np
from interpolation import interp
from numba import njit, prange
from scipy.stats import lognorm
import matplotlib.pyplot as plt
%matplotlib inline
```

```
/home/runner/miniconda3/envs/quantecon/lib/python3.11/site-packages/numba/core/
↪decorators.py:262: NumbaDeprecationWarning: numba.generated_jit is deprecated.␣
↪Please see the documentation at: https://numba.readthedocs.io/en/stable/
↪reference/deprecation.html#deprecation-of-generated-jit for more information and␣
↪advice on a suitable replacement.
  warnings.warn(msg, NumbaDeprecationWarning)
```

10.2 The Lucas Model

Lucas studied a pure exchange economy with a representative consumer (or household), where

- *Pure exchange* means that all endowments are exogenous.
- *Representative* consumer means that either
  - there is a single consumer (sometimes also referred to as a household), or
  - all consumers have identical endowments and preferences

Either way, the assumption of a representative agent means that prices adjust to eradicate desires to trade.

This makes it very easy to compute competitive equilibrium prices.

10.2.1 Basic Setup

Let's review the setup.

Assets

There is a single "productive unit" that costlessly generates a sequence of consumption goods $\{y_t\}_{t=0}^{\infty}$.

Another way to view $\{y_t\}_{t=0}^{\infty}$ is as a *consumption endowment* for this economy.

We will assume that this endowment is Markovian, following the exogenous process

$$y_{t+1} = G(y_t, \xi_{t+1})$$

Here $\{\xi_t\}$ is an IID shock sequence with known distribution ϕ and $y_t \geq 0$.

An asset is a claim on all or part of this endowment stream.

The consumption goods $\{y_t\}_{t=0}^{\infty}$ are nonstorable, so holding assets is the only way to transfer wealth into the future.

For the purposes of intuition, it's common to think of the productive unit as a "tree" that produces fruit.

Based on this idea, a "Lucas tree" is a claim on the consumption endowment.

Consumers

A representative consumer ranks consumption streams $\{c_t\}$ according to the time separable utility functional

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t u(c_t) \tag{10.1}$$

Here

- $\beta \in (0, 1)$ is a fixed discount factor.
- u is a strictly increasing, strictly concave, continuously differentiable period utility function.
- $\mathbb{E}$ is a mathematical expectation.

10.2.2 Pricing a Lucas Tree

What is an appropriate price for a claim on the consumption endowment?

We'll price an *ex-dividend* claim, meaning that

- the seller retains this period's dividend
- the buyer pays p_t today to purchase a claim on
  - y_{t+1} and
  - the right to sell the claim tomorrow at price p_{t+1}

Since this is a competitive model, the first step is to pin down consumer behavior, taking prices as given.

Next, we'll impose equilibrium constraints and try to back out prices.

In the consumer problem, the consumer's control variable is the share π_t of the claim held in each period.

Thus, the consumer problem is to maximize (10.1) subject to

$$c_t + \pi_{t+1} p_t \leq \pi_t y_t + \pi_t p_t$$

along with $c_t \geq 0$ and $0 \leq \pi_t \leq 1$ at each t.

The decision to hold share π_t is actually made at time $t - 1$.

But this value is inherited as a state variable at time t, which explains the choice of subscript.

The Dynamic Program

We can write the consumer problem as a dynamic programming problem.

Our first observation is that prices depend on current information, and current information is really just the endowment process up until the current period.

In fact, the endowment process is Markovian, so that the only relevant information is the current state $y \in \mathbb{R}_+$ (dropping the time subscript).

This leads us to guess an equilibrium where price is a function p of y.

Remarks on the solution method

- Since this is a competitive (read: price taking) model, the consumer will take this function p as given.
- In this way, we determine consumer behavior given p and then use equilibrium conditions to recover p.

- This is the standard way to solve competitive equilibrium models.

Using the assumption that price is a given function p of y, we write the value function and constraint as

$$v(\pi, y) = \max_{c, \pi'} \left\{ u(c) + \beta \int v(\pi', G(y, z)) \phi(dz) \right\}$$

subject to

$$c + \pi' p(y) \leq \pi y + \pi p(y) \tag{10.2}$$

We can invoke the fact that utility is increasing to claim equality in (10.2) and hence eliminate the constraint, obtaining

$$v(\pi, y) = \max_{\pi'} \left\{ u[\pi(y + p(y)) - \pi' p(y)] + \beta \int v(\pi', G(y, z)) \phi(dz) \right\} \tag{10.3}$$

The solution to this dynamic programming problem is an optimal policy expressing either π' or c as a function of the state (π, y).

- Each one determines the other, since $c(\pi, y) = \pi(y + p(y)) - \pi'(\pi, y) p(y)$

Next Steps

What we need to do now is determine equilibrium prices.

It seems that to obtain these, we will have to

1. Solve this two-dimensional dynamic programming problem for the optimal policy.
2. Impose equilibrium constraints.
3. Solve out for the price function $p(y)$ directly.

However, as Lucas showed, there is a related but more straightforward way to do this.

Equilibrium Constraints

Since the consumption good is not storable, in equilibrium we must have $c_t = y_t$ for all t.

In addition, since there is one representative consumer (alternatively, since all consumers are identical), there should be no trade in equilibrium.

In particular, the representative consumer owns the whole tree in every period, so $\pi_t = 1$ for all t.

Prices must adjust to satisfy these two constraints.

The Equilibrium Price Function

Now observe that the first-order condition for (10.3) can be written as

$$u'(c) p(y) = \beta \int v_1'(\pi', G(y, z)) \phi(dz)$$

where v_1' is the derivative of v with respect to its first argument.

To obtain v_1' we can simply differentiate the right-hand side of (10.3) with respect to π, yielding

$$v_1'(\pi, y) = u'(c)(y + p(y))$$

Next, we impose the equilibrium constraints while combining the last two equations to get

$$p(y) = \beta \int \frac{u'[G(y,z)]}{u'(y)} [G(y,z) + p(G(y,z))] \phi(dz) \tag{10.4}$$

In sequential rather than functional notation, we can also write this as

$$p_t = \mathbb{E}_t \left[\beta \frac{u'(c_{t+1})}{u'(c_t)} (y_{t+1} + p_{t+1}) \right] \tag{10.5}$$

This is the famous consumption-based asset pricing equation.

Before discussing it further we want to solve out for prices.

10.2.3 Solving the Model

Equation (10.4) is a *functional equation* in the unknown function p.

The solution is an equilibrium price function p^*.

Let's look at how to obtain it.

Setting up the Problem

Instead of solving for it directly we'll follow Lucas' indirect approach, first setting

$$f(y) := u'(y) p(y) \tag{10.6}$$

so that (10.4) becomes

$$f(y) = h(y) + \beta \int f[G(y,z)] \phi(dz) \tag{10.7}$$

Here $h(y) := \beta \int u'[G(y,z)] G(y,z) \phi(dz)$ is a function that depends only on the primitives.

Equation (10.7) is a functional equation in f.

The plan is to solve out for f and convert back to p via (10.6).

To solve (10.7) we'll use a standard method: convert it to a fixed point problem.

First, we introduce the operator T mapping f into Tf as defined by

$$(Tf)(y) = h(y) + \beta \int f[G(y,z)] \phi(dz) \tag{10.8}$$

In what follows, we refer to T as the Lucas operator.

The reason we do this is that a solution to (10.7) now corresponds to a function f^* satisfying $(Tf^*)(y) = f^*(y)$ for all y.

In other words, a solution is a *fixed point* of T.

This means that we can use fixed point theory to obtain and compute the solution.

A Little Fixed Point Theory

Let $cb\mathbb{R}_+$ be the set of continuous bounded functions $f \colon \mathbb{R}_+ \to \mathbb{R}_+$.

We now show that

1. T has exactly one fixed point f^* in $cb\mathbb{R}_+$.
2. For any $f \in cb\mathbb{R}_+$, the sequence $T^k f$ converges uniformly to f^*.

Note: If you find the mathematics heavy going you can take 1–2 as given and skip to the *next section*

Recall the Banach contraction mapping theorem.

It tells us that the previous statements will be true if we can find an $\alpha < 1$ such that

$$\|Tf - Tg\| \leq \alpha \|f - g\|, \qquad \forall\, f, g \in cb\mathbb{R}_+ \tag{10.9}$$

Here $\|h\| := \sup_{x \in \mathbb{R}_+} |h(x)|$.

To see that (10.9) is valid, pick any $f, g \in cb\mathbb{R}_+$ and any $y \in \mathbb{R}_+$.

Observe that, since integrals get larger when absolute values are moved to the inside,

$$\begin{aligned}
|Tf(y) - Tg(y)| &= \left| \beta \int f[G(y,z)]\phi(dz) - \beta \int g[G(y,z)]\phi(dz) \right| \\
&\leq \beta \int |f[G(y,z)] - g[G(y,z)]|\, \phi(dz) \\
&\leq \beta \int \|f - g\| \phi(dz) \\
&= \beta \|f - g\|
\end{aligned}$$

Since the right-hand side is an upper bound, taking the sup over all y on the left-hand side gives (10.9) with $\alpha := \beta$.

10.2.4 Computation – An Example

The preceding discussion tells that we can compute f^* by picking any arbitrary $f \in cb\mathbb{R}_+$ and then iterating with T.

The equilibrium price function p^* can then be recovered by $p^*(y) = f^*(y)/u'(y)$.

Let's try this when $\ln y_{t+1} = \alpha \ln y_t + \sigma \epsilon_{t+1}$ where $\{\epsilon_t\}$ is IID and standard normal.

Utility will take the isoelastic form $u(c) = c^{1-\gamma}/(1-\gamma)$, where $\gamma > 0$ is the coefficient of relative risk aversion.

We will set up a `LucasTree` class to hold parameters of the model

```
class LucasTree:
    """
    Class to store parameters of the Lucas tree model.

    """

    def __init__(self,
                 γ=2,            # CRRA utility parameter
                 β=0.95,         # Discount factor
                 α=0.90,         # Correlation coefficient
                 σ=0.1,          # Volatility coefficient
```

(continues on next page)

(continued from previous page)

```
                 grid_size=100):

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

        # Set the grid interval to contain most of the mass of the
        # stationary distribution of the consumption endowment
        ssd = self.σ / np.sqrt(1 - self.α**2)
        grid_min, grid_max = np.exp(-4 * ssd), np.exp(4 * ssd)
        self.grid = np.linspace(grid_min, grid_max, grid_size)
        self.grid_size = grid_size

        # Set up distribution for shocks
        self.ϕ = lognorm(σ)
        self.draws = self.ϕ.rvs(500)

        self.h = np.empty(self.grid_size)
        for i, y in enumerate(self.grid):
            self.h[i] = β * np.mean((y**α * self.draws)**(1 - γ))
```

The following function takes an instance of the `LucasTree` and generates a jitted version of the Lucas operator

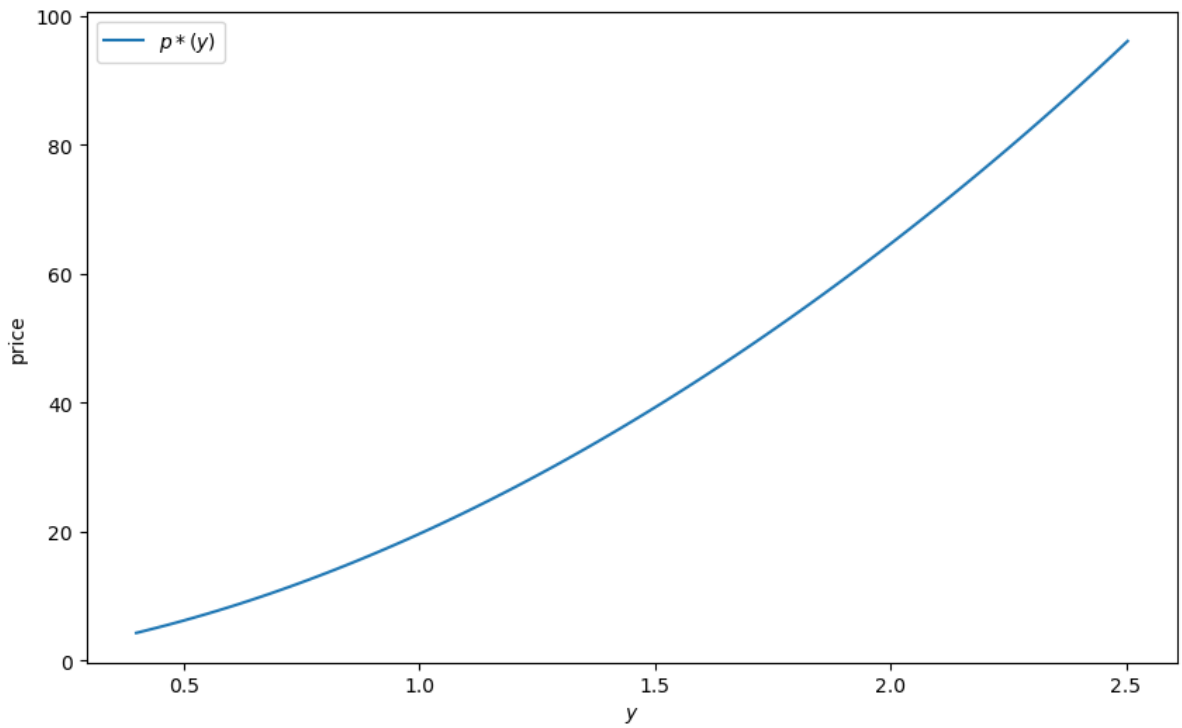
```
def operator_factory(tree, parallel_flag=True):

    """
    Returns approximate Lucas operator, which computes and returns the
    updated function Tf on the grid points.

    tree is an instance of the LucasTree class

    """

    grid, h = tree.grid, tree.h
    α, β = tree.α, tree.β
    z_vec = tree.draws

    @njit(parallel=parallel_flag)
    def T(f):
        """
        The Lucas operator
        """

        # Turn f into a function
        Af = lambda x: interp(grid, f, x)

        Tf = np.empty_like(f)
        # Apply the T operator to f using Monte Carlo integration
        for i in prange(len(grid)):
            y = grid[i]
            Tf[i] = h[i] + β * np.mean(Af(y**α * z_vec))

        return Tf

    return T
```

To solve the model, we write a function that iterates using the Lucas operator to find the fixed point.

```
def solve_model(tree, tol=1e-6, max_iter=500):
    """
    Compute the equilibrium price function associated with Lucas
    tree

    * tree is an instance of LucasTree

    """
    # Simplify notation
    grid, grid_size = tree.grid, tree.grid_size
    y = tree.y

    T = operator_factory(tree)

    i = 0
    f = np.ones_like(grid)  # Initial guess of f
    error = tol + 1
    while error > tol and i < max_iter:
        Tf = T(f)
        error = np.max(np.abs(Tf - f))
        f = Tf
        i += 1

    price = f * grid**y  # Back out price vector

    return price
```

Solving the model and plotting the resulting price function

```
tree = LucasTree()
price_vals = solve_model(tree)

fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(tree.grid, price_vals, label='$p*(y)$')
ax.set_xlabel('$y$')
ax.set_ylabel('price')
ax.legend()
plt.show()
```



We see that the price is increasing, even if we remove all serial correlation from the endowment process.

The reason is that a larger current endowment reduces current marginal utility.

The price must therefore rise to induce the household to consume the entire endowment (and hence satisfy the resource constraint).

What happens with a more patient consumer?

Here the orange line corresponds to the previous parameters and the green line is price when $\beta = 0.98$.

We see that when consumers are more patient the asset becomes more valuable, and the price of the Lucas tree shifts up.

Exercise 1 asks you to replicate this figure.

10.3 Exercises

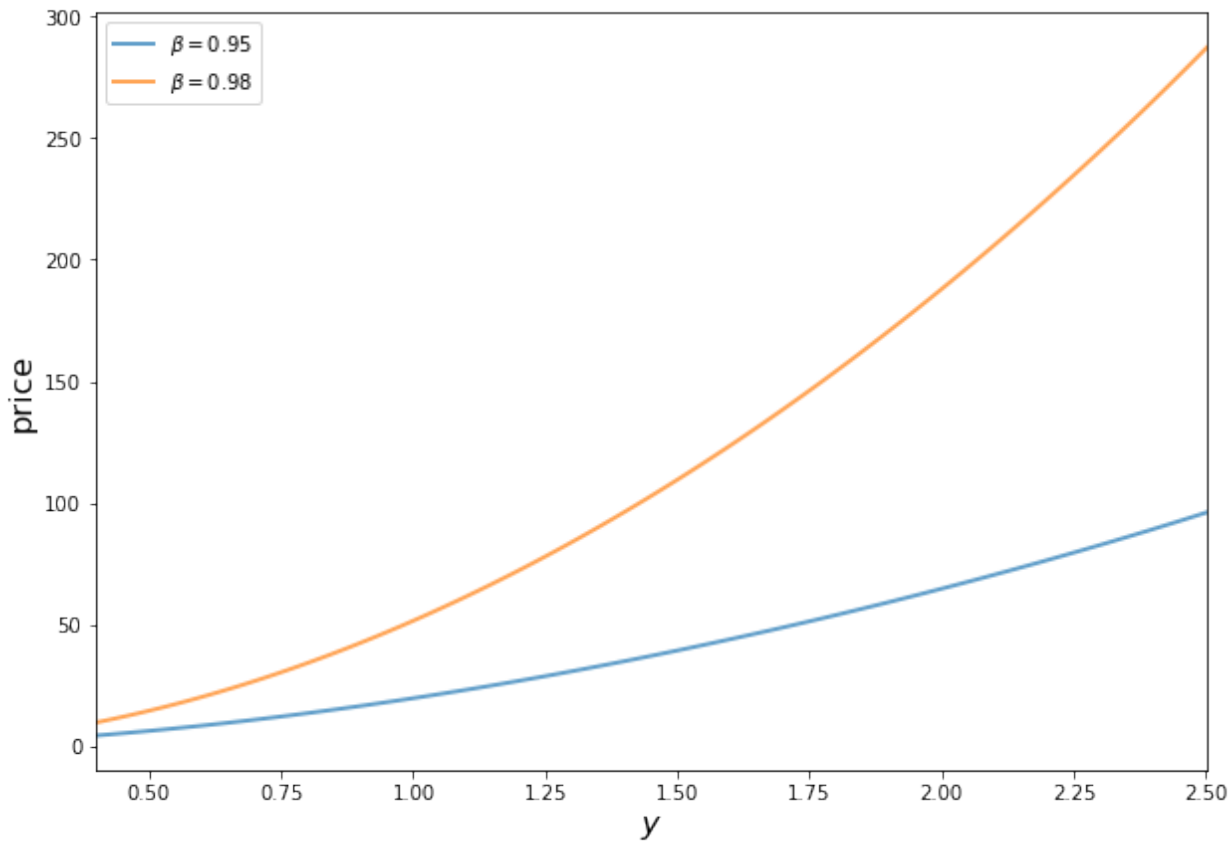
Exercise 10.3.1

Replicate *the figure* to show how discount factors affect prices.

Solution to Exercise 10.3.1

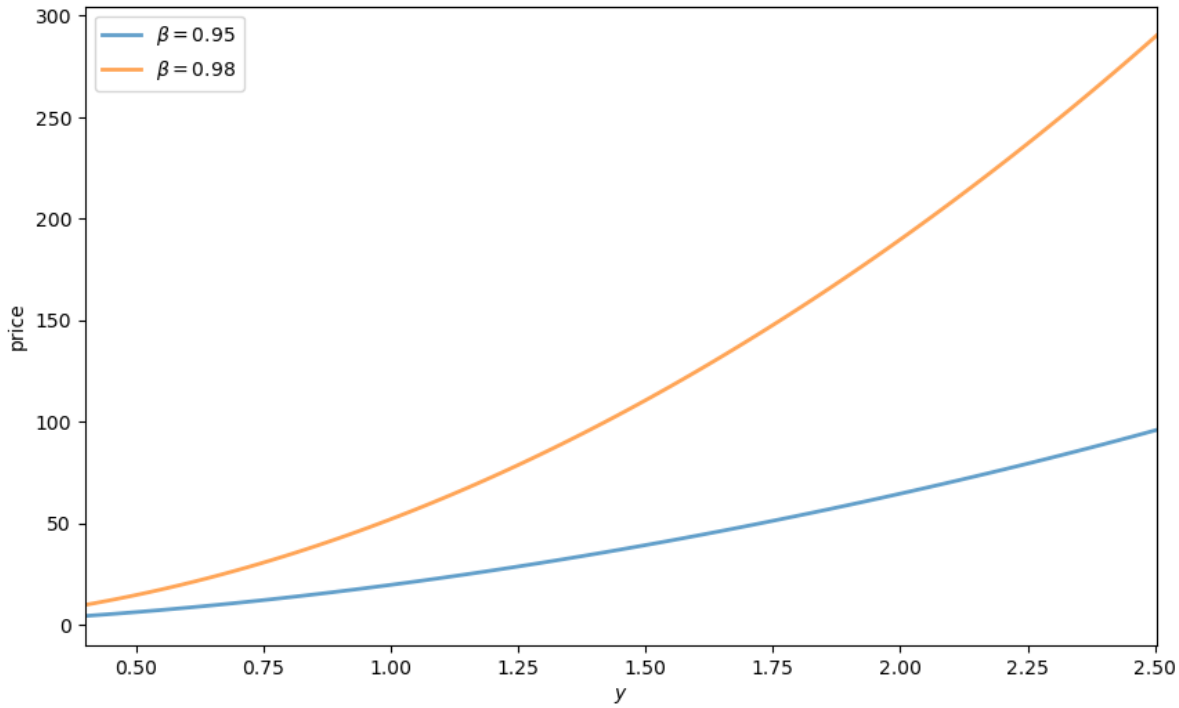
```
fig, ax = plt.subplots(figsize=(10, 6))

for β in (.95, 0.98):
    tree = LucasTree(β=β)
    grid = tree.grid
    price_vals = solve_model(tree)
```

(continues on next page)


$\beta = 0.95$
$\beta = 0.98$
price
y
300
250
200
150
100
50
0
0.50
0.75
1.00
1.25
1.50
1.75
2.00
2.25
2.50

(continued from previous page)

```
    label = rf'$\beta = {β}$'
    ax.plot(grid, price_vals, lw=2, alpha=0.7, label=label)

ax.legend(loc='upper left')
ax.set(xlabel='$y$', ylabel='price', xlim=(min(grid), max(grid)))
plt.show()
```



CHAPTER

ELEVEN

ELEMENTARY ASSET PRICING THEORY

Contents

- *Elementary Asset Pricing Theory*
  - *Overview*
  - *Key Equation*
  - *Implications of Key Equation*
  - *Expected Return - Beta Representation*
  - *Mean-Variance Frontier*
  - *Sharpe Ratios and the Price of Risk*
  - *Mathematical Structure of Frontier*
  - *Multi-factor Models*
  - *Empirical Implementations*
  - *Exercises*

11.1 Overview

This lecture is about some implications of asset-pricing theories that are based on the equation $EmR = 1$, where R is the gross return on an asset, m is a stochastic discount factor, and E is a mathematical expectation with respect to a joint probability distribution of R and m.

Instances of this equation occur in many models.

Note: Chapter 1 of [LS18] describes the role that this equation plays in a diverse set of models in macroeconomics, monetary economics, and public finance.

We aim to convey insights about empirical implications of this equation brought out in the work of Lars Peter Hansen [HR87] and Lars Peter Hansen and Ravi Jagannathan [HJ91].

By following their footsteps, from that single equation we'll derive

- a mean-variance frontier
- a single-factor model of excess returns

To do this, we use two ideas:

- the equation $EmR = 1$ that is implied by an application of a *law of one price*
- a Cauchy-Schwartz inequality

In particular, we'll apply a Cauchy-Schwartz inequality to a population linear least squares regression equation that is implied by $EmR = 1$.

We'll also describe how practitioners have implemented the model using

- cross sections of returns on many assets
- time series of returns on various assets

For background and basic concepts about linear least squares projections, see our lecture orthogonal projections and their applications.

As a sequel to the material here, please see our lecture two modifications of mean-variance portfolio theory.

11.2 Key Equation

We begin with a **key asset pricing equation**:

$$EmR^i = 1 \tag{11.1}$$

for $i = 1, \dots, I$ and where

$$\begin{aligned} m &= \text{ stochastic discount factor} \\ R^i &= \text{random gross return on asset } i \\ E &\sim \text{ mathematical expectation} \end{aligned}$$

The random gross return R^i for every asset i and the scalar stochastic discount factor m live in a common probability space.

[HR87] and [HJ91] explain how **existence** of a scalar stochastic discount factor that verifies equation (11.1) is implied by a **law of one price** that requires that all portfolios of assets that bring the same payouts have the same price.

They also explain how the **absence of an arbitrage** opportunity implies that the stochastic discount factor $m \geq 0$.

In order to say something about the **uniqueness** of a stochastic discount factor, we would have to impose more theoretical structure than we do in this lecture.

For example, in **complete markets** models like those illustrated in this lecture equilibrium capital structures with incomplete markets, the stochastic discount factor is unique.

In **incomplete markets** models like those illustrated in this lecture the Aiyagari model, the stochastic discount factor is not unique.

11.3 Implications of Key Equation

We combine key equation (11.1) with a remark of Lars Peter Hansen that "asset pricing theory is all about covariances".

Note: Lars Hansen's remark is a concise summary of ideas in [HR87] and [HJ91]. Important foundations of these ideas were set down by [Ros76], [Ros78], [HK79b], [Kre81], and [CR83].

This remark of Lars Hansen refers to the fact that interesting restrictions can be deduced by recognizing that EmR^i is a component of the covariance between m and R^i and then using that fact to rearrange equation (11.1).

Let's do this step by step.

First note that the definition of a covariance $\text{cov}\left(m, R^i\right) = E(m - Em)(R^i - ER^i)$ implies that

$$EmR^i = EmER^i + \text{cov}\left(m, R^i\right)$$

Substituting this result into equation (11.1) gives

$$1 = EmER^i + \text{cov}\left(m, R^i\right) \tag{11.2}$$

Next note that for a risk-free asset with non-random gross return R^f, equation (11.1) becomes

$$1 = ER^f m = R^f Em.$$

This is true because we can pull the constant R^f outside the mathematical expectation.

It follows that the gross return on a risk-free asset is

$$R^f = 1/E(m)$$

Using this formula for R^f in equation (11.2) and rearranging, it follows that

$$R^f = ER^i + \text{cov}\left(m, R^i\right) R^f$$

which can be rearranged to become

$$ER^i = R^f - \text{cov}\left(m, R^i\right) R^f.$$

It follows that we can express an **excess return** $ER^i - R^f$ on asset i relative to the risk-free rate as

$$ER^i - R^f = -\text{cov}\left(m, R^i\right) R^f \tag{11.3}$$

Equation (11.3) can be rearranged to display important parts of asset pricing theory.

11.4 Expected Return - Beta Representation

We can obtain the celebrated **expected-return-Beta -representation** for gross return R^i by simply rearranging excess return equation (11.3) to become

$$ER^i = R^f + \left(\underbrace{\frac{\text{cov}\left(R^i, m\right)}{\text{var}(m)}}_{\beta_{i,m}=\text{regression coefficient}} \right) \left(\underbrace{-\frac{\text{var}(m)}{E(m)}}_{\lambda_m=\text{price of risk}} \right)$$

or

$$ER^i = R^f + \beta_{i,m}\lambda_m \tag{11.4}$$

Here

- $\beta_{i,m}$ is a (population) least squares regression coefficient of gross return R^i on stochastic discount factor m
- λ_m is minus the variance of m divided by the mean of m, an object that is sometimes called a **price of risk**.

Because $\lambda_m < 0$, equation (11.4) asserts that

- assets whose returns are **positively** correlated with the stochastic discount factor (SDF) m have expected returns **lower** than the risk-free rate R^f
- assets whose returns are **negatively** correlated with the SDF m have expected returns **higher** than the risk-free rate R^f

These patterns will be discussed more below.

In particular, we'll see that returns that are **perfectly** negatively correlated with the SDF m have a special status:

- they are on a **mean-variance frontier**

Before we dive into that more, we'll pause to look at an example of an SDF.

To interpret representation (11.4), the following widely used example helps.

Example

Let c_t be the logarithm of the consumption of a *representative consumer* or just a single consumer for whom we have consumption data.

A popular model of m is

$$m_{t+1} = \beta \frac{U'(C_{t+1})}{U'(C_t)}$$

where C_t is consumption at time t, $\beta = \exp(-\rho)$ is a discount **factor** with ρ being the discount **rate**, and $U(\cdot)$ is a concave, twice-diffential utility function.

For a constant relative risk aversion (CRRA) utility function $U(C) = \frac{C^{1-\gamma}}{1-\gamma}$ utility function $U'(C) = C^{-\gamma}$.

In this case, letting $c_t = \log(C_t)$, we can write m_{t+1} as

$$m_{t+1} = \exp(-\rho)\exp(-\gamma(c_{t+1} - c_t))$$

where $\rho > 0$, $\gamma > 0$.

A popular model for the growth of log of consumption is

$$c_{t+1} - c_t = \mu + \sigma_c \epsilon_{t+1}$$

where $\epsilon_{t+1} \sim \mathcal{N}(0,1)$.

Here $\{c_t\}$ is a random walk with drift μ, a good approximation to US per capital consumption growth.

Again here

- $\gamma > 0$ is a coefficient of relative risk aversion
- $\rho > 0$ is a fixed intertemporal discount rate

So we have

$$m_{t+1} = \exp(-\rho)\exp(-\gamma\mu - \gamma\sigma_c\epsilon_{t+1})$$

In this case

$$Em_{t+1} = \exp(-\rho)\exp\left(-\gamma\mu + \frac{\sigma_c^2\gamma^2}{2}\right)$$

and

$$\text{var}(m_{t+1}) = E(m)[\exp(\sigma_c^2\gamma^2) - 1)]$$

When $\gamma > 0$, it is true that

- when consumption growth is **high**, m is **low**
- when consumption growth is **low**, m is **high**

According to representation (11.4), an asset with a gross return R^i that is expected to be **high** when consumption growth is **low** has $\beta_{i,m}$ positive and a **low** expected return.

- because it has a high gross return when consumption growth is low, it is a good hedge against consumption risk. That justifies its low average return.

An asset with an R^i that is **low** when consumption growth is **low** has $\beta_{i,m}$ negative and a **high** expected return.

- because it has a low gross return when consumption growth is low, it is a poor hedge against consumption risk. That justifies its high average return.

11.5 Mean-Variance Frontier

Now we'll derive the celebrated **mean-variance frontier**.

We do this using a method deployed by Lars Peter Hansen and Scott Richard [HR87].

Note: Methods of Hansen and Richard are described and used extensively by [Coc05].

Their idea was rearrange the key equation (11.1), namely, $EmR^i = 1$, and then to apply a Cauchy-Schwarz inequality.

A convenient way to remember the Cauchy-Schwartz inequality in our context is that it says that an R^2 in any regression has to be less than or equal to 1.

(Please note that here R^2 denotes the coefficient of determination in a regression, not a return on an asset!)

Let's apply that idea to deduce

$$1 = E\left(mR^i\right) = E(m)E\left(R^i\right) + \rho_{m,R^i}\frac{\sigma(m)}{E(m)}\sigma\left(R^i\right) \tag{11.5}$$

where the correlation coefficient ρ_{m,R^i} is defined as

$$\rho_{m,R^i} \equiv \frac{\text{cov}\left(m, R^i\right)}{\sigma(m)\sigma\left(R^i\right)}$$

and where $\sigma(\cdot)$ denotes the standard deviation of the variable in parentheses

Equation (11.5) implies

$$ER^i = R^f - \rho_{m,R^i}\frac{\sigma(m)}{E(m)}\sigma\left(R^i\right)$$

Because $\rho_{m,R^i} \in [-1, 1]$, it follows that $|\rho_{m,R^i}| \leq 1$ and that

$$\left|ER^i - R^f\right| \leqslant \frac{\sigma(m)}{E(m)}\sigma\left(R^i\right) \tag{11.6}$$

Inequality (11.6) delineates a **mean-variance frontier**

(Actually, it looks more like a **mean-standard-deviation frontier**)

Evidently, points on the frontier correspond to gross returns that are perfectly correlated (either positively or negatively) with the stochastic discount factor m.

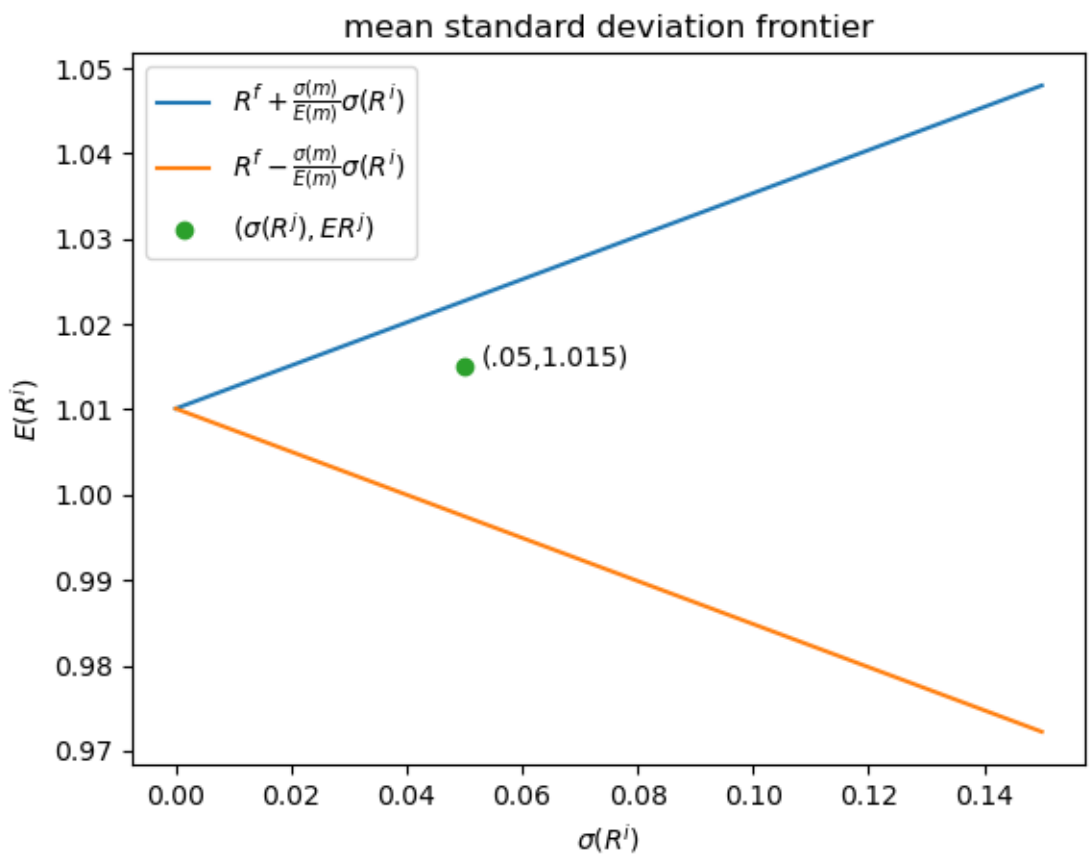
We summarize this observation as

$$\rho_{m,R^i} = \begin{cases} +1 & \implies R^i \text{ is on lower frontier} \\ -1 & \implies R^i \text{ is on an upper frontier} \end{cases}$$

Now let's use matplotlib to draw a mean variance frontier.

In drawing a frontier, we'll set $\sigma(m) = .25$ and $Em = .99$, values roughly consistent with what many studies calibrate from quarterly US data.

```
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline

# Define the function to plot
def y(x, alpha, beta):
    return alpha + beta*x
def z(x, alpha, beta):
    return alpha - beta*x

sigmam = .25
Em = .99

# Set the values of alpha and beta
alpha = 1/Em
beta = sigmam/Em

# Create a range of values for x
x = np.linspace(0, .15, 100)

# Calculate the values of y and z
y_values = y(x, alpha, beta)
z_values = z(x, alpha, beta)

# Create a figure and axes object
fig, ax = plt.subplots()

# Plot y
ax.plot(x, y_values, label=r'$R^f + \frac{\sigma(m)}{E(m)} \sigma(R^i)$')
ax.plot(x, z_values, label=r'$R^f - \frac{\sigma(m)}{E(m)} \sigma(R^i)$')

plt.title('mean standard deviation frontier')
plt.xlabel(r"$\sigma(R^i)$")
plt.ylabel(r"$E (R^i) $")
plt.text(.053, 1.015, "(.05,1.015)")
ax.plot(.05, 1.015, 'o', label="$(\sigma(R^j), E R^j)$")
# Add a legend and show the plot
ax.legend()
plt.show()
```



The figure shows two straight lines, the blue upper one being the locus of $(\sigma(R^i), E(R^i)$ pairs that are on the **mean-variance frontier** or **mean-standard-deviation frontier**.

The green dot refers to a return R^j that is **not** on the frontier and that has moments $(\sigma(R^j), ER^j) = (.05, 1.015)$.

It is described by the statistical model

$$R^j = R^i + \epsilon^j$$

where R^i is a return that is on the frontier and ϵ^j is a random variable that has mean zero and that is orthogonal to R^i.

Then $ER^j = ER^i$ and, as a consequence of R^j not being on the frontier,

$$\sigma^2(R^j) = \sigma^2(R^i) + \sigma^2(\epsilon^j)$$

The length of a horizontal line from the point $\sigma(R^j), E(R^j) = .05, 1.015$ to the frontier equals

$$\sqrt{\sigma^2(R^i) + \sigma^2(\epsilon^j)} - \sigma(R^i)$$

This is a measure of the part of the risk in R^j that is not priced because it is uncorrelated with the stochastic discount factor and so can be diversified away (i.e., averaged out to zero by holding a diversified portfolio).

11.6 Sharpe Ratios and the Price of Risk

An asset's **Sharpe ratio** is defined as

$$\frac{E(R^i) - R^f}{\sigma(R^i)}$$

The above figure reminds us that all assets R^i whose returns are on the mean-standard deviation frontier satisfy

$$\frac{E(R^i) - R^f}{\sigma(R^i)} = \frac{\sigma(m)}{Em}$$

The ratio $\frac{\sigma(m)}{Em}$ is often called the **market price of risk**.

Evidently it equals the maximum Sharpe ratio for any asset or portfolio of assets.

11.7 Mathematical Structure of Frontier

The mathematical structure of the mean-variance frontier described by inequality (11.6) implies that

- all returns on the frontier are perfectly correlated.

  Thus,

  - Let R^m, R^{mv} be two returns on the frontier.
  - Then for some scalar a, a return R^{mv} on the mean-variance frontier satisfies the affine equation $R^{mv} = R^f + a\left(R^m - R^f\right)$. This is an **exact** equation with no **residual**.
- each return R^{mv} that is on the mean-variance frontier is perfectly (negatively) correlated with m
  - $\left(\rho_{m,R^{mv}} = -1\right) \Rightarrow \begin{cases} m = a + bR^{mv} \\ R^{mv} = e + dm \end{cases}$ for some scalars a, b, e, d,

  Therefore, **any return on the mean-variance frontier is a legitimate stochastic discount factor**
- for any mean-variance-efficient return R^{mv} that is on the frontier but that is **not** R^f, there exists a **single-beta representation** for any return R^i that takes the form:

$$ER^i = R^f + \beta_{i,R^{mv}}\left[E\left(R^{mv}\right) - R^f\right] \tag{11.7}$$

- the regression coefficient $\beta_{i,R^{mv}}$ is often called asset i's **beta**
- The special case of a single-beta representation (11.7) with $R^i = R^{mv}$ is

  $ER^{mv} = R^f + 1 \cdot \left[E\left(R^{mv}\right) - R^f\right]$

11.8 Multi-factor Models

The single-beta representation (11.7) is a special case of the multi-factor model

$$ER^i = \gamma + \beta_{i,a}\lambda_a + \beta_{i,b}\lambda_b + \cdots$$

where λ_j is the price of being exposed to risk factor f_t^j and $\beta_{i,j}$ is asset i's exposure to that risk factor.

To uncover the $\beta_{i,j}$'s, one takes data on time series of the risk factors f_t^j that are being priced and specifies the following least squares regression

$$\begin{aligned} R_t^i = a_i + \beta_{i,a} f_t^a + \beta_{i,b} f_t^b + ... + \epsilon_t^i, \quad t = 1, 2, ..., T \\ \epsilon_t^i \perp f_t^j, i = 1, 2, ..., I; j = a, b, ... \end{aligned} \tag{11.8}$$

Special cases are:

- a popular **single-factor** model specifies the single factor f_t to be the return on the market portfolio
- another popular **single-factor** model called the **consumption-based model** specifies the factor to be $m_{t+1} = \beta \frac{u'(c_{t+1})}{u'(c_t)}$, where c_t is a representative consumer's time t consumption.

As a reminder, model objects are interpreted as follows:

- $\beta_{i,a}$ is the exposure of return R^i to risk factor f_a
- λ_a is the price of exposure to risk factor f_a

11.9 Empirical Implementations

We briefly describe empirical implementations of multi-factor generalizations of the single-factor model described above.

Two representations of a multi-factor model play importnt roles in empirical applications.

One is the time series regression (11.8)

The other representation entails a **cross-section regression** of **average returns** ER^i for assets $i = 1, 2, ..., I$ on **prices of risk** λ_j for $j = a, b, c, ...$

Here is the cross-section regression specification for a multi-factor model:

$$ER^i = \gamma + \beta_{i,a} \lambda_a + \beta_{i,b} \lambda_b + \cdots$$

Testing strategies:

Time-series and cross-section regressions play roles in both **estimating** and **testing** beta representation models.

The basic idea is to implement the following two steps.

Step 1:

- Estimate $a_i, \beta_{i,a}, \beta_{i,b}, \cdots$ by running a **time series regression:** R_t^i on a constant and $f_t^a, f_t^b, ...$

Step 2:

- take the $\beta_{i,j}$'s estimated in step one as regressors together with data on average returns ER^i over some period and then estimate the **cross-section regression**

$$\underbrace{E(R^i)}_{\text{average return over time series}} = \gamma + \underbrace{\beta_{i,a}}_{\text{regressor}} \underbrace{\lambda_a}_{\text{regressioncoefficient}} + \underbrace{\beta_{i,b}}_{\text{regressor}} \underbrace{\lambda_b}_{\text{regressioncoefficient}} + \cdots + \underbrace{\alpha_i}_{\text{pricing errors}}, i = 1, ..., I; \quad \underbrace{\alpha_i \perp \beta_{i,j}, j = a, b, ...}_{\text{least squares orthogonality condition}}$$

- Here $\perp$ means **orthogonal to**
- estimate $\gamma, \lambda_a, \lambda_b, ...$ by an appropriate regression technique, recognizing that the regressors have been generated by a step 1 regression.

Note that presumably the risk-free return $ER^f = \gamma$.

For excess returns $R^{ei} = R^i - R^f$ we have

$$ER^{ei} = \beta_{i,a} \lambda_a + \beta_{i,b} \lambda_b + \cdots + \alpha_i, i = 1, ..., I$$

In the following exercises, we illustrate aspects of these empirical strategies on artificial data.

Our basic tools are random number generator that we shall use to create artificial samples that conform to the theory and least squares regressions that let us watch aspects of the theory at work.

These exercises will further convince us that asset pricing theory is mostly about covariances and least squares regressions.

11.10 Exercises

Let's start with some imports.

```
import numpy as np
import statsmodels.api as sm
import matplotlib.pyplot as plt
%matplotlib inline
```

Lots of our calculations will involve computing population and sample OLS regressions.

So we define a function for simple univariate OLS regression that calls the `OLS` routine from `statsmodels`.

```
def simple_ols(X, Y, constant=False):

    if constant:
        X = sm.add_constant(X)

    model = sm.OLS(Y, X)
    res = model.fit()

    β_hat = res.params[-1]
    σ_hat = np.sqrt(res.resid @ res.resid / res.df_resid)

    return β_hat, σ_hat
```

Exercise 11.10.1

Look at the equation,

$$R_t^i - R^f = \beta_{i,R^m}(R_t^m - R^f) + \sigma_i \varepsilon_{i,t}.$$

Verify that this equation is a regression equation.

Solution to Exercise 11.10.1

To verify that it is a **regression equation** we must show that the residual is orthogonal to the regressor.

Our assumptions about mutual orthogonality imply that

$$E\left[\epsilon_{i,t}\right] = 0, \quad E\left[\epsilon_{i,t} u_t\right] = 0$$

It follows that

$$\begin{aligned} E\left[\sigma_i \epsilon_{i,t}\left(R_t^m - R^f\right)\right] &= E\left[\sigma_i \epsilon_{i,t}\left(\xi + \lambda u_t\right)\right] \\ &= \sigma_i \xi E\left[\epsilon_{i,t}\right] + \sigma_i \lambda E\left[\epsilon_{i,t} u_t\right] \\ &= 0 \end{aligned}$$

Exercise 11.10.2

Give a formula for the regression coefficient β_{i,R^m}.

Solution to Exercise 11.10.2

The regression coefficient β_{i,R^m} is

$$\beta_{i,R^m} = \frac{Cov\left(R_t^i - R^f, R_t^m - R^f\right)}{Var\left(R_t^m - R^f\right)}$$

Exercise 11.10.3

As in many sciences, it is useful to distinguish a **direct problem** from an **inverse problem**.

- A direct problem involves simulating a particular model with known parameter values.
- An inverse problem involves using data to **estimate** or **choose** a particular parameter vector from a manifold of models indexed by a set of parameter vectors.

Please assume the parameter values provided below and then simulate 2000 observations from the theory specified above for 5 assets, $i = 1, \ldots, 5$.

$$\begin{aligned}
E\left[R^f\right] &= 0.02 \\
\sigma_f &= 0.00 \\
\xi &= 0.06 \\
\lambda &= 0.04 \\
\beta_{1,R^m} &= 0.2 \\
\sigma_1 &= 0.04 \\
\beta_{2,R^m} &= .4 \\
\sigma_2 &= 0.04 \\
\beta_{3,R^m} &= .6 \\
\sigma_3 &= 0.04 \\
\beta_{4,R^m} &= .8 \\
\sigma_4 &= 0.04 \\
\beta_{5,R^m} &= 1.0 \\
\sigma_5 &= 0.04
\end{aligned}$$

More Exercises

Now come some even more fun parts!

Our theory implies that there exist values of two scalars, a and b, such that a legitimate stochastic discount factor is:

$$m_t = a + bR_t^m$$

The parameters a, b must satisfy the following equations:

$$\begin{aligned}
E[(a + bR_t^m)R_t^m)] &= 1 \\
E[(a + bR_t^m)R_t^f)] &= 1
\end{aligned}$$

Solution to Exercise 11.10.3

Direct Problem:

```
# Code for the direct problem

# assign the parameter values
ERf = 0.02
σf = 0.00 # Zejin: Hi tom, here is where you manipulate σf
ξ = 0.06
λ = 0.08
βi = np.array([0.2, .4, .6, .8, 1.0])
σi = np.array([0.04, 0.04, 0.04, 0.04, 0.04])
```

```
# in this cell we set the number of assets and number of observations
# we first set T to a large number to verify our computation results
T = 2000
N = 5
```

```
# simulate i.i.d. random shocks
e = np.random.normal(size=T)
u = np.random.normal(size=T)
ϵ = np.random.normal(size=(N, T))
```

```
# simulate the return on a risk-free asset
Rf = ERf + σf * e

# simulate the return on the market portfolio
excess_Rm = ξ + λ * u
Rm = Rf + excess_Rm

# simulate the return on asset i
Ri = np.empty((N, T))
for i in range(N):
    Ri[i, :] = Rf + βi[i] * excess_Rm + σi[i] * ϵ[i, :]
```

Now that we have a panel of data, we'd like to solve the inverse problem by assuming the theory specified above and estimating the coefficients given above.

```
# Code for the inverse problem
```

Inverse Problem:

We will solve the inverse problem by simple OLS regressions.

1. estimate $E\left[R^f\right]$ and σ_f

```
ERf_hat, σf_hat = simple_ols(np.ones(T), Rf)
```

```
ERf_hat, σf_hat
```

```
(0.020000000000000046, 4.5114090308141905e-17)
```

Let's compare these with the *true* population parameter values.

```
ERf, σf
```

```
(0.02, 0.0)
```

2. ξ and λ

```
ξ_hat, λ_hat = simple_ols(np.ones(T), Rm - Rf)
```

```
ξ_hat, λ_hat
```

```
(0.05986028501738618, 0.07962127020932139)
```

```
ξ, λ
```

```
(0.06, 0.08)
```

3. β_{i,R^m} and σ_i

```
βi_hat = np.empty(N)
σi_hat = np.empty(N)

for i in range(N):
    βi_hat[i], σi_hat[i] = simple_ols(Rm - Rf, Ri[i, :] - Rf)
```

```
βi_hat, σi_hat
```

```
(array([0.1928035 , 0.39546209, 0.59346563, 0.76993033, 1.00700414]),
 array([0.03977189, 0.03956167, 0.04105691, 0.03959442, 0.03967038]))
```

```
βi, σi
```

```
(array([0.2, 0.4, 0.6, 0.8, 1. ]), array([0.04, 0.04, 0.04, 0.04, 0.04]))
```

Q: How close did your estimates come to the parameters we specified?

Exercise 11.10.4

Using the equations above, find a system of two **linear** equations that you can solve for a and b as functions of the parameters $(\lambda, \xi, E[R_f])$.

Write a function that can solve these equations.

Please check the **condition number** of a key matrix that must be inverted to determine a, b

---

Solution to Exercise 11.10.4

The system of two linear equations is shown below:

$$a((E(R^f)+\xi)+b((E(R^f)+\xi)^2+\lambda^2+\sigma_f^2)=1$$
$$aE(R^f)+b(E(R^f)^2+\xi E(R^f)+\sigma_f^2)=1$$

```
# Code here
def solve_ab(ERf, σf, λ, ξ):

    M = np.empty((2, 2))
    M[0, 0] = ERf + ξ
    M[0, 1] = (ERf + ξ) ** 2 + λ ** 2 + σf ** 2
    M[1, 0] = ERf
    M[1, 1] = ERf ** 2 + ξ * ERf + σf ** 2

    a, b = np.linalg.solve(M, np.ones(2))
    condM = np.linalg.cond(M)

    return a, b, condM
```

Let's try to solve a and b using the actual model parameters.

```
a, b, condM = solve_ab(ERf, σf, λ, ξ)
```

```
a, b, condM
```

```
(87.49999999999999, -468.7499999999999, 54.406619883717504)
```

Exercise 11.10.5

Using the estimates of the parameters that you generated above, compute the implied stochastic discount factor.

Solution to Exercise 11.10.5

Now let's pass $\hat{E}(R^f), \hat{\sigma}^f, \hat{\lambda}, \hat{\xi}$ to the function `solve_ab`.

```
a_hat, b_hat, M_hat = solve_ab(ERf_hat, σf_hat, λ_hat, ξ_hat)
```

```
a_hat, b_hat, M_hat
```

```
(87.70348001004712, -472.1180246456532, 54.732834096498)
```

CHAPTER

TWELVE

TWO MODIFICATIONS OF MEAN-VARIANCE PORTFOLIO THEORY

Contents

- *Two Modifications of Mean-Variance Portfolio Theory*
  - *Overview*
  - *Mean-Variance Portfolio Choice*
  - *Estimating Mean and Variance*
  - *Black-Litterman Starting Point*
  - *Details*
  - *Adding Views*
  - *Bayesian Interpretation*
  - *Curve Decolletage*
  - *Black-Litterman Recommendation as Regularization*
  - *A Robust Control Operator*
  - *A Robust Mean-Variance Portfolio Model*
  - *Appendix*
  - *Special Case – IID Sample*
  - *Dependence and Sampling Frequency*
  - *Frequency and the Mean Estimator*

12.1 Overview

This lecture describes extensions to the classical mean-variance portfolio theory summarized in our lecture Elementary Asset Pricing Theory.

The classic theory described there assumes that a decision maker completely trusts the statistical model that he posits to govern the joint distribution of returns on a list of available assets.

Both extensions described here put distrust of that statistical model into the mind of the decision maker.

One is a model of Black and Litterman [BL92] that imputes to the decision maker distrust of historically estimated mean returns but still complete trust of estimated covariances of returns.

The second model also imputes to the decision maker doubts about his statistical model, but now by saying that, because of that distrust, the decision maker uses a version of robust control theory described in this lecture Robustness.

The famous **Black-Litterman** (1992) [BL92] portfolio choice model was motivated by the finding that with high frequency or moderately high frequency data, means are more difficult to estimate than variances.

A model of **robust portfolio choice** that we'll describe below also begins from the same starting point.

To begin, we'll take for granted that means are more difficult to estimate that covariances and will focus on how Black and Litterman, on the one hand, an robust control theorists, on the other, would recommend modifying the **mean-variance portfolio choice model** to take that into account.

At the end of this lecture, we shall use some rates of convergence results and some simulations to verify how means are more difficult to estimate than variances.

Among the ideas in play in this lecture will be

- Mean-variance portfolio theory
- Bayesian approaches to estimating linear regressions
- A risk-sensitivity operator and its connection to robust control theory

In summary, we'll describe two ways to modify the classic mean-variance portfolio choice model in ways designed to make its recommendations more plausible.

Both of the adjustments that we describe are designed to confront a widely recognized embarrassment to mean-variance portfolio theory, namely, that it usually implies taking very extreme long-short portfolio positions.

The two approaches build on a common and widespread hunch – that because it is much easier statistically to estimate covariances of excess returns than it is to estimate their means, it makes sense to adjust investors' subjective beliefs about mean returns in order to render more plausible decisions.

Let's start with some imports:

```
import numpy as np
import scipy.stats as stat
import matplotlib.pyplot as plt
%matplotlib inline
from ipywidgets import interact, FloatSlider
```

12.2 Mean-Variance Portfolio Choice

A risk-free security earns one-period net return r_f.

An $n \times 1$ vector of risky securities earns an $n \times 1$ vector $\vec{r} - r_f \mathbf{1}$ of *excess returns*, where $\mathbf{1}$ is an $n \times 1$ vector of ones.

The excess return vector is multivariate normal with mean μ and covariance matrix Σ, which we express either as

$$\vec{r} - r_f \mathbf{1} \sim \mathcal{N}(\mu, \Sigma)$$

or

$$\vec{r} - r_f \mathbf{1} = \mu + C\epsilon$$

where $\epsilon \sim \mathcal{N}(0, I)$ is an $n \times 1$ random vector.

Let w be an $n \times 1$ vector of portfolio weights.

A portfolio consisting w earns returns

$$w'(\vec{r} - r_f \mathbf{1}) \sim \mathcal{N}(w'\mu, w'\Sigma w)$$

The **mean-variance portfolio choice problem** is to choose w to maximize

$$U(\mu, \Sigma; w) = w'\mu - \frac{\delta}{2} w'\Sigma w \tag{12.1}$$

where $\delta > 0$ is a risk-aversion parameter. The first-order condition for maximizing (12.1) with respect to the vector w is

$$\mu = \delta \Sigma w$$

which implies the following design of a risky portfolio:

$$w = (\delta\Sigma)^{-1}\mu \tag{12.2}$$

12.3 Estimating Mean and Variance

The key inputs into the portfolio choice model (12.2) are

- estimates of the parameters μ, Σ of the random excess return vector$(\vec{r} - r_f \mathbf{1})$
- the risk-aversion parameter δ

A standard way of estimating μ is maximum-likelihood or least squares; that amounts to estimating μ by a sample mean of excess returns and estimating Σ by a sample covariance matrix.

12.4 Black-Litterman Starting Point

When estimates of μ and Σ from historical sample means and covariances have been combined with **plausible** values of the risk-aversion parameter δ to compute an optimal portfolio from formula (12.2), a typical outcome has been w's with **extreme long and short positions**.

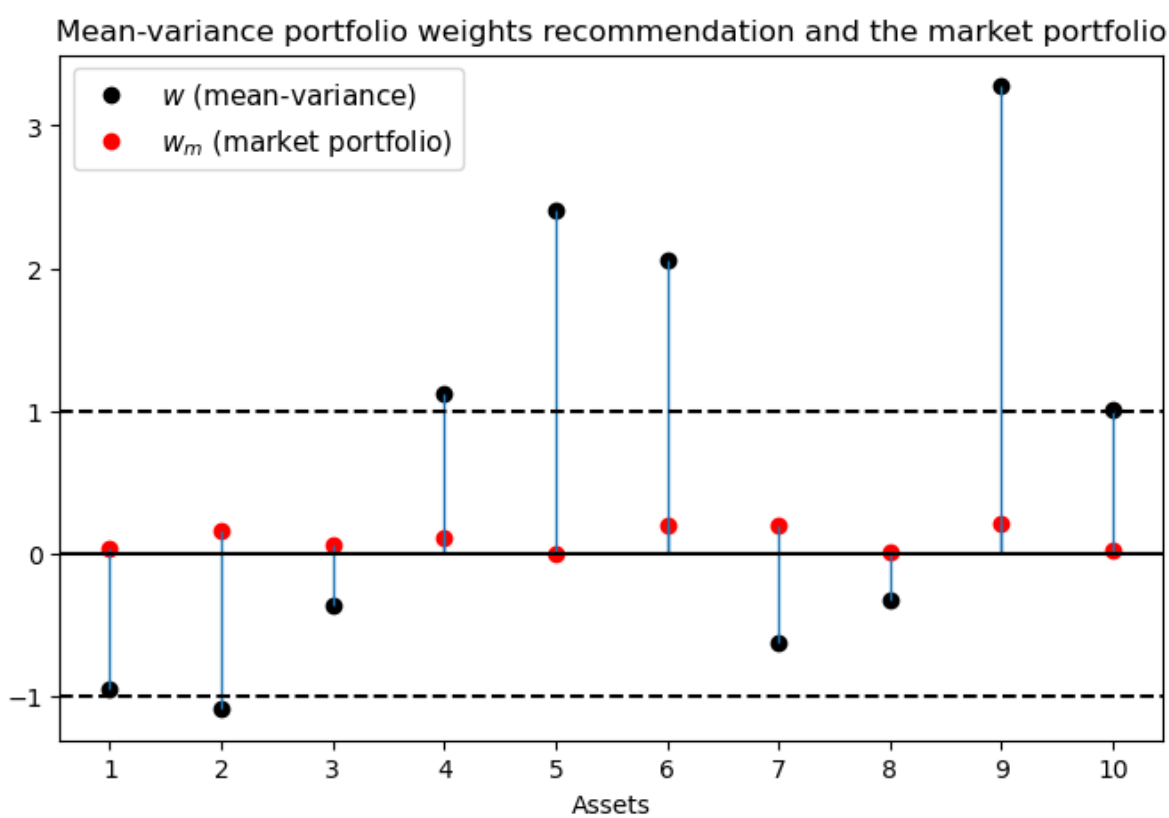
A common reaction to these outcomes is that they are so implausible that a portfolio manager cannot recommend them to a customer.

```
np.random.seed(12)

N = 10                                          # Number of assets
T = 200                                         # Sample size

# random market portfolio (sum is normalized to 1)
w_m = np.random.rand(N)
w_m = w_m / (w_m.sum())

# True risk premia and variance of excess return (constructed
# so that the Sharpe ratio is 1)
μ = (np.random.randn(N) + 5)  /100      # Mean excess return (risk premium)
S = np.random.randn(N, N)        # Random matrix for the covariance matrix
V = S @ S.T           # Turn the random matrix into symmetric psd
# Make sure that the Sharpe ratio is one
Σ = V * (w_m @ μ)**2 / (w_m @ V @ w_m)
```

(continues on next page)

(continued from previous page)

```
# Risk aversion of market portfolio holder
δ = 1 / np.sqrt(w_m @ Σ @ w_m)

# Generate a sample of excess returns
excess_return = stat.multivariate_normal(μ, Σ)
sample = excess_return.rvs(T)

# Estimate μ and Σ
μ_est = sample.mean(0).reshape(N, 1)
Σ_est = np.cov(sample.T)

w = np.linalg.solve(δ * Σ_est, μ_est)

fig, ax = plt.subplots(figsize=(8, 5))
ax.set_title('Mean-variance portfolio weights recommendation and the market portfolio
↪')
ax.plot(np.arange(N)+1, w, 'o', c='k', label='$w$ (mean-variance)')
ax.plot(np.arange(N)+1, w_m, 'o', c='r', label='$w_m$ (market portfolio)')
ax.vlines(np.arange(N)+1, 0, w, lw=1)
ax.vlines(np.arange(N)+1, 0, w_m, lw=1)
ax.axhline(0, c='k')
ax.axhline(-1, c='k', ls='--')
ax.axhline(1, c='k', ls='--')
ax.set_xlabel('Assets')
ax.xaxis.set_ticks(np.arange(1, N+1, 1))
plt.legend(numpoints=1, fontsize=11)
plt.show()
```



Black and Litterman's responded to this situation in the following way:

- They continue to accept (12.2) as a good model for choosing an optimal portfolio w.
- They want to continue to allow the customer to express his or her risk tolerance by setting δ.
- Leaving Σ at its maximum-likelihood value, they push μ away from its maximum-likelihood value in a way designed to make portfolio choices that are more plausible in terms of conforming to what most people actually do.

In particular, given Σ and a plausible value of δ, Black and Litterman reverse engineered a vector μ_{BL} of mean excess returns that makes the w implied by formula (12.2) equal the **actual** market portfolio w_m, so that

$$w_m = (\delta\Sigma)^{-1}\mu_{BL}$$

12.5 Details

Let's define

$$w_m'\mu \equiv (r_m - r_f)$$

as the (scalar) excess return on the market portfolio w_m.

Define

$$\sigma^2 = w_m'\Sigma w_m$$

as the variance of the excess return on the market portfolio w_m.

Define

$$\mathbf{SR}_m = \frac{r_m - r_f}{\sigma}$$

as the **Sharpe-ratio** on the market portfolio w_m.

Let δ_m be the value of the risk aversion parameter that induces an investor to hold the market portfolio in light of the optimal portfolio choice rule (12.2).

Evidently, portfolio rule (12.2) then implies that $r_m - r_f = \delta_m\sigma^2$ or

$$\delta_m = \frac{r_m - r_f}{\sigma^2}$$

or

$$\delta_m = \frac{\mathbf{SR}_m}{\sigma}$$

Following the Black-Litterman philosophy, our first step will be to back a value of δ_m from

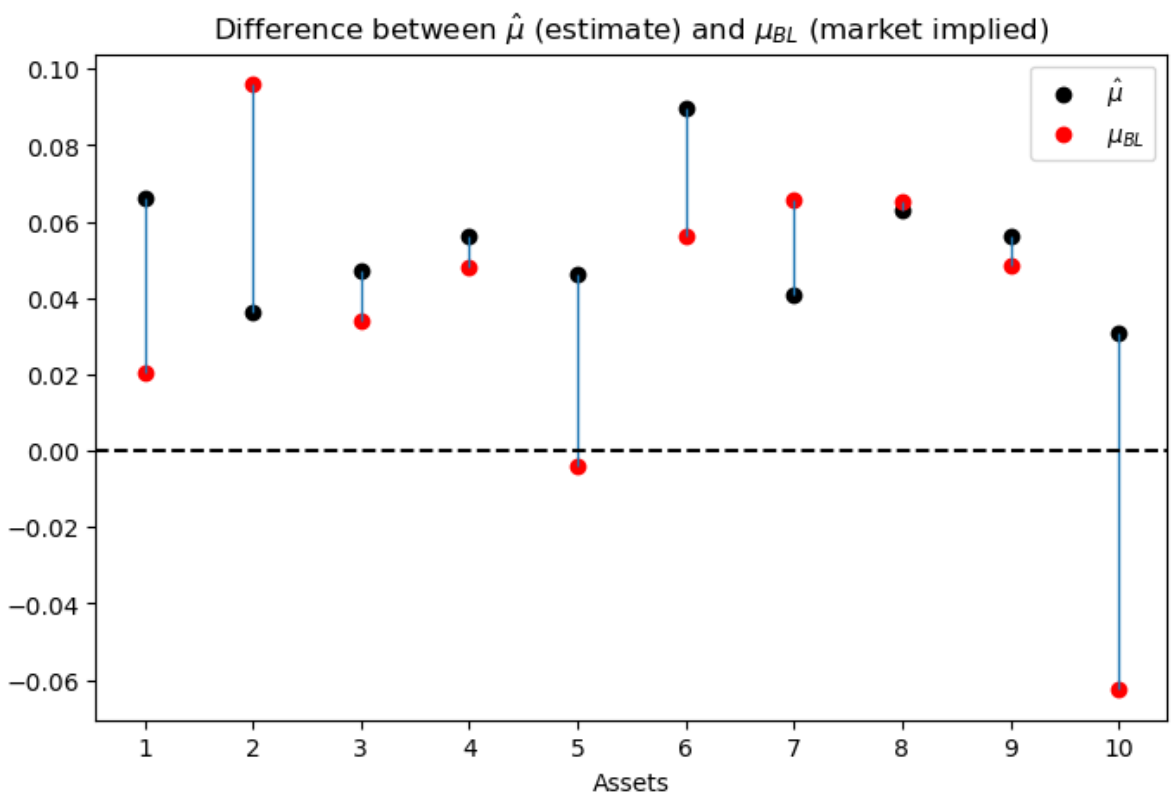
- an estimate of the Sharpe-ratio, and
- our maximum likelihood estimate of σ drawn from our estimates or w_m and Σ

The second key Black-Litterman step is then to use this value of δ together with the maximum likelihood estimate of Σ to deduce a $\mu_{\mathbf{BL}}$ that verifies portfolio rule (12.2) at the market portfolio $w = w_m$

$$\mu_m = \delta_m\Sigma w_m$$

The starting point of the Black-Litterman portfolio choice model is thus a pair (δ_m, μ_m) that tells the customer to hold the market portfolio.

```
# Observed mean excess market return
r_m = w_m @ μ_est

# Estimated variance of the market portfolio
σ_m = w_m @ Σ_est @ w_m

# Sharpe-ratio
sr_m = r_m / np.sqrt(σ_m)

# Risk aversion of market portfolio holder
d_m = r_m / σ_m

# Derive "view" which would induce the market portfolio
μ_m = (d_m * Σ_est @ w_m).reshape(N, 1)

fig, ax = plt.subplots(figsize=(8, 5))
ax.set_title(r'Difference between $\hat{\mu}$ (estimate) and $\mu_{BL}$ (market␣
↪implied)')
ax.plot(np.arange(N)+1, μ_est, 'o', c='k', label='$\hat{\mu}$')
ax.plot(np.arange(N)+1, μ_m, 'o', c='r', label='$\mu_{BL}$')
ax.vlines(np.arange(N) + 1, μ_m, μ_est, lw=1)
ax.axhline(0, c='k', ls='--')
ax.set_xlabel('Assets')
ax.xaxis.set_ticks(np.arange(1, N+1, 1))
plt.legend(numpoints=1)
plt.show()
```



12.6 Adding Views

Black and Litterman start with a baseline customer who asserts that he or she shares the **market's views**, which means that he or she believes that excess returns are governed by

$$\vec{r} - r_f \mathbf{1} \sim \mathcal{N}(\mu_{BL}, \Sigma) \tag{12.3}$$

Black and Litterman would advise that customer to hold the market portfolio of risky securities.

Black and Litterman then imagine a consumer who would like to express a view that differs from the market's.

The consumer wants appropriately to mix his view with the market's before using (12.2) to choose a portfolio.

Suppose that the customer's view is expressed by a hunch that rather than (12.3), excess returns are governed by

$$\vec{r} - r_f \mathbf{1} \sim \mathcal{N}(\hat{\mu}, \tau\Sigma)$$

where $\tau > 0$ is a scalar parameter that determines how the decision maker wants to mix his view $\hat{\mu}$ with the market's view $\mu_{\mathbf{BL}}$.

Black and Litterman would then use a formula like the following one to mix the views $\hat{\mu}$ and $\mu_{\mathbf{BL}}$

$$\tilde{\mu} = (\Sigma^{-1} + (\tau\Sigma)^{-1})^{-1}(\Sigma^{-1}\mu_{BL} + (\tau\Sigma)^{-1}\hat{\mu}) \tag{12.4}$$

Black and Litterman would then advise the customer to hold the portfolio associated with these views implied by rule (12.2):

$$\tilde{w} = (\delta\Sigma)^{-1}\tilde{\mu}$$

This portfolio $\tilde{w}$ will deviate from the portfolio w_{BL} in amounts that depend on the mixing parameter τ.

If $\hat{\mu}$ is the maximum likelihood estimator and τ is chosen heavily to weight this view, then the customer's portfolio will involve big short-long positions.

```
def black_litterman(λ, μ1, μ2, Σ1, Σ2):
    """
    This function calculates the Black-Litterman mixture
    mean excess return and covariance matrix
    """
    Σ1_inv = np.linalg.inv(Σ1)
    Σ2_inv = np.linalg.inv(Σ2)

    μ_tilde = np.linalg.solve(Σ1_inv + λ * Σ2_inv,
                              Σ1_inv @ μ1 + λ * Σ2_inv @ μ2)
    return μ_tilde

τ = 1
μ_tilde = black_litterman(1, μ_m, μ_est, Σ_est, τ * Σ_est)

# The Black-Litterman recommendation for the portfolio weights
w_tilde = np.linalg.solve(δ * Σ_est, μ_tilde)

τ_slider = FloatSlider(min=0.05, max=10, step=0.5, value=τ)

@interact(τ=τ_slider)
def BL_plot(τ):
    μ_tilde = black_litterman(1, μ_m, μ_est, Σ_est, τ * Σ_est)
    w_tilde = np.linalg.solve(δ * Σ_est, μ_tilde)
```

(continues on next page)

(continued from previous page)

```
    fig, ax = plt.subplots(1, 2, figsize=(16, 6))
    ax[0].plot(np.arange(N)+1, μ_est, 'o', c='k',
               label=r'$\hat{\mu}$ (subj view)')
    ax[0].plot(np.arange(N)+1, μ_m, 'o', c='r',
               label=r'$\mu_{BL}$ (market)')
    ax[0].plot(np.arange(N)+1, μ_tilde, 'o', c='y',
               label=r'$\tilde{\mu}$ (mixture)')
    ax[0].vlines(np.arange(N)+1, μ_m, μ_est, lw=1)
    ax[0].axhline(0, c='k', ls='--')
    ax[0].set(xlim=(0, N+1), xlabel='Assets',
              title=r'Relationship between $\hat{\mu}$, $\mu_{BL}$, and  $ \tilde{\mu}
↪$')
    ax[0].xaxis.set_ticks(np.arange(1, N+1, 1))
    ax[0].legend(numpoints=1)

    ax[1].set_title('Black-Litterman portfolio weight recommendation')
    ax[1].plot(np.arange(N)+1, w, 'o', c='k', label=r'$w$ (mean-variance)')
    ax[1].plot(np.arange(N)+1, w_m, 'o', c='r', label=r'$w_{m}$ (market, BL)')
    ax[1].plot(np.arange(N)+1, w_tilde, 'o', c='y',
               label=r'$\tilde{w}$ (mixture)')
    ax[1].vlines(np.arange(N)+1, 0, w, lw=1)
    ax[1].vlines(np.arange(N)+1, 0, w_m, lw=1)
    ax[1].axhline(0, c='k')
    ax[1].axhline(-1, c='k', ls='--')
    ax[1].axhline(1, c='k', ls='--')
    ax[1].set(xlim=(0, N+1), xlabel='Assets',
              title='Black-Litterman portfolio weight recommendation')
    ax[1].xaxis.set_ticks(np.arange(1, N+1, 1))
    ax[1].legend(numpoints=1)
    plt.show()
```

```
interactive(children=(FloatSlider(value=1.0, description='τ', max=10.0, min=0.05,␣
↪step=0.5), Output()), _dom_c…
```

12.7 Bayesian Interpretation

Consider the following Bayesian interpretation of the Black-Litterman recommendation.

The prior belief over the mean excess returns is consistent with the market portfolio and is given by

$$\mu \sim \mathcal{N}(\mu_{BL}, \Sigma)$$

Given a particular realization of the mean excess returns μ one observes the average excess returns $\hat{\mu}$ on the market according to the distribution

$$\hat{\mu} \mid \mu, \Sigma \sim \mathcal{N}(\mu, \tau\Sigma)$$

where τ is typically small capturing the idea that the variation in the mean is smaller than the variation of the individual random variable.

Given the realized excess returns one should then update the prior over the mean excess returns according to Bayes rule.

The corresponding posterior over mean excess returns is normally distributed with mean

$$(\Sigma^{-1} + (\tau\Sigma)^{-1})^{-1}(\Sigma^{-1}\mu_{BL} + (\tau\Sigma)^{-1}\hat{\mu})$$

The covariance matrix is

$$(\Sigma^{-1} + (\tau\Sigma)^{-1})^{-1}$$

Hence, the Black-Litterman recommendation is consistent with the Bayes update of the prior over the mean excess returns in light of the realized average excess returns on the market.

12.8 Curve Decolletage

Consider two independent "competing" views on the excess market returns

$$\vec{r}_e \sim \mathcal{N}(\mu_{BL}, \Sigma)$$

and

$$\vec{r}_e \sim \mathcal{N}(\hat{\mu}, \tau\Sigma)$$

A special feature of the multivariate normal random variable Z is that its density function depends only on the (Euclidiean) length of its realization z.

Formally, let the k-dimensional random vector be

$$Z \sim \mathcal{N}(\mu, \Sigma)$$

then

$$\bar{Z} \equiv \Sigma(Z - \mu) \sim \mathcal{N}(\mathbf{0}, I)$$

and so the points where the density takes the same value can be described by the ellipse

$$\bar{z} \cdot \bar{z} = (z - \mu)'\Sigma^{-1}(z - \mu) = \bar{d} \tag{12.5}$$

where $\bar{d} \in \mathbb{R}_+$ denotes the (transformation) of a particular density value.

The curves defined by equation (12.5) can be labeled as iso-likelihood ellipses

> **Remark:** More generally there is a class of density functions that possesses this feature, i.e.
>
> $$\exists g : \mathbb{R}_+ \mapsto \mathbb{R}_+ \quad \text{and} \quad c \geq 0, \quad \text{s.t. the density} \quad f \quad \text{of} \quad Z \quad \text{has the form} \qquad f(z) = cg(z \cdot z)$$
>
> This property is called **spherical symmetry** (see p 81. in Leamer (1978) [Lea78]).

In our specific example, we can use the pair $(\bar{d}_1, \bar{d}_2)$ as being two "likelihood" values for which the corresponding iso-likelihood ellipses in the excess return space are given by

$$\begin{aligned}(\vec{r}_e - \mu_{BL})'\Sigma^{-1}(\vec{r}_e - \mu_{BL}) &= \bar{d}_1 \\ (\vec{r}_e - \hat{\mu})' \left(\tau\Sigma\right)^{-1} (\vec{r}_e - \hat{\mu}) &= \bar{d}_2\end{aligned}$$

Notice that for particular $\bar{d}_1$ and $\bar{d}_2$ values the two ellipses have a tangency point.

These tangency points, indexed by the pairs $(\bar{d}_1, \bar{d}_2)$, characterize points $\vec{r}_e$ from which there exists no deviation where one can increase the likelihood of one view without decreasing the likelihood of the other view.

The pairs $(\bar{d}_1, \bar{d}_2)$ for which there is such a point outlines a curve in the excess return space. This curve is reminiscent of the Pareto curve in an Edgeworth-box setting.

Dickey (1975) [Dic75] calls it a *curve decolletage*.

Leamer (1978) [Lea78] calls it an *information contract curve* and describes it by the following program: maximize the likelihood of one view, say the Black-Litterman recommendation while keeping the likelihood of the other view at least at a prespecified constant $\bar{d}_2$

$$\bar{d}_1(\bar{d}_2) \equiv \max_{\vec{r}_e} \ (\vec{r}_e - \mu_{BL})'\Sigma^{-1}(\vec{r}_e - \mu_{BL})$$
$$\text{subject to} \quad (\vec{r}_e - \hat{\mu})'(\tau\Sigma)^{-1}(\vec{r}_e - \hat{\mu}) \geq \bar{d}_2$$

Denoting the multiplier on the constraint by λ, the first-order condition is

$$2(\vec{r}_e - \mu_{BL})'\Sigma^{-1} + \lambda 2(\vec{r}_e - \hat{\mu})'(\tau\Sigma)^{-1} = \mathbf{0}$$

which defines the *information contract curve* between μ_{BL} and $\hat{\mu}$

$$\vec{r}_e = (\Sigma^{-1} + \lambda(\tau\Sigma)^{-1})^{-1}(\Sigma^{-1}\mu_{BL} + \lambda(\tau\Sigma)^{-1}\hat{\mu}) \tag{12.6}$$

Note that if $\lambda = 1$, (12.6) is equivalent with (12.4) and it identifies one point on the information contract curve.

Furthermore, because λ is a function of the minimum likelihood $\bar{d}_2$ on the RHS of the constraint, by varying $\bar{d}_2$ (or λ), we can trace out the whole curve as the figure below illustrates.

```
np.random.seed(1987102)

N = 2                                          # Number of assets
T = 200                                        # Sample size
τ = 0.8

# Random market portfolio (sum is normalized to 1)
w_m = np.random.rand(N)
w_m = w_m / (w_m.sum())

μ = (np.random.randn(N) + 5) / 100
S = np.random.randn(N, N)
V = S @ S.T
Σ = V * (w_m @ μ)**2 / (w_m @ V @ w_m)

excess_return = stat.multivariate_normal(μ, Σ)
sample = excess_return.rvs(T)

μ_est = sample.mean(0).reshape(N, 1)
Σ_est = np.cov(sample.T)

σ_m = w_m @ Σ_est @ w_m
d_m = (w_m @ μ_est) / σ_m
μ_m = (d_m * Σ_est @ w_m).reshape(N, 1)

N_r1, N_r2 = 100, 100
r1 = np.linspace(-0.04, .1, N_r1)
r2 = np.linspace(-0.02, .15, N_r2)

λ_grid = np.linspace(.001, 20, 100)
curve = np.asarray([black_litterman(λ, μ_m, μ_est, Σ_est,
                                    τ * Σ_est).flatten() for λ in λ_grid])

λ_slider = FloatSlider(min=.1, max=7, step=.5, value=1)

@interact(λ=λ_slider)
def decolletage(λ):
```

(continues on next page)

(continued from previous page)

```
    dist_r_BL = stat.multivariate_normal(μ_m.squeeze(), Σ_est)
    dist_r_hat = stat.multivariate_normal(μ_est.squeeze(), τ * Σ_est)

    X, Y = np.meshgrid(r1, r2)
    Z_BL = np.zeros((N_r1, N_r2))
    Z_hat = np.zeros((N_r1, N_r2))

    for i in range(N_r1):
        for j in range(N_r2):
            Z_BL[i, j] = dist_r_BL.pdf(np.hstack([X[i, j], Y[i, j]]))
            Z_hat[i, j] = dist_r_hat.pdf(np.hstack([X[i, j], Y[i, j]]))

    μ_tilde = black_litterman(λ, μ_m, μ_est, Σ_est, τ * Σ_est).flatten()

    fig, ax = plt.subplots(figsize=(10, 6))
    ax.contourf(X, Y, Z_hat, cmap='viridis', alpha =.4)
    ax.contourf(X, Y, Z_BL, cmap='viridis', alpha =.4)
    ax.contour(X, Y, Z_BL, [dist_r_BL.pdf(μ_tilde)], cmap='viridis', alpha=.9)
    ax.contour(X, Y, Z_hat, [dist_r_hat.pdf(μ_tilde)], cmap='viridis', alpha=.9)
    ax.scatter(μ_est[0], μ_est[1])
    ax.scatter(μ_m[0], μ_m[1])
    ax.scatter(μ_tilde[0], μ_tilde[1], c='k', s=20*3)

    ax.plot(curve[:, 0], curve[:, 1], c='k')
    ax.axhline(0, c='k', alpha=.8)
    ax.axvline(0, c='k', alpha=.8)
    ax.set_xlabel(r'Excess return on the first asset, $r_{e, 1}$')
    ax.set_ylabel(r'Excess return on the second asset, $r_{e, 2}$')
    ax.text(μ_est[0] + 0.003, μ_est[1], r'$\hat{\mu}$')
    ax.text(μ_m[0] + 0.003, μ_m[1] + 0.005, r'$\mu_{BL}$')
    plt.show()
```

```
interactive(children=(FloatSlider(value=1.0, description='λ', max=7.0, min=0.1,␣
↪step=0.5), Output()), _dom_cla…
```

Note that the line that connects the two points $\hat{\mu}$ and μ_{BL} is linear, which comes from the fact that the covariance matrices of the two competing distributions (views) are proportional to each other.

To illustrate the fact that this is not necessarily the case, consider another example using the same parameter values, except that the "second view" constituting the constraint has covariance matrix τI instead of $\tau\Sigma$.

This leads to the following figure, on which the curve connecting $\hat{\mu}$ and μ_{BL} are bending

```
λ_grid = np.linspace(.001, 20000, 1000)
curve = np.asarray([black_litterman(λ, μ_m, μ_est, Σ_est,
                                    τ * np.eye(N)).flatten() for λ in λ_grid])

λ_slider = FloatSlider(min=5, max=1500, step=100, value=200)

@interact(λ=λ_slider)
def decolletage(λ):
    dist_r_BL = stat.multivariate_normal(μ_m.squeeze(), Σ_est)
    dist_r_hat = stat.multivariate_normal(μ_est.squeeze(), τ * np.eye(N))

    X, Y = np.meshgrid(r1, r2)
    Z_BL = np.zeros((N_r1, N_r2))
```

(continues on next page)

(continued from previous page)

```
    Z_hat = np.zeros((N_r1, N_r2))

    for i in range(N_r1):
        for j in range(N_r2):
            Z_BL[i, j] = dist_r_BL.pdf(np.hstack([X[i, j], Y[i, j]]))
            Z_hat[i, j] = dist_r_hat.pdf(np.hstack([X[i, j], Y[i, j]]))

    μ_tilde = black_litterman(λ, μ_m, μ_est, Σ_est, τ * np.eye(N)).flatten()

    fig, ax = plt.subplots(figsize=(10, 6))
    ax.contourf(X, Y, Z_hat, cmap='viridis', alpha=.4)
    ax.contourf(X, Y, Z_BL, cmap='viridis', alpha=.4)
    ax.contour(X, Y, Z_BL, [dist_r_BL.pdf(μ_tilde)], cmap='viridis', alpha=.9)
    ax.contour(X, Y, Z_hat, [dist_r_hat.pdf(μ_tilde)], cmap='viridis', alpha=.9)
    ax.scatter(μ_est[0], μ_est[1])
    ax.scatter(μ_m[0], μ_m[1])

    ax.scatter(μ_tilde[0], μ_tilde[1], c='k', s=20*3)

    ax.plot(curve[:, 0], curve[:, 1], c='k')
    ax.axhline(0, c='k', alpha=.8)
    ax.axvline(0, c='k', alpha=.8)
    ax.set_xlabel(r'Excess return on the first asset, $r_{e, 1}$')
    ax.set_ylabel(r'Excess return on the second asset, $r_{e, 2}$')
    ax.text(μ_est[0] + 0.003, μ_est[1], r'$\hat{\mu}$')
    ax.text(μ_m[0] + 0.003, μ_m[1] + 0.005, r'$\mu_{BL}$')
    plt.show()
```

```
interactive(children=(FloatSlider(value=200.0, description='λ', max=1500.0, min=5.
↪0, step=100.0), Output()), _…
```

12.9 Black-Litterman Recommendation as Regularization

First, consider the OLS regression

$$\min_{\beta} \|X\beta - y\|^2$$

which yields the solution

$$\hat{\beta}_{OLS} = (X'X)^{-1}X'y$$

A common performance measure of estimators is the *mean squared error (MSE)*.

An estimator is "good" if its MSE is relatively small. Suppose that β_0 is the "true" value of the coefficient, then the MSE of the OLS estimator is

$$\text{mse}(\hat{\beta}_{OLS}, \beta_0) := \mathbb{E}\|\hat{\beta}_{OLS} - \beta_0\|^2 = \underbrace{\mathbb{E}\|\hat{\beta}_{OLS} - \mathbb{E}\beta_{OLS}\|^2}_{\text{variance}} + \underbrace{\|\mathbb{E}\hat{\beta}_{OLS} - \beta_0\|^2}_{\text{bias}}$$

From this decomposition, one can see that in order for the MSE to be small, both the bias and the variance terms must be small.

For example, consider the case when X is a T-vector of ones (where T is the sample size), so $\hat{\beta}_{OLS}$ is simply the sample average, while $\beta_0 \in \mathbb{R}$ is defined by the true mean of y.

In this example the MSE is

$$\text{mse}(\hat{\beta}_{OLS}, \beta_0) = \underbrace{\frac{1}{T^2}\mathbb{E}\left(\sum_{t=1}^{T}(y_t - \beta_0)\right)^2}_{\text{variance}} + \underbrace{0}_{\text{bias}}$$

However, because there is a trade-off between the estimator's bias and variance, there are cases when by permitting a small bias we can substantially reduce the variance so overall the MSE gets smaller.

A typical scenario when this proves to be useful is when the number of coefficients to be estimated is large relative to the sample size.

In these cases, one approach to handle the bias-variance trade-off is the so called *Tikhonov regularization*.

A general form with regularization matrix Γ can be written as

$$\min_{\beta}\left\{\|X\beta - y\|^2 + \|\Gamma(\beta - \tilde{\beta})\|^2\right\}$$

which yields the solution

$$\hat{\beta}_{Reg} = (X'X + \Gamma'\Gamma)^{-1}(X'y + \Gamma'\Gamma\tilde{\beta})$$

Substituting the value of $\hat{\beta}_{OLS}$ yields

$$\hat{\beta}_{Reg} = (X'X + \Gamma'\Gamma)^{-1}(X'X\hat{\beta}_{OLS} + \Gamma'\Gamma\tilde{\beta})$$

Often, the regularization matrix takes the form $\Gamma = \lambda I$ with $\lambda > 0$ and $\tilde{\beta} = \mathbf{0}$.

Then the Tikhonov regularization is equivalent to what is called *ridge regression* in statistics.

To illustrate how this estimator addresses the bias-variance trade-off, we compute the MSE of the ridge estimator

$$\text{mse}(\hat{\beta}_{\text{ridge}}, \beta_0) = \underbrace{\frac{1}{(T+\lambda)^2}\mathbb{E}\left(\sum_{t=1}^{T}(y_t - \beta_0)\right)^2}_{\text{variance}} + \underbrace{\left(\frac{\lambda}{T+\lambda}\right)^2\beta_0^2}_{\text{bias}}$$

The ridge regression shrinks the coefficients of the estimated vector towards zero relative to the OLS estimates thus reducing the variance term at the cost of introducing a "small" bias.

However, there is nothing special about the zero vector.

When $\tilde{\beta} \neq \mathbf{0}$ shrinkage occurs in the direction of $\tilde{\beta}$.

Now, we can give a regularization interpretation of the Black-Litterman portfolio recommendation.

To this end, first simplify the equation (12.4) that characterizes the Black-Litterman recommendation

$$\begin{aligned}\tilde{\mu} &= (\Sigma^{-1} + (\tau\Sigma)^{-1})^{-1}(\Sigma^{-1}\mu_{BL} + (\tau\Sigma)^{-1}\hat{\mu}) \\ &= (1+\tau^{-1})^{-1}\Sigma\Sigma^{-1}(\mu_{BL} + \tau^{-1}\hat{\mu}) \\ &= (1+\tau^{-1})^{-1}(\mu_{BL} + \tau^{-1}\hat{\mu})\end{aligned}$$

In our case, $\hat{\mu}$ is the estimated mean excess returns of securities. This could be written as a vector autoregression where

- y is the stacked vector of observed excess returns of size $(NT \times 1)$ – N securities and T observations.
- $X = \sqrt{T^{-1}}(I_N \otimes \iota_T)$ where I_N is the identity matrix and ι_T is a column vector of ones.

Correspondingly, the OLS regression of y on X would yield the mean excess returns as coefficients.

With $\Gamma = \sqrt{\tau T^{-1}}(I_N \otimes \iota_T)$ we can write the regularized version of the mean excess return estimation

$$\begin{aligned}\hat{\beta}_{Reg} &= (X'X + \Gamma'\Gamma)^{-1}(X'X\hat{\beta}_{OLS} + \Gamma'\Gamma\tilde{\beta}) \\ &= (1+\tau)^{-1}X'X(X'X)^{-1}(\hat{\beta}_{OLS} + \tau\tilde{\beta}) \\ &= (1+\tau)^{-1}(\hat{\beta}_{OLS} + \tau\tilde{\beta}) \\ &= (1+\tau^{-1})^{-1}(\tau^{-1}\hat{\beta}_{OLS} + \tilde{\beta})\end{aligned}$$

Given that $\hat{\beta}_{OLS} = \hat{\mu}$ and $\tilde{\beta} = \mu_{BL}$ in the Black-Litterman model, we have the following interpretation of the model's recommendation.

The estimated (personal) view of the mean excess returns, $\hat{\mu}$ that would lead to extreme short-long positions are "shrunk" towards the conservative market view, μ_{BL}, that leads to the more conservative market portfolio.

So the Black-Litterman procedure results in a recommendation that is a compromise between the conservative market portfolio and the more extreme portfolio that is implied by estimated "personal" views.

12.10 A Robust Control Operator

The Black-Litterman approach is partly inspired by the econometric insight that it is easier to estimate covariances of excess returns than the means.

That is what gave Black and Litterman license to adjust investors' perception of mean excess returns while not tampering with the covariance matrix of excess returns.

The robust control theory is another approach that also hinges on adjusting mean excess returns but not covariances.

Associated with a robust control problem is what Hansen and Sargent [HS01], [HS08] call a T operator.

Let's define the T operator as it applies to the problem at hand.

Let x be an $n \times 1$ Gaussian random vector with mean vector μ and covariance matrix $\Sigma = CC'$. This means that x can be represented as

$$x = \mu + C\epsilon$$

where $\epsilon \sim \mathcal{N}(0, I)$.

Let $\phi(\epsilon)$ denote the associated standardized Gaussian density.

Let $m(\epsilon, \mu)$ be a **likelihood ratio**, meaning that it satisfies

- $m(\epsilon, \mu) > 0$
- $\int m(\epsilon, \mu)\phi(\epsilon)d\epsilon = 1$

That is, $m(\epsilon, \mu)$ is a non-negative random variable with mean 1.

Multiplying $\phi(\epsilon)$ by the likelihood ratio $m(\epsilon, \mu)$ produces a distorted distribution for ϵ, namely

$$\tilde{\phi}(\epsilon) = m(\epsilon, \mu)\phi(\epsilon)$$

The next concept that we need is the **entropy** of the distorted distribution $\tilde{\phi}$ with respect to ϕ.

Entropy is defined as

$$\text{ent} = \int \log m(\epsilon, \mu) m(\epsilon, \mu)\phi(\epsilon)d\epsilon$$

or

$$\text{ent} = \int \log m(\epsilon, \mu)\tilde{\phi}(\epsilon)d\epsilon$$

That is, relative entropy is the expected value of the likelihood ratio m where the expectation is taken with respect to the twisted density $\tilde{\phi}$.

Relative entropy is non-negative. It is a measure of the discrepancy between two probability distributions.

As such, it plays an important role in governing the behavior of statistical tests designed to discriminate one probability distribution from another.

We are ready to define the T operator.

Let $V(x)$ be a value function.

Define

$$\begin{aligned}\mathsf{T}\left(V(x)\right) &= \min_{m(\epsilon,\mu)} \int m(\epsilon,\mu)[V(\mu + C\epsilon) + \theta \log m(\epsilon,\mu)]\phi(\epsilon)d\epsilon \\ &= -\log\theta \int \exp\left(\frac{-V(\mu + C\epsilon)}{\theta}\right)\phi(\epsilon)d\epsilon\end{aligned}$$

This asserts that T is an indirect utility function for a minimization problem in which an **adversary** chooses a distorted probability distribution $\tilde{\phi}$ to lower expected utility, subject to a penalty term that gets bigger the larger is relative entropy.

Here the penalty parameter

$$\theta \in [\underline{\theta}, +\infty]$$

is a robustness parameter when it is $+\infty$, there is no scope for the minimizing agent to distort the distribution, so no robustness to alternative distributions is acquired.

As θ is lowered, more robustness is achieved.

Note: The T operator is sometimes called a *risk-sensitivity* operator.

We shall apply T to the special case of a linear value function $w'(\vec{r} - r_f 1)$ where $\vec{r} - r_f 1 \sim \mathcal{N}(\mu, \Sigma)$ or $\vec{r} - r_f \mathbf{1} = \mu + C\epsilon$ and $\epsilon \sim \mathcal{N}(0, I)$.

The associated worst-case distribution of ϵ is Gaussian with mean $v = -\theta^{-1}C'w$ and covariance matrix I

(When the value function is affine, the worst-case distribution distorts the mean vector of ϵ but not the covariance matrix of ϵ).

For utility function argument $w'(\vec{r} - r_f 1)$

$$\mathsf{T}(\vec{r} - r_f \mathbf{1}) = w'\mu + \zeta - \frac{1}{2\theta}w'\Sigma w$$

and entropy is

$$\frac{v'v}{2} = \frac{1}{2\theta^2}w'CC'w$$

12.11 A Robust Mean-Variance Portfolio Model

According to criterion (12.1), the mean-variance portfolio choice problem chooses w to maximize

$$E[w(\vec{r} - r_f \mathbf{1})]] - \text{var}[w(\vec{r} - r_f \mathbf{1})]$$

which equals

$$w'\mu - \frac{\delta}{2} w' \Sigma w$$

A robust decision maker can be modeled as replacing the mean return $E[w(\vec{r} - r_f \mathbf{1})]$ with the risk-sensitive criterion

$$\mathsf{T}[w(\vec{r} - r_f \mathbf{1})] = w'\mu - \frac{1}{2\theta} w' \Sigma w$$

that comes from replacing the mean μ of $\vec{r} - r_f \mathbf{1}$ with the worst-case mean

$$\mu - \theta^{-1} \Sigma w$$

Notice how the worst-case mean vector depends on the portfolio w.

The operator T is the indirect utility function that emerges from solving a problem in which an agent who chooses probabilities does so in order to minimize the expected utility of a maximizing agent (in our case, the maximizing agent chooses portfolio weights w).

The robust version of the mean-variance portfolio choice problem is then to choose a portfolio w that maximizes

$$\mathsf{T}[w(\vec{r} - r_f \mathbf{1})] - \frac{\delta}{2} w' \Sigma w$$

or

$$w'(\mu - \theta^{-1} \Sigma w) - \frac{\delta}{2} w' \Sigma w \tag{12.7}$$

The minimizer of (12.7) is

$$w_{\text{rob}} = \frac{1}{\delta + \gamma} \Sigma^{-1} \mu$$

where $\gamma \equiv \theta^{-1}$ is sometimes called the risk-sensitivity parameter.

An increase in the risk-sensitivity parameter γ shrinks the portfolio weights toward zero in the same way that an increase in risk aversion does.

12.12 Appendix

We want to illustrate the "folk theorem" that with high or moderate frequency data, it is more difficult to estimate means than variances.

In order to operationalize this statement, we take two analog estimators:

- sample average: $\bar{X}_N = \frac{1}{N} \sum_{i=1}^N X_i$
- sample variance: $S_N = \frac{1}{N-1} \sum_{t=1}^N (X_i - \bar{X}_N)^2$

to estimate the unconditional mean and unconditional variance of the random variable X, respectively.

To measure the "difficulty of estimation", we use *mean squared error* (MSE), that is the average squared difference between the estimator and the true value.

Assuming that the process $\{X_i\}$is ergodic, both analog estimators are known to converge to their true values as the sample size N goes to infinity.

More precisely for all $\varepsilon > 0$

$$\lim_{N\to\infty} \quad P\left\{\left|\bar{X}_N - \mathbb{E}X\right| > \varepsilon\right\} = 0$$

and

$$\lim_{N\to\infty} \quad P\left\{\left|S_N - \mathbb{V}X\right| > \varepsilon\right\} = 0$$

A necessary condition for these convergence results is that the associated MSEs vanish as N goes to infinity, or in other words,

$$\text{MSE}(\bar{X}_N, \mathbb{E}X) = o(1) \qquad \text{and} \qquad \text{MSE}(S_N, \mathbb{V}X) = o(1)$$

Even if the MSEs converge to zero, the associated rates might be different. Looking at the limit of the *relative MSE* (as the sample size grows to infinity)

$$\frac{\text{MSE}(S_N, \mathbb{V}X)}{\text{MSE}(\bar{X}_N, \mathbb{E}X)} = \frac{o(1)}{o(1)} \underset{N\to\infty}{\to} B$$

can inform us about the relative (asymptotic) rates.

We will show that in general, with dependent data, the limit B depends on the sampling frequency.

In particular, we find that the rate of convergence of the variance estimator is less sensitive to increased sampling frequency than the rate of convergence of the mean estimator.

Hence, we can expect the relative asymptotic rate, B, to get smaller with higher frequency data, illustrating that "it is more difficult to estimate means than variances".

That is, we need significantly more data to obtain a given precision of the mean estimate than for our variance estimate.

12.13 Special Case – IID Sample

We start our analysis with the benchmark case of IID data.

Consider a sample of size N generated by the following IID process,

$$X_i \sim \mathcal{N}(\mu, \sigma^2)$$

Taking $\bar{X}_N$ to estimate the mean, the MSE is

$$\text{MSE}(\bar{X}_N, \mu) = \frac{\sigma^2}{N}$$

Taking S_N to estimate the variance, the MSE is

$$\text{MSE}(S_N, \sigma^2) = \frac{2\sigma^4}{N-1}$$

Both estimators are unbiased and hence the MSEs reflect the corresponding variances of the estimators.

Furthermore, both MSEs are $o(1)$ with a (multiplicative) factor of difference in their rates of convergence:

$$\frac{\text{MSE}(S_N, \sigma^2)}{\text{MSE}(\bar{X}_N, \mu)} = \frac{N2\sigma^2}{N-1} \quad \underset{N\to\infty}{\to} \quad 2\sigma^2$$

We are interested in how this (asymptotic) relative rate of convergence changes as increasing sampling frequency puts dependence into the data.

12.14 Dependence and Sampling Frequency

To investigate how sampling frequency affects relative rates of convergence, we assume that the data are generated by a mean-reverting continuous time process of the form

$$dX_t = -\kappa(X_t - \mu)dt + \sigma dW_t$$

where $\mu$is the unconditional mean, $\kappa > 0$ is a persistence parameter, and $\{W_t\}$ is a standardized Brownian motion.

Observations arising from this system in particular discrete periods $\mathcal{T}(h) \equiv \{nh : n \in \mathbb{Z}\}$with$h > 0$ can be described by the following process

$$X_{t+1} = (1 - \exp(-\kappa h))\mu + \exp(-\kappa h)X_t + \epsilon_{t,h}$$

where

$$\epsilon_{t,h} \sim \mathcal{N}(0, \Sigma_h) \quad \text{with} \quad \Sigma_h = \frac{\sigma^2(1 - \exp(-2\kappa h))}{2\kappa}$$

We call h the *frequency* parameter, whereas n represents the number of *lags* between observations.

Hence, the effective distance between two observations X_t and X_{t+n} in the discrete time notation is equal to $h \cdot n$ in terms of the underlying continuous time process.

Straightforward calculations show that the autocorrelation function for the stochastic process $\{X_t\}_{t\in\mathcal{T}(h)}$ is

$$\Gamma_h(n) \equiv \text{corr}(X_{t+hn}, X_t) = \exp(-\kappa hn)$$

and the auto-covariance function is

$$\gamma_h(n) \equiv \text{cov}(X_{t+hn}, X_t) = \frac{\exp(-\kappa hn)\sigma^2}{2\kappa}.$$

It follows that if $n = 0$, the unconditional variance is given by $\gamma_h(0) = \frac{\sigma^2}{2\kappa}$ irrespective of the sampling frequency.

The following figure illustrates how the dependence between the observations is related to the sampling frequency

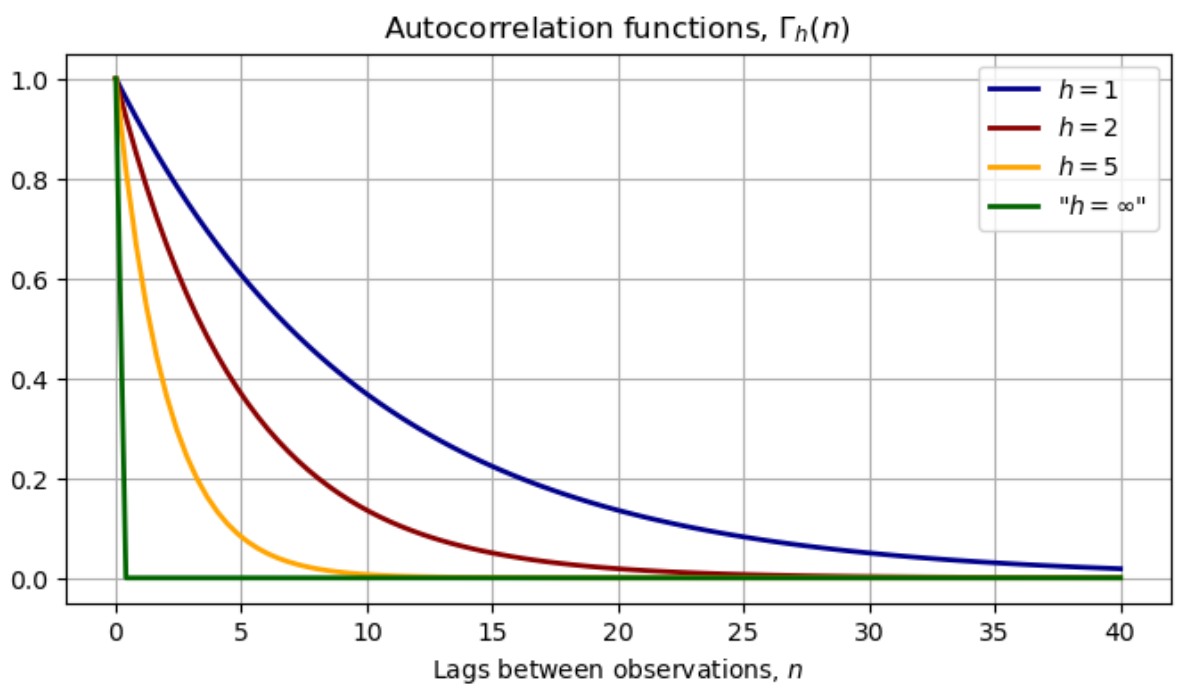
- For any given h, the autocorrelation converges to zero as we increase the distance – n– between the observations. This represents the "weak dependence" of the X process.
- Moreover, for a fixed lag length, n, the dependence vanishes as the sampling frequency goes to infinity. In fact, letting h go to ∞ gives back the case of IID data.

```
μ = .0
κ = .1
σ = .5
var_uncond = σ**2 / (2 * κ)

n_grid = np.linspace(0, 40, 100)
```

(continues on next page)

(continued from previous page)

```
autocorr_h1 = np.exp(-κ * n_grid * 1)
autocorr_h2 = np.exp(-κ * n_grid * 2)
autocorr_h5 = np.exp(-κ * n_grid * 5)
autocorr_h1000 = np.exp(-κ * n_grid * 1e8)

fig, ax = plt.subplots(figsize=(8, 4))
ax.plot(n_grid, autocorr_h1, label=r'$h=1$', c='darkblue', lw=2)
ax.plot(n_grid, autocorr_h2, label=r'$h=2$', c='darkred', lw=2)
ax.plot(n_grid, autocorr_h5, label=r'$h=5$', c='orange', lw=2)
ax.plot(n_grid, autocorr_h1000, label=r'"$h=\infty$"', c='darkgreen', lw=2)
ax.legend()
ax.grid()
ax.set(title=r'Autocorrelation functions, $\Gamma_h(n)$',
       xlabel=r'Lags between observations, $n$')
plt.show()
```



12.15 Frequency and the Mean Estimator

Consider again the AR(1) process generated by discrete sampling with frequency h. Assume that we have a sample of size N and we would like to estimate the unconditional mean – in our case the true mean is μ.

Again, the sample average is an unbiased estimator of the unconditional mean

$$\mathbb{E}[\bar{X}_N] = \frac{1}{N}\sum_{i=1}^{N}\mathbb{E}[X_i] = \mathbb{E}[X_0] = \mu$$

The variance of the sample mean is given by

$$
\begin{aligned}
\mathbb{V}\left(\bar{X}_N\right) &= \mathbb{V}\left(\frac{1}{N}\sum_{i=1}^{N} X_i\right) \\
&= \frac{1}{N^2}\left(\sum_{i=1}^{N}\mathbb{V}(X_i) + 2\sum_{i=1}^{N-1}\sum_{s=i+1}^{N}\operatorname{cov}(X_i, X_s)\right) \\
&= \frac{1}{N^2}\left(N\gamma(0) + 2\sum_{i=1}^{N-1} i\cdot\gamma\left(h\cdot(N-i)\right)\right) \\
&= \frac{1}{N^2}\left(N\frac{\sigma^2}{2\kappa} + 2\sum_{i=1}^{N-1} i\cdot\exp(-\kappa h(N-i))\frac{\sigma^2}{2\kappa}\right)
\end{aligned}
$$

It is explicit in the above equation that time dependence in the data inflates the variance of the mean estimator through the covariance terms.

Moreover, as we can see, a higher sampling frequency—smaller h—makes all the covariance terms larger, everything else being fixed.

This implies a relatively slower rate of convergence of the sample average for high-frequency data.

Intuitively, stronger dependence across observations for high-frequency data reduces the "information content" of each observation relative to the IID case.

We can upper bound the variance term in the following way

$$
\begin{aligned}
\mathbb{V}(\bar{X}_N) &= \frac{1}{N^2}\left(N\sigma^2 + 2\sum_{i=1}^{N-1} i\cdot\exp(-\kappa h(N-i))\sigma^2\right) \\
&\leq \frac{\sigma^2}{2\kappa N}\left(1 + 2\sum_{i=1}^{N-1}\cdot\exp(-\kappa h(i))\right) \\
&= \underbrace{\frac{\sigma^2}{2\kappa N}}_{\text{IID case}}\left(1 + 2\frac{1-\exp(-\kappa h)^{N-1}}{1-\exp(-\kappa h)}\right)
\end{aligned}
$$

Asymptotically, the term $\exp(-\kappa h)^{N-1}$ vanishes and the dependence in the data inflates the benchmark IID variance by a factor of

$$
\left(1 + 2\frac{1}{1-\exp(-\kappa h)}\right)
$$

This long run factor is larger the higher is the frequency (the smaller is h).

Therefore, we expect the asymptotic relative MSEs, B, to change with time-dependent data. We just saw that the mean estimator's rate is roughly changing by a factor of

$$
\left(1 + 2\frac{1}{1-\exp(-\kappa h)}\right)
$$

Unfortunately, the variance estimator's MSE is harder to derive.

Nonetheless, we can approximate it by using (large sample) simulations, thus getting an idea about how the asymptotic relative MSEs changes in the sampling frequency h relative to the IID case that we compute in closed form.

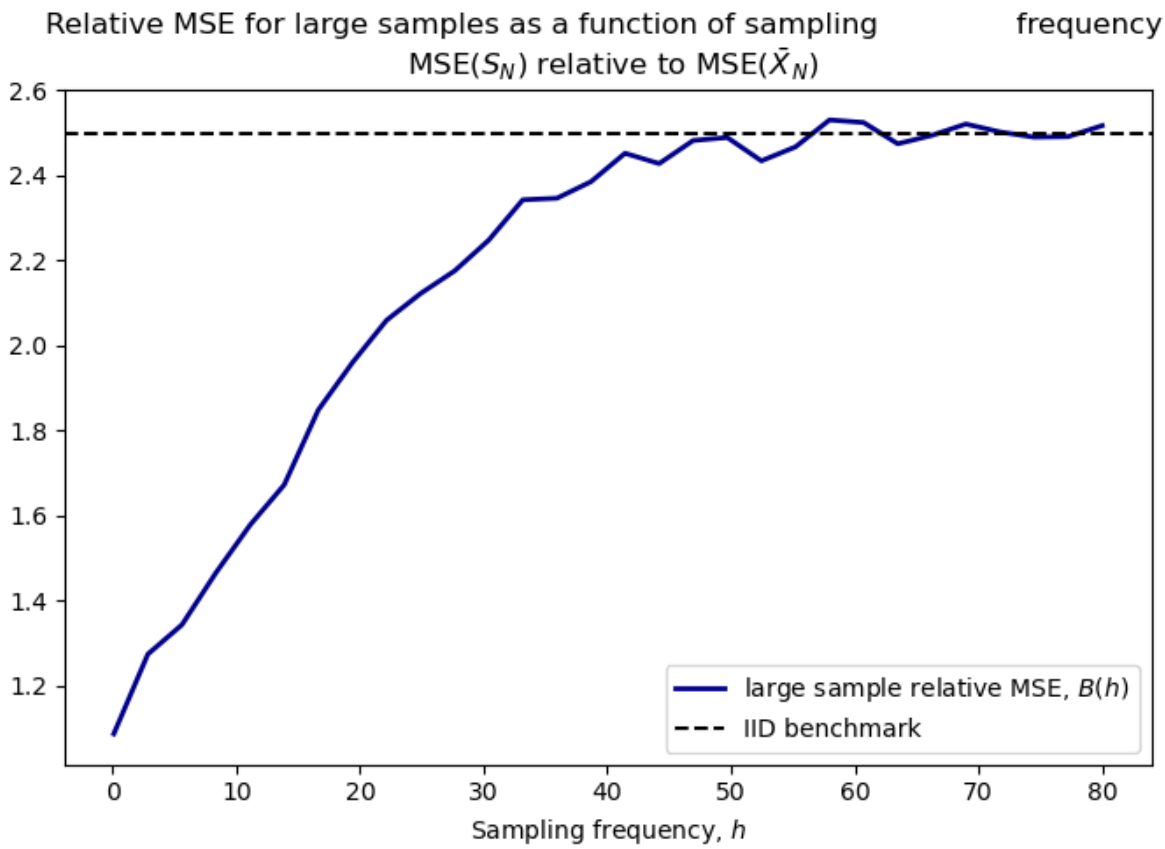
```
def sample_generator(h, N, M):
    ϕ = (1 - np.exp(-κ * h)) * μ
    ρ = np.exp(-κ * h)
```

(continues on next page)

(continued from previous page)

```
    s = σ**2 * (1 - np.exp(-2 * κ * h)) / (2 * κ)

    mean_uncond = μ
    std_uncond = np.sqrt(σ**2 / (2 * κ))

    ε_path = stat.norm(0, np.sqrt(s)).rvs((M, N))

    y_path = np.zeros((M, N + 1))
    y_path[:, 0] = stat.norm(mean_uncond, std_uncond).rvs(M)

    for i in range(N):
        y_path[:, i + 1] = ϕ + ρ * y_path[:, i] + ε_path[:, i]

    return y_path
```

```
# Generate large sample for different frequencies
N_app, M_app = 1000, 30000        # Sample size, number of simulations
h_grid = np.linspace(.1, 80, 30)

var_est_store = []
mean_est_store = []
labels = []

for h in h_grid:
    labels.append(h)
    sample = sample_generator(h, N_app, M_app)
    mean_est_store.append(np.mean(sample, 1))
    var_est_store.append(np.var(sample, 1))

var_est_store = np.array(var_est_store)
mean_est_store = np.array(mean_est_store)

# Save mse of estimators
mse_mean = np.var(mean_est_store, 1) + (np.mean(mean_est_store, 1) - μ)**2
mse_var = np.var(var_est_store, 1) \
          + (np.mean(var_est_store, 1) - var_uncond)**2

benchmark_rate = 2 * var_uncond       # IID case

# Relative MSE for large samples
rate_h = mse_var / mse_mean

fig, ax = plt.subplots(figsize=(8, 5))
ax.plot(h_grid, rate_h, c='darkblue', lw=2,
        label=r'large sample relative MSE, $B(h)$')
ax.axhline(benchmark_rate, c='k', ls='--', label=r'IID benchmark')
ax.set_title('Relative MSE for large samples as a function of sampling \
              frequency \n MSE($S_N$) relative to MSE($\\bar X_N$)')
ax.set_xlabel('Sampling frequency, $h$')
ax.legend()
plt.show()
```



The above figure illustrates the relationship between the asymptotic relative MSEs and the sampling frequency

- We can see that with low-frequency data – large values of h – the ratio of asymptotic rates approaches the IID case.
- As h gets smaller – the higher the frequency – the relative performance of the variance estimator is better in the sense that the ratio of asymptotic rates gets smaller. That is, as the time dependence gets more pronounced, the rate of convergence of the mean estimator's MSE deteriorates more than that of the variance estimator.

CHAPTER

THIRTEEN

IRRELEVANCE OF CAPITAL STRUCTURES WITH COMPLETE MARKETS

Contents

- *Irrelevance of Capital Structures with Complete Markets*
  - *Introduction*
  - *Competitive equilibrium*
  - *Code*

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

```
!pip install --upgrade quantecon
!pip install interpolation
!conda install -y -c plotly plotly plotly-orca
```

13.1 Introduction

This is a prolegomenon to another lecture *Equilibrium Capital Structures with Incomplete Markets* about a model with incomplete markets authored by Bisin, Clementi, and Gottardi [BCG18].

We adopt specifications of preferences and technologies very close to Bisin, Clemente, and Gottardi's but unlike them assume that there are complete markets in one-period Arrow securities.

This simplification of BCG's setup helps us by

- creating a benchmark economy to compare with outcomes in BCG's incomplete markets economy
- creating a good guess for initial values of some equilibrium objects to be computed in BCG's incomplete markets economy via an iterative algorithm
- illustrating classic complete markets outcomes that include
  - indeterminacy of consumers' portfolio choices
  - indeterminacy of firms' financial structures that underlies a Modigliani-Miller theorem [MM58]
- introducing `Big K, little k` issues in a simple context that will recur in the BCG incomplete markets environment

A Big K, little k analysis also played roles in this quantecon lecture as well as here and here.

13.1.1 Setup

The economy lasts for two periods, $t = 0, 1$.

There are two types of consumers named $i = 1, 2$.

A scalar random variable ϵ with probability density $g(\epsilon)$ affects both

- the return in period 1 from investing $k \geq 0$ in physical capital in period 0.
- exogenous period 1 endowments of the consumption good for agents of types $i = 1$ and $i = 2$.

Type $i = 1$ and $i = 2$ agents' period 1 endowments are correlated with the return on physical capital in different ways.

We discuss two arrangements:

- a command economy in which a benevolent planner chooses k and allocates goods to the two types of consumers in each period and each random second period state
- a competitive equilibrium with markets in claims on physical capital and a complete set (possibly a continuum) of one-period Arrow securities that pay period 1 consumption goods contingent on the realization of random variable ϵ.

13.1.2 Endowments

There is a single consumption good in period 0 and at each random state ϵ in period 1.

Economy-wide endowments in periods 0 and 1 are

$$\begin{aligned} w_0 \\ w_1(\epsilon) \text{ in state } \epsilon \end{aligned}$$

Soon we'll explain how aggregate endowments are divided between type $i = 1$ and type $i = 2$ consumers.

We don't need to do that in order to describe a social planning problem.

13.1.3 Technology:

Where $\alpha \in (0, 1)$ and $A > 0$

$$\begin{aligned} c_0^1 + c_0^2 + k &= w_0^1 + w_0^2 \\ c_1^1(\epsilon) + c_1^2(\epsilon) &= w_1^1(\epsilon) + w_1^2(\epsilon) + e^\epsilon A k^\alpha, \quad k \geq 0 \end{aligned}$$

13.1.4 Preferences:

A consumer of type i orders period 0 consumption c_0^i and state ϵ, period 1 consumption $c_1^i(\epsilon)$ by

$$u^i = u(c_0^i) + \beta \int u(c_1^i(\epsilon)) g(\epsilon) d\epsilon, \quad i = 1, 2$$

$\beta \in (0, 1)$ and the one-period utility function is

$$u(c) = \begin{cases} \frac{c^{1-\gamma}}{1-\gamma} & \text{if } \gamma \neq 1 \\ \log c & \text{if } \gamma = 1 \end{cases}$$

13.1.5 Parameterizations

Following BCG, we shall employ the following parameterizations:

$$\begin{aligned}
\epsilon &\sim \mathcal{N}(\mu, \sigma^2) \\
u(c) &= \frac{c^{1-\gamma}}{1-\gamma} \\
w_1^i(\epsilon) &= e^{-\chi_i \mu - .5\chi_i^2 \sigma^2 + \chi_i \epsilon}, \quad \chi_i \in [0,1]
\end{aligned}$$

Sometimes instead of asuming $\epsilon \sim g(\epsilon) = \mathcal{N}(0, \sigma^2)$, we'll assume that $g(\cdot)$ is a probability mass function that serves as a discrete approximation to a standardized normal density.

13.1.6 Pareto criterion and planning problem

The planner's objective function is

$$\text{obj} = \phi_1 u^1 + \phi_2 u^2, \quad \phi_i \geq 0, \quad \phi_1 + \phi_2 = 1$$

where $\phi_i \geq 0$ is a Pareto weight that the planner attaches to a consumer of type i.

We form the following Lagrangian for the planner's problem:

$$\begin{aligned}
L = &\sum_{i=1}^{2} \phi_i \left[u(c_0^i) + \beta \int u(c_1^i(\epsilon)) g(\epsilon) d\epsilon \right] \\
&+ \lambda_0 \left[w_0^1 + w_0^2 - k - c_0^1 - c_0^2 \right] \\
&+ \beta \int \lambda_1(\epsilon) \left[w_1^1(\epsilon) + w_1^2(\epsilon) + e^\epsilon A k^\alpha - c_1^1(\epsilon) - c_1^2(\epsilon) \right] g(\epsilon) d\epsilon
\end{aligned}$$

First-order necessary optimality conditions for the planning problem are:

$$\begin{aligned}
c_0^1: &\quad \phi_1 u'(c_0^1) - \lambda_0 = 0 \\
c_0^2: &\quad \phi_2 u'(c_0^2) - \lambda_0 = 0 \\
c_1^1(\epsilon): &\quad \phi_1 \beta u'(c_1^1(\epsilon)) g(\epsilon) - \beta \lambda_1(\epsilon) g(\epsilon) = 0 \\
c_1^2(\epsilon): &\quad \phi_2 \beta u'(c_1^2(\epsilon)) g(\epsilon) - \beta \lambda_1(\epsilon) g(\epsilon) = 0 \\
k: &\quad -\lambda_0 + \beta \alpha A k^{\alpha-1} \int \lambda_1(\epsilon) e^\epsilon g(\epsilon) d\epsilon = 0
\end{aligned}$$

The first four equations imply that

$$\begin{aligned}
\frac{u'(c_1^1(\epsilon))}{u'(c_0^1))} &= \frac{u'(c_1^2(\epsilon))}{u'(c_0^2))} = \frac{\lambda_1(\epsilon)}{\lambda_0} \\
\frac{u'(c_0^1)}{u'(c_0^2)} &= \frac{u'(c_1^1(\epsilon))}{u'(c_1^2(\epsilon))} = \frac{\phi_2}{\phi_1}
\end{aligned}$$

These together with the fifth first-order condition for the planner imply the following equation that determines an optimal choice of capital

$$1 = \beta \alpha A k^{\alpha-1} \int \frac{u'(c_1^i(\epsilon))}{u'(c_0^i)} e^\epsilon g(\epsilon) d\epsilon$$

for $i = 1, 2$.

13.1.7 Helpful observations and bookkeeping

Evidently,

$$u'(c) = c^{-\gamma}$$

and

$$\frac{u'(c^1)}{u'(c^2)} = \left(\frac{c^1}{c^2}\right)^{-\gamma} = \frac{\phi_2}{\phi_1}$$

where it is to be understood that this equation holds for $c^1 = c_0^1$ and $c^2 = c_0^2$ and also for $c^1 = c^1(\epsilon)$ and $c^2 = c^2(\epsilon)$ for all ϵ.

With the same understanding, it follows that

$$\left(\frac{c^1}{c^2}\right) = \left(\frac{\phi_2}{\phi_1}\right)^{-\gamma^{-1}}$$

Let $c = c^1 + c^2$.

It follows from the preceding equation that

$$\begin{aligned} c^1 &= \eta c \\ c^2 &= (1-\eta)c \end{aligned}$$

where $\eta \in [0,1]$ is a function of ϕ_1 and γ.

Consequently, we can write the planner's first-order condition for k as

$$1 = \beta\alpha A k^{\alpha-1} \int \left(\frac{w_1(\epsilon) + Ak^\alpha e^\epsilon}{w_0 - k}\right)^{-\gamma} e^\epsilon g(\epsilon) d\epsilon$$

which is one equation to be solved for $k \geq 0$.

Anticipating a `Big K, little k` idea widely used in macroeconomics, to be discussed in detail below, let K be the value of k that solves the preceding equation so that

$$1 = \beta\alpha A K^{\alpha-1} \int \left(\frac{w_1(\epsilon) + AK^\alpha e^\epsilon}{w_0 - K}\right)^{-\gamma} g(\epsilon) e^\epsilon d\epsilon \tag{13.1}$$

The associated optimal consumption allocation is

$$\begin{aligned} C_0 &= w_0 - K \\ C_1(\epsilon) &= w_1(\epsilon) + AK^\alpha e^\epsilon \\ c_0^1 &= \eta C_0 \\ c_0^2 &= (1-\eta) C_0 \\ c_1^1(\epsilon) &= \eta C_1(\epsilon) \\ c_1^2(\epsilon) &= (1-\eta) C_1(\epsilon) \end{aligned}$$

where $\eta \in [0,1]$ is the consumption share parameter mentioned above that is a function of the Pareto weight ϕ_1 and the utility curvature parameter γ.

Remarks

The relative Pareto weight parameter η does not appear in equation (13.1) that determines K.

Neither does it influence C_0 or $C_1(\epsilon)$, which depend solely on K.

The role of η is to determine how to allocate total consumption between the two types of consumers.

Thus, the planner's choice of K does not interact with how it wants to allocate consumption.

13.2 Competitive equilibrium

We now describe a competitive equilibrium for an economy that has specifications of consumer preferences, technology, and aggregate endowments that are identical to those in the preceding planning problem.

While prices do not appear in the planning problem – only quantities do – prices play an important role in a competitive equilibrium.

To understand how the planning economy is related to a competitive equilibrium, we now turn to the `Big K, little k` distinction.

13.2.1 Measures of agents and firms

We follow BCG in assuming that there are unit measures of

- consumers of type $i = 1$
- consumers of type $i = 2$
- firms with access to the production technology that converts k units of time 0 good into $Ak^\alpha e^\epsilon$ units of the time 1 good in random state ϵ

Thus, let $\omega \in [0, 1]$ index a particular consumer of type i.

Then define Big C^i as

$$C^i = \int_0^1 c^i(\omega) d\,\omega$$

In the same spirit, let $\zeta \in [0, 1]$ index a particular firm. Then define Big K as

$$K = \int_0^1 k(\zeta) d\,\zeta$$

The assumption that there are continua of our three types of agents plays an important role making each individual agent into a powerless **price taker**:

- an individual consumer chooses its own (infinesimal) part $c^i(\omega)$ of C^i taking prices as given
- an individual firm chooses its own (infinitesmimal) part $k(\zeta)$ of K taking prices as
- equilibrium prices depend on the `Big K, Big C` objects K and C

Nevertheless, in equilibrium, $K = k, C^i = c^i$

The assumption about measures of agents is thus a powerful device for making a host of competitive agents take as given equilibrium prices that are determined by the independent decisions of hosts of agents who behave just like they do.

Ownership

Consumers of type i own the following exogenous quantities of the consumption good in periods 0 and 1:

$$\begin{aligned} w_0^i, \quad & i = 1, 2 \\ w_1^i(\epsilon) \quad & i = 1, 2 \end{aligned}$$

where

$$\begin{aligned} \sum_i w_0^i &= w_0 \\ \sum_i w_1^i(\epsilon) &= w_1(\epsilon) \end{aligned}$$

Consumers also own shares in a firm that operates the technology for converting nonnegative amounts of the time 0 consumption good one-for-one into a capital good k that produces $Ak^\alpha e^\epsilon$ units of the time 1 consumption good in time 1 state ϵ.

Consumers of types $i = 1, 2$ are endowed with θ_0^i shares of a firm and

$$\theta_0^1 + \theta_0^2 = 1$$

Asset markets

At time 0, consumers trade the following assets with other consumers and with firms:

- equities (also known as stocks) issued by firms
- one-period Arrow securities that pay one unit of consumption at time 1 when the shock ϵ assumes a particular value

Later, we'll allow the firm to issue bonds too, but not now.

13.2.2 Objects appearing in a competitive equilibrium

Let

- $a^i(\epsilon)$ be consumer i 's purchases of claims on time 1 consumption in state ϵ
- $q(\epsilon)$ be a pricing kernel for one-period Arrow securities
- $\theta_0^i \geq 0$ be consumer i's intial share of the firm, $\sum_i \theta_0^i = 1$
- θ^i be the fraction of a firm's shares purchased by consumer i at time $t = 0$
- V be the value of the representative firm
- $\tilde{V}$ be the value of equity issued by the representative firm
- K, C_0 be two scalars and $C_1(\epsilon)$ a function that we use to construct a guess about an equilibrium pricing kernel for Arrow securities

We proceed to describe constrained optimum problems faced by consumers and a representative firm in a competitive equilibrium.

13.2.3 A representative firm's problem

A representative firm takes Arrow security prices $q(\epsilon)$ as given.

The firm purchases capital $k \geq 0$ from consumers at time 0 and finances itself by issuing equity at time 0.

The firm produces time 1 goods $Ak^\alpha e^\epsilon$ in state ϵ and pays all of these `earnings` to owners of its equity.

The value of a firm's equity at time 0 can be computed by multiplying its state-contingent earnings by their Arrow securities prices and then adding over all contingencies:

$$\tilde{V} = \int Ak^\alpha e^\epsilon q(\epsilon) d\epsilon$$

Owners of a firm want it to choose k to maximize

$$V = -k + \int Ak^\alpha e^\epsilon q(\epsilon) d\epsilon$$

The firm's first-order necessary condition for an optimal k is

$$-1 + \alpha A k^{\alpha-1} \int e^\epsilon q(\epsilon) d\epsilon = 0$$

The time 0 value of a representative firm is

$$V = -k + \tilde{V}$$

The right side equals the value of equity minus the cost of the time 0 goods that it purchases and uses as capital.

13.2.4 A consumer's problem

We now pose a consumer's problem in a competitive equilibrium.

As a price taker, each consumer faces a given Arrow securities pricing kernel $q(\epsilon)$, a given value of a firm V that has chosen capital stock k, a price of equity $\tilde{V}$, and prospective next period random dividends $Ak^\alpha e^\epsilon$.

If we evaluate consumer i's time 1 budget constraint at zero consumption $c_1^i(\epsilon) = 0$ and solve for $-a^i(\epsilon)$ we obtain

$$-\bar{a}^i(\epsilon; \theta^i) = w_1^i(\epsilon) + \theta^i A k^\alpha e^\epsilon \tag{13.2}$$

The quantity $-\bar{a}^i(\epsilon; \theta^i)$ is the maximum amount that it is feasible for consumer i to repay to his Arrow security creditors at time 1 in state ϵ.

Notice that $-\bar{a}^i(\epsilon; \theta^i)$ defined in (13.2) depends on

- his endowment $w_1^i(\epsilon)$ at time 1 in state ϵ
- his share θ^i of a representive firm's dividends

These constitute two sources of **collateral** that back the consumer's issues of Arrow securities that pay off in state ϵ

Consumer i chooses a scalar c_0^i and a function $c_1^i(\epsilon)$ to maximize

$$u(c_0^i) + \beta \int u(c_1^i(\epsilon)) g(\epsilon) d\epsilon$$

subject to time 0 and time 1 budget constraints

$$\begin{aligned} c_0^i &\leq w_0^i + \theta_0^i V - \int q(\epsilon) a^i(\epsilon) d\epsilon - \theta^i \tilde{V} \\ c_1^i(\epsilon) &\leq w_1^i(\epsilon) + \theta^i A k^\alpha e^\epsilon + a^i(\epsilon) \end{aligned}$$

Attach Lagrange multiplier λ_0^i to the budget constraint at time 0 and scaled Lagrange multiplier $\beta\lambda_1^i(\epsilon)g(\epsilon)$ to the budget constraint at time 1 and state ϵ, then form the Lagrangian

$$\begin{aligned} L^i &= u(c_0^i) + \beta \int u(c_1^i(\epsilon))g(\epsilon)d\epsilon \\ &+ \lambda_0^i[w_0^i + \theta_0^i - \int q(\epsilon)a^i(\epsilon)d\epsilon - \theta^i\tilde{V} - c_0^i] \\ &+ \beta \int \lambda_1^i(\epsilon)[w_1^i(\epsilon) + \theta^i Ak^\alpha e^\epsilon + a^i(\epsilon)c_1^i(\epsilon)]g(\epsilon)d\epsilon \end{aligned}$$

Off corners, first-order necessary conditions for an optimum with respect to c_0^i, $c_1^i(\epsilon)$, and $a^i(\epsilon)$ are

$$\begin{aligned} c_0^i: &\quad u'(c_0^i) - \lambda_0^i = 0 \\ c_1^i(\epsilon): &\quad \beta u'(c_1^i(\epsilon))g(\epsilon) - \beta\lambda_1^i(\epsilon)g(\epsilon) = 0 \\ a^i(\epsilon): &\quad -\lambda_0^i q(\epsilon) + \beta\lambda_1^i(\epsilon) = 0 \end{aligned}$$

These equations imply that consumer i adjusts its consumption plan to satisfy

$$q(\epsilon) = \beta\left(\frac{u'(c_1^i(\epsilon))}{u'(c_0^i)}\right)g(\epsilon) \tag{13.3}$$

To deduce a restriction on equilibrium prices, we solve the period 1 budget constraint to express $a^i(\epsilon)$ as

$$a^i(\epsilon) = c_1^i(\epsilon) - w_1^i(\epsilon) - \theta^i Ak^\alpha e^\epsilon$$

then substitute the expression on the right side into the time 0 budget constraint and rearrange to get the single intertemporal budget constraint

$$w_0^i + \theta_0^i V + \int w_1^i(\epsilon)q(\epsilon)d\epsilon + \theta^i\left[Ak^\alpha \int e^\epsilon q(\epsilon)d\epsilon - \tilde{V}\right] \geq c_0^i + \int c_1^i(\epsilon)q(\epsilon)d\epsilon \tag{13.4}$$

The right side of inequality (13.4) is the present value of consumer i's consumption while the left side is the present value of consumer i's endowment when consumer i buys θ^i shares of equity.

From inequality (13.4), we deduce two findings.

1. No arbitrage profits condition:

Unless

$$\tilde{V} = Ak^\alpha \int e^\epsilon q(\epsilon)d\epsilon \tag{13.5}$$

an **arbitrage** opportunity would be open.

If

$$\tilde{V} > Ak^\alpha \int e^\epsilon q(\epsilon)d\epsilon$$

the consumer could afford an arbitrarily high present value of consumption by setting θ^i to an arbitrarily large **negative** number.

If

$$\tilde{V} < Ak^\alpha \int e^\epsilon q(\epsilon)d\epsilon$$

the consumer could afford an arbitrarily high present value of consumption by setting θ^i to be arbitrarily large **positive** number.

Since resources are finite, there can exist no such arbitrage opportunity in a competitive equilibrium.

Therefore, it must be true that the following no arbitrage condition prevails:

$$\tilde{V} = \int Ak^{\alpha} e^{\epsilon} q(\epsilon; K) d\epsilon \tag{13.6}$$

Equation (13.6) asserts that the value of equity equals the value of the state-contingent dividends $Ak^{\alpha}e^{\epsilon}$ evaluated at the Arrow security prices $q(\epsilon; K)$ that we have expressed as a function of K.

We'll say more about this equation later.

2. Indeterminacy of portfolio

When the no-arbitrage pricing equation (13.6) prevails, a consumer of type i's choice θ^i of equity is indeterminate.

Consumer of type i can offset any choice of θ^i by setting an appropriate schedule $a^i(\epsilon)$ for purchasing state-contingent securities.

13.2.5 Computing competitive equilibrium prices and quantities

Having computed an allocation that solves the planning problem, we can readily compute a competitive equilibrium via the following steps that, as we'll see, relies heavily on the `Big K, little k`, `Big C, little c` logic mentioned earlier:

- a competitive equilbrium allocation equals the allocation chosen by the planner
- competitive equilibrium prices and the value of a firm's equity are encoded in shadow prices from the planning problem that depend on Big K and Big C.

To substantiate that this procedure is valid, we proceed as follows.

With K in hand, we make the following guess for competitive equilibrium Arrow securities prices

$$q(\epsilon; K) = \beta \left(\frac{u'\left(w_1(\epsilon) + AK^{\alpha}e^{\epsilon}\right)}{u'(w_0 - K)} \right)^{-\gamma} \tag{13.7}$$

To confirm the guess, we begin by considering its consequences for the firm's choice of k.

With Arrow securities prices (13.7), the firm's first-order necessary condition for choosing k becomes

$$-1 + \alpha A k^{\alpha - 1} \int e^{\epsilon} q(\epsilon; K) d\epsilon = 0 \tag{13.8}$$

which can be verified to be satisfied if the firm sets

$$k = K$$

because by setting $k = K$ equation (13.8) becomes equivalent with the planner's first-order condition (13.1) for setting K.

To pose a consumer's problem in a competitive equilibrium, we require not only the above guess for the Arrow securities pricing kernel $q(\epsilon)$ but the value of equity $\tilde{V}$:

$$\tilde{V} = \int AK^{\alpha} e^{\epsilon} q(\epsilon; K) d\epsilon \tag{13.9}$$

Let $\tilde{V}$ be the value of equity implied by Arrow securities price function (13.7) and formula (13.9).

At the Arrow securities prices $q(\epsilon)$ given by (13.7) and equity value $\tilde{V}$ given by (13.9), consumer $i = 1, 2$ choose consumption allocations and portolios that satisfy the first-order necessary conditions

$$\beta \left(\frac{u'(c_1^i(\epsilon))}{u'(c_0^i)} \right) g(\epsilon) = q(\epsilon; K)$$

It can be verified directly that the following choices satisfy these equations

$$
\begin{aligned}
c_0^1 + c_0^2 &= C_0 = w_0 - K \\
c_0^1(\epsilon) + c_0^2(\epsilon) &= C_1(\epsilon) = w_1(\epsilon) + Ak^\alpha e^\epsilon \\
\frac{c_1^2(\epsilon)}{c_1^1(\epsilon)} &= \frac{c_0^2}{c_0^1} = \frac{1-\eta}{\eta}
\end{aligned}
$$

for an $\eta \in (0,1)$ that depends on consumers' endowments $[w_0^1, w_0^2, w_1^1(\epsilon), w_1^2(\epsilon), \theta_0^1, \theta_0^2]$.

Remark: Multiple arrangements of endowments $[w_0^1, w_0^2, w_1^1(\epsilon), w_1^2(\epsilon), \theta_0^1, \theta_0^2]$ associated with the same distribution of wealth η. Can you explain why?

Hint: Think about the portfolio indeterminacy finding above.

13.2.6 Modigliani-Miller theorem

We now allow a firm to issue both bonds and equity.

Payouts from equity and bonds, respectively, are

$$
\begin{aligned}
d^e(k,b;\epsilon) &= \max\{e^\epsilon Ak^\alpha - b, 0\} \\
d^b(k,b;\epsilon) &= \min\left\{\frac{e^\epsilon Ak^\alpha}{b}, 1\right\}
\end{aligned}
$$

Thus, one unit of the bond pays one unit of consumption at time 1 in state ϵ if $Ak^\alpha e^\epsilon - b \geq 0$, which is true when $\epsilon \geq \epsilon^* = \log \frac{b}{Ak^\alpha}$, and pays $\frac{Ak^\alpha e^\epsilon}{b}$ units of time 1 consumption in state ϵ when $\epsilon < \epsilon^*$.

The value of the firm is now the sum of equity plus the value of bonds, which we denote

$$
\tilde{V} + bp(k,b)
$$

where $p(k,b)$ is the price of one unit of the bond when a firm with k units of physical capital issues b bonds.

We continue to assume that there are complete markets in Arrow securities with pricing kernel $q(\epsilon)$.

A version of the no-arbitrage-in-equilibrium argument that we presented earlier implies that the value of equity and the price of bonds are

$$
\begin{aligned}
\tilde{V} &= Ak^\alpha \int_{\epsilon^*}^{\infty} e^\epsilon q(\epsilon) d\epsilon - b \int_{\epsilon^*}^{\infty} q(\epsilon) d\epsilon \\
p(k,b) &= \frac{Ak^\alpha}{b} \int_{-\infty}^{\epsilon^*} e^\epsilon q(\epsilon) d\epsilon + \int_{\epsilon^*}^{\infty} q(\epsilon) d\epsilon
\end{aligned}
$$

Consequently, the value of the firm is

$$
\tilde{V} + p(k,b)b = Ak^\alpha \int_{-\infty}^{\infty} e^\epsilon q(\epsilon) d\epsilon,
$$

which is the same expression that we obtained above when we assumed that the firm issued only equity.

We thus obtain a version of the celebrated Modigliani-Miller theorem [MM58] about firms' finance:

Modigliani-Miller theorem:

- The value of a firm is independent the mix of equity and bonds that it uses to finance its physical capital.

- The firms's decision about how much physical capital to purchase does not depend on whether it finances those purchases by issuing bonds or equity
- The firm's choice of whether to finance itself by issuing equity or bonds is indeterminant

Please note the role of the assumption of complete markets in Arrow securities in substantiating these claims.

In *Equilibrium Capital Structures with Incomplete Markets*, we will assume that markets are (very) incomplete – we'll shut down markets in almost all Arrow securities.

That will pull the rug from underneath the Modigliani-Miller theorem.

13.3 Code

We create a class object `BCG_complete_markets` to compute equilibrium allocations of the complete market BCG model given a list of parameter values.

It consists of 4 functions that do the following things:

- `opt_k` computes the planner's optimal capital K
  - First, create a grid for capital.
  - Then for each value of capital stock in the grid, compute the left side of the planner's first-order necessary condition for k, that is,

$$\beta\alpha AK^{\alpha-1}\int\left(\frac{w_1(\epsilon)+AK^{\alpha}e^{\epsilon}}{w_0-K}\right)^{-\gamma}e^{\epsilon}g(\epsilon)d\epsilon-1=0$$

  - Find k that solves this equation.
- `q` computes Arrow security prices as a function of the productivity shock ϵ and capital K:

$$q(\epsilon;K)=\beta\left(\frac{u'\left(w_1(\epsilon)+AK^{\alpha}e^{\epsilon}\right)}{u'(w_0-K)}\right)$$

- `V` solves for the firm value given capital k:

$$V=-k+\int Ak^{\alpha}e^{\epsilon}q(\epsilon;K)d\epsilon$$

- `opt_c` computes optimal consumptions c_0^i, and $c^i(\epsilon)$:
  - The function first computes weight η using the budget constraint for agent 1:

$$w_0^1+\theta_0^1V+\int w_1^1(\epsilon)q(\epsilon)d\epsilon=c_0^1+\int c_1^1(\epsilon)q(\epsilon)d\epsilon=\eta\left(C_0+\int C_1(\epsilon)q(\epsilon)d\epsilon\right)$$

    where

$$\begin{aligned}C_0&=w_0-K\\C_1(\epsilon)&=w_1(\epsilon)+AK^{\alpha}e^{\epsilon}\end{aligned}$$

  - It computes consumption for each agent as

$$\begin{aligned}c_0^1&=\eta C_0\\c_0^2&=(1-\eta)C_0\\c_1^1(\epsilon)&=\eta C_1(\epsilon)\\c_1^2(\epsilon)&=(1-\eta)C_1(\epsilon)\end{aligned}$$

The list of parameters includes:

- χ_1, χ_2: Correlation parameters for agents 1 and 2. Default values are 0 and 0.9, respectively.
- w_0^1, w_0^2: Initial endowments. Default values are 1.
- θ_0^1, θ_0^2: Consumers' initial shares of a representative firm. Default values are 0.5.
- ψ: CRRA risk parameter. Default value is 3.
- α: Returns to scale production function parameter. Default value is 0.6.
- A: Productivity of technology. Default value is 2.5.
- μ, σ: Mean and standard deviation of the log of the shock. Default values are -0.025 and 0.4, respectively.
- β: time preference discount factor. Default value is .96.
- `nb_points_integ`: number of points used for integration through Gauss-Hermite quadrature: default value is 10

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm
from numba import njit, prange
from quantecon.optimize import root_finding
%matplotlib inline
```

```
#=========== Class: BCG for complete markets ===========#
class BCG_complete_markets:

    # init method or constructor
    def __init__(self,
                 ?1 = 0,
                 ?2 = 0.9,
                 w10 = 1,
                 w20 = 1,
                 ?10 = 0.5,
                 ?20 = 0.5,
                 ? = 3,
                 ? = 0.6,
                 A = 2.5,
                 ? = -0.025,
                 ? = 0.4,
                 ? = 0.96,
                 nb_points_integ = 10):

        #=========== Setup ===========#
        # Risk parameters
        self.?1 = ?1
        self.?2 = ?2

        # Other parameters
        self.? = ?
        self.? = ?
        self.A = A
        self.? = ?
        self.? = ?
        self.? = ?
```

(continues on next page)

(continued from previous page)

```
        # Utility
        self.u = lambda c: (c**(1-γ)) / (1-γ)

        # Production
        self.f = njit(lambda k: A * (k ** α))
        self.Y = lambda ξ, k: np.exp(ξ) * self.f(k)

        # Initial endowments
        self.w10 = w10
        self.w20 = w20
        self.w0 = w10 + w20

        # Initial holdings
        self.ξ10 = ξ10
        self.ξ20 = ξ20

        # Endowments at t=1
        w11 = njit(lambda ξ: np.exp(-ξ1*ξ - 0.5*(ξ1**2)*(ξ**2) + ξ1*ξ))
        w21 = njit(lambda ξ: np.exp(-ξ2*ξ - 0.5*(ξ2**2)*(ξ**2) + ξ2*ξ))
        self.w11 = w11
        self.w21 = w21

        self.w1 = njit(lambda ξ: w11(ξ) + w21(ξ))

        # Normal PDF
        self.g = lambda x: norm.pdf(x, loc=ξ, scale=ξ)

        # Integration
        x, self.weights = np.polynomial.hermite.hermgauss(nb_points_integ)
        self.points_integral = np.sqrt(2) * ξ * x + ξ

        self.k_foc = k_foc_factory(self)

    #=========== Optimal k ===========#
    # Function: solve for optimal k
    def opt_k(self, plot=False):
        w0 = self.w0

        # Grid for k
        kgrid = np.linspace(1e-4, w0-1e-4, 100)

        # get FONC values for each k in the grid
        kfoc_list = [];
        for k in kgrid:
            kfoc = self.k_foc(k, self.ξ1, self.ξ2)
            kfoc_list.append(kfoc)

        # Plot FONC for k
        if plot:
            fig, ax = plt.subplots(figsize=(8,7))
            ax.plot(kgrid, kfoc_list, color='blue', label=r'FONC for k')
            ax.axhline(0, color='red', linestyle='--')
            ax.legend()
            ax.set_xlabel(r'k')
            plt.show()
```

(continues on next page)

(continued from previous page)

```
        # Find k that solves the FONC
        kk = root_finding.newton_secant(self.k_foc, 1e-2, args=(self.𝜃1, self.𝜃2)).
↪root

        return kk

    #=========== Arrow security price ===========#
    # Function: Compute Arrow security price
    def q(self,𝜃,k):
        𝜃 = self.𝜃
        𝜃 = self.𝜃
        w0 = self.w0
        w1 = self.w1
        fk = self.f(k)
        g = self.g

        return 𝜃 * ((w1(𝜃) + np.exp(𝜃)*fk) / (w0 - k))**(-𝜃)


    #=========== Firm value V ===========#
    # Function: compute firm value V
    def V(self, k):
        q = self.q
        fk = self.f(k)
        weights = self.weights
        integ = lambda 𝜃: np.exp(𝜃) * fk * q(𝜃, k)

        return -k + np.sum(weights * integ(self.points_integral)) / np.sqrt(np.pi)

    #=========== Optimal c ===========#
    # Function: Compute optimal consumption choices c
    def opt_c(self, k=None, plot=False):
        w1 = self.w1
        w0 = self.w0
        w10 = self.w10
        w11 = self.w11
        𝜃10 = self.𝜃10
        Y = self.Y
        q = self.q
        V = self.V
        weights = self.weights

        if k is None:
            k = self.opt_k()

        # Solve for the ratio of consumption 𝜃 from the intertemporal B.C.
        fk = self.f(k)

        c1 = lambda 𝜃: (w1(𝜃) + np.exp(𝜃)*fk)*q(𝜃,k)
        denom = np.sum(weights * c1(self.points_integral)) / np.sqrt(np.pi) + (w0 - k)

        w11q = lambda 𝜃: w11(𝜃)*q(𝜃,k)
        num = w10 + 𝜃10 * V(k) + np.sum(weights * w11q(self.points_integral)) / np.
↪sqrt(np.pi)

        𝜃 = num / denom
```

(continues on next page)

(continued from previous page)

```
        # Consumption choices
        c10 = ⍰ * (w0 - k)
        c20 = (1-⍰) * (w0 - k)
        c11 = lambda ⍰: ⍰ * (w1(⍰)+Y(⍰,k))
        c21 = lambda ⍰: (1-⍰) * (w1(⍰)+Y(⍰,k))

        return c10, c20, c11, c21


def k_foc_factory(model):
    ⍰ = model.⍰
    f = model.f
    ⍰ = model.⍰
    ⍰ = model.⍰
    A = model.A
    ⍰ = model.⍰
    w0 = model.w0
    ⍰ = model.⍰
    ⍰ = model.⍰

    weights = model.weights
    points_integral = model.points_integral

    w11 = njit(lambda ⍰, ⍰1, : np.exp(-⍰1*⍰ - 0.5*(⍰1**2)*(⍰**2) + ⍰1*⍰))
    w21 = njit(lambda ⍰, ⍰2: np.exp(-⍰2*⍰ - 0.5*(⍰2**2)*(⍰**2) + ⍰2*⍰))
    w1 = njit(lambda ⍰, ⍰1, ⍰2: w11(⍰, ⍰1) + w21(⍰, ⍰2))

    @njit
    def integrand(⍰, ⍰1, ⍰2, k=1e-4):
        fk = f(k)
        return (w1(⍰, ⍰1, ⍰2) + np.exp(⍰) * fk) ** (-⍰) * np.exp(⍰)

    @njit
    def k_foc(k, ⍰1, ⍰2):
        int_k = np.sum(weights * integrand(points_integral, ⍰1, ⍰2, k=k)) / np.
↪sqrt(np.pi)

        mul = ⍰ * ⍰ * A * k ** (⍰ - 1) / ((w0 - k) ** (-⍰))
        val = mul * int_k - 1

        return val

    return k_foc
```

13.3.1 Examples

Below we provide some examples of how to use `BCG_complete markets`.

1st example

In the first example, we set up instances of BCG complete markets models.

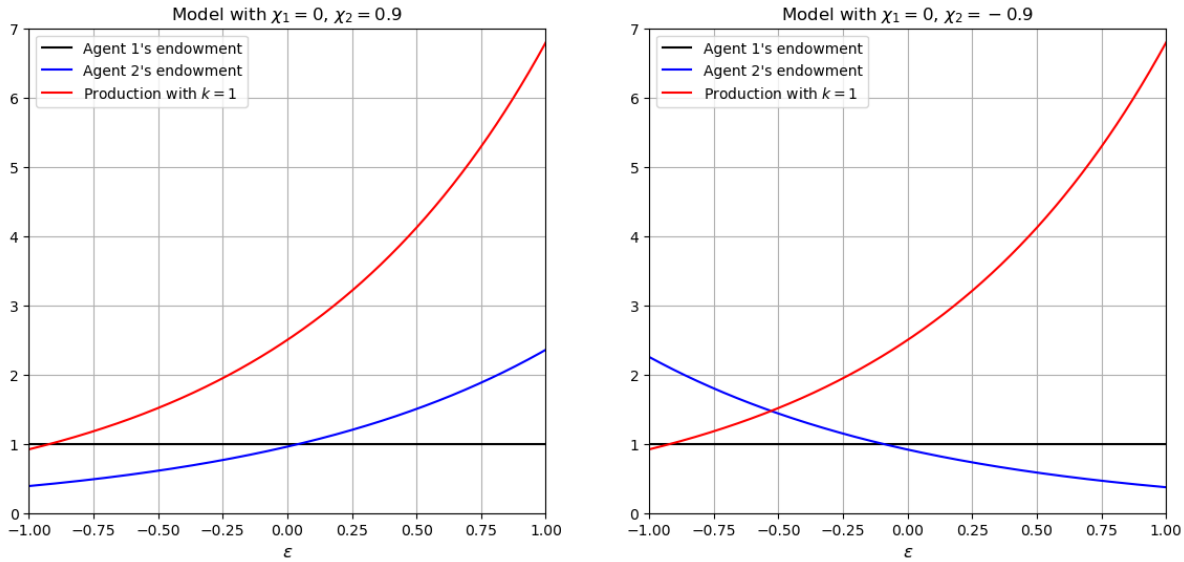
We can use either default parameter values or set parameter values as we want.

The two instances of the BCG complete markets model, `mdl1` and `mdl2`, represent the model with default parameter settings and with agent 2's income correlation altered to be $\chi_2 = -0.9$, respectively.

```
# Example: BCG model for complete markets
mdl1 = BCG_complete_markets()
mdl2 = BCG_complete_markets(χ2=-0.9)
```

Let's plot the agents' time-1 endowments with respect to shocks to see the difference in the two models:

```
#==== Figure 1: HH endowments and firm productivity ====#
# Realizations of innovation from -3 to 3
epsgrid = np.linspace(-1,1,1000)


fig, ax = plt.subplots(1,2,figsize=(14,6))
ax[0].plot(epsgrid, mdl1.w11(epsgrid), color='black', label='Agent 1\'s endowment')
ax[0].plot(epsgrid, mdl1.w21(epsgrid), color='blue', label='Agent 2\'s endowment')
ax[0].plot(epsgrid, mdl1.Y(epsgrid,1), color='red', label=r'Production with $k=1$')
ax[0].set_xlim([-1,1])
ax[0].set_ylim([0,7])
ax[0].set_xlabel(r'$\epsilon$',fontsize=12)
ax[0].set_title(r'Model with $\chi_1 = 0$, $\chi_2 = 0.9$')
ax[0].legend()
ax[0].grid()

ax[1].plot(epsgrid, mdl2.w11(epsgrid), color='black', label='Agent 1\'s endowment')
ax[1].plot(epsgrid, mdl2.w21(epsgrid), color='blue', label='Agent 2\'s endowment')
ax[1].plot(epsgrid, mdl2.Y(epsgrid,1), color='red', label=r'Production with $k=1$')
ax[1].set_xlim([-1,1])
ax[1].set_ylim([0,7])
ax[1].set_xlabel(r'$\epsilon$',fontsize=12)
ax[1].set_title(r'Model with $\chi_1 = 0$, $\chi_2 = -0.9$')
ax[1].legend()
ax[1].grid()

plt.show()
```



Let's also compare the optimal capital stock, k, and optimal time-0 consumption of agent 2, c_0^2, for the two models:

```
# Print optimal k
kk_1 = mdl1.opt_k()
kk_2 = mdl2.opt_k()

print('The optimal k for model 1: {:.5f}'.format(kk_1))
print('The optimal k for model 2: {:.5f}'.format(kk_2))

# Print optimal time-0 consumption for agent 2
c20_1 = mdl1.opt_c(k=kk_1)[1]
c20_2 = mdl2.opt_c(k=kk_2)[1]

print('The optimal c20 for model 1: {:.5f}'.format(c20_1))
print('The optimal c20 for model 2: {:.5f}'.format(c20_2))
```

```
The optimal k for model 1: 0.14235
The optimal k for model 2: 0.13791
```

```
The optimal c20 for model 1: 0.90205
The optimal c20 for model 2: 0.92862
```

2nd example

In the second example, we illustrate how the optimal choice of k is influenced by the correlation parameter χ_i.

We will need to install the `plotly` package for 3D illustration. See https://plotly.com/python/getting-started/ for further instructions.

```
# Mesh grid of 
N = 30
1grid, 2grid = np.meshgrid(np.linspace(-1,1,N),
                           np.linspace(-1,1,N))
```

(continues on next page)

(continued from previous page)

```
k_foc = k_foc_factory(mdl1)

# Create grid for k
kgrid = np.zeros_like(⍰1grid)

w0 = mdl1.w0

@njit(parallel=True)
def fill_k_grid(kgrid):
    # Loop: Compute optimal k and
    for i in prange(N):
        for j in prange(N):
            X1 = ⍰1grid[i, j]
            X2 = ⍰2grid[i, j]
            k = root_finding.newton_secant(k_foc, 1e-2, args=(X1, X2)).root
            kgrid[i, j] = k
```

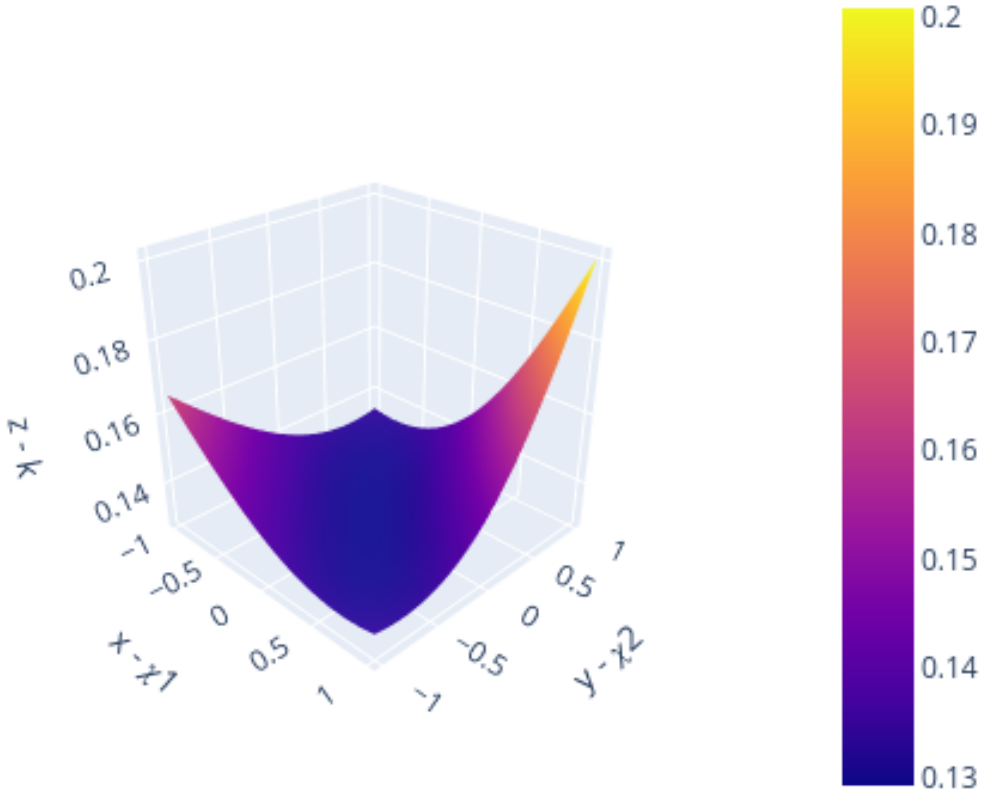
```
%%time
fill_k_grid(kgrid)
```

```
CPU times: user 4.5 s, sys: 111 ms, total: 4.62 s
Wall time: 4.61 s
```

```
%%time
# Second-run
fill_k_grid(kgrid)
```

```
CPU times: user 7.99 ms, sys: 0 ns, total: 7.99 ms
Wall time: 2.46 ms
```

```
#=== Example: Plot optimal k with different correlations ===#

from IPython.display import Image
# Import plotly
import plotly.graph_objs as go

# Plot optimal k
fig = go.Figure(data=[go.Surface(x=⍰1grid, y=⍰2grid, z=kgrid)])
fig.update_layout(scene = dict(xaxis_title='x - ⍰1',
                               yaxis_title='y - ⍰2',
                               zaxis_title='z - k',
                               aspectratio=dict(x=1,y=1,z=1)))
fig.update_layout(width=500,
                  height=500,
                  margin=dict(l=50, r=50, b=65, t=90))
fig.update_layout(scene_camera=dict(eye=dict(x=2, y=-2, z=1.5)))

# Export to PNG file
Image(fig.to_image(format="png"))
# fig.show() will provide interactive plot when running
# notebook locally
```


0.2
0.18
0.16
0.14
z - x
-1
-0.5
0
0.5
1
x - x1
-1
-0.5
0
0.5
1
y - x2
0.2
0.19
0.18
0.17
0.16
0.15
0.14
0.13

CHAPTER

FOURTEEN

EQUILIBRIUM CAPITAL STRUCTURES WITH INCOMPLETE MARKETS

Contents

- *Equilibrium Capital Structures with Incomplete Markets*
  - *Introduction*
  - *Asset Markets*
  - *Equilibrium verification*
  - *Pseudo Code*
  - *Code*
  - *Examples*
  - *A picture worth a thousand words*

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

```
!pip install --upgrade quantecon
!pip install interpolation
!conda install -y -c plotly plotly plotly-orca
```

14.1 Introduction

This is an extension of an earlier lecture *Irrelevance of Capital Structure with Complete Markets* about a **complete markets** model.

In contrast to that lecture, this one describes an instance of a model authored by Bisin, Clementi, and Gottardi [BCG18] in which financial markets are **incomplete**.

Instead of being able to trade equities and a full set of one-period Arrow securities as they can in *Irrelevance of Capital Structure with Complete Markets*, here consumers and firms trade only equity and a bond.

It is useful to watch how outcomes differ in the two settings.

In the complete markets economy in *Irrelevance of Capital Structure with Complete Markets*

- there is a unique stochastic discount factor that prices all assets
- consumers' portfolio choices are indeterminate

- firms' financial structures are indeterminate, so the model embodies an instance of a Modigliani-Miller irrelevance theorem [MM58]
- the aggregate of all firms' financial structures are indeterminate, a consequence of there being redundant assets

In the incomplete markets economy studied here

- there is a not a unique equilibrium stochastic discount factor
- different stochastic discount factors price different assets
- consumers' portfolio choices are determinate
- while **individual** firms' financial structures are indeterminate, thus conforming to part of a Modigliani-Miller theorem, [MM58], the **aggregate** of all firms' financial structures **is** determinate.

A `Big K, little k` analysis played an important role in the previous lecture *Irrelevance of Capital Structure with Complete Markets*.

A more subtle version of a `Big K, little k` features in the BCG incomplete markets environment here.

We use it to convey the heart of what BCG call a **rational conjectures** equilibrium in which conjectures are about equilibrium pricing functions in regions of the state space that an average consumer or firm does not visit in equilibrium.

Note that the absence of complete markets means that now we cannot compute competitive equilibrium prices and allocations by first solving the simple planning problem that we did in *Irrelevance of Capital Structure with Complete Markets*.

Instead, we compute an equilibrium by solving a system of simultaneous inequalities.

(Here we do not address the interesting question of whether there is a *different* planning problem that we could use to compute a competitive equlibrium allocation.)

14.1.1 Setup

We adopt specifications of preferences and technologies used by Bisin, Clemente, and Gottardi (2018) [BCG18] and in our earlier lecture on a complete markets version of their model.

The economy lasts for two periods, $t = 0, 1$.

There are two types of consumers named $i = 1, 2$.

A scalar random variable ϵ affects both

- a representative firm's physical return $f(k)e^{\epsilon}$ in period 1 from investing $k \geq 0$ in capital in period 0.
- period 1 endowments $w_1^i(\epsilon)$ of the consumption good for agents $i = 1$ and $i = 2$.

14.1.2 Ownership

A consumer of type i is endowed with w_0^i units of the time 0 good and $w_1^i(\epsilon)$ of the time 1 good when the random variable takes value ϵ.

At the start of period 0, a consumer of type i also owns θ_0^i shares of a representative firm.

14.1.3 Measures of agents and firms

As in the companion lecture *Irrelevance of Capital Structure with Complete Markets* that studies a complete markets version of the model, we follow BCG in assuming that there are unit measures of

- consumers of type $i = 1$
- consumers of type $i = 2$
- firms with access to a production technology that converts k units of time 0 good into $Ak^\alpha e^\epsilon$ units of the time 1 good in random state ϵ

Thus, let $\omega \in [0, 1]$ index a particular consumer of type i.

Then define Big C^i as

$$C^i = \int_0^1 c^i(\omega) d\,\omega$$

with components

$$C_0^i = \int_0^1 c_0^i(\omega) d\,\omega$$
$$C_1^i(\epsilon) = \int_0^1 c_1^i(\epsilon;\omega) d\,\omega$$

In the same spirit, let $\zeta \in [0, 1]$ index a particular firm and let firm ζ purchase $k(\zeta)$ units of capital and issue $b(\zeta)$ bonds.

Then define Big K and Big B as

$$K = \int_0^1 k(\zeta) d\,\zeta, \quad B = \int_0^1 b(\zeta) d\,\zeta$$

The assumption that there are equal measures of our three types of agents justifies our assumption that each individual agent is a powerless **price taker**:

- an individual consumer chooses its own (infinitesimal) part $c^i(\omega)$ of C^i taking prices as given
- an individual firm chooses its own (infinitesmimal) part $k(\zeta)$ of K and $b(\zeta)$ of B taking pricing functions as given
- However, equilibrium prices depend on the `Big K, Big B, Big C` objects K, B, and C

The assumption about measures of agents is a powerful device for making a host of competitive agents take as given the equilibrium prices that turn out to be determined by the decisions of hosts of agents who are just like them.

We call an equilibrium **symmetric** if

- all type i consumers choose the same consumption profiles so that $c^i(\omega) = C^i$ for all $\omega \in [0, 1]$
- all firms choose the same levels of k and b so that $k(\zeta) = K$, $b(\zeta) = B$ for all $\zeta \in [0, 1]$

In this lecture, we restrict ourselves to describing symmetric equilibria.

14.1.4 Endowments

Per capital economy-wide endowments in periods 0 and 1 are

$$\begin{aligned} w_0 &= w_0^1 + w_0^2 \\ w_1(\epsilon) &= w_1^1(\epsilon) + w_1^2(\epsilon) \text{ in state } \epsilon \end{aligned}$$

14.1.5 Feasibility:

Where $\alpha \in (0,1)$ and $A > 0$

$$\begin{aligned} C_0^1 + C_0^2 &= w_0^1 + w_0^2 - K \\ C_1^1(\epsilon) + C_1^2(\epsilon) &= w_1^1(\epsilon) + w_1^2(\epsilon) + e^{\epsilon} \int_0^1 f(k(\zeta))d\zeta, \quad k \geq 0 \end{aligned}$$

where $f(k) = Ak^{\alpha}, A > 0, \alpha \in (0,1)$.

14.1.6 Parameterizations

Following BCG, we shall employ the following parameterizations:

$$\begin{aligned} \epsilon &\sim \mathcal{N}(\mu, \sigma^2) \\ u(c) &= \frac{c^{1-\gamma}}{1-\gamma} \\ w_1^i(\epsilon) &= e^{-\chi_i \mu - .5\chi_i^2 \sigma^2 + \chi_i \epsilon}, \quad \chi_i \in [0,1] \end{aligned}$$

Sometimes instead of asuming $\epsilon \sim g(\epsilon) = \mathcal{N}(0, \sigma^2)$, we'll assume that $g(\cdot)$ is a probability mass function that serves as a discrete approximation to a standardized normal density.

14.1.7 Preferences:

A consumer of type i orders period 0 consumption c_0^i and state ϵ-period 1 consumption $c^i(\epsilon)$ by

$$u^i = u(c_0^i) + \beta \int u(c_1^i(\epsilon))g(\epsilon)d\epsilon, \quad i = 1,2$$

$\beta \in (0,1)$ and the one-period utility function is

$$u(c) = \begin{cases} \frac{c^{1-\gamma}}{1-\gamma} & \text{if } \gamma \neq 1 \\ \log c & \text{if } \gamma = 1 \end{cases}$$

14.1.8 Risk-sharing motives

The two types of agents' period 1 endowments have different correlations with the physical return on capital.

Endowment differences give agents incentives to trade risks that in the complete market version of the model showed up in their demands for equity and in their demands and supplies of one-period Arrow securities.

In the incomplete-markets setting under study here, these differences show up in differences in the two types of consumers' demands for a typical firm's bonds and equity, the only two assets that agents can now trade.

14.2 Asset Markets

Markets are incomplete: *ex cathedra* we the model builders declare that only equities and bonds issued by representative firms can be traded.

Let θ^i and ξ^i be a consumer of type i's post-trade holdings of equity and bonds, respectively.

A firm issues bonds promising to pay b units of consumption at time $t = 1$ and purchases k units of physical capital at time $t = 0$.

When $e^\epsilon Ak^\alpha < b$ at time 1, the firm defaults and its output is divided equally among bondholders.

Evidently, when the productivity shock $\epsilon < \epsilon^* = \log\left(\frac{b}{Ak^\alpha}\right)$, the firm defaults on its debt

Payoffs to equity and debt at date 1 as functions of the productivity shock ϵ are thus

$$\begin{aligned} d^e(k,b;\epsilon) &= \max\left\{e^\epsilon Ak^\alpha - b, 0\right\} \\ d^b(k,b;\epsilon) &= \min\left\{\frac{e^\epsilon Ak^\alpha}{b}, 1\right\} \end{aligned} \tag{14.1}$$

A firm faces a bond price function $p(k,b)$ when it issues b bonds and purchases k units of physical capital.

A firm's equity is worth $q(k,b)$ when it issues b bonds and purchases k units of physical capital.

A firm regards an equity-pricing function $q(k,b)$ and a bond pricing function $p(k,b)$ as exogenous in the sense that they are not affected by its choices of k and b.

Consumers face equilibrium prices $\breve{q}$ and $\breve{p}$ for bonds and equities, where $\breve{q}$ and $\breve{p}$ are both scalars.

Consumers are price takers and only need to know the scalars $\breve{q}, \breve{p}$.

Firms are *price function* takers and must know the functions $q(k,b), p(k,b)$ in order completely to pose their optimum problems.

14.2.1 Consumers

Each consumer of type i is endowed with w_0^i of the time 0 consumption good, $w_1^i(\epsilon)$ of the time 1, state ϵ consumption good and also owns a fraction $\theta_0^i \in (0,1)$ of the initial value of a representative firm, where $\theta_0^1 + \theta_0^2 = 1$.

The initial value of a representative firm is V (an object to be determined in a rational expectations equilibrium).

Consumer i buys θ^i shares of equity and buys bonds worth $\breve{p}\xi^i$ where $\breve{p}$ is the bond price.

Being a price-taker, a consumer takes V, $\breve{q}$, $\breve{p}$, and K, B as given.

Consumers know that equilibrium payoff functions for bonds and equities take the form

$$\begin{aligned} d^e(K,B;\epsilon) &= \max\left\{e^\epsilon AK^\alpha - B, 0\right\} \\ d^b(K,B;\epsilon) &= \min\left\{\frac{e^\epsilon AK^\alpha}{B}, 1\right\} \end{aligned}$$

Consumer i's optimization problem is

$$\begin{aligned} \max_{c_0^i, \theta^i, \xi^i, c_1^i(\epsilon)} \quad & u(c_0^i) + \beta \int u(c^i(\epsilon)) g(\epsilon)\, d\epsilon \\ \text{subject to} \quad & c_0^i = w_0^i + \theta_0^i V - \breve{q}\theta^i - \breve{p}\xi^i, \\ & c_1^i(\epsilon) = w_1^i(\epsilon) + \theta^i d^e(K,B;\epsilon) + \xi^i d^b(K,B;\epsilon) \;\forall\; \epsilon, \\ & \theta^i \geq 0, \xi^i \geq 0. \end{aligned}$$

The last two inequalities impose that the consumer cannot short sell either equity or bonds.

In a rational expectations equilibrium, $\check{q} = q(K, B)$ and $\check{p} = p(K, B)$

We form consumer i's Lagrangian:

$$\begin{aligned} L^i :=& u(c_0^i) + \beta \int u(c^i(\epsilon))g(\epsilon)\, d\epsilon \\ &+ \lambda_0^i [w_0^i + \theta_0 V - \check{q}\theta^i - \check{p}\xi^i - c_0^i] \\ &+ \beta \int \lambda_1^i(\epsilon) \left[w_1^i(\epsilon) + \theta^i d^e(K, B; \epsilon) + \xi^i d^b(K, B; \epsilon) - c_1^i(\epsilon)\right] g(\epsilon)\, d\epsilon \end{aligned}$$

Consumer i's first-order necessary conditions for an optimum include:

$$\begin{aligned} c_0^i : &\quad u'(c_0^i) = \lambda_0^i \\ c_1^i(\epsilon) : &\quad u'(c_1^i(\epsilon)) = \lambda_1^i(\epsilon) \\ \theta^i : &\quad \beta \int \lambda_1^i(\epsilon) d^e(K, B; \epsilon) g(\epsilon)\, d\epsilon \leq \lambda_0^i \check{q} \quad (= \text{ if } \theta^i > 0) \\ \xi^i : &\quad \beta \int \lambda_1^i(\epsilon) d^b(K, B; \epsilon) g(\epsilon)\, d\epsilon \leq \lambda_0^i \check{p} \quad (= \text{ if } b^i > 0) \end{aligned}$$

We can combine and rearrange consumer i's first-order conditions to become:

$$\begin{aligned} \check{q} &\geq \beta \int \frac{u'(c_1^i(\epsilon))}{u'(c_0^i)} d^e(K, B; \epsilon) g(\epsilon)\, d\epsilon \quad (= \text{ if } \theta^i > 0) \\ \check{p} &\geq \beta \int \frac{u'(c_1^i(\epsilon))}{u'(c_0^i)} d^b(K, B; \epsilon) g(\epsilon)\, d\epsilon \quad (= \text{ if } b^i > 0) \end{aligned}$$

These inequalities imply that in a symmetric rational expectations equilibrium consumption allocations and prices satisfy

$$\begin{aligned} \check{q} &= \max_i \beta \int \frac{u'(c_1^i(\epsilon))}{u'(c_0^i)} d^e(K, B; \epsilon) g(\epsilon)\, d\epsilon \\ \check{p} &= \max_i \beta \int \frac{u'(c_1^i(\epsilon))}{u'(c_0^i)} d^b(K, B; \epsilon) g(\epsilon)\, d\epsilon \end{aligned}$$

14.2.2 Pricing functions

When individual firms solve their optimization problems, they take big C^i's as fixed objects that they don't influence.

A representative firm faces a price function $q(k, b)$ for its equity and a price function $p(k, b)$ per unit of bonds that satisfy

$$\begin{aligned} q(k, b) &= \max_i \beta \int \frac{u'(C_1^i(\epsilon))}{u'(C_0^i)} d^e(k, b; \epsilon) g(\epsilon)\, d\epsilon \\ p(k, b) &= \max_i \beta \int \frac{u'(C_1^i(\epsilon))}{u'(C_0^i)} d^b(k, b; \epsilon) g(\epsilon)\, d\epsilon \end{aligned}$$

where the payoff functions are described by equations (14.1).

Notice the appearance of big C^i's on the right sides of these two equations that define equilibrium pricing functions.

The two price functions describe outcomes not only for equilibrium choices K, B of capital k and debt b, but also for any **out-of-equilibrium** pairs $(k, b) \neq (K, B)$.

The firm is assumed to know both price functions.

This means that the firm understands that its choice of k, b influences how markets price its equity and debt.

This package of assumptions is sometimes called **rational conjectures** (about price functions).

BCG give credit to Makowski for emphasizing and clarifying how rational conjectures are components of rational expectations equilibria.

14.2.3 Firms

The firm chooses capital k and debt b to maximize its market value:

$$V \equiv \max_{k,b} -k + q(k,b) + p(k,b)b$$

Attributing value maximization to the firm is a good idea because in equilibrium consumers of both types *want* a firm to maximize its value.

In the special quantitative examples studied here

- consumers of types $i = 1, 2$ both hold equity
- only consumers of type $i = 2$ hold debt; consumers of type $i = 1$ hold none.

These outcomes occur because we follow BCG and set parameters so that a type 2 consumer's stochastic endowment of the consumption good in period 1 is more correlated with the firm's output than is a type 1 consumer's.

This gives consumers of type 2 a motive to hedge their second period endowment risk by holding bonds (they also choose to hold some equity).

These outcomes mean that the pricing functions end up satisfying

$$\begin{aligned} q(k,b) &= \beta \int \frac{u'(C_1^1(\epsilon))}{u'(C_0^1)} d^e(k,b;\epsilon) g(\epsilon)\, d\epsilon = \beta \int \frac{u'(C_1^2(\epsilon))}{u'(C_0^2)} d^e(k,b;\epsilon) g(\epsilon)\, d\epsilon \\ p(k,b) &= \beta \int \frac{u'(C_1^2(\epsilon))}{u'(C_0^2)} d^b(k,b;\epsilon) g(\epsilon)\, d\epsilon \end{aligned}$$

Recall that $\epsilon^*(k,b) \equiv \log\left(\frac{b}{Ak^\alpha}\right)$ is a firm's default threshold.

We can rewrite the pricing functions as:

$$\begin{aligned} q(k,b) &= \beta \int_{\epsilon^*}^{\infty} \frac{u'(C_1^i(\epsilon))}{u'(C_0^i)} \left(e^\epsilon A k^\alpha - b\right) g(\epsilon)\, d\epsilon, \quad i = 1, 2 \\ p(k,b) &= \beta \int_{-\infty}^{\epsilon^*} \frac{u'(C_1^2(\epsilon))}{u'(C_0^2)} \left(\frac{e^\epsilon A k^\alpha}{b}\right) g(\epsilon)\, d\epsilon + \beta \int_{\epsilon^*}^{\infty} \frac{u'(C_1^2(\epsilon))}{u'(C_0^2)} g(\epsilon)\, d\epsilon \end{aligned}$$

Firm's optimization problem

The firm's optimization problem is

$$V \equiv \max_{k,b} \{-k + q(k,b) + p(k,b)b\}$$

The firm's first-order necessary conditions with respect to k and b, respectively, are

$$\begin{aligned} k: \quad & -1 + \frac{\partial q(k,b)}{\partial k} + b\frac{\partial p(q,b)}{\partial k} = 0 \\ b: \quad & \frac{\partial q(k,b)}{\partial b} + p(k,b) + b\frac{\partial p(k,b)}{\partial b} = 0 \end{aligned}$$

We use the Leibniz integral rule several times to arrive at the following derivatives:

$$\frac{\partial q(k,b)}{\partial k} = \beta \alpha A k^{\alpha-1} \int_{\epsilon^*}^{\infty} \frac{u'(C_1^i(\epsilon))}{u'(C_0^i)} e^\epsilon g(\epsilon) d\epsilon, \quad i = 1, 2$$

$$\frac{\partial q(k,b)}{\partial b} = -\beta \int_{\epsilon^*}^{\infty} \frac{u'(C_1^i(\epsilon))}{u'(C_0^i)} g(\epsilon) d\epsilon, \quad i = 1, 2$$

$$\frac{\partial p(k,b)}{\partial k} = \beta\alpha\frac{Ak^{\alpha-1}}{b}\int_{-\infty}^{\epsilon^*}\frac{u'(C_1^2(\epsilon))}{u'(C_0^2)}g(\epsilon)d\epsilon$$
$$\frac{\partial p(k,b)}{\partial b} = -\beta\frac{Ak^{\alpha}}{b^2}\int_{-\infty}^{\epsilon^*}\frac{u'(C_1^2(\epsilon))}{u'(C_0^2)}e^{\epsilon}g(\epsilon)d\epsilon$$

Special case: We confine ourselves to a special case in which both types of consumer hold positive equities so that $\frac{\partial q(k,b)}{\partial k}$ and $\frac{\partial q(k,b)}{\partial b}$ are related to rates of intertemporal substitution for both agents.

Substituting these partial derivatives into the above first-order conditions for k and b, respectively, we obtain the following versions of those first order conditions:

$$k: \quad -1 + \beta\alpha Ak^{\alpha-1}\int_{-\infty}^{\infty}\frac{u'(C_1^2(\epsilon))}{u'(C_0^2)}e^{\epsilon}g(\epsilon)d\epsilon = 0 \tag{14.2}$$

$$b: \quad \int_{\epsilon^*}^{\infty}\left(\frac{u'(C_1^1(\epsilon))}{u'(C_0^1)}\right)g(\epsilon)\,d\epsilon = \int_{\epsilon^*}^{\infty}\left(\frac{u'(C_1^2(\epsilon))}{u'(C_0^2)}\right)g(\epsilon)\,d\epsilon \tag{14.3}$$

where again recall that $\epsilon^*(k,b) \equiv \log\left(\frac{b}{Ak^{\alpha}}\right)$.

Taking $C_0^i, C_1^i(\epsilon)$ as given, these are two equations that we want to solve for the firm's optimal decisions k, b.

14.3 Equilibrium verification

On page 5 of BCG (2018), the authors say

If the price conjectures corresponding to the plan chosen by firms in equilibrium are correct, that is equal to the market prices $\check{q}$ and $\check{p}$, it is immediate to verify that the rationality of the conjecture coincides with the agents' Euler equations.

Here BCG are describing how they go about verifying that when they set little k, little b from the firm's first-order conditions equal to the big K, big B at the big C's that appear in the pricing functions, then

- consumers' Euler equations are satisfied if little c's are equated to Big C's
- firms' first-order necessary conditions for k, b are satisfied.
- $\check{q} = q(K,B)$ and $\check{p} = p(K,B)$.

14.4 Pseudo Code

Before displaying our Python code for computing a BCG incomplete markets equilibrium, we'll sketch some pseudo code that describes its logical flow.

Here goes:

1. Set upper and lower bounds for firm value as V_h and V_l, for capital as k_h and k_l, and for debt as b_h and b_l.
2. Conjecture firm value $V = \frac{1}{2}(V_h + V_l)$
3. Conjecture debt level $b = \frac{1}{2}(b_h + b_l)$.
4. Conjecture capital $k = \frac{1}{2}(k_h + k_l)$.
5. Compute the default threshold $\epsilon^* \equiv \log\left(\frac{b}{Ak^{\alpha}}\right)$.
6. (In this step we abuse notation by freezing V, k, b and in effect temporarily treating them as Big K, B values. Thus, in this step 6 little k, b are frozen at guessed at value of K, B.) Fixing the values of V, b and k, compute optimal choices of consumption c^i with consumers' FOCs. Assume that only agent 2 holds debt: $\xi^2 = b$ and that both agents hold equity: $0 < \theta^i < 1$ for $i = 1, 2$.

7. Set high and low bounds for equity holdings for agent 1 as θ_h^1 and θ_l^1. Guess $\theta^1 = \frac{1}{2}(\theta_h^1 + \theta_l^1)$, and $\theta^2 = 1 - \theta^1$. While $|\theta_h^1 - \theta_l^1|$ is large:
   - Compute agent 1's valuation of the equity claim with a fixed-point iteration:

     $q_1 = \beta \int \frac{u'(c_1^1(\epsilon))}{u'(c_0^1)} d^e(k, b; \epsilon) g(\epsilon)\ d\epsilon$

     where

     $c_1^1(\epsilon) = w_1^1(\epsilon) + \theta^1 d^e(k, b; \epsilon)$

     and

     $c_0^1 = w_0^1 + \theta_0^1 V - q_1 \theta^1$
   - Compute agent 2's valuation of the bond claim with a fixed-point iteration:

     $p = \beta \int \frac{u'(c_1^2(\epsilon))}{u'(c_0^2)} d^b(k, b; \epsilon) g(\epsilon)\ d\epsilon$

     where

     $c_1^2(\epsilon) = w_1^2(\epsilon) + \theta^2 d^e(k, b; \epsilon) + b$

     and

     $c_0^2 = w_0^2 + \theta_0^2 V - q_1 \theta^2 - pb$
   - Compute agent 2's valuation of the equity claim with a fixed-point iteration:

     $q_2 = \beta \int \frac{u'(c_1^2(\epsilon))}{u'(c_0^2)} d^e(k, b; \epsilon) g(\epsilon)\ d\epsilon$

     where

     $c_1^2(\epsilon) = w_1^2(\epsilon) + \theta^2 d^e(k, b; \epsilon) + b$

     and

     $c_0^2 = w_0^2 + \theta_0^2 V - q_2 \theta^2 - pb$
   - If $q_1 > q_2$, Set $\theta_l = \theta^1$; otherwise, set $\theta_h = \theta^1$.
   - Repeat steps 6Aa through 6Ad until $|\theta_h^1 - \theta_l^1|$ is small.
8. Set bond price as p and equity price as $q = \max(q_1, q_2)$.
9. Compute optimal choices of consumption:

$$
\begin{aligned}
c_0^1 &= w_0^1 + \theta_0^1 V - q\theta^1 \\
c_0^2 &= w_0^2 + \theta_0^2 V - q\theta^2 - pb \\
c_1^1(\epsilon) &= w_1^1(\epsilon) + \theta^1 d^e(k, b; \epsilon) \\
c_1^2(\epsilon) &= w_1^2(\epsilon) + \theta^2 d^e(k, b; \epsilon) + b
\end{aligned}
$$

10. (Here we confess to abusing notation again, but now in a different way. In step 7, we interpret frozen c^is as Big C^i. We do this to solve the firm's problem.) Fixing the values of c_0^i and $c_1^i(\epsilon)$, compute optimal choices of capital k and debt level b using the firm's first order necessary conditions.
11. Compute deviations from the firm's FONC for capital k as:

    $kfoc = \beta \alpha A k^{\alpha - 1} \left(\int \frac{u'(c_1^2(\epsilon))}{u'(c_0^2)} e^\epsilon g(\epsilon)\ d\epsilon \right) - 1$
    - If $kfoc > 0$, Set $k_l = k$; otherwise, set $k_h = k$.
    - Repeat steps 4 through 7A until $|k_h - k_l|$ is small.

12. Compute deviations from the firm's FONC for debt level b as:

    $bfoc = \beta \left[\int_{\epsilon^*}^{\infty} \left(\frac{u'(c_1^1(\epsilon))}{u'(c_0^1)} \right) g(\epsilon)\, d\epsilon - \int_{\epsilon^*}^{\infty} \left(\frac{u'(c_1^2(\epsilon))}{u'(c_0^2)} \right) g(\epsilon)\, d\epsilon \right]$

    - If $bfoc > 0$, Set $b_h = b$; otherwise, set $b_l = b$.
    - Repeat steps 3 through 7B until $|b_h - b_l|$ is small.
13. Given prices q and p from step 6, and the firm choices of k and b from step 7, compute the synthetic firm value:

    $V_x = -k + q + pb$

    - If $V_x > V$, then set $V_l = V$; otherwise, set $V_h = V$.
    - Repeat steps 1 through 8 until $|V_x - V|$ is small.
14. Ultimately, the algorithm returns equilibrium capital k^*, debt b^* and firm value V^*, as well as the following equilibrium values:

    - Equity holdings $\theta^{1,*} = \theta^1(k^*, b^*)$
    - Prices $q^* = q(k^*, b^*),\ p^* = p(k^*, b^*)$
    - Consumption plans $C_0^{1,*} = c_0^1(k^*, b^*),\ C_0^{2,*} = c_0^2(k^*, b^*),\ C_1^{1,*}(\epsilon) = c_1^1(k^*, b^*; \epsilon),\ C_1^{1,*}(\epsilon) = c_1^2(k^*, b^*; \epsilon)$.

14.5 Code

We create a Python class `BCG_incomplete_markets` to compute the equilibrium allocations of the incomplete market BCG model, given a set of parameter values.

The class includes the following methods, i.e., functions:

- `solve_eq`: solves the BCG model and returns the equilibrium values of capital k, debt b and firm value V, as well as
  - agent 1's equity holdings $\theta^{1,*}$
  - prices q^*, p^*
  - consumption plans $C_0^{1,*}, C_0^{2,*}, C_1^{1,*}(\epsilon), C_1^{2,*}(\epsilon)$.
- `eq_valuation`: inputs equilibrium consumpion plans C^* and outputs the following valuations for each pair of (k, b) in the grid:
  - the firm $V(k, b)$
  - the equity $q(k, b)$
  - the bond $p(k, b)$.

Parameters include:

- χ_1, χ_2: correlation parameter for agent 1 and 2. Default values are respectively 0 and 0.9.
- w_0^1, w_0^2: initial endowments. Default values are respectively 0.9 and 1.1.
- θ_0^1, θ_0^2: initial holding of the firm. Default values are 0.5.
- ψ: risk parameter. Default value is 3.
- α: Production function parameter. Default value is 0.6.
- A: Productivity of the firm. Default value is 2.5.
- μ, σ: Mean and standard deviation of the shock distribution. Default values are respectively -0.025 and 0.4

- β: Discount factor. Default value is 0.96.
- bound: Bound for truncated normal distribution. Default value is 3.

```
import numpy as np
from scipy.stats import truncnorm
from scipy.integrate import quad
from numba import njit
from interpolation import interp
```

```
/home/runner/miniconda3/envs/quantecon/lib/python3.11/site-packages/numba/core/
↪decorators.py:262: NumbaDeprecationWarning: numba.generated_jit is deprecated.␣
↪Please see the documentation at: https://numba.readthedocs.io/en/stable/
↪reference/deprecation.html#deprecation-of-generated-jit for more information and␣
↪advice on a suitable replacement.
  warnings.warn(msg, NumbaDeprecationWarning)
```

```
class BCG_incomplete_markets:

    # init method or constructor
    def __init__(self,
                 ⍰1 = 0,
                 ⍰2 = 0.9,
                 w10 = 0.9,
                 w20 = 1.1,
                 ⍰10 = 0.5,
                 ⍰20 = 0.5,
                 ⍰1 = 3,
                 ⍰2 = 3,
                 ⍰ = 0.6,
                 A = 2.5,
                 ⍰ = -0.025,
                 ⍰ = 0.4,
                 ⍰ = 0.96,
                 bound = 3,
                 Vl = 0,
                 Vh = 0.5,
                 kbot = 0.01,
                 #ktop = (⍰*A)**(1/(1-⍰)),
                 ktop = 0.25,
                 bbot = 0.1,
                 btop = 0.8):

        #=========== Setup ===========#
        # Risk parameters
        self.⍰1 = ⍰1
        self.⍰2 = ⍰2

        # Other parameters
        self.⍰1 = ⍰1
        self.⍰2 = ⍰2
        self.⍰ = ⍰
        self.A = A
        self.⍰ = ⍰
        self.⍰ = ⍰
        self.⍰ = ⍰
        self.bound = bound
```

(continues on next page)

(continued from previous page)

```
        # Bounds for firm value, capital, and debt
        self.Vl = Vl
        self.Vh = Vh
        self.kbot = kbot
        #self.kbot = (θ*A)**(1/(1-θ))
        self.ktop = ktop
        self.bbot = bbot
        self.btop = btop

        # Utility
        self.u = njit(lambda c: (c**(1-γ)) / (1-γ))

        # Initial endowments
        self.w10 = w10
        self.w20 = w20
        self.w0 = w10 + w20

        # Initial holdings
        self.θ10 = θ10
        self.θ20 = θ20

        # Endowments at t=1
        self.w11 = njit(lambda ε: np.exp(-σ1*ε - 0.5*(σ1**2)*(ε**2) + μ1*ε))
        self.w21 = njit(lambda ε: np.exp(-σ2*ε - 0.5*(σ2**2)*(ε**2) + μ2*ε))
        self.w1 = njit(lambda ε: self.w11(ε) + self.w21(ε))

        # Truncated normal
        ta, tb = (-bound - μ) / σ, (bound - μ) / σ
        rv = truncnorm(ta, tb, loc=μ, scale=σ)
        ε_range = np.linspace(ta, tb, 1000000)
        pdf_range = rv.pdf(ε_range)
        self.g = njit(lambda ε: interp(ε_range, pdf_range, ε))


    #*************************************************************
    # Function: Solve for equilibrium of the BCG model
    #*************************************************************
    def solve_eq(self, print_crit=True):

        # Load parameters
        σ1 = self.σ1
        σ2 = self.σ2
        γ = self.γ
        A = self.A
        θ = self.θ
        bound = self.bound
        Vl = self.Vl
        Vh = self.Vh
        kbot = self.kbot
        ktop = self.ktop
        bbot = self.bbot
        btop = self.btop
        w10 = self.w10
        w20 = self.w20
        θ10 = self.θ10
```

(continues on next page)

(continued from previous page)

```
        ⍰20 = self.⍰20
        w11 = self.w11
        w21 = self.w21
        g = self.g

        # We need to find a fixed point on the value of the firm
        V_crit = 1

        Y = njit(lambda ⍰, fk: np.exp(⍰)*fk)
        intqq1 = njit(lambda ⍰, fk, ⍰1, ⍰1, b: (w11(⍰) + ⍰1*(Y(⍰, fk) - b))**(-
↪⍰1)*(Y(⍰, fk) - b)*g(⍰))
        intp1 = njit(lambda ⍰, fk, ⍰2, b: (Y(⍰, fk)/b)*(w21(⍰) + Y(⍰, fk))**(-
↪⍰2)*g(⍰))
        intp2 = njit(lambda ⍰, fk, ⍰2, ⍰2, b: (w21(⍰) + ⍰2*(Y(⍰, fk)-b) + b)**(-
↪⍰2)*g(⍰))
        intqq2 = njit(lambda ⍰, fk, ⍰2, ⍰2, b: (w21(⍰) + ⍰2*(Y(⍰, fk)-b) + b)**(-
↪⍰2)*(Y(⍰, fk) - b)*g(⍰))
        intk1 = njit(lambda ⍰, fk, ⍰2: (w21(⍰) + Y(⍰, fk))**(-⍰2)*np.exp(⍰)*g(⍰))
        intk2 = njit(lambda ⍰, fk, ⍰2, ⍰2, b: (w21(⍰) + ⍰2*(Y(⍰, fk)-b) + b)**(-
↪⍰2)*np.exp(⍰)*g(⍰))
        intB1 = njit(lambda ⍰, fk, ⍰1, ⍰1, b: (w11(⍰) + ⍰1*(Y(⍰, fk) - b))**(-
↪⍰1)*g(⍰))
        intB2 = njit(lambda ⍰, fk, ⍰2, ⍰2, b: (w21(⍰) + ⍰2*(Y(⍰, fk) - b) + b)**(-
↪⍰2)*g(⍰))

        while V_crit>1e-4:

            # We begin by adding the guess for the value of the firm to endowment
            V = (Vl+Vh)/2
            ww10 = w10 + ⍰10*V
            ww20 = w20 + ⍰20*V

            # Figure out the optimal level of debt
            bl = bbot
            bh = btop
            b_crit=1

            while b_crit>1e-5:

                # Setting the conjecture for debt
                b = (bl+bh)/2

                # Figure out the optimal level of capital
                kl = kbot
                kh = ktop
                k_crit=1

                while k_crit>1e-5:

                    # Setting the conjecture for capital
                    k = (kl+kh)/2

                    # Production
                    fk = A*(k**⍰)
#                     Y = lambda ⍰: np.exp(⍰)*fk
```

(continues on next page)

(continued from previous page)

```
                    # Compute integration threshold
                    epstar = np.log(b/fk)


                    #*************************************************************
                    # Compute the prices and allocations consistent with consumers'
                    # Euler equations
                    #*************************************************************

                    # We impose the following:
                    # Agent 1 buys equity
                    # Agent 2 buys equity and all debt
                    # Agents trade such that prices converge

                    #========
                    # Agent 1
                    #========
                    # Holdings
                    θ1 = 0
                    θ1a = 0.3
                    θ1b = 1

                    while abs(θ1b - θ1a) > 0.001:

                        θ1 = (θ1a + θ1b) / 2

                        # qq1 is the equity price consistent with agent-1 Euler␣
 ↪Equation
                        ## Note: Price is in the date-0 budget constraint of the agent

                        ## First, compute the constant term that is not influenced by␣
 ↪q
                        ## that is, θE[u'(c^{1}_{1})d^{e}(k,B)]
#                         intqq1 = lambda θ: (w11(θ) + θ1*(Y(θ, fk) - b))**(-θ1)*(Y(θ,
 ↪ fk) - b)*g(θ)
#                         const_qq1 = θ * quad(intqq1,epstar,bound)[0]

                        const_qq1 = θ * quad(intqq1,epstar,bound, args=(fk, θ1, θ1,␣
 ↪b))[0]


                        ## Second, iterate to get the equity price q
                        qq1l = 0
                        qq1h = ww10
                        diff = 1
                        while diff > 1e-7:
                            qq1 = (qq1l+qq1h)/2
                            rhs = const_qq1/((ww10-qq1*θ1)**(-θ1));
                            if (rhs > qq1):
                                qq1l = qq1
                            else:
                                qq1h = qq1
                            diff = abs(qq1l-qq1h)

                        #========
                        # Agent 2
```

(continues on next page)

(continued from previous page)

```
                        #========
                        ⍰2 = b - ⍰1
                        ⍰2 = 1 - ⍰1

                        # p is the bond price consistent with agent-2 Euler Equation
                        ## Note: Price is in the date-0 budget constraint of the agent

                        ## First, compute the constant term that is not influenced by␣
 ↪p
                        ## that is, ⍰E[u'(c^{2}_{1})d^{b}(k,B)]
#                         intp1 = lambda ⍰: (Y(⍰, fk)/b)*(w21(⍰) + Y(⍰, fk))**(-
 ↪⍰2)*g(⍰)
#                         intp2 = lambda ⍰: (w21(⍰) + ⍰2*(Y(⍰, fk)-b) + b)**(-⍰2)*g(⍰)
#                         const_p = ⍰ * (quad(intp1,-bound,epstar)[0] + quad(intp2,
 ↪epstar,bound)[0])
                        const_p = ⍰ * (quad(intp1,-bound,epstar, args=(fk, ⍰2, b))[0]\
                                       + quad(intp2,epstar,bound, args=(fk, ⍰2, ⍰2,␣
 ↪b))[0])

                        ## iterate to get the bond price p
                        pl = 0
                        ph = ww20/b
                        diff = 1
                        while diff > 1e-7:
                            p = (pl+ph)/2
                            rhs = const_p/((ww20-qq1*⍰2-p*b)**(-⍰2))
                            if (rhs > p):
                                pl = p
                            else:
                                ph = p
                            diff = abs(pl-ph)

                        # qq2 is the equity price consistent with agent-2 Euler␣
 ↪Equation
#                         intqq2 = lambda ⍰: (w21(⍰) + ⍰2*(Y(⍰, fk)-b) + b)**(-
 ↪⍰2)*(Y(⍰, fk) - b)*g(⍰)
                        const_qq2 = ⍰ * quad(intqq2,epstar,bound, args=(fk, ⍰2, ⍰2,␣
 ↪b))[0]
                        qq2l = 0
                        qq2h = ww20
                        diff = 1
                        while diff > 1e-7:
                            qq2 = (qq2l+qq2h)/2
                            rhs = const_qq2/((ww20-qq2*⍰2-p*b)**(-⍰2));
                            if (rhs > qq2):
                                qq2l = qq2
                            else:
                                qq2h = qq2
                            diff = abs(qq2l-qq2h)

                        # q be the maximum valuation for the equity among agents
                        ## This will be the equity price based on Makowski's criterion
                        q = max(qq1,qq2)

                        #================
                        # Update holdings
```

(continues on next page)

(continued from previous page)

```
                        #================
                        if qq1 > qq2:
                            ⍰1a = ⍰1
                        else:
                            ⍰1b = ⍰1

                    #================
                    # Get consumption
                    #================
                    c10 = ww10 - q*⍰1
                    c11 = lambda ⍰: w11(⍰) + ⍰1*max(Y(⍰, fk)-b,0)
                    c20 = ww20 - q*(1-⍰1) - p*b
                    c21 = lambda ⍰: w21(⍰) + (1-⍰1)*max(Y(⍰, fk)-b,0) + min(Y(⍰, fk),
 ↪b)


                    #**************************************************
                    # Compute the first order conditions for the firm
                    #**************************************************

                    #===========
                    # Equity FOC
                    #===========
                    # Only agent 2's IMRS is relevent
#                     intk1 = lambda ⍰: (w21(⍰) + Y(⍰, fk))**(-⍰2)*np.exp(⍰)*g(⍰)
#                     intk2 = lambda ⍰: (w21(⍰) + ⍰2*(Y(⍰, fk)-b) + b)**(-⍰2)*np.
 ↪exp(⍰)*g(⍰)
#                     kfoc_num = quad(intk1,-bound,epstar)[0] + quad(intk2,epstar,
 ↪bound)[0]
                    kfoc_num = quad(intk1,-bound,epstar, args=(fk, ⍰2))[0] +␣
 ↪quad(intk2,epstar,bound, args=(fk, ⍰2, ⍰2, b))[0]
                    kfoc_denom = (ww20- q*⍰2 - p*b)**(-⍰2)
                    kfoc = ⍰*⍰*A*(k**(⍰-1))*(kfoc_num/kfoc_denom) - 1

                    if (kfoc > 0):
                        kl = k
                    else:
                        kh = k
                    k_crit = abs(kh-kl)

                    if print_crit:
                        print("critical value of k: {:.5f}".format(k_crit))


                #=========
                # Bond FOC
                #=========
#                   intB1 = lambda ⍰: (w11(⍰) + ⍰1*(Y(⍰, fk) - b))**(-⍰1)*g(⍰)
#                   intB2 = lambda ⍰: (w21(⍰) + ⍰2*(Y(⍰, fk) - b) + b)**(-⍰2)*g(⍰)

#                   bfoc1 = quad(intB1,epstar,bound)[0] / (ww10 - q*⍰1)**(-⍰1)
#                   bfoc2 = quad(intB2,epstar,bound)[0] / (ww20 - q*⍰2 - p*b)**(-⍰2)

                bfoc1 = quad(intB1,epstar,bound, args=(fk, ⍰1, ⍰1, b))[0] / (ww10 -␣
 ↪q*⍰1)**(-⍰1)
                bfoc2 = quad(intB2,epstar,bound, args=(fk, ⍰2, ⍰2, b))[0] / (ww20 -␣
 ↪q*⍰2 - p*b)**(-⍰2)
```

(continues on next page)

(continued from previous page)

```
            bfoc = bfoc1 - bfoc2

            if (bfoc > 0):
                bh = b
            else:
                bl = b
            b_crit = abs(bh-bl)

            if print_crit:
                print("#=== critical value of b: {:.5f}".format(b_crit))

        # Compute the value of the firm
        value_x = -k + q + p*b
        if (value_x > V):
             Vl = V
        else:
             Vh = V
        V_crit = abs(value_x-V)

        if print_crit:
            print("#====== critical value of V: {:.5f}".format(V_crit))

        print('k,b,p,q,kfoc,bfoc,epstar,V,V_crit')
        formattedList = ["%.3f" % member for member in [k,
                                                        b,
                                                        p,
                                                        q,
                                                        kfoc,
                                                        bfoc,
                                                        epstar,
                                                        V,
                                                        V_crit]]
        print(formattedList)

    #*********************************
    # Equilibrium values
    #*********************************

    # Return the results
    kss = k
    bss = b
    Vss = V
    qss = q
    pss = p
    c10ss = c10
    c11ss = c11
    c20ss = c20
    c21ss = c21
    ⍰1ss = ⍰1


    # Print the results
    print('finished')
   # print('k,b,p,q,kfoc,bfoc,epstar,V,V_crit')
    #formattedList = ["%.3f" % member for member in [kss,
    #                                                bss,
```

(continues on next page)

(continued from previous page)

```
        #                                                   pss,
        #                                                   qss,
        #                                                   kfoc,
        #                                                   bfoc,
        #                                                   epstar,
        #                                                   Vss,
        #                                                   V_crit]]
        #print(formattedList)

        return kss,bss,Vss,qss,pss,c10ss,c11ss,c20ss,c21ss,𝜃1ss


    #*************************************************************
    # Function: Equity and bond valuations by different agents
    #*************************************************************
    def valuations_by_agent(self,
                            c10, c11, c20, c21,
                            k, b):

        # Load parameters
        𝜃1 = self.𝜃1
        𝜃2 = self.𝜃2
        𝜃 = self.𝜃
        A = self.A
        𝜌 = self.𝜌
        bound = self.bound
        Vl = self.Vl
        Vh = self.Vh
        kbot = self.kbot
        ktop = self.ktop
        bbot = self.bbot
        btop = self.btop
        w10 = self.w10
        w20 = self.w20
        𝜆10 = self.𝜆10
        𝜆20 = self.𝜆20
        w11 = self.w11
        w21 = self.w21
        g = self.g

        # Get functions for IMRS/state price density
        IMRS1 = lambda 𝜀: 𝜌 * (c11(𝜀)/c10)**(-𝜃1)*g(𝜀)
        IMRS2 = lambda 𝜀: 𝜌 * (c21(𝜀)/c20)**(-𝜃2)*g(𝜀)

        # Production
        fk = A*(k**𝜃)
        Y = lambda 𝜀: np.exp(𝜀)*fk

        # Compute integration threshold
        epstar = np.log(b/fk)

        # Compute equity valuation with agent 1's IMRS
        intQ1 = lambda 𝜀: IMRS1(𝜀)*(Y(𝜀) - b)
        Q1 = quad(intQ1, epstar, bound)[0]

        # Compute bond valuation with agent 1's IMRS
```

(continues on next page)

(continued from previous page)

```
        intP1 = lambda □: IMRS1(□)*Y(□)/b
        P1 = quad(intP1, -bound, epstar)[0] + quad(IMRS1, epstar, bound)[0]

        # Compute equity valuation with agent 2's IMRS
        intQ2 = lambda □: IMRS2(□)*(Y(□) - b)
        Q2 = quad(intQ2, epstar, bound)[0]

        # Compute bond valuation with agent 2's IMRS
        intP2 = lambda □: IMRS2(□)*Y(□)/b
        P2 = quad(intP2, -bound, epstar)[0] + quad(IMRS2, epstar, bound)[0]

        return Q1,Q2,P1,P2


    #*************************************************************
    # Function: equilibrium valuations for firm, equity, bond
    #*************************************************************
    def eq_valuation(self, c10, c11, c20, c21, N=30):

        # Load parameters
        □1 = self.□1
        □2 = self.□2
        □ = self.□
        A = self.A
        □ = self.□
        bound = self.bound
        Vl = self.Vl
        Vh = self.Vh
        kbot = self.kbot
        ktop = self.ktop
        bbot = self.bbot
        btop = self.btop
        w10 = self.w10
        w20 = self.w20
        □10 = self.□10
        □20 = self.□20
        w11 = self.w11
        w21 = self.w21
        g = self.g

        # Create grids
        kgrid, bgrid = np.meshgrid(np.linspace(kbot,ktop,N),
                                   np.linspace(bbot,btop,N))
        Vgrid = np.zeros_like(kgrid)
        Qgrid = np.zeros_like(kgrid)
        Pgrid = np.zeros_like(kgrid)

        # Loop: firm value
        for i in range(N):
            for j in range(N):

                # Get capital and debt
                k = kgrid[i,j]
                b = bgrid[i,j]

                # Valuations by each agent
```

(continues on next page)

(continued from previous page)

```
                Q1,Q2,P1,P2 = self.valuations_by_agent(c10,
                                                       c11,
                                                       c20,
                                                       c21,
                                                       k,
                                                       b)

                # The prices will be the maximum of the valuations
                Q = max(Q1,Q2)
                P = max(P1,P2)

                # Compute firm value
                V = -k + Q + P*b
                Vgrid[i,j] = V
                Qgrid[i,j] = Q
                Pgrid[i,j] = P

        return kgrid, bgrid, Vgrid, Qgrid, Pgrid
```

14.6 Examples

Below we show some examples computed with the class `BCG_incomplete markets`.

14.6.1 First example

In the first example, we set up an instance of the BCG incomplete markets model with default parameter values.

```
mdl = BCG_incomplete_markets()
kss,bss,Vss,qss,pss,c10ss,c11ss,c20ss,c21ss,?1ss = mdl.solve_eq(print_crit=False)
```

```
print(-kss+qss+pss*bss)
print(Vss)
print(?1ss)
```

```
0.10073912888808995
0.100830078125
0.98564453125
```

Python reports to us that the equilibrium firm value is $V = 0.101$, with capital $k = 0.151$ and debt $b = 0.484$.

Let's verify some things that have to be true if our algorithm has truly found an equilibrium.

Thus, let's see if the firm is actually maximizing its firm value given the equilibrium pricing function $q(k, b)$ for equity and $p(k, b)$ for bonds.

```
kgrid, bgrid, Vgrid, Qgrid, Pgrid = mdl.eq_valuation(c10ss, c11ss, c20ss, c21ss,N=30)

print('Maximum valuation of the firm value in the (k,B) grid: {:.5f}'.format(Vgrid.
 ↪max()))
print('Equilibrium firm value: {:.5f}'.format(Vss))
```

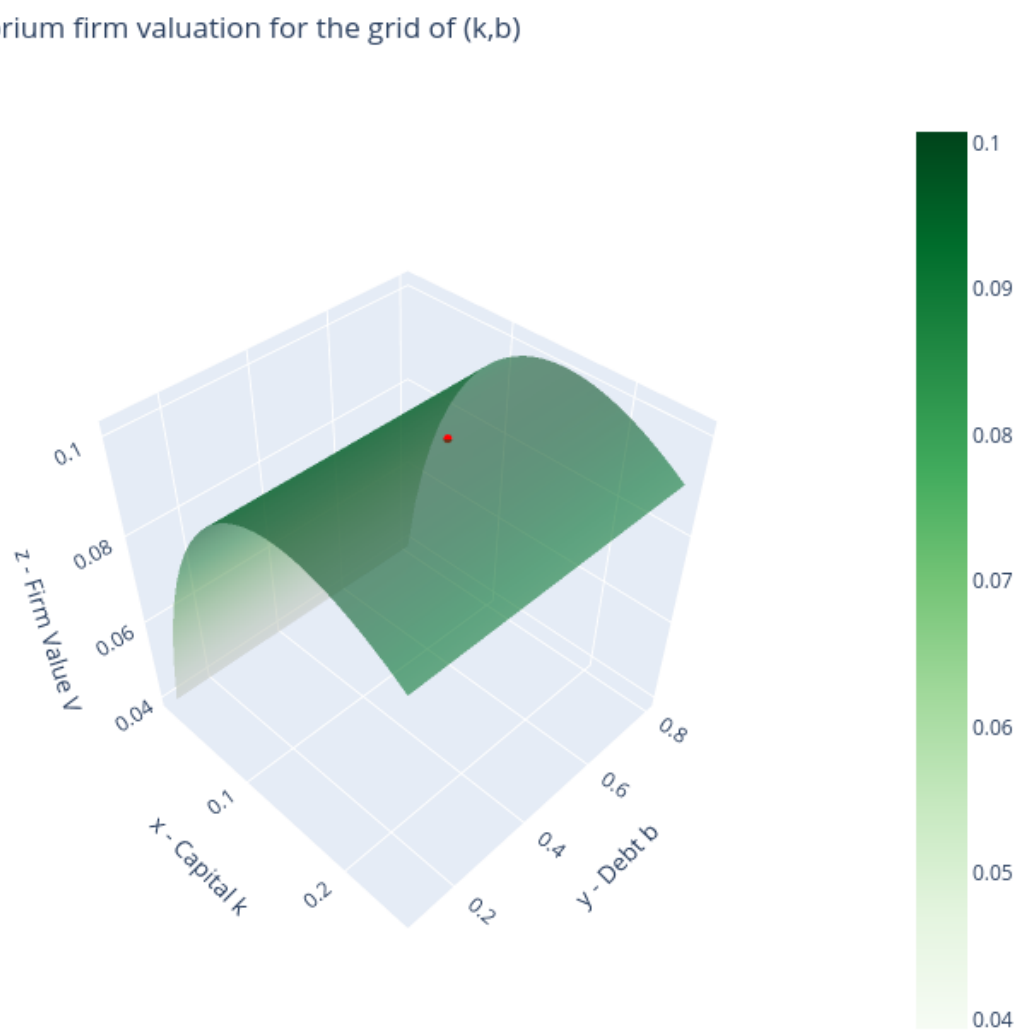
```
Maximum valuation of the firm value in the (k,B) grid: 0.10074
Equilibrium firm value: 0.10083
```

Up to the approximation involved in using a discrete grid, these numbers give us comfort that the firm does indeed seem to be maximizing its value at the top of the value hill on the (k, b) plane that it faces.

Below we will plot the firm's value as a function of k, b.

We'll also plot the equilibrium price functions $q(k, b)$ and $p(k, b)$.

```
from IPython.display import Image
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d
import plotly.graph_objs as go

# Firm Valuation
fig = go.Figure(data=[go.Scatter3d(x=[kss],
                                   y=[bss],
                                   z=[Vss],
                                   mode='markers',
                                   marker=dict(size=3, color='red')),
                      go.Surface(x=kgrid,
                                 y=bgrid,
                                 z=Vgrid,
                                 colorscale='Greens',opacity=0.6)])

fig.update_layout(scene = dict(
                     xaxis_title='x - Capital k',
                     yaxis_title='y - Debt b',
                     zaxis_title='z - Firm Value V',
                     aspectratio = dict(x=1,y=1,z=1)),
                  width=700,
                  height=700,
                  margin=dict(l=50, r=50, b=65, t=90))
fig.update_layout(scene_camera=dict(eye=dict(x=1.5, y=-1.5, z=2)))
fig.update_layout(title='Equilibrium firm valuation for the grid of (k,b)')

# Export to PNG file
Image(fig.to_image(format="png"))
# fig.show() will provide interactive plot when running
# code locally
```



A Modigliani-Miller theorem?

The red dot in the above graph is **both** an equilibrium (b, k) chosen by a representative firm **and** the equilibrium B, K pair chosen by the aggregate of all firms.

Thus, **in equilibrium** it is true that

$$(b, k) = (B, K)$$

But an individual firm named $\zeta \in [0, 1]$ neither knows nor cares whether it sets $(b(\zeta), k(\zeta)) = (B, K)$.

Indeed the above graph has a ridge of $b(\zeta)$'s that also maximize the firm's value so long as it sets $k(\zeta) = K$.

Here it is important that the measure of firms that deviate from setting b at the red dot is very small – measure zero – so that B remains at the red dot even while one firm ζ deviates.

So within this equilibrium, there is a *qualified* Modigliani-Miller theorem that asserts that firm ζ's value is independent of how it mixes its financing between equity and bonds (so long as it is not what other firms do on average).

Thus, while an individual firm ζ's financial structure is indeterminate, the **market's** financial structure is determinant and sits at the red dot in the above graph.

This contrasts sharply with the *unqualified* Modigliani-Miller theorem descibed in the complete markets model in the lecture *Irrelevance of Capital Structure with Complete Markets*.

There the **market's** financial structure was indeterminate.

These subtle distinctions bear more thought and exploration.

So we will do some calculations to ferret out a sense in which the equilibrium $(k, b) = (K, B)$ outcome at the red dot in the above graph is **stable**.

In particular, we'll explore the consequences of some choices of $b = B$ that deviate from the red dot and ask whether firm ζ would want to remain at that b.

In more detail, here is what we'll do:

1. Obtain equilibrium values of capital and debt as $k^* = K$ and $b^* = B$, the red dot above.
2. Now fix k^* and let $b^{**} = b^* - e$ for some $e > 0$. Conjecture that big $K = k^*$ but big $B = b^{**}$.
3. Take K and B and compute intertermporal marginal rates of substitution (IMRS's) as we did before.
4. Taking the **new** IMRS to the firm's problem. Plot 3D surface for the valuations of the firm with this **new** IMRS.
5. Check if the value at k^*, b^{**} is at the top of this new 3D surface.
6. Repeat these calculations for $b^{**} = b^* + e$.

To conduct the above procedures, we create a function `off_eq_check` that inputs the BCG model instance parameters, equilibrium capital $K = k^*$ and debt $B = b^*$, and a perturbation of debt e.

The function outputs the fixed point firm values V^{**}, prices q^{**}, p^{**}, and consumption choices c^{**}.

Importantly, we relax the condition that only agent 2 holds bonds.

Now **both** agents can hold bonds, i.e., $0 \leq \xi^1 \leq B$ and $\xi^1 + \xi^2 = B$.

That implies the consumers' budget constraints are:

$$
\begin{aligned}
c_0^1 &= w_0^1 + \theta_0^1 V - q\theta^1 - p\xi^1 \\
c_0^2 &= w_0^2 + \theta_0^2 V - q\theta^2 - p\xi^2 \\
c_1^1(\epsilon) &= w_1^1(\epsilon) + \theta^1 d^e(k, b; \epsilon) + \xi^1 \\
c_1^2(\epsilon) &= w_1^2(\epsilon) + \theta^2 d^e(k, b; \epsilon) + \xi^2
\end{aligned}
$$

The function also outputs agent 1's bond holdings ξ_1.

```
def off_eq_check(mdl,kss,bss,e=0.1):
    # Big K and big B
    k = kss
    b = bss + e

    # Load parameters
    ?1 = mdl.?1
    ?2 = mdl.?2
    ? = mdl.?
    A = mdl.A
    ? = mdl.?
    bound = mdl.bound
```

(continues on next page)

(continued from previous page)

```
    Vl = mdl.Vl
    Vh = mdl.Vh
    kbot = mdl.kbot
    ktop = mdl.ktop
    bbot = mdl.bbot
    btop = mdl.btop
    w10 = mdl.w10
    w20 = mdl.w20
    ϕ10 = mdl.ϕ10
    ϕ20 = mdl.ϕ20
    w11 = mdl.w11
    w21 = mdl.w21
    g = mdl.g

    Y = njit(lambda ϵ, fk: np.exp(ϵ)*fk)
    intqq1 = njit(lambda ϵ, fk, ϕ1, ϕ1, ϕ1, b: (w11(ϵ) + ϕ1*(Y(ϵ, fk) - b) + ϕ1)**(-
↪ϕ1)*(Y(ϵ, fk) - b)*g(ϵ))
    intpp1a = njit(lambda ϵ, fk, ϕ1, ϕ1, b: (Y(ϵ, fk)/b)*(w11(ϵ) + Y(ϵ, fk)/b*ϕ1)**(-
↪ϕ1)*g(ϵ))
    intpp1b = njit(lambda ϵ, fk, ϕ1, ϕ1, ϕ1, b: (w11(ϵ) + ϕ1*(Y(ϵ, fk)-b) + ϕ1)**(-
↪ϕ1)*g(ϵ))
    intpp2a = njit(lambda ϵ, fk, ϕ2, ϕ2, b: (Y(ϵ, fk)/b)*(w21(ϵ) + Y(ϵ, fk)/b*ϕ2)**(-
↪ϕ2)*g(ϵ))
    intpp2b = njit(lambda ϵ, fk, ϕ2, ϕ2, ϕ2, b: (w21(ϵ) + ϕ2*(Y(ϵ, fk)-b) + ϕ2)**(-
↪ϕ2)*g(ϵ))
    intqq2 = njit(lambda ϵ, fk, ϕ2, ϕ2, b: (w21(ϵ) + ϕ2*(Y(ϵ, fk)-b) + b)**(-ϕ2)*(Y(ϵ,
↪ fk) - b)*g(ϵ))


    # Loop: Find fixed points V, q and p
    V_crit = 1
    while V_crit>1e-5:

        # We begin by adding the guess for the value of the firm to endowment
        V = (Vl+Vh)/2
        ww10 = w10 + ϕ10*V
        ww20 = w20 + ϕ20*V

        # Production
        fk = A*(k**ϕ)
#         Y = lambda ϵ: np.exp(ϵ)*fk

        # Compute integration threshold
        epstar = np.log(b/fk)


        #*************************************************************
        # Compute the prices and allocations consistent with consumers'
        # Euler equations
        #*************************************************************

        # We impose the following:
        # Agent 1 buys equity
        # Agent 2 buys equity and all debt
        # Agents trade such that prices converge
```

(continues on next page)

(continued from previous page)

```
        #========
        # Agent 1
        #========
        # Holdings
        ⍰1a = 0
        ⍰1b = b/2
        p = 0.3

        while abs(⍰1b - ⍰1a) > 0.001:

            ⍰1 = (⍰1a + ⍰1b) / 2
            ⍰1a = 0.3
            ⍰1b = 1

            while abs(⍰1b - ⍰1a) > (0.001/b):

                ⍰1 = (⍰1a + ⍰1b) / 2

                # qq1 is the equity price consistent with agent-1 Euler Equation
                ## Note: Price is in the date-0 budget constraint of the agent

                ## First, compute the constant term that is not influenced by q
                ## that is, ⍰E[u'(c^{1}_{1})d^{e}(k,B)]
#                 intqq1 = lambda ⍰: (w11(⍰) + ⍰1*(Y(⍰, fk) - b) + ⍰1)**(-⍰1)*(Y(⍰,␣
↪fk) - b)*g(⍰)
#                 const_qq1 = ⍰ * quad(intqq1,epstar,bound)[0]
                const_qq1 = ⍰ * quad(intqq1,epstar,bound, args=(fk, ⍰1, ⍰1, ⍰1, b))[0]

                ## Second, iterate to get the equity price q
                qq1l = 0
                qq1h = ww10
                diff = 1
                while diff > 1e-7:
                    qq1 = (qq1l+qq1h)/2
                    rhs = const_qq1/((ww10-qq1*⍰1-p*⍰1)**(-⍰1));
                    if (rhs > qq1):
                        qq1l = qq1
                    else:
                        qq1h = qq1
                    diff = abs(qq1l-qq1h)

                # pp1 is the bond price consistent with agent-2 Euler Equation
                ## Note: Price is in the date-0 budget constraint of the agent

                ## First, compute the constant term that is not influenced by p
                ## that is, ⍰E[u'(c^{1}_{1})d^{b}(k,B)]
#                 intpp1a = lambda ⍰: (Y(⍰, fk)/b)*(w11(⍰) + Y(⍰, fk)/b*⍰1)**(-
↪⍰1)*g(⍰)
#                 intpp1b = lambda ⍰: (w11(⍰) + ⍰1*(Y(⍰, fk)-b) + ⍰1)**(-⍰1)*g(⍰)
#                 const_pp1 = ⍰ * (quad(intpp1a,-bound,epstar)[0] + quad(intpp1b,
↪epstar,bound)[0])
                const_pp1 = ⍰ * (quad(intpp1a,-bound,epstar, args=(fk, ⍰1, ⍰1, b))[0]␣
↪\
                                 + quad(intpp1b,epstar,bound, args=(fk, ⍰1, ⍰1, ⍰1,␣
↪b))[0])
```

(continues on next page)

(continued from previous page)

```
                ## iterate to get the bond price p
                pp1l = 0
                pp1h = ww10/b
                diff = 1
                while diff > 1e-7:
                    pp1 = (pp1l+pp1h)/2
                    rhs = const_pp1/((ww10-qq1*ϕ1-pp1*ϕ1)**(-ϕ1))
                    if (rhs > pp1):
                        pp1l = pp1
                    else:
                        pp1h = pp1
                    diff = abs(pp1l-pp1h)

                #========
                # Agent 2
                #========
                ϕ2 = b - ϕ1
                ϕ2 = 1 - ϕ1

                # pp2 is the bond price consistent with agent-2 Euler Equation
                ## Note: Price is in the date-0 budget constraint of the agent

                ## First, compute the constant term that is not influenced by p
                ## that is, ϕE[u'(c^{2}_{1})d^{b}(k,B)]
#                 intpp2a = lambda ϕ: (Y(ϕ, fk)/b)*(w21(ϕ) + Y(ϕ, fk)/b*ϕ2)**(-
↪ϕ2)*g(ϕ)
#                 intpp2b = lambda ϕ: (w21(ϕ) + ϕ2*(Y(ϕ, fk)-b) + ϕ2)**(-ϕ2)*g(ϕ)
#                 const_pp2 = ϕ * (quad(intpp2a,-bound,epstar)[0] + quad(intpp2b,
↪epstar,bound)[0])
                const_pp2 = ϕ * (quad(intpp2a,-bound,epstar, args=(fk, ϕ2, ϕ2, b))[0]␣
↪\
                                 + quad(intpp2b,epstar,bound, args=(fk, ϕ2, ϕ2, ϕ2,␣
↪b))[0])

                ## iterate to get the bond price p
                pp2l = 0
                pp2h = ww20/b
                diff = 1
                while diff > 1e-7:
                    pp2 = (pp2l+pp2h)/2
                    rhs = const_pp2/((ww20-qq1*ϕ2-pp2*ϕ2)**(-ϕ2))
                    if (rhs > pp2):
                        pp2l = pp2
                    else:
                        pp2h = pp2
                    diff = abs(pp2l-pp2h)

                # p be the maximum valuation for the bond among agents
                ## This will be the equity price based on Makowski's criterion
                p = max(pp1,pp2)


                # qq2 is the equity price consistent with agent-2 Euler Equation
#                 intqq2 = lambda ϕ: (w21(ϕ) + ϕ2*(Y(ϕ, fk)-b) + b)**(-ϕ2)*(Y(ϕ, fk) -
↪ b)*g(ϕ)
#                 const_qq2 = ϕ * quad(intqq2,epstar,bound)[0]
```

(continues on next page)

(continued from previous page)

```
                const_qq2 = ⍰ * quad(intqq2,epstar,bound, args=(fk, ⍰2, ⍰2, b))[0]
                qq2l = 0
                qq2h = ww20
                diff = 1
                while diff > 1e-7:
                    qq2 = (qq2l+qq2h)/2
                    rhs = const_qq2/((ww20-qq2*⍰2-p*⍰2)**(-⍰2));
                    if (rhs > qq2):
                        qq2l = qq2
                    else:
                        qq2h = qq2
                    diff = abs(qq2l-qq2h)

                # q be the maximum valuation for the equity among agents
                ## This will be the equity price based on Makowski's criterion
                q = max(qq1,qq2)

                #================
                # Update holdings
                #================
                if qq1 > qq2:
                    ⍰1a = ⍰1
                else:
                    ⍰1b = ⍰1

                #print(p,q,⍰1,⍰1)

            if pp1 > pp2:
                ⍰1a = ⍰1
            else:
                ⍰1b = ⍰1


        #================
        # Get consumption
        #================
        c10 = ww10 - q*⍰1 - p*⍰1
        c11 = lambda ⍰: w11(⍰) + ⍰1*max(Y(⍰, fk)-b,0) + ⍰1*min(Y(⍰, fk)/b,1)
        c20 = ww20 - q*(1-⍰1) - p*(b-⍰1)
        c21 = lambda ⍰: w21(⍰) + (1-⍰1)*max(Y(⍰, fk)-b,0) + (b-⍰1)*min(Y(⍰, fk)/b,1)

        # Compute the value of the firm
        value_x = -k + q + p*b
        if (value_x > V):
            Vl = V
        else:
            Vh = V
        V_crit = abs(value_x-V)

    return V,k,b,p,q,c10,c11,c20,c21,⍰1
```

Here is our strategy for checking *stability* of an equilibrium.

We use `off_eq_check` to obtain consumption plans for both agents at the conjectured big K and big B.

Then we input consumption plans into the function `eq_valuation` from the BCG model class and plot the agents' valuations associated with different choices of k and b.

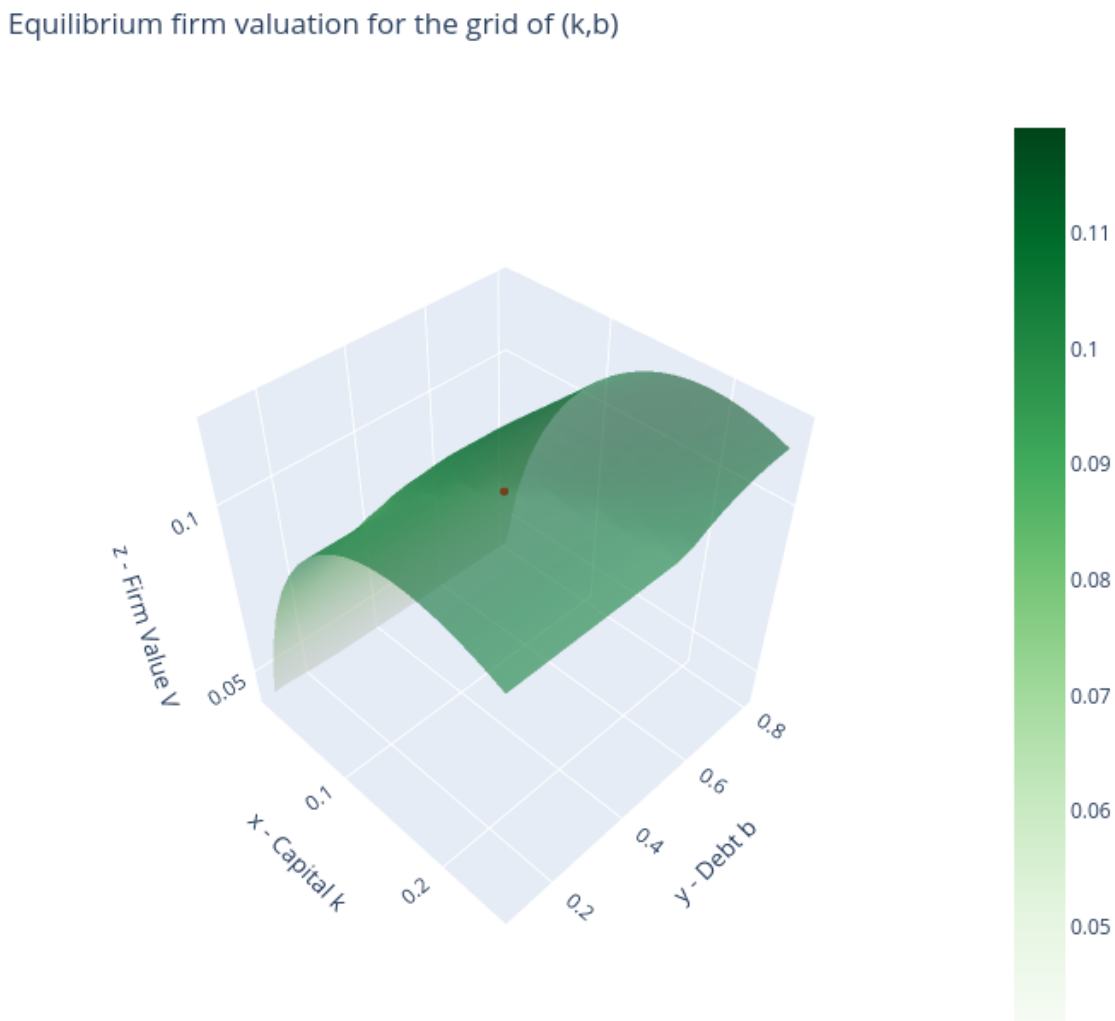
Our hunch is that (k^*, b^{**}) is **not** at the top of the firm valuation 3D surface so that the firm is **not** maximizing its value if it chooses $k = K = k^*$ and $b = B = b^{**}$.

That indicates that (k^*, b^{**}) is not an equilibrium capital structure for the firm.

We first check the case in which $b^{**} = b^* - e$ where $e = 0.1$:

```
#====================== Experiment 1 =====================#
Ve1,ke1,be1,pe1,qe1,c10e1,c11e1,c20e1,c21e1,?1e1 = off_eq_check(mdl,
                                                                kss,
                                                                bss,
                                                                e=-0.1)

# Firm Valuation
kgride1, bgride1, Vgride1, Qgride1, Pgride1 = mdl.eq_valuation(c10e1, c11e1, c20e1,␣
↪c21e1,N=20)

print('Maximum valuation of the firm value in the (k,b) grid: {:.4f}'.format(Vgride1.
↪max()))
print('Equilibrium firm value: {:.4f}'.format(Ve1))

fig = go.Figure(data=[go.Scatter3d(x=[ke1],
                                   y=[be1],
                                   z=[Ve1],
                                   mode='markers',
                                   marker=dict(size=3, color='red')),
                      go.Surface(x=kgride1,
                                 y=bgride1,
                                 z=Vgride1,
                                 colorscale='Greens',opacity=0.6)])

fig.update_layout(scene = dict(
                    xaxis_title='x - Capital k',
                    yaxis_title='y - Debt b',
                    zaxis_title='z - Firm Value V',
                    aspectratio = dict(x=1,y=1,z=1)),
                  width=700,
                  height=700,
                  margin=dict(l=50, r=50, b=65, t=90))
fig.update_layout(scene_camera=dict(eye=dict(x=1.5, y=-1.5, z=2)))
fig.update_layout(title='Equilibrium firm valuation for the grid of (k,b)')


# Export to PNG file
Image(fig.to_image(format="png"))
# fig.show() will provide interactive plot when running
# code locally
```

```
Maximum valuation of the firm value in the (k,b) grid: 0.1191
Equilibrium firm value: 0.1118
```



In the above 3D surface of prospective firm valuations, the perturbed choice $(k^*, b^* - e)$, represented by the red dot, is not at the top.

The firm could issue more debts and attain a higher firm valuation from the market.

Therefore, $(k^*, b^* - e)$ would not be an equilibrium.

Next, we check for $b^{**} = b^* + e$.

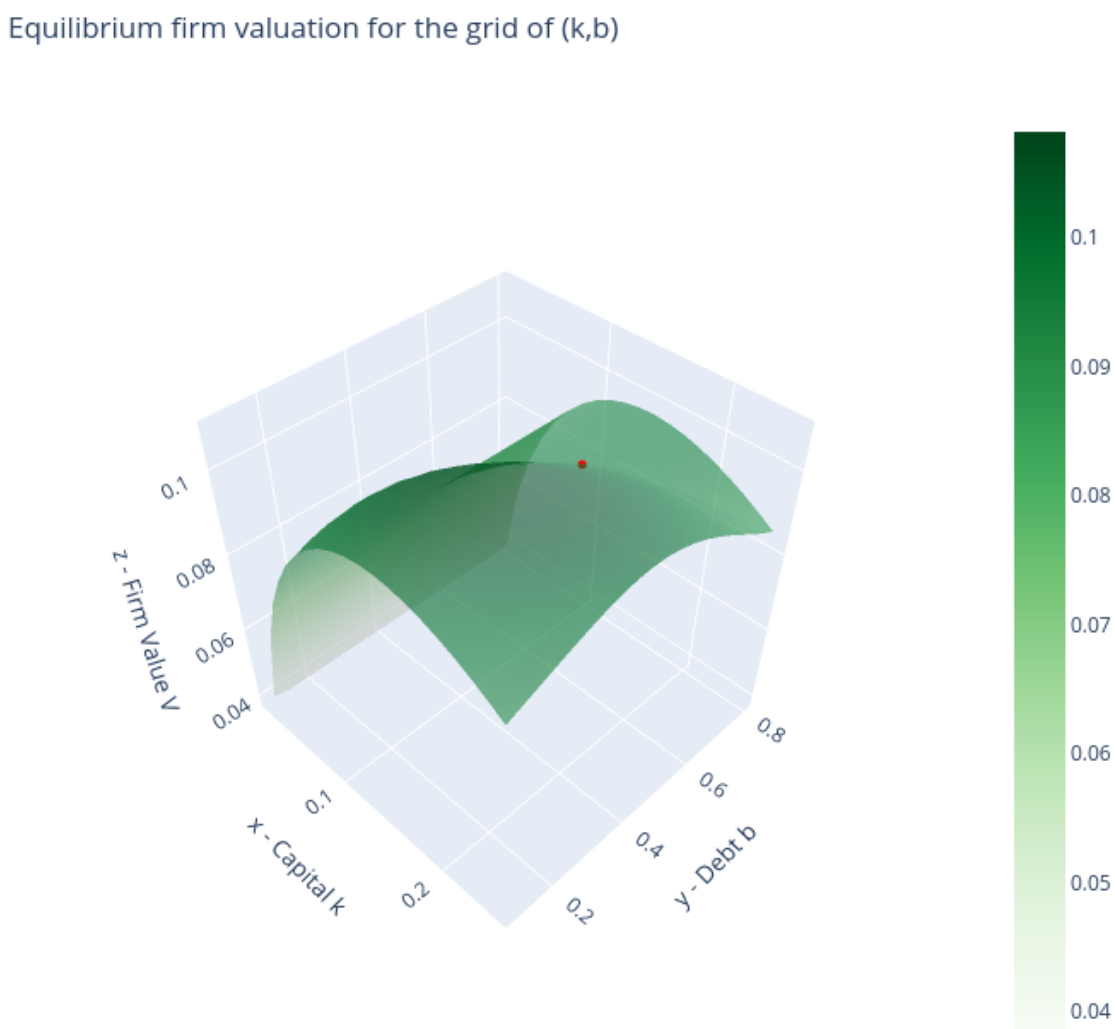
```
#====================== Experiment 2 =======================#
Ve2,ke2,be2,pe2,qe2,c10e2,c11e2,c20e2,c21e2,1e2 = off_eq_check(mdl,
                                                               kss,
                                                               bss,
                                                               e=0.1)

# Firm Valuation
kgride2, bgride2, Vgride2, Qgride2, Pgride2 = mdl.eq_valuation(c10e2, c11e2, c20e2,␣
↪c21e2,N=20)
```

(continues on next page)

(continued from previous page)

```
print('Maximum valuation of the firm value in the (k,b) grid: {:.4f}'.format(Vgride2.
 ↪max()))
print('Equilibrium firm value: {:.4f}'.format(Ve2))

fig = go.Figure(data=[go.Scatter3d(x=[ke2],
                                   y=[be2],
                                   z=[Ve2],
                                   mode='markers',
                                   marker=dict(size=3, color='red')),
                      go.Surface(x=kgride2,
                                 y=bgride2,
                                 z=Vgride2,
                                 colorscale='Greens',opacity=0.6)])

fig.update_layout(scene = dict(
                    xaxis_title='x - Capital k',
                    yaxis_title='y - Debt b',
                    zaxis_title='z - Firm Value V',
                    aspectratio = dict(x=1,y=1,z=1)),
                  width=700,
                  height=700,
                  margin=dict(l=50, r=50, b=65, t=90))
fig.update_layout(scene_camera=dict(eye=dict(x=1.5, y=-1.5, z=2)))
fig.update_layout(title='Equilibrium firm valuation for the grid of (k,b)')

# Export to PNG file
Image(fig.to_image(format="png"))
# fig.show() will provide interactive plot when running
# code locally
```

```
Maximum valuation of the firm value in the (k,b) grid: 0.1082
Equilibrium firm value: 0.0974
```



In contrast to $(k^*, b^* - e)$, the 3D surface for $(k^*, b^* + e)$ now indicates that a firm would want o *decrease* its debt issuance to attain a higher valuation.

That incentive to deviate means that $(k^*, b^* + e)$ is not an equilibrium capital structure for the firm.

Interestingly, if consumers were to anticipate that firms would over-issue debt, i.e. $B > b^*$, then both types of consumer would want to hold corporate debt.

For example, $\xi^1 > 0$:

```
print('Bond holdings of agent 1: {:.3f}'.format(1e2))
```

```
Bond holdings of agent 1: 0.039
```

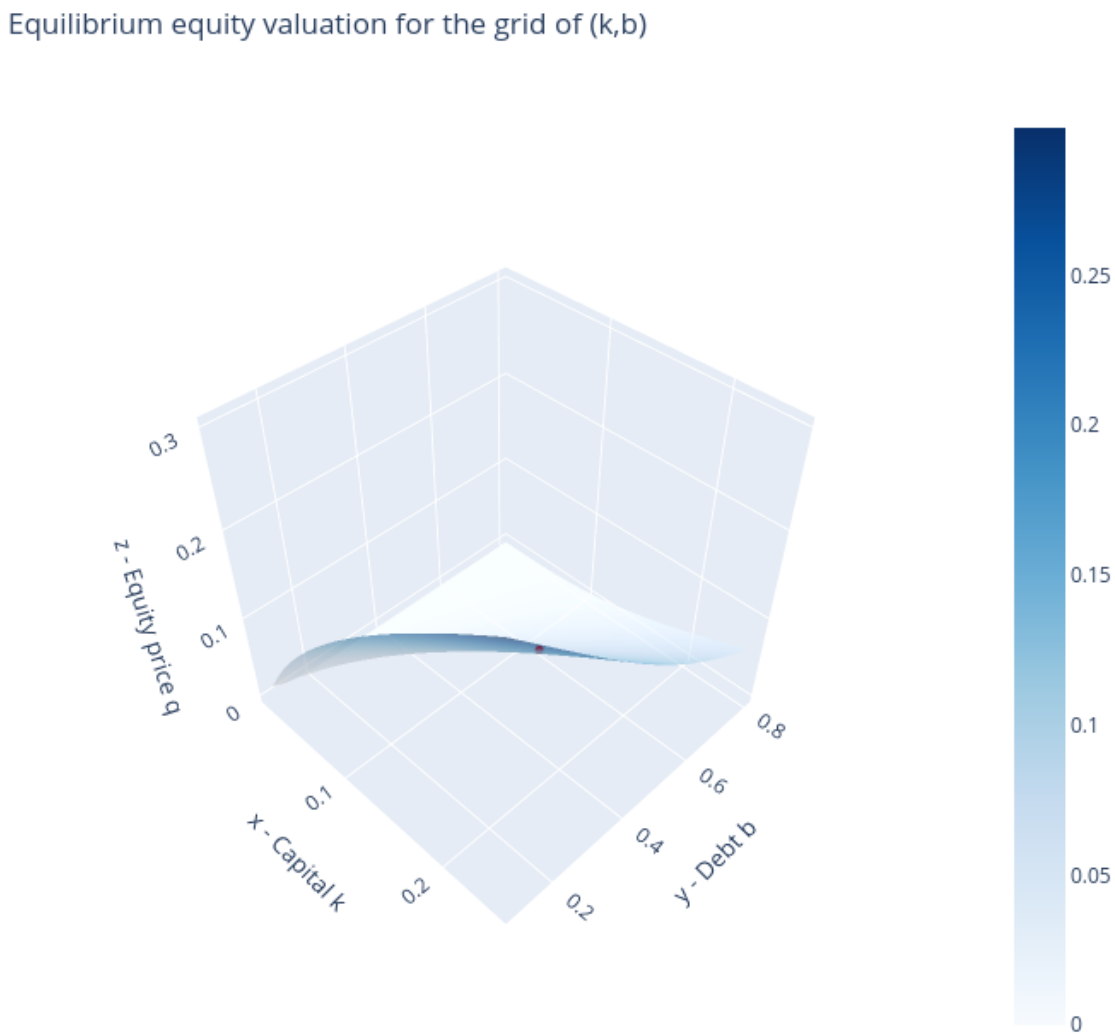
Our two *stability experiments* suggest that the equilibrium capital structure (k^*, b^*) is locally unique even though **at the equilibrium** an individual firm would be willing to deviate from the representative firms' equilibrium debt choice.

These experiments thus refine our discussion of the *qualified* Modigliani-Miller theorem that prevails in this example economy.

Equilibrium equity and bond price functions

It is also interesting to look at the equilibrium price functions $q(k, b)$ and $p(k, b)$ faced by firms in our rational expectations equilibrium.

```
# Equity Valuation
fig = go.Figure(data=[go.Scatter3d(x=[kss],
                                   y=[bss],
                                   z=[qss],
                                   mode='markers',
                                   marker=dict(size=3, color='red')),
                      go.Surface(x=kgrid,
                                 y=bgrid,
                                 z=Qgrid,
                                 colorscale='Blues',opacity=0.6)])

fig.update_layout(scene = dict(
                    xaxis_title='x - Capital k',
                    yaxis_title='y - Debt b',
                    zaxis_title='z - Equity price q',
                    aspectratio = dict(x=1,y=1,z=1)),
                  width=700,
                  height=700,
                  margin=dict(l=50, r=50, b=65, t=90))
fig.update_layout(scene_camera=dict(eye=dict(x=1.5, y=-1.5, z=2)))
fig.update_layout(title='Equilibrium equity valuation for the grid of (k,b)')


# Export to PNG file
Image(fig.to_image(format="png"))
# fig.show() will provide interactive plot when running
# code locally
```

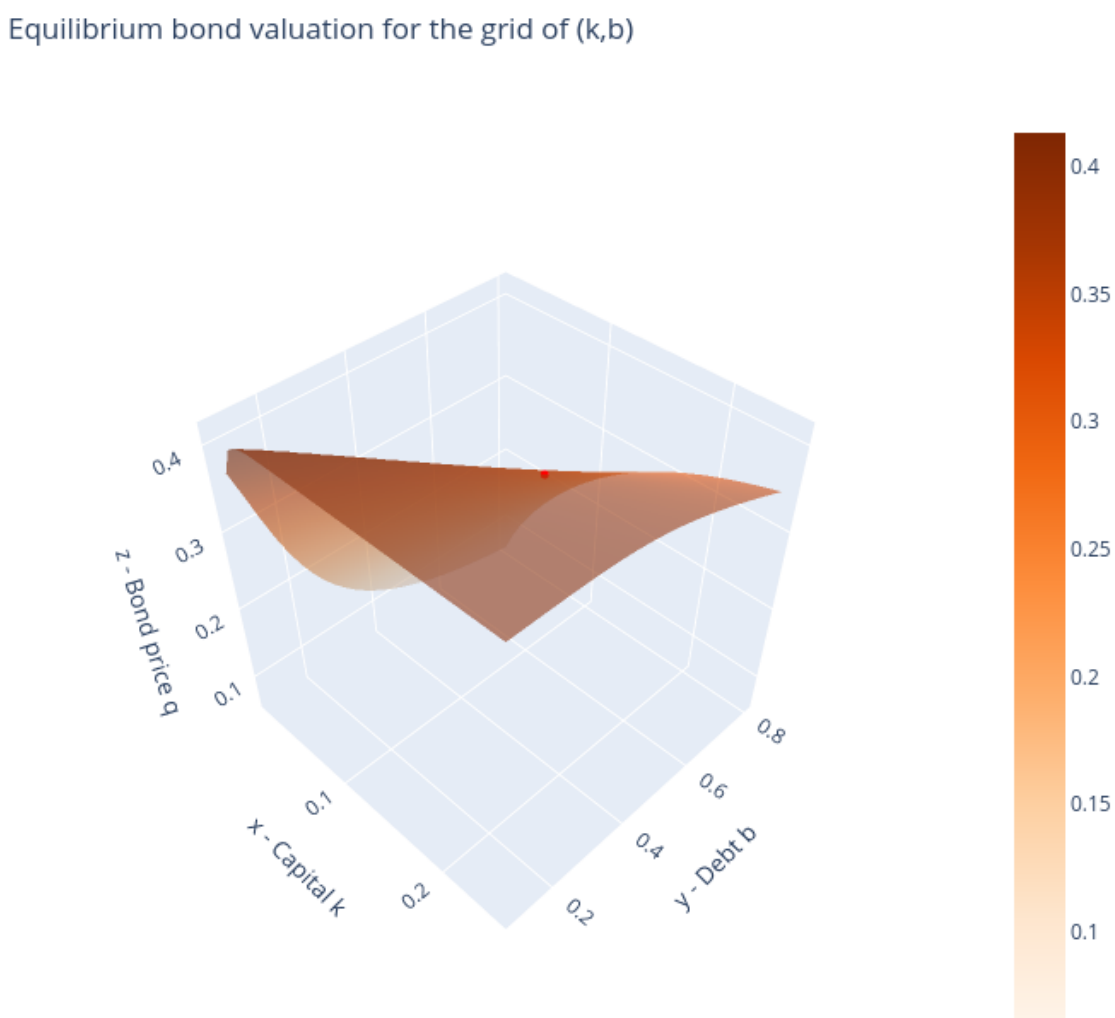



```
# Bond Valuation
fig = go.Figure(data=[go.Scatter3d(x=[kss],
                                   y=[bss],
                                   z=[pss],
                                   mode='markers',
                                   marker=dict(size=3, color='red')),
                      go.Surface(x=kgrid,
                                 y=bgrid,
                                 z=Pgrid,
                                 colorscale='Oranges',opacity=0.6)])

fig.update_layout(scene = dict(
                    xaxis_title='x - Capital k',
                    yaxis_title='y - Debt b',
                    zaxis_title='z - Bond price q',
                    aspectratio = dict(x=1,y=1,z=1)),
```

(continues on next page)

(continued from previous page)

```
                    width=700,
                    height=700,
                    margin=dict(l=50, r=50, b=65, t=90))
fig.update_layout(scene_camera=dict(eye=dict(x=1.5, y=-1.5, z=2)))
fig.update_layout(title='Equilibrium bond valuation for the grid of (k,b)')


# Export to PNG file
Image(fig.to_image(format="png"))
# fig.show() will provide interactive plot when running
# code locally
```



14.6.2 Comments on equilibrium pricing functions

The equilibrium pricing functions displayed above merit study and reflection.

They reveal the countervailing effects on a firm's valuations of bonds and equities that lie beneath the Modigliani-Miller ridge apparent in our earlier graph of an individual firm ζ's value as a function of $k(\zeta), b(\zeta)$.

14.6.3 Another example economy

We illustrate how the fraction of initial endowments held by agent 2, $w_0^2/(w_0^1+w_0^2)$ affects an equilibrium capital structure $(k, b) = (K, B)$ well as associated equilibrium allocations.

We are interested in how agents 1 and 2 value equity and bond.

$$Q^i = \beta \int \frac{u'(C_1^{i,*}(\epsilon))}{u'(C_0^{i,*})} d^e(k^*, b^*; \epsilon) g(\epsilon)\, d\epsilon$$
$$P^i = \beta \int \frac{u'(C_1^{i,*}(\epsilon))}{u'(C_0^{i,*})} d^b(k^*, b^*; \epsilon) g(\epsilon)\, d\epsilon$$

The function `valuations_by_agent` is used in calculating these valuations.

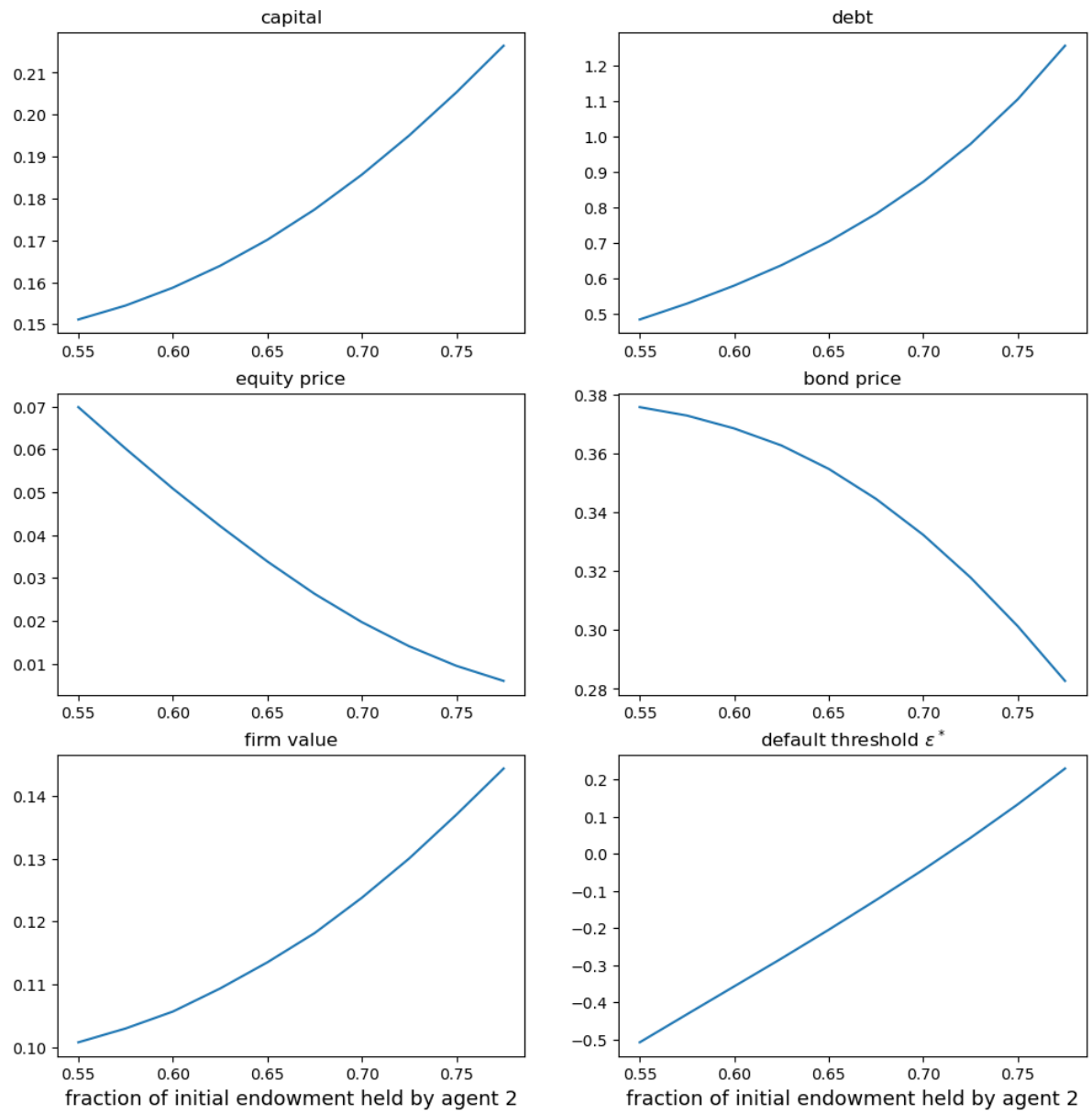
```
# Lists for storage
wlist = []
klist = []
blist = []
qlist = []
plist = []
Vlist = []
tlist = []
q1list = []
q2list = []
p1list = []
p2list = []

# For loop: optimization for each endowment combination
for i in range(10):
    print(i)

    # Save fraction
    w10 = 0.9 - 0.05*i
    w20 = 1.1 + 0.05*i
    wlist.append(w20/(w10+w20))

    # Create the instance
    mdl = BCG_incomplete_markets(w10 = w10, w20 = w20, ktop = 0.5, btop = 2.5)

    # Solve for equilibrium
    kss,bss,Vss,qss,pss,c10ss,c11ss,c20ss,c21ss,q1ss = mdl.solve_eq(print_crit=False)

    # Store the equilibrium results
    klist.append(kss)
    blist.append(bss)
    qlist.append(qss)
    plist.append(pss)
    Vlist.append(Vss)
```

(continues on next page)

(continued from previous page)

```
    tlist.append(τ1ss)

    # Evaluations of equity and bond by each agent
    Q1,Q2,P1,P2 = mdl.valuations_by_agent(c10ss, c11ss, c20ss, c21ss, kss, bss)

    # Save the valuations
    q1list.append(Q1)
    q2list.append(Q2)
    p1list.append(P1)
    p2list.append(P2)
```

```
# Plot
fig, ax = plt.subplots(3,2,figsize=(12,12))
ax[0,0].plot(wlist,klist)
ax[0,0].set_title('capital')
ax[0,1].plot(wlist,blist)
ax[0,1].set_title('debt')
ax[1,0].plot(wlist,qlist)
ax[1,0].set_title('equity price')
ax[1,1].plot(wlist,plist)
ax[1,1].set_title('bond price')
ax[2,0].plot(wlist,Vlist)
ax[2,0].set_title('firm value')
ax[2,0].set_xlabel('fraction of initial endowment held by agent 2',fontsize=13)

# Create a list of Default thresholds
A = mdl.A
τ = mdl.τ
epslist = []
for i in range(len(wlist)):
    bb = blist[i]
    kk = klist[i]
    eps = np.log(bb/(A*kk**τ))
    epslist.append(eps)

# Plot (cont.)
ax[2,1].plot(wlist,epslist)
ax[2,1].set_title(r'default threshold $\epsilon^*$')
ax[2,1].set_xlabel('fraction of initial endowment held by agent 2',fontsize=13)
plt.show()
```



14.7 A picture worth a thousand words

Please stare at the above panels.

They describe how equilibrium prices and quantities respond to alterations in the structure of society's *hedging desires* across economies with different allocations of the initial endowment to our two types of agents.

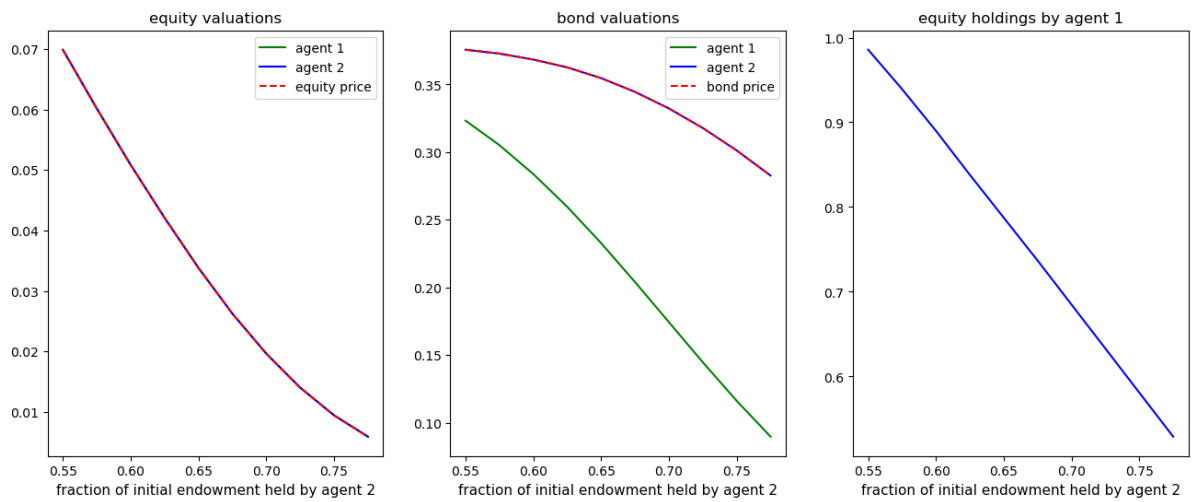
Now let's see how the two types of agents value bonds and equities, keeping in mind that the type that values the asset highest determines the equilibrium price (and thus the pertinent set of Big C's).

```
# Comparing the prices
fig, ax = plt.subplots(1,3,figsize=(16,6))
```

(continues on next page)

(continued from previous page)

```
ax[0].plot(wlist,q1list,label='agent 1',color='green')
ax[0].plot(wlist,q2list,label='agent 2',color='blue')
ax[0].plot(wlist,qlist,label='equity price',color='red',linestyle='--')
ax[0].legend()
ax[0].set_title('equity valuations')
ax[0].set_xlabel('fraction of initial endowment held by agent 2',fontsize=11)

ax[1].plot(wlist,p1list,label='agent 1',color='green')
ax[1].plot(wlist,p2list,label='agent 2',color='blue')
ax[1].plot(wlist,plist,label='bond price',color='red',linestyle='--')
ax[1].legend()
ax[1].set_title('bond valuations')
ax[1].set_xlabel('fraction of initial endowment held by agent 2',fontsize=11)

ax[2].plot(wlist,tlist,color='blue')
ax[2].set_title('equity holdings by agent 1')
ax[2].set_xlabel('fraction of initial endowment held by agent 2',fontsize=11)

plt.show()
```



It is rewarding to stare at the above plots too.

In equilibrium, equity valuations are the same across the two types of agents but bond valuations are not.

Agents of type 2 value bonds more highly (they want more hedging).

Taken together with our earlier plot of equity holdings, these graphs confirm our earlier conjecture that while both type of agents hold equities, only agents of type 2 holds bonds.

Part III

Search

CHAPTER

FIFTEEN

JOB SEARCH I: THE MCCALL SEARCH MODEL

Contents

- *Job Search I: The McCall Search Model*
  - *Overview*
  - *The McCall Model*
  - *Computing the Optimal Policy: Take 1*
  - *Computing an Optimal Policy: Take 2*
  - *Exercises*

> "Questioning a McCall worker is like having a conversation with an out-of-work friend: 'Maybe you are setting your sights too high', or 'Why did you quit your old job before you had a new one lined up?' This is real social science: an attempt to model, to understand, human behavior by visualizing the situation people find themselves in, the options they face and the pros and cons as they themselves see them." – Robert E. Lucas, Jr.

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

```
!pip install quantecon
```

15.1 Overview

The McCall search model [McC70] helped transform economists' way of thinking about labor markets.

To clarify notions such as "involuntary" unemployment, McCall modeled the decision problem of an unemployed worker in terms of factors including

- current and likely future wages
- impatience
- unemployment compensation

To solve the decision problem McCall used dynamic programming.

Here we set up McCall's model and use dynamic programming to analyze it.

As we'll see, McCall's model is not only interesting in its own right but also an excellent vehicle for learning dynamic programming.

Let's start with some imports:

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from numba import jit, float64
from numba.experimental import jitclass
import quantecon as qe
from quantecon.distributions import BetaBinomial
```

15.2 The McCall Model

An unemployed agent receives in each period a job offer at wage w_t.

In this lecture, we adopt the following simple environment:

- The offer sequence $\{w_t\}_{t \geq 0}$ is IID, with $q(w)$ being the probability of observing wage w in finite set $\mathbb{W}$.
- The agent observes w_t at the start of t.
- The agent knows that $\{w_t\}$ is IID with common distribution q and can use this when computing expectations.

(In later lectures, we will relax these assumptions.)

At time t, our agent has two choices:

1. Accept the offer and work permanently at constant wage w_t.
2. Reject the offer, receive unemployment compensation c, and reconsider next period.

The agent is infinitely lived and aims to maximize the expected discounted sum of earnings

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t y_t$$

The constant β lies in $(0, 1)$ and is called a **discount factor**.

The smaller is β, the more the agent discounts future utility relative to current utility.

The variable y_t is income, equal to

- his/her wage w_t when employed
- unemployment compensation c when unemployed

15.2.1 A Trade-Off

The worker faces a trade-off:

- Waiting too long for a good offer is costly, since the future is discounted.
- Accepting too early is costly, since better offers might arrive in the future.

To decide optimally in the face of this trade-off, we use dynamic programming.

Dynamic programming can be thought of as a two-step procedure that

1. first assigns values to "states" and
2. then deduces optimal actions given those values

We'll go through these steps in turn.

15.2.2 The Value Function

In order to optimally trade-off current and future rewards, we need to think about two things:

1. the current payoffs we get from different choices
2. the different states that those choices will lead to in next period

To weigh these two aspects of the decision problem, we need to assign *values* to states.

To this end, let $v^*(w)$ be the total lifetime *value* accruing to an unemployed worker who enters the current period unemployed when the wage is $w \in \mathbb{W}$.

In particular, the agent has wage offer w in hand.

More precisely, $v^*(w)$ denotes the value of the objective function (16.1) when an agent in this situation makes *optimal* decisions now and at all future points in time.

Of course $v^*(w)$ is not trivial to calculate because we don't yet know what decisions are optimal and what aren't!

But think of v^* as a function that assigns to each possible wage s the maximal lifetime value that can be obtained with that offer in hand.

A crucial observation is that this function v^* must satisfy the recursion

$$v^*(w) = \max \left\{ \frac{w}{1-\beta}, \, c + \beta \sum_{w' \in \mathbb{W}} v^*(w') q(w') \right\} \tag{15.1}$$

for every possible w in $\mathbb{W}$.

This important equation is a version of the **Bellman equation**, which is ubiquitous in economic dynamics and other fields involving planning over time.

The intuition behind it is as follows:

- the first term inside the max operation is the lifetime payoff from accepting current offer, since
$$\frac{w}{1-\beta} = w + \beta w + \beta^2 w + \cdots$$
- the second term inside the max operation is the **continuation value**, which is the lifetime payoff from rejecting the current offer and then behaving optimally in all subsequent periods

If we optimize and pick the best of these two options, we obtain maximal lifetime value from today, given current offer w.

But this is precisely $v^*(w)$, which is the left-hand side of (15.1).

15.2.3 The Optimal Policy

Suppose for now that we are able to solve (15.1) for the unknown function v^*.

Once we have this function in hand we can behave optimally (i.e., make the right choice between accept and reject).

All we have to do is select the maximal choice on the right-hand side of (15.1).

The optimal action is best thought of as a **policy**, which is, in general, a map from states to actions.

Given *any* w, we can read off the corresponding best choice (accept or reject) by picking the max on the right-hand side of (15.1).

Thus, we have a map from $\mathbb{R}$ to $\{0, 1\}$, with 1 meaning accept and 0 meaning reject.

We can write the policy as follows

$$\sigma(w) := \mathbf{1}\left\{\frac{w}{1-\beta} \geq c + \beta \sum_{w' \in \mathbb{W}} v^*(w')q(w')\right\}$$

Here $\mathbf{1}\{P\} = 1$ if statement P is true and equals 0 otherwise.

We can also write this as

$$\sigma(w) := \mathbf{1}\{w \geq \bar{w}\}$$

where

$$\bar{w} := (1-\beta)\left\{c + \beta \sum_{w'} v^*(w')q(w')\right\} \tag{15.2}$$

Here $\bar{w}$ (called the *reservation wage*) is a constant depending on β, c and the wage distribution.

The agent should accept if and only if the current wage offer exceeds the reservation wage.

In view of (15.2), we can compute this reservation wage if we can compute the value function.

15.3 Computing the Optimal Policy: Take 1

To put the above ideas into action, we need to compute the value function at each possible state $w \in \mathbb{W}$.

To simplify notation, let's set

$$\mathbb{W} := \{w_1, \dots, w_n\} \quad \text{and} \quad v^*(i) := v^*(w_i)$$

The value function is then represented by the vector $v^* = (v^*(i))_{i=1}^n$.

In view of (15.1), this vector satisfies the nonlinear system of equations

$$v^*(i) = \max\left\{\frac{w(i)}{1-\beta}, c + \beta \sum_{1 \leq j \leq n} v^*(j)q(j)\right\} \quad \text{for } i = 1, \dots, n \tag{15.3}$$

15.3.1 The Algorithm

To compute this vector, we use successive approximations:

Step 1: pick an arbitrary initial guess $v \in \mathbb{R}^n$.

Step 2: compute a new vector $v' \in \mathbb{R}^n$ via

$$v'(i) = \max\left\{\frac{w(i)}{1-\beta}, c + \beta \sum_{1 \leq j \leq n} v(j)q(j)\right\} \quad \text{for } i = 1, \dots, n \tag{15.4}$$

Step 3: calculate a measure of a discrepancy between v and v', such as $\max_i |v(i) - v'(i)|$.

Step 4: if the deviation is larger than some fixed tolerance, set $v = v'$ and go to step 2, else continue.

Step 5: return v.

For a small tolerance, the returned function v is a close approximation to the value function v^*.

The theory below elaborates on this point.

15.3.2 Fixed Point Theory

What's the mathematics behind these ideas?

First, one defines a mapping T from $\mathbb{R}^n$ to itself via

$$(Tv)(i) = \max \left\{ \frac{w(i)}{1-\beta}, \, c + \beta \sum_{1 \leq j \leq n} v(j) q(j) \right\} \quad \text{for } i = 1, \dots, n \tag{15.5}$$

(A new vector Tv is obtained from given vector v by evaluating the r.h.s. at each i.)

The element v_k in the sequence $\{v_k\}$ of successive approximations corresponds to $T^k v$.

- This is T applied k times, starting at the initial guess v

One can show that the conditions of the Banach fixed point theorem are satisfied by T on $\mathbb{R}^n$.

One implication is that T has a unique fixed point in $\mathbb{R}^n$.

- That is, a unique vector $\bar{v}$ such that $T\bar{v} = \bar{v}$.

Moreover, it's immediate from the definition of T that this fixed point is v^*.

A second implication of the Banach contraction mapping theorem is that $\{T^k v\}$ converges to the fixed point v^* regardless of v.

15.3.3 Implementation

Our default for q, the distribution of the state process, will be Beta-binomial.

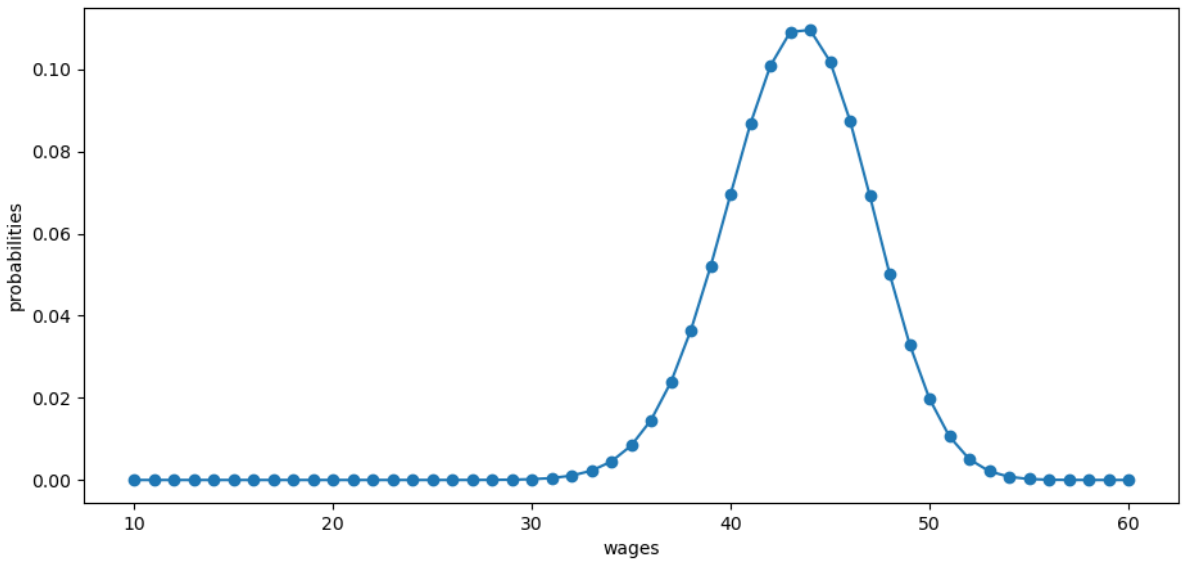
```
n, a, b = 50, 200, 100                        # default parameters
q_default = BetaBinomial(n, a, b).pdf()       # default choice of q
```

Our default set of values for wages will be

```
w_min, w_max = 10, 60
w_default = np.linspace(w_min, w_max, n+1)
```

Here's a plot of the probabilities of different wage outcomes:

```
fig, ax = plt.subplots()
ax.plot(w_default, q_default, '-o', label='$q(w(i))$')
ax.set_xlabel('wages')
ax.set_ylabel('probabilities')

plt.show()
```



We are going to use Numba to accelerate our code.

- See, in particular, the discussion of `@jitclass` in our lecture on Numba.

The following helps Numba by providing some type

```
mccall_data = [
    ('c', float64),       # unemployment compensation
    ('β', float64),       # discount factor
    ('w', float64[:]),    # array of wage values, w[i] = wage at state i
    ('q', float64[:])     # array of probabilities
]
```

Here's a class that stores the data and computes the values of state-action pairs, i.e. the value in the maximum bracket on the right hand side of the Bellman equation (15.4), given the current state and an arbitrary feasible action.

Default parameter values are embedded in the class.

```
@jitclass(mccall_data)
class McCallModel:

    def __init__(self, c=25, β=0.99, w=w_default, q=q_default):

        self.c, self.β = c, β
        self.w, self.q = w_default, q_default

    def state_action_values(self, i, v):
        """
        The values of state-action pairs.
        """
        # Simplify names
        c, β, w, q = self.c, self.β, self.w, self.q
        # Evaluate value for each state-action pair
        # Consider action = accept or reject the current offer
        accept = w[i] / (1 - β)
        reject = c + β * np.sum(v * q)

        return np.array([accept, reject])
```

Based on these defaults, let's try plotting the first few approximate value functions in the sequence $\{T^k v\}$.

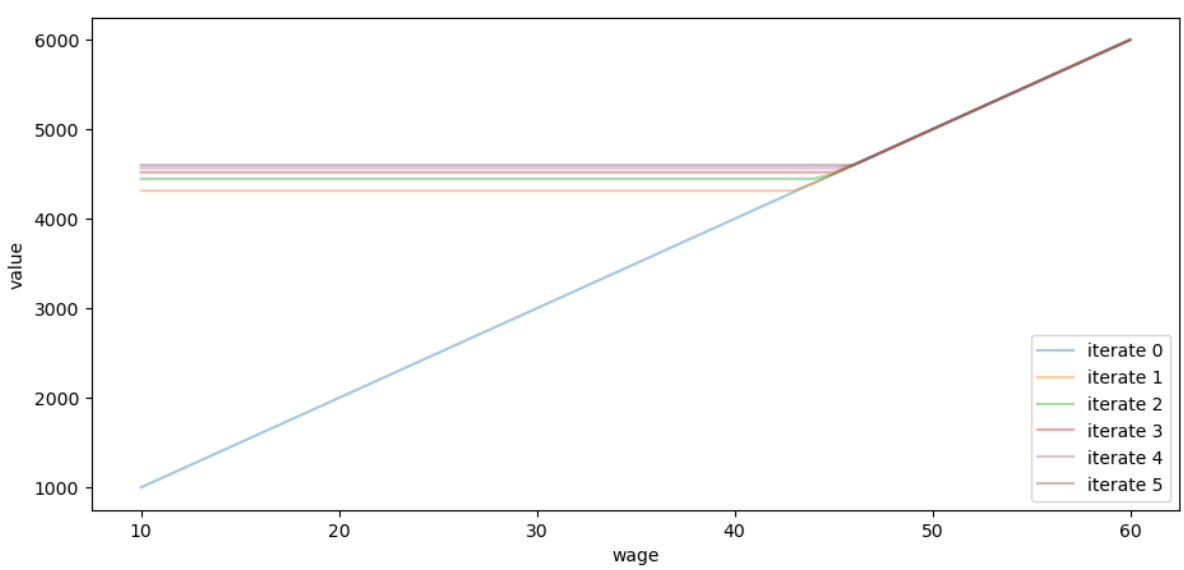
We will start from guess v given by $v(i) = w(i)/(1-\beta)$, which is the value of accepting at every given wage.

Here's a function to implement this:

```
def plot_value_function_seq(mcm, ax, num_plots=6):
    """
    Plot a sequence of value functions.

        * mcm is an instance of McCallModel
        * ax is an axes object that implements a plot method.

    """

    n = len(mcm.w)
    v = mcm.w / (1 - mcm.β)
    v_next = np.empty_like(v)
    for i in range(num_plots):
        ax.plot(mcm.w, v, '-', alpha=0.4, label=f"iterate {i}")
        # Update guess
        for j in range(n):
            v_next[j] = np.max(mcm.state_action_values(j, v))
        v[:] = v_next  # copy contents into v

    ax.legend(loc='lower right')
```

Now let's create an instance of `McCallModel` and watch iterations $T^k v$ converge from below:

```
mcm = McCallModel()

fig, ax = plt.subplots()
ax.set_xlabel('wage')
ax.set_ylabel('value')
plot_value_function_seq(mcm, ax)
plt.show()
```



You can see that convergence is occurring: successive iterates are getting closer together.

Here's a more serious iteration effort to compute the limit, which continues until measured deviation between successive iterates is below tol.

Once we obtain a good approximation to the limit, we will use it to calculate the reservation wage.

We'll be using JIT compilation via Numba to turbocharge our loops.

```
@jit(nopython=True)
def compute_reservation_wage(mcm,
                             max_iter=500,
                             tol=1e-6):

    # Simplify names
    c, β, w, q = mcm.c, mcm.β, mcm.w, mcm.q

    # == First compute the value function == #

    n = len(w)
    v = w / (1 - β)          # initial guess
    v_next = np.empty_like(v)
    j = 0
    error = tol + 1
    while j < max_iter and error > tol:

        for j in range(n):
            v_next[j] = np.max(mcm.state_action_values(j, v))

        error = np.max(np.abs(v_next - v))
        j += 1

        v[:] = v_next  # copy contents into v

    # == Now compute the reservation wage == #

    return (1 - β) * (c + β * np.sum(v * q))
```

The next line computes the reservation wage at default parameters

```
compute_reservation_wage(mcm)
```

```
47.316499710024964
```

15.3.4 Comparative Statics

Now that we know how to compute the reservation wage, let's see how it varies with parameters.

In particular, let's look at what happens when we change β and c.

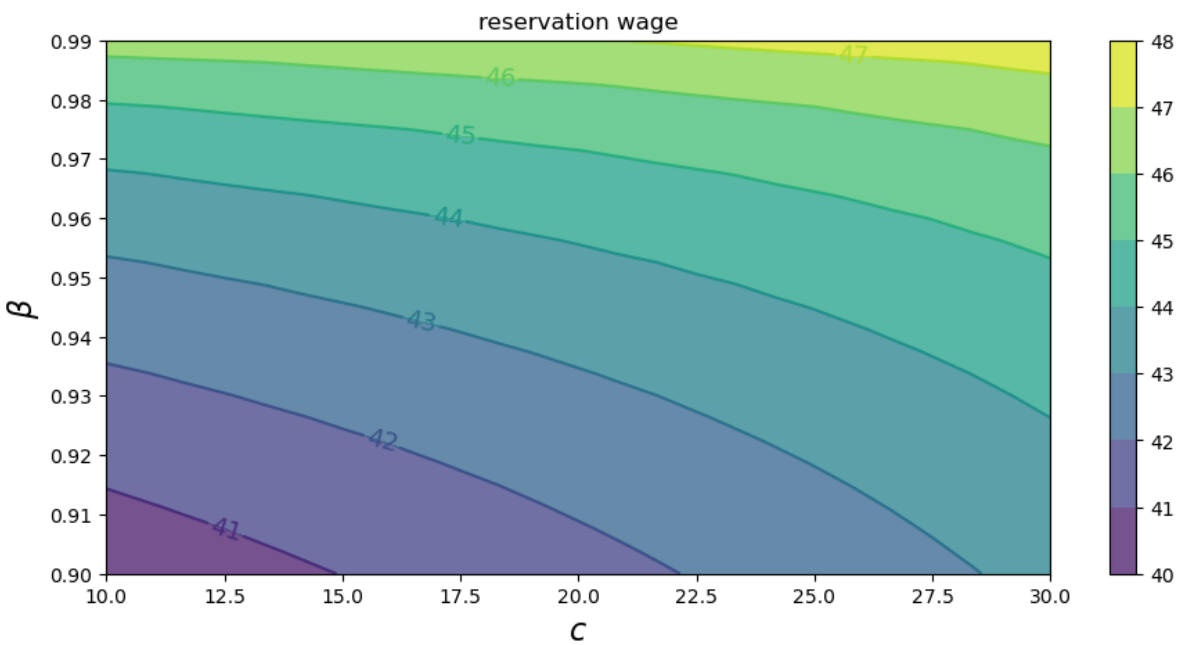
```
grid_size = 25
R = np.empty((grid_size, grid_size))

c_vals = np.linspace(10.0, 30.0, grid_size)
β_vals = np.linspace(0.9, 0.99, grid_size)

for i, c in enumerate(c_vals):
```

(continues on next page)

(continued from previous page)

```
    for j, β in enumerate(β_vals):
        mcm = McCallModel(c=c, β=β)
        R[i, j] = compute_reservation_wage(mcm)
```

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

cs1 = ax.contourf(c_vals, β_vals, R.T, alpha=0.75)
ctr1 = ax.contour(c_vals, β_vals, R.T)

plt.clabel(ctr1, inline=1, fontsize=13)
plt.colorbar(cs1, ax=ax)


ax.set_title("reservation wage")
ax.set_xlabel("$c$", fontsize=16)
ax.set_ylabel("$β$", fontsize=16)

ax.ticklabel_format(useOffset=False)

plt.show()
```



As expected, the reservation wage increases both with patience and with unemployment compensation.

15.4 Computing an Optimal Policy: Take 2

The approach to dynamic programming just described is standard and broadly applicable.

But for our McCall search model there's also an easier way that circumvents the need to compute the value function.

Let h denote the continuation value:

$$h = c + \beta \sum_{s'} v^*(s') q(s') \tag{15.6}$$

The Bellman equation can now be written as

$$v^*(s') = \max \left\{ \frac{w(s')}{1-\beta}, h \right\}$$

Substituting this last equation into (15.6) gives

$$h = c + \beta \sum_{s' \in \mathbb{S}} \max \left\{ \frac{w(s')}{1-\beta}, h \right\} q(s') \tag{15.7}$$

This is a nonlinear equation that we can solve for h.

As before, we will use successive approximations:

Step 1: pick an initial guess h.

Step 2: compute the update h' via

$$h' = c + \beta \sum_{s' \in \mathbb{S}} \max \left\{ \frac{w(s')}{1-\beta}, h \right\} q(s') \tag{15.8}$$

Step 3: calculate the deviation $|h - h'|$.

Step 4: if the deviation is larger than some fixed tolerance, set $h = h'$ and go to step 2, else return h.

One can again use the Banach contraction mapping theorem to show that this process always converges.

The big difference here, however, is that we're iterating on a scalar h, rather than an n-vector, $v(i), i = 1, \ldots, n$.

Here's an implementation:

```
@jit(nopython=True)
def compute_reservation_wage_two(mcm,
                                 max_iter=500,
                                 tol=1e-5):

    # Simplify names
    c, β, w, q = mcm.c, mcm.β, mcm.w, mcm.q

    # == First compute h == #

    h = np.sum(w * q) / (1 - β)
    i = 0
    error = tol + 1
    while i < max_iter and error > tol:

        s = np.maximum(w / (1 - β), h)
        h_next = c + β * np.sum(s * q)
```

(continues on next page)

(continued from previous page)

```
        error = np.abs(h_next - h)
        i += 1

        h = h_next

    # == Now compute the reservation wage == #

    return (1 - β) * h
```

You can use this code to solve the exercise below.

15.5 Exercises

Exercise 15.5.1

Compute the average duration of unemployment when $\beta = 0.99$ and c takes the following values

```
c_vals = np.linspace(10, 40, 25)
```

That is, start the agent off as unemployed, compute their reservation wage given the parameters, and then simulate to see how long it takes to accept.

Repeat a large number of times and take the average.

Plot mean unemployment duration as a function of c in `c_vals`.

Solution to Exercise 15.5.1

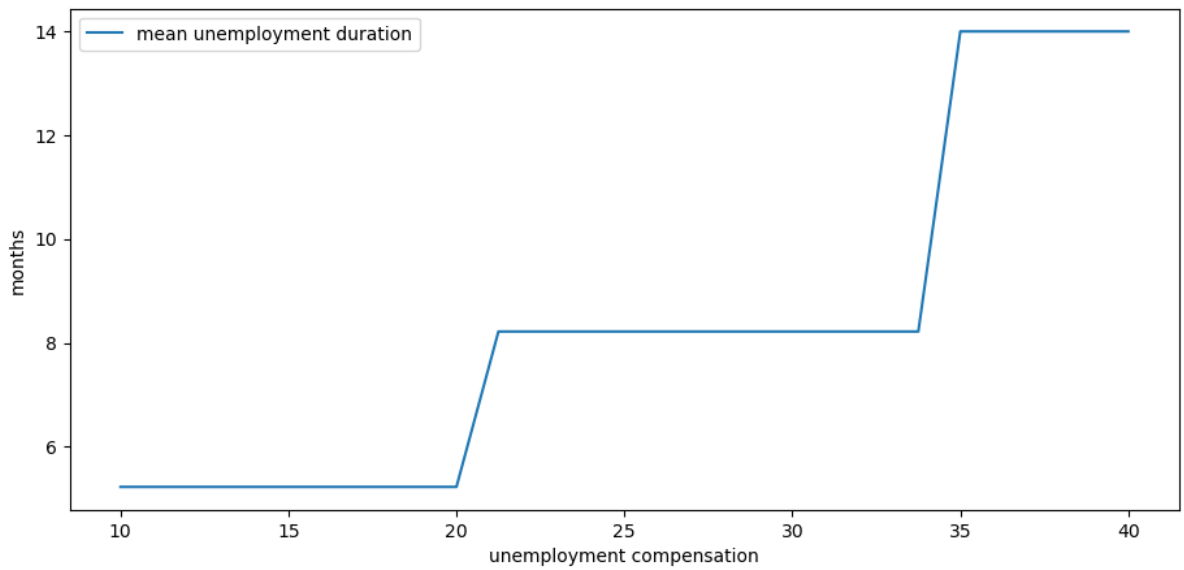
Here's one solution

```
cdf = np.cumsum(q_default)

@jit(nopython=True)
def compute_stopping_time(w_bar, seed=1234):

    np.random.seed(seed)
    t = 1
    while True:
        # Generate a wage draw
        w = w_default[qe.random.draw(cdf)]
        # Stop when the draw is above the reservation wage
        if w >= w_bar:
            stopping_time = t
            break
        else:
            t += 1
    return stopping_time

@jit(nopython=True)
def compute_mean_stopping_time(w_bar, num_reps=100000):
    obs = np.empty(num_reps)
    for i in range(num_reps):
        obs[i] = compute_stopping_time(w_bar, seed=i)
```

(continues on next page)

(continued from previous page)

```
    return obs.mean()

c_vals = np.linspace(10, 40, 25)
stop_times = np.empty_like(c_vals)
for i, c in enumerate(c_vals):
    mcm = McCallModel(c=c)
    w_bar = compute_reservation_wage_two(mcm)
    stop_times[i] = compute_mean_stopping_time(w_bar)

fig, ax = plt.subplots()

ax.plot(c_vals, stop_times, label="mean unemployment duration")
ax.set(xlabel="unemployment compensation", ylabel="months")
ax.legend()

plt.show()
```



Exercise 15.5.2

The purpose of this exercise is to show how to replace the discrete wage offer distribution used above with a continuous distribution.

This is a significant topic because many convenient distributions are continuous (i.e., have a density).

Fortunately, the theory changes little in our simple model.

Recall that h in (15.6) denotes the value of not accepting a job in this period but then behaving optimally in all subsequent periods:

To shift to a continuous offer distribution, we can replace (15.6) by

$$h = c + \beta \int v^*(s')q(s')ds'. \tag{15.9}$$

Equation (15.7) becomes

$$h = c + \beta \int \max\left\{\frac{w(s')}{1-\beta}, h\right\} q(s')ds' \tag{15.10}$$

The aim is to solve this nonlinear equation by iteration, and from it obtain the reservation wage.

Try to carry this out, setting

- the state sequence $\{s_t\}$ to be IID and standard normal and
- the wage function to be $w(s) = \exp(\mu + \sigma s)$.

You will need to implement a new version of the `McCallModel` class that assumes a lognormal wage distribution.

Calculate the integral by Monte Carlo, by averaging over a large number of wage draws.

For default parameters, use `c=25, β=0.99, σ=0.5, μ=2.5`.

Once your code is working, investigate how the reservation wage changes with c and β.

Solution to Exercise 15.5.2

Here is one solution:

```
mccall_data_continuous = [
    ('c', float64),          # unemployment compensation
    ('β', float64),          # discount factor
    ('σ', float64),          # scale parameter in lognormal distribution
    ('μ', float64),          # location parameter in lognormal distribution
    ('w_draws', float64[:])  # draws of wages for Monte Carlo
]

@jitclass(mccall_data_continuous)
class McCallModelContinuous:

    def __init__(self, c=25, β=0.99, σ=0.5, μ=2.5, mc_size=1000):

        self.c, self.β, self.σ, self.μ = c, β, σ, μ

        # Draw and store shocks
        np.random.seed(1234)
        s = np.random.randn(mc_size)
        self.w_draws = np.exp(μ+ σ * s)


@jit(nopython=True)
def compute_reservation_wage_continuous(mcmc, max_iter=500, tol=1e-5):

    c, β, σ, μ, w_draws = mcmc.c, mcmc.β, mcmc.σ, mcmc.μ, mcmc.w_draws

    h = np.mean(w_draws) / (1 - β)  # initial guess
    i = 0
    error = tol + 1
    while i < max_iter and error > tol:

        integral = np.mean(np.maximum(w_draws / (1 - β), h))
        h_next = c + β * integral
```

(continues on next page)

(continued from previous page)

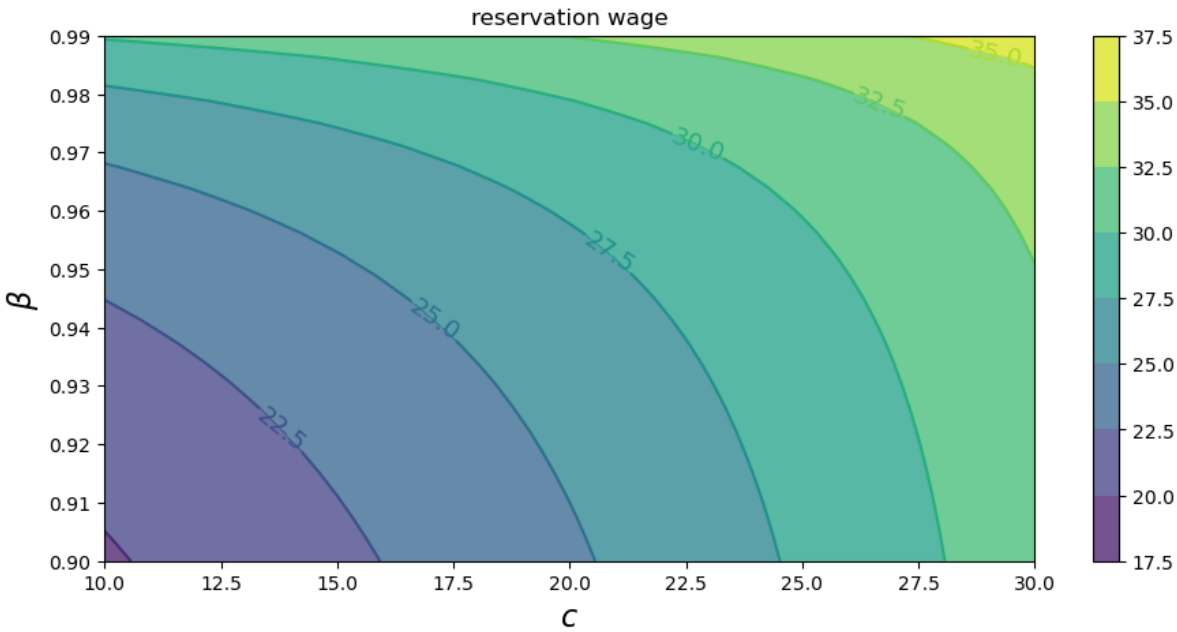
```
        error = np.abs(h_next - h)
        i += 1

        h = h_next

    # == Now compute the reservation wage == #

    return (1 - β) * h
```

Now we investigate how the reservation wage changes with c and β.

We will do this using a contour plot.

```
grid_size = 25
R = np.empty((grid_size, grid_size))

c_vals = np.linspace(10.0, 30.0, grid_size)
β_vals = np.linspace(0.9, 0.99, grid_size)

for i, c in enumerate(c_vals):
    for j, β in enumerate(β_vals):
        mcmc = McCallModelContinuous(c=c, β=β)
        R[i, j] = compute_reservation_wage_continuous(mcmc)
```

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

cs1 = ax.contourf(c_vals, β_vals, R.T, alpha=0.75)
ctr1 = ax.contour(c_vals, β_vals, R.T)

plt.clabel(ctr1, inline=1, fontsize=13)
plt.colorbar(cs1, ax=ax)


ax.set_title("reservation wage")
ax.set_xlabel("$c$", fontsize=16)
ax.set_ylabel("$β$", fontsize=16)

ax.ticklabel_format(useOffset=False)

plt.show()
```


reservation wage
0.99
0.98
0.97
0.96
0.95
0.94
0.93
0.92
0.91
0.90
β
10.0
12.5
15.0
17.5
20.0
22.5
25.0
27.5
30.0
c
35.0
32.5
30.0
27.5
25.0
22.5
37.5
35.0
32.5
30.0
27.5
25.0
22.5
20.0
17.5

CHAPTER

SIXTEEN

JOB SEARCH II: SEARCH AND SEPARATION

Contents

- *Job Search II: Search and Separation*
  - *Overview*
  - *The Model*
  - *Solving the Model*
  - *Implementation*
  - *Impact of Parameters*
  - *Exercises*

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

```
!pip install quantecon
```

16.1 Overview

Previously *we looked* at the McCall job search model [McC70] as a way of understanding unemployment and worker decisions.

One unrealistic feature of the model is that every job is permanent.

In this lecture, we extend the McCall model by introducing job separation.

Once separation enters the picture, the agent comes to view

- the loss of a job as a capital loss, and
- a spell of unemployment as an *investment* in searching for an acceptable job

The other minor addition is that a utility function will be included to make worker preferences slightly more sophisticated.

We'll need the following imports

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from numba import njit, float64
```

(continues on next page)

(continued from previous page)

```
from numba.experimental import jitclass
from quantecon.distributions import BetaBinomial
```

16.2 The Model

The model is similar to the *baseline McCall job search model*.

It concerns the life of an infinitely lived worker and

- the opportunities he or she (let's say he to save one character) has to work at different wages
- exogenous events that destroy his current job
- his decision making process while unemployed

The worker can be in one of two states: employed or unemployed.

He wants to maximize

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t u(y_t) \tag{16.1}$$

At this stage the only difference from the *baseline model* is that we've added some flexibility to preferences by introducing a utility function u.

It satisfies $u' > 0$ and $u'' < 0$.

16.2.1 The Wage Process

For now we will drop the separation of state process and wage process that we maintained for the *baseline model*.

In particular, we simply suppose that wage offers $\{w_t\}$ are IID with common distribution q.

The set of possible wage values is denoted by $\mathbb{W}$.

(Later we will go back to having a separate state process $\{s_t\}$ driving random outcomes, since this formulation is usually convenient in more sophisticated models.)

16.2.2 Timing and Decisions

At the start of each period, the agent can be either

- unemployed or
- employed at some existing wage level w_e.

At the start of a given period, the current wage offer w_t is observed.

If currently *employed*, the worker

1. receives utility $u(w_e)$ and
2. is fired with some (small) probability α.

If currently *unemployed*, the worker either accepts or rejects the current offer w_t.

If he accepts, then he begins work immediately at wage w_t.

If he rejects, then he receives unemployment compensation c.

The process then repeats.

Note: We do not allow for job search while employed—this topic is taken up in a *later lecture*.

16.3 Solving the Model

We drop time subscripts in what follows and primes denote next period values.

Let

- $v(w_e)$ be total lifetime value accruing to a worker who enters the current period *employed* with existing wage w_e
- $h(w)$ be total lifetime value accruing to a worker who who enters the current period *unemployed* and receives wage offer w.

Here *value* means the value of the objective function (16.1) when the worker makes optimal decisions at all future points in time.

Our first aim is to obtain these functions.

16.3.1 The Bellman Equations

Suppose for now that the worker can calculate the functions v and h and use them in his decision making.

Then v and h should satisfy

$$v(w_e) = u(w_e) + \beta \left[(1-\alpha) v(w_e) + \alpha \sum_{w' \in \mathbb{W}} h(w') q(w') \right] \tag{16.2}$$

and

$$h(w) = \max \left\{ v(w),\, u(c) + \beta \sum_{w' \in \mathbb{W}} h(w') q(w') \right\} \tag{16.3}$$

Equation (16.2) expresses the value of being employed at wage w_e in terms of

- current reward $u(w_e)$ plus
- discounted expected reward tomorrow, given the α probability of being fired

Equation (16.3) expresses the value of being unemployed with offer w in hand as a maximum over the value of two options: accept or reject the current offer.

Accepting transitions the worker to employment and hence yields reward $v(w)$.

Rejecting leads to unemployment compensation and unemployment tomorrow.

Equations (16.2) and (16.3) are the Bellman equations for this model.

They provide enough information to solve for both v and h.

16.3.2 A Simplifying Transformation

Rather than jumping straight into solving these equations, let's see if we can simplify them somewhat.

(This process will be analogous to our *second pass* at the plain vanilla McCall model, where we simplified the Bellman equation.)

First, let

$$d := \sum_{w' \in \mathbb{W}} h(w')q(w') \tag{16.4}$$

be the expected value of unemployment tomorrow.

We can now write (16.3) as

$$h(w) = \max\{v(w),\, u(c) + \beta d\}$$

or, shifting time forward one period

$$\sum_{w' \in \mathbb{W}} h(w')q(w') = \sum_{w' \in \mathbb{W}} \max\{v(w'),\, u(c) + \beta d\}\, q(w')$$

Using (16.4) again now gives

$$d = \sum_{w' \in \mathbb{W}} \max\{v(w'),\, u(c) + \beta d\}\, q(w') \tag{16.5}$$

Finally, (16.2) can now be rewritten as

$$v(w) = u(w) + \beta\left[(1-\alpha)v(w) + \alpha d\right] \tag{16.6}$$

In the last expression, we wrote w_e as w to make the notation simpler.

16.3.3 The Reservation Wage

Suppose we can use (16.5) and (16.6) to solve for d and v.

(We will do this soon.)

We can then determine optimal behavior for the worker.

From (16.3), we see that an unemployed agent accepts current offer w if $v(w) \geq u(c) + \beta d$.

This means precisely that the value of accepting is higher than the expected value of rejecting.

It is clear that v is (at least weakly) increasing in w, since the agent is never made worse off by a higher wage offer.

Hence, we can express the optimal choice as accepting wage offer w if and only if

$$w \geq \bar{w} \quad \text{where} \quad \bar{w} \text{ solves } v(\bar{w}) = u(c) + \beta d$$

16.3.4 Solving the Bellman Equations

We'll use the same iterative approach to solving the Bellman equations that we adopted in the *first job search lecture*.

Here this amounts to

1. make guesses for d and v
2. plug these guesses into the right-hand sides of (16.5) and (16.6)

3. update the left-hand sides from this rule and then repeat

In other words, we are iterating using the rules

$$d_{n+1} = \sum_{w' \in \mathbb{W}} \max \{v_n(w'),\, u(c) + \beta d_n\}\, q(w') \tag{16.7}$$

$$v_{n+1}(w) = u(w) + \beta \left[(1-\alpha) v_n(w) + \alpha d_n\right] \tag{16.8}$$

starting from some initial conditions d_0, v_0.

As before, the system always converges to the true solutions—in this case, the v and d that solve (16.5) and (16.6).

(A proof can be obtained via the Banach contraction mapping theorem.)

16.4 Implementation

Let's implement this iterative process.

In the code, you'll see that we use a class to store the various parameters and other objects associated with a given model.

This helps to tidy up the code and provides an object that's easy to pass to functions.

The default utility function is a CRRA utility function

```
@njit
def u(c, σ=2.0):
    return (c**(1 - σ) - 1) / (1 - σ)
```

Also, here's a default wage distribution, based around the BetaBinomial distribution:

```
n = 60                                  # n possible outcomes for w
w_default = np.linspace(10, 20, n)      # wages between 10 and 20
a, b = 600, 400                         # shape parameters
dist = BetaBinomial(n-1, a, b)
q_default = dist.pdf()
```

Here's our jitted class for the McCall model with separation.

```
mccall_data = [
    ('α', float64),       # job separation rate
    ('β', float64),       # discount factor
    ('c', float64),       # unemployment compensation
    ('w', float64[:]),    # list of wage values
    ('q', float64[:])     # pmf of random variable w
]

@jitclass(mccall_data)
class McCallModel:
    """
    Stores the parameters and functions associated with a given model.
    """

    def __init__(self, α=0.2, β=0.98, c=6.0, w=w_default, q=q_default):

        self.α, self.β, self.c, self.w, self.q = α, β, c, w, q
```

(continues on next page)

(continued from previous page)

```
    def update(self, v, d):

        α, β, c, w, q = self.α, self.β, self.c, self.w, self.q

        v_new = np.empty_like(v)

        for i in range(len(w)):
            v_new[i] = u(w[i]) + β * ((1 - α) * v[i] + α * d)

        d_new = np.sum(np.maximum(v, u(c) + β * d) * q)

        return v_new, d_new
```

Now we iterate until successive realizations are closer together than some small tolerance level.

We then return the current iterate as an approximate solution.

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

    * mcm is an instance of McCallModel
    """

    v = np.ones_like(mcm.w)     # Initial guess of v
    d = 1                       # Initial guess of d
    i = 0
    error = tol + 1

    while error > tol and i < max_iter:
        v_new, d_new = mcm.update(v, d)
        error_1 = np.max(np.abs(v_new - v))
        error_2 = np.abs(d_new - d)
        error = max(error_1, error_2)
        v = v_new
        d = d_new
        i += 1

    return v, d
```

16.4.1 The Reservation Wage: First Pass

The optimal choice of the agent is summarized by the reservation wage.

As discussed above, the reservation wage is the $\bar{w}$ that solves $v(\bar{w}) = h$ where $h := u(c) + \beta d$ is the continuation value.

Let's compare v and h to see what they look like.

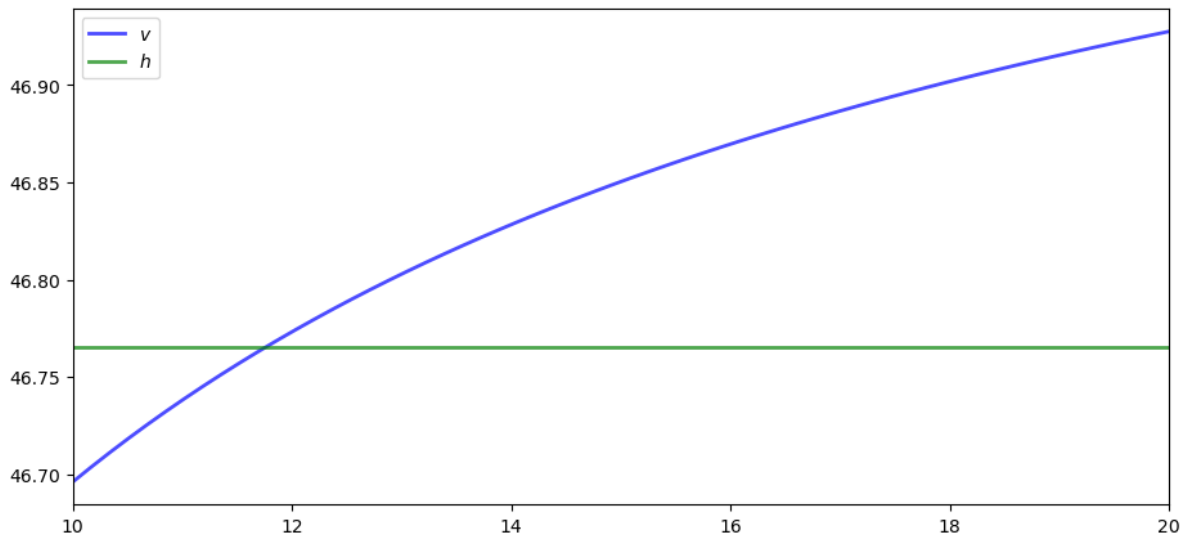
We'll use the default parameterizations found in the code above.

```
mcm = McCallModel()
v, d = solve_model(mcm)
h = u(mcm.c) + mcm.β * d
```

(continues on next page)

(continued from previous page)

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

ax.plot(mcm.w, v, 'b-', lw=2, alpha=0.7, label='$v$')
ax.plot(mcm.w, [h] * len(mcm.w),
        'g-', lw=2, alpha=0.7, label='$h$')
ax.set_xlim(min(mcm.w), max(mcm.w))
ax.legend()

plt.show()
```



The value v is increasing because higher w generates a higher wage flow conditional on staying employed.

16.4.2 The Reservation Wage: Computation

Here's a function `compute_reservation_wage` that takes an instance of `McCallModel` and returns the associated reservation wage.

```
@njit
def compute_reservation_wage(mcm):
    """
    Computes the reservation wage of an instance of the McCall model
    by finding the smallest w such that v(w) >= h.

    If no such w exists, then w_bar is set to np.inf.
    """

    v, d = solve_model(mcm)
    h = u(mcm.c) + mcm.β * d

    i = np.searchsorted(v, h, side='right')
    w_bar = mcm.w[i]

    return w_bar
```

Next we will investigate how the reservation wage varies with parameters.

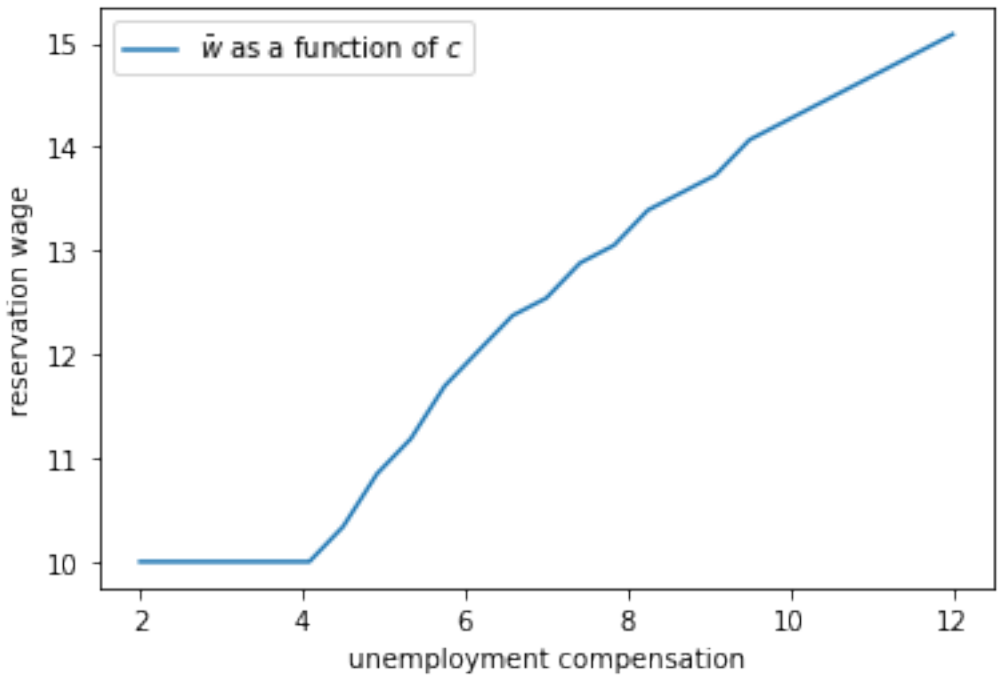
16.5 Impact of Parameters

In each instance below, we'll show you a figure and then ask you to reproduce it in the exercises.

16.5.1 The Reservation Wage and Unemployment Compensation

First, let's look at how $\bar{w}$ varies with unemployment compensation.

In the figure below, we use the default parameters in the `McCallModel` class, apart from c (which takes the values given on the horizontal axis)



As expected, higher unemployment compensation causes the worker to hold out for higher wages.

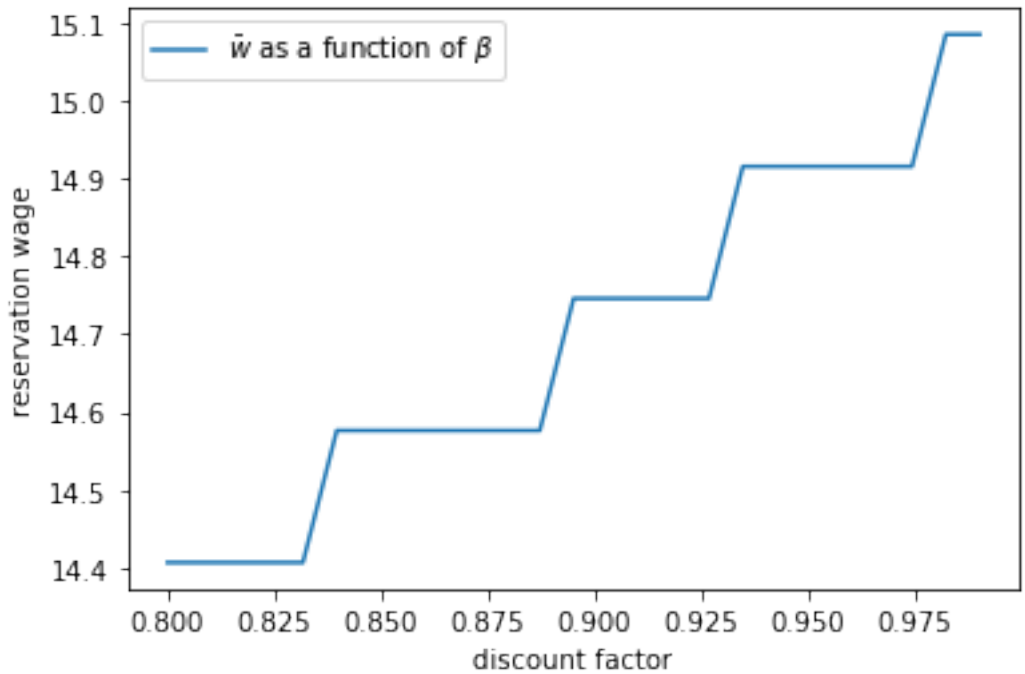
In effect, the cost of continuing job search is reduced.

16.5.2 The Reservation Wage and Discounting

Next, let's investigate how $\bar{w}$ varies with the discount factor.

The next figure plots the reservation wage associated with different values of β

Again, the results are intuitive: More patient workers will hold out for higher wages.



16.5.3 The Reservation Wage and Job Destruction

Finally, let's look at how $\bar{w}$ varies with the job separation rate α.

Higher α translates to a greater chance that a worker will face termination in each period once employed.

Once more, the results are in line with our intuition.

If the separation rate is high, then the benefit of holding out for a higher wage falls.

Hence the reservation wage is lower.

16.6 Exercises

Exercise 16.6.1

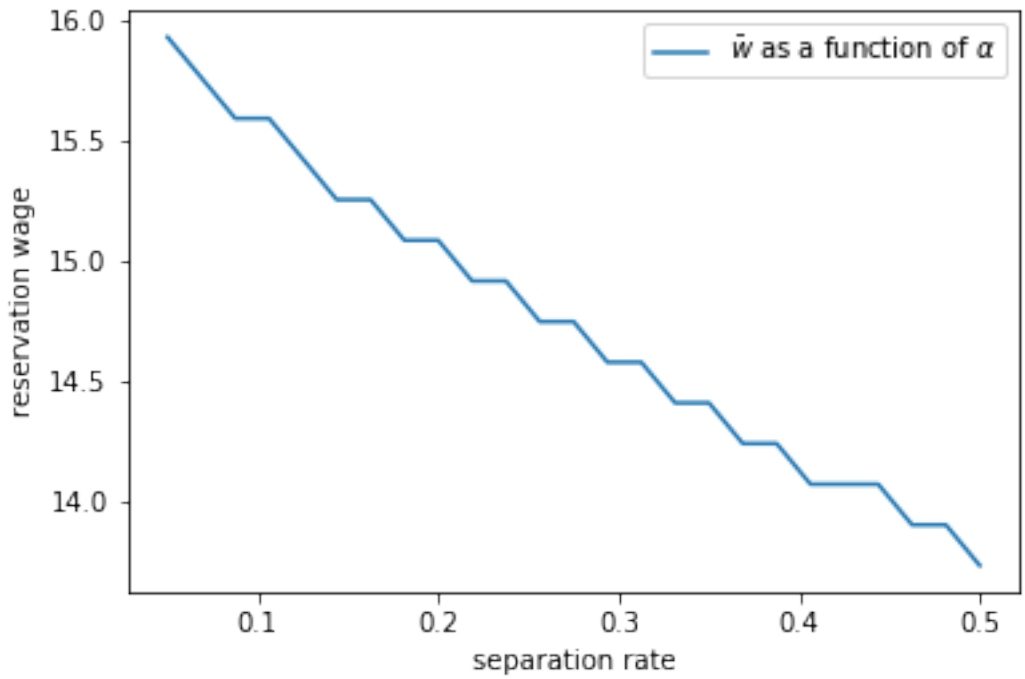
Reproduce all the reservation wage figures shown above.

Regarding the values on the horizontal axis, use

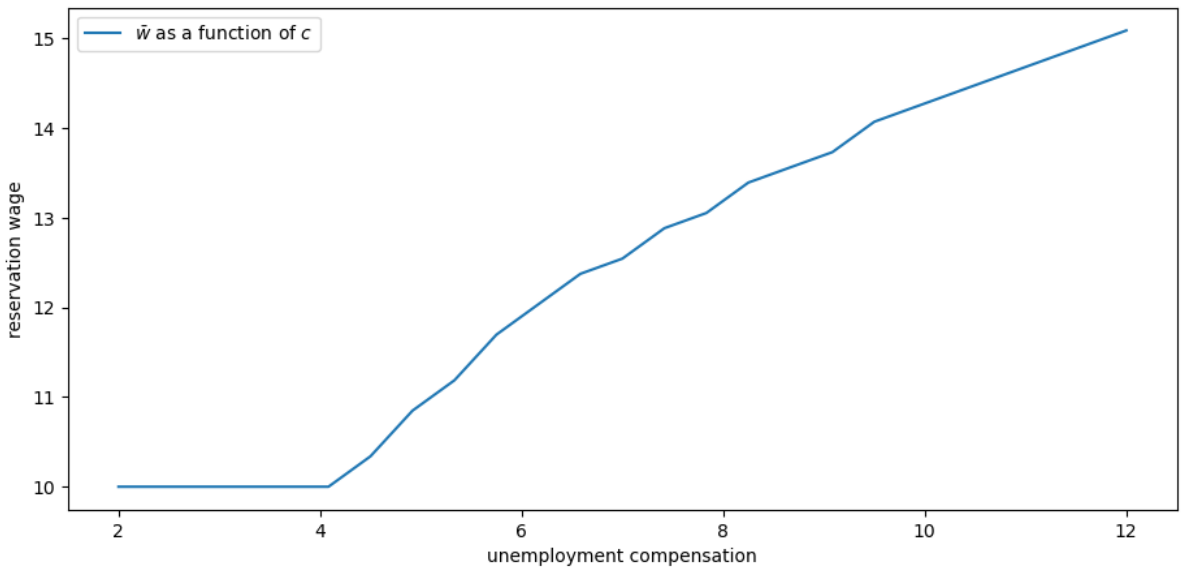
```
grid_size = 25
c_vals = np.linspace(2, 12, grid_size)          # unemployment compensation
beta_vals = np.linspace(0.8, 0.99, grid_size)   # discount factors
alpha_vals = np.linspace(0.05, 0.5, grid_size)  # separation rate
```

Solution to Exercise 16.6.1

Here's the first figure.

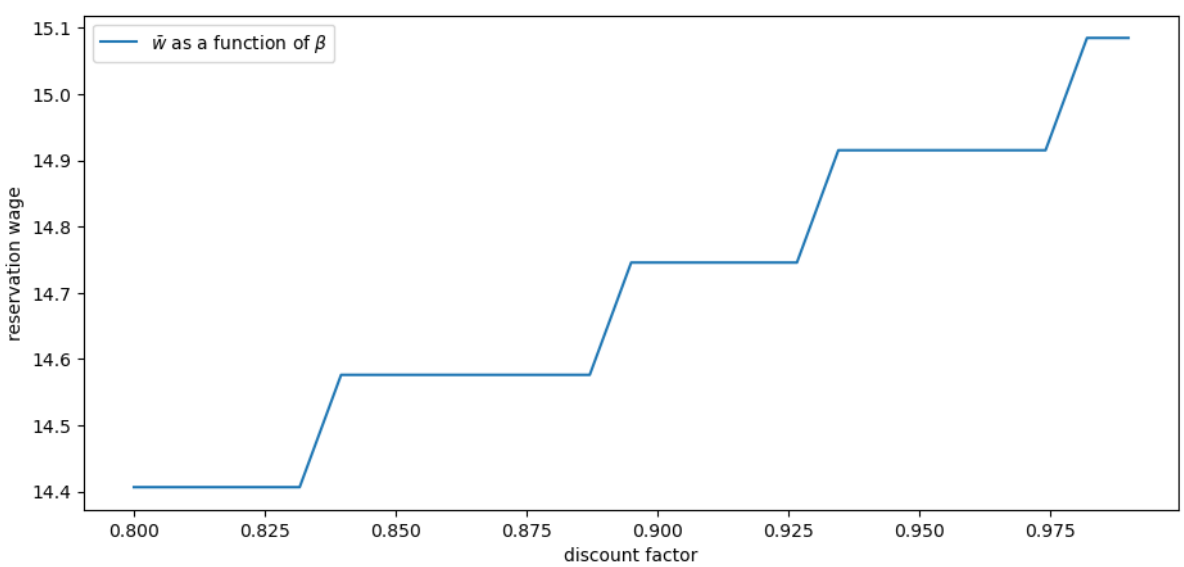



```
mcm = McCallModel()

w_bar_vals = np.empty_like(c_vals)

fig, ax = plt.subplots()

for i, c in enumerate(c_vals):
    mcm.c = c
    w_bar = compute_reservation_wage(mcm)
    w_bar_vals[i] = w_bar

ax.set(xlabel='unemployment compensation',
       ylabel='reservation wage')
ax.plot(c_vals, w_bar_vals, label=r'$\bar w$ as a function of $c$')
ax.legend()

plt.show()
```



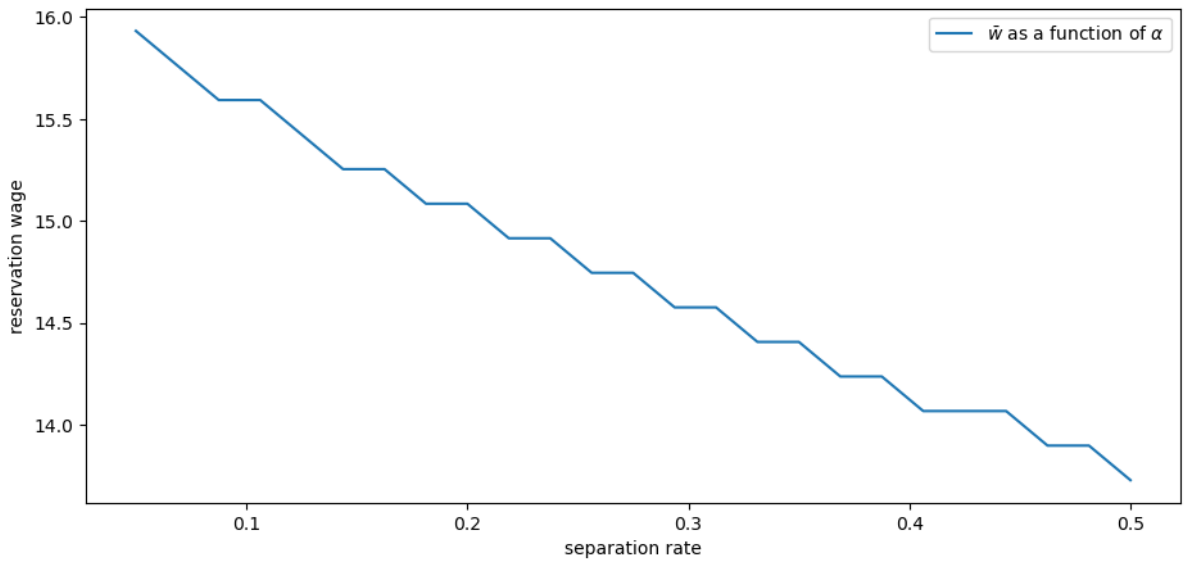
Here's the second one.

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

for i, β in enumerate(beta_vals):
    mcm.β = β
    w_bar = compute_reservation_wage(mcm)
    w_bar_vals[i] = w_bar

ax.set(xlabel='discount factor', ylabel='reservation wage')
ax.plot(beta_vals, w_bar_vals, label=r'$\bar w$ as a function of $\beta$')
ax.legend()

plt.show()
```



Here's the third.

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

for i, α in enumerate(alpha_vals):
    mcm.α = α
    w_bar = compute_reservation_wage(mcm)
    w_bar_vals[i] = w_bar

ax.set(xlabel='separation rate', ylabel='reservation wage')
ax.plot(alpha_vals, w_bar_vals, label=r'$\bar w$ as a function of $\alpha$')
ax.legend()

plt.show()
```



CHAPTER

SEVENTEEN

JOB SEARCH III: FITTED VALUE FUNCTION ITERATION

Contents

- *Job Search III: Fitted Value Function Iteration*
  - *Overview*
  - *The Algorithm*
  - *Implementation*
  - *Exercises*

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

```
!pip install interpolation
```

17.1 Overview

In this lecture we again study the *McCall job search model with separation*, but now with a continuous wage distribution.

While we already considered continuous wage distributions briefly in the exercises of the *first job search lecture*, the change was relatively trivial in that case.

This is because we were able to reduce the problem to solving for a single scalar value (the continuation value).

Here, with separation, the change is less trivial, since a continuous wage distribution leads to an uncountably infinite state space.

The infinite state space leads to additional challenges, particularly when it comes to applying value function iteration (VFI).

These challenges will lead us to modify VFI by adding an interpolation step.

The combination of VFI and this interpolation step is called **fitted value function iteration** (fitted VFI).

Fitted VFI is very common in practice, so we will take some time to work through the details.

We will use the following imports:

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

(continues on next page)

(continued from previous page)

```
from numba import njit, float64
from numba.experimental import jitclass
```

```
/home/runner/miniconda3/envs/quantecon/lib/python3.11/site-packages/numba/core/
↪decorators.py:262: NumbaDeprecationWarning: numba.generated_jit is deprecated.␣
↪Please see the documentation at: https://numba.readthedocs.io/en/stable/
↪reference/deprecation.html#deprecation-of-generated-jit for more information and␣
↪advice on a suitable replacement.
  warnings.warn(msg, NumbaDeprecationWarning)
```

17.2 The Algorithm

The model is the same as the McCall model with job separation we *studied before*, except that the wage offer distribution is continuous.

We are going to start with the two Bellman equations we obtained for the model with job separation after *a simplifying transformation*.

Modified to accommodate continuous wage draws, they take the following form:

$$d = \int \max\left\{v(w'),\, u(c) + \beta d\right\} q(w') dw' \tag{17.1}$$

and

$$v(w) = u(w) + \beta\left[(1-\alpha)v(w) + \alpha d\right] \tag{17.2}$$

The unknowns here are the function v and the scalar d.

The difference between these and the pair of Bellman equations we previously worked on are

1. in (17.1), what used to be a sum over a finite number of wage values is an integral over an infinite set.
2. The function v in (17.2) is defined over all $w \in \mathbb{R}_+$.

The function q in (17.1) is the density of the wage offer distribution.

Its support is taken as equal to $\mathbb{R}_+$.

17.2.1 Value Function Iteration

In theory, we should now proceed as follows:

1. Begin with a guess v, d for the solutions to (17.1)–(17.2).
2. Plug v, d into the right hand side of (17.1)–(17.2) and compute the left hand side to obtain updates v', d'
3. Unless some stopping condition is satisfied, set $(v, d) = (v', d')$ and go to step 2.

However, there is a problem we must confront before we implement this procedure: The iterates of the value function can neither be calculated exactly nor stored on a computer.

To see the issue, consider (17.2).

Even if v is a known function, the only way to store its update v' is to record its value $v'(w)$ for every $w \in \mathbb{R}_+$.

Clearly, this is impossible.

17.2.2 Fitted Value Function Iteration

What we will do instead is use **fitted value function iteration**.

The procedure is as follows:

Let a current guess v be given.

Now we record the value of the function v' at only finitely many "grid" points $w_1 < w_2 < \cdots < w_I$ and then reconstruct v' from this information when required.

More precisely, the algorithm will be

1. Begin with an array **v** representing the values of an initial guess of the value function on some grid points $\{w_i\}$.
2. Build a function v on the state space $\mathbb{R}_+$ by interpolation or approximation, based on **v** and $\{w_i\}$.
3. Obtain and record the samples of the updated function $v'(w_i)$ on each grid point w_i.
4. Unless some stopping condition is satisfied, take this as the new array and go to step 1.

How should we go about step 2?

This is a problem of function approximation, and there are many ways to approach it.

What's important here is that the function approximation scheme must not only produce a good approximation to each v, but also that it combines well with the broader iteration algorithm described above.

One good choice from both respects is continuous piecewise linear interpolation.

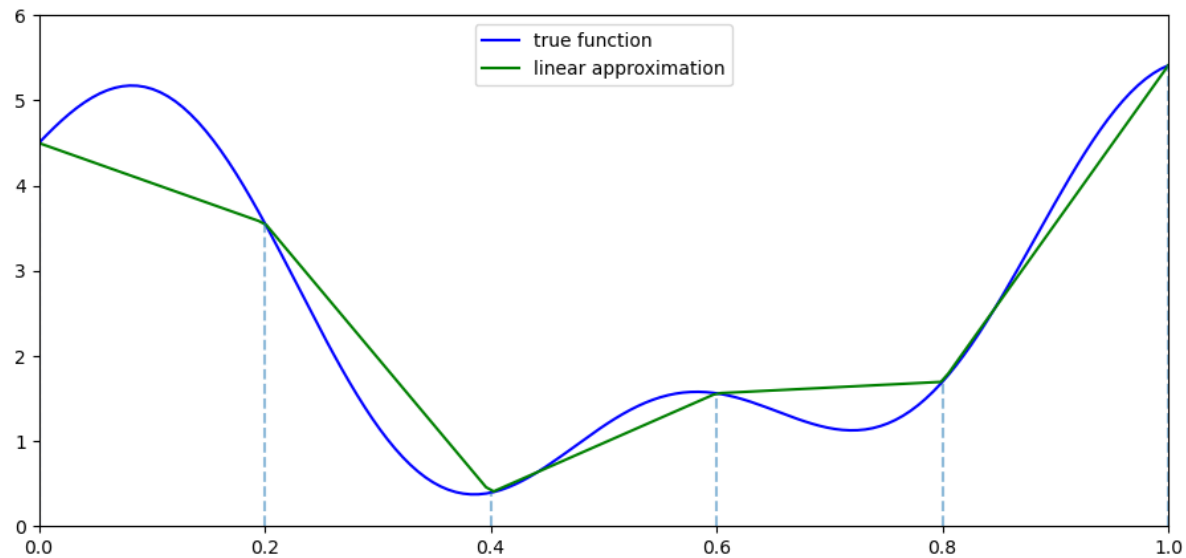
This method

1. combines well with value function iteration (see., e.g., [Gor95] or [Sta08]) and
2. preserves useful shape properties such as monotonicity and concavity/convexity.

Linear interpolation will be implemented using a JIT-aware Python interpolation library called interpolation.py.

The next figure illustrates piecewise linear interpolation of an arbitrary function on grid points $0, 0.2, 0.4, 0.6, 0.8, 1$.

```
def f(x):
    y1 = 2 * np.cos(6 * x) + np.sin(14 * x)
    return y1 + 2.5

c_grid = np.linspace(0, 1, 6)
f_grid = np.linspace(0, 1, 150)

def Af(x):
    return interp(c_grid, f(c_grid), x)

fig, ax = plt.subplots()

ax.plot(f_grid, f(f_grid), 'b-', label='true function')
ax.plot(f_grid, Af(f_grid), 'g-', label='linear approximation')
ax.vlines(c_grid, c_grid * 0, f(c_grid), linestyle='dashed', alpha=0.5)

ax.legend(loc="upper center")

ax.set(xlim=(0, 1), ylim=(0, 6))
plt.show()
```



17.3 Implementation

The first step is to build a jitted class for the McCall model with separation and a continuous wage offer distribution.

We will take the utility function to be the log function for this application, with $u(c) = \ln c$.

We will adopt the lognormal distribution for wages, with $w = \exp(\mu + \sigma z)$ when z is standard normal and μ, σ are parameters.

```
@njit
def lognormal_draws(n=1000, μ=2.5, σ=0.5, seed=1234):
    np.random.seed(seed)
    z = np.random.randn(n)
    w_draws = np.exp(μ + σ * z)
    return w_draws
```

Here's our class.

```
mccall_data_continuous = [
    ('c', float64),          # unemployment compensation
    ('α', float64),          # job separation rate
    ('β', float64),          # discount factor
    ('w_grid', float64[:]),  # grid of points for fitted VFI
    ('w_draws', float64[:])  # draws of wages for Monte Carlo
]

@jitclass(mccall_data_continuous)
class McCallModelContinuous:

    def __init__(self,
                 c=1,
                 α=0.1,
                 β=0.96,
                 grid_min=1e-10,
                 grid_max=5,
```

(continues on next page)

(continued from previous page)

```
                 grid_size=100,
                 w_draws=lognormal_draws()):

        self.c, self.α, self.β = c, α, β

        self.w_grid = np.linspace(grid_min, grid_max, grid_size)
        self.w_draws = w_draws

    def update(self, v, d):

        # Simplify names
        c, α, β = self.c, self.α, self.β
        w = self.w_grid
        u = lambda x: np.log(x)

        # Interpolate array represented value function
        vf = lambda x: interp(w, v, x)

        # Update d using Monte Carlo to evaluate integral
        d_new = np.mean(np.maximum(vf(self.w_draws), u(c) + β * d))

        # Update v
        v_new = u(w) + β * ((1 - α) * v + α * d)

        return v_new, d_new
```

We then return the current iterate as an approximate solution.

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

    * mcm is an instance of McCallModel
    """

    v = np.ones_like(mcm.w_grid)    # Initial guess of v
    d = 1                           # Initial guess of d
    i = 0
    error = tol + 1

    while error > tol and i < max_iter:
        v_new, d_new = mcm.update(v, d)
        error_1 = np.max(np.abs(v_new - v))
        error_2 = np.abs(d_new - d)
        error = max(error_1, error_2)
        v = v_new
        d = d_new
        i += 1

    return v, d
```

Here's a function `compute_reservation_wage` that takes an instance of `McCallModelContinuous` and returns the associated reservation wage.

If $v(w) < h$ for all w, then the function returns np.inf

```
@njit
def compute_reservation_wage(mcm):
    """
    Computes the reservation wage of an instance of the McCall model
    by finding the smallest w such that v(w) >= h.

    If no such w exists, then w_bar is set to np.inf.
    """
    u = lambda x: np.log(x)

    v, d = solve_model(mcm)
    h = u(mcm.c) + mcm.β * d

    w_bar = np.inf
    for i, wage in enumerate(mcm.w_grid):
        if v[i] > h:
            w_bar = wage
            break

    return w_bar
```

The exercises ask you to explore the solution and how it changes with parameters.

17.4 Exercises

Exercise 17.4.1

Use the code above to explore what happens to the reservation wage when the wage parameter μ changes.

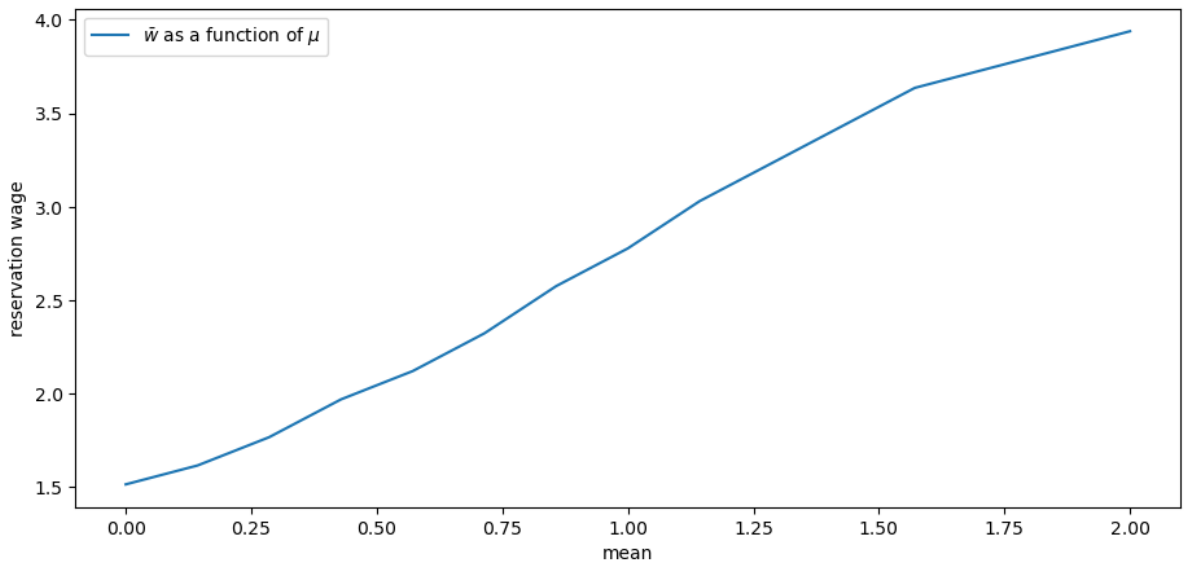
Use the default parameters and μ in `mu_vals = np.linspace(0.0, 2.0, 15)`.

Is the impact on the reservation wage as you expected?

Solution to Exercise 17.4.1

Here is one solution

```
mcm = McCallModelContinuous()
mu_vals = np.linspace(0.0, 2.0, 15)
w_bar_vals = np.empty_like(mu_vals)

fig, ax = plt.subplots()

for i, m in enumerate(mu_vals):
    mcm.w_draws = lognormal_draws(μ=m)
    w_bar = compute_reservation_wage(mcm)
    w_bar_vals[i] = w_bar

ax.set(xlabel='mean', ylabel='reservation wage')
ax.plot(mu_vals, w_bar_vals, label=r'$\bar w$ as a function of $\mu$')
ax.legend()

plt.show()
```



Not surprisingly, the agent is more inclined to wait when the distribution of offers shifts to the right.

Exercise 17.4.2

Let us now consider how the agent responds to an increase in volatility.

To try to understand this, compute the reservation wage when the wage offer distribution is uniform on $(m - s, m + s)$ and s varies.

The idea here is that we are holding the mean constant and spreading the support.

(This is a form of *mean-preserving spread*.)

Use `s_vals = np.linspace(1.0, 2.0, 15)` and `m = 2.0`.

State how you expect the reservation wage to vary with s.

Now compute it. Is this as you expected?

Solution to Exercise 17.4.2

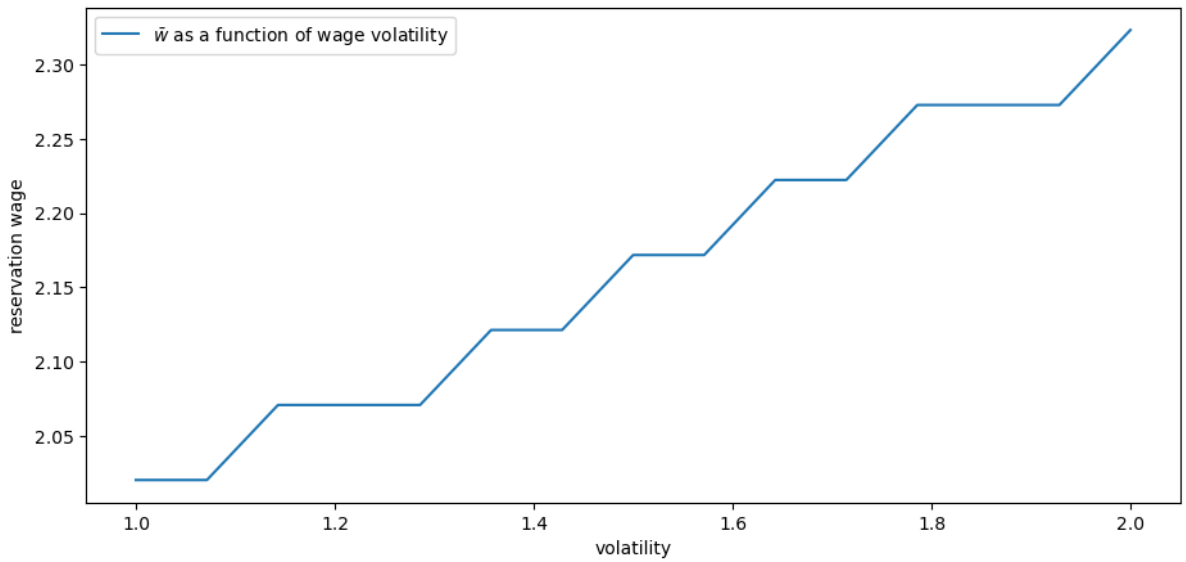
Here is one solution

```
mcm = McCallModelContinuous()
s_vals = np.linspace(1.0, 2.0, 15)
m = 2.0
w_bar_vals = np.empty_like(s_vals)

fig, ax = plt.subplots()

for i, s in enumerate(s_vals):
    a, b = m - s, m + s
    mcm.w_draws = np.random.uniform(low=a, high=b, size=10_000)
    w_bar = compute_reservation_wage(mcm)
    w_bar_vals[i] = w_bar
```

(continues on next page)

(continued from previous page)

```
ax.set(xlabel='volatility', ylabel='reservation wage')
ax.plot(s_vals, w_bar_vals, label=r'$\bar w$ as a function of wage volatility')
ax.legend()

plt.show()
```



The reservation wage increases with volatility.

One might think that higher volatility would make the agent more inclined to take a given offer, since doing so represents certainty and waiting represents risk.

But job search is like holding an option: the worker is only exposed to upside risk (since, in a free market, no one can force them to take a bad offer).

More volatility means higher upside potential, which encourages the agent to wait.

CHAPTER

EIGHTEEN

JOB SEARCH IV: CORRELATED WAGE OFFERS

Contents

- *Job Search IV: Correlated Wage Offers*
  - *Overview*
  - *The Model*
  - *Implementation*
  - *Unemployment Duration*
  - *Exercises*

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

```
!pip install quantecon
!pip install interpolation
```

18.1 Overview

In this lecture we solve a *McCall style job search model* with persistent and transitory components to wages.

In other words, we relax the unrealistic assumption that randomness in wages is independent over time.

At the same time, we will go back to assuming that jobs are permanent and no separation occurs.

This is to keep the model relatively simple as we study the impact of correlation.

We will use the following imports:

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
import quantecon as qe
from interpolation import interp
from numpy.random import randn
from numba import njit, prange, float64
from numba.experimental import jitclass
```

```
/home/runner/miniconda3/envs/quantecon/lib/python3.11/site-packages/numba/core/
↪decorators.py:262: NumbaDeprecationWarning: numba.generated_jit is deprecated.␣
↪Please see the documentation at: https://numba.readthedocs.io/en/stable/
↪reference/deprecation.html#deprecation-of-generated-jit for more information and␣
↪advice on a suitable replacement.
  warnings.warn(msg, NumbaDeprecationWarning)
```

18.2 The Model

Wages at each point in time are given by

$$w_t = \exp(z_t) + y_t$$

where

$$y_t \sim \exp(\mu + s\zeta_t) \quad \text{and} \quad z_{t+1} = d + \rho z_t + \sigma \epsilon_{t+1}$$

Here $\{\zeta_t\}$ and $\{\epsilon_t\}$ are both IID and standard normal.

Here $\{y_t\}$ is a transitory component and $\{z_t\}$ is persistent.

As before, the worker can either

1. accept an offer and work permanently at that wage, or
2. take unemployment compensation c and wait till next period.

The value function satisfies the Bellman equation

$$v^*(w, z) = \max \left\{ \frac{u(w)}{1-\beta}, u(c) + \beta \, \mathbb{E}_z v^*(w', z') \right\}$$

In this express, u is a utility function and $\mathbb{E}_z$ is expectation of next period variables given current z.

The variable z enters as a state in the Bellman equation because its current value helps predict future wages.

18.2.1 A Simplification

There is a way that we can reduce dimensionality in this problem, which greatly accelerates computation.

To start, let f^* be the continuation value function, defined by

$$f^*(z) := u(c) + \beta \, \mathbb{E}_z v^*(w', z')$$

The Bellman equation can now be written

$$v^*(w, z) = \max \left\{ \frac{u(w)}{1-\beta}, \, f^*(z) \right\}$$

Combining the last two expressions, we see that the continuation value function satisfies

$$f^*(z) = u(c) + \beta \, \mathbb{E}_z \max \left\{ \frac{u(w')}{1-\beta}, f^*(z') \right\}$$

We'll solve this functional equation for f^* by introducing the operator

$$Qf(z) = u(c) + \beta \, \mathbb{E}_z \max \left\{ \frac{u(w')}{1-\beta}, f(z') \right\}$$

By construction, f^* is a fixed point of Q, in the sense that $Qf^* = f^*$.

Under mild assumptions, it can be shown that Q is a contraction mapping over a suitable space of continuous functions on $\mathbb{R}$.

By Banach's contraction mapping theorem, this means that f^* is the unique fixed point and we can calculate it by iterating with Q from any reasonable initial condition.

Once we have f^*, we can solve the search problem by stopping when the reward for accepting exceeds the continuation value, or

$$\frac{u(w)}{1-\beta} \geq f^*(z)$$

For utility we take $u(c) = \ln(c)$.

The reservation wage is the wage where equality holds in the last expression.

That is,

$$\bar{w}(z) := \exp(f^*(z)(1-\beta)) \tag{18.1}$$

Our main aim is to solve for the reservation rule and study its properties and implications.

18.3 Implementation

Let f be our initial guess of f^*.

When we iterate, we use the *fitted value function iteration* algorithm.

In particular, f and all subsequent iterates are stored as a vector of values on a grid.

These points are interpolated into a function as required, using piecewise linear interpolation.

The integral in the definition of Qf is calculated by Monte Carlo.

The following list helps Numba by providing some type information about the data we will work with.

```
job_search_data = [
     ('μ', float64),             # transient shock log mean
     ('s', float64),             # transient shock log variance
     ('d', float64),             # shift coefficient of persistent state
     ('ρ', float64),             # correlation coefficient of persistent state
     ('σ', float64),             # state volatility
     ('β', float64),             # discount factor
     ('c', float64),             # unemployment compensation
     ('z_grid', float64[:]),     # grid over the state space
     ('e_draws', float64[:,:])   # Monte Carlo draws for integration
]
```

Here's a class that stores the data and the right hand side of the Bellman equation.

Default parameter values are embedded in the class.

```
@jitclass(job_search_data)
class JobSearch:

    def __init__(self,
                 μ=0.0,        # transient shock log mean
```

(continues on next page)

(continued from previous page)

```
                 s=1.0,        # transient shock log variance
                 d=0.0,        # shift coefficient of persistent state
                 ρ=0.9,        # correlation coefficient of persistent state
                 σ=0.1,        # state volatility
                 β=0.98,       # discount factor
                 c=5,          # unemployment compensation
                 mc_size=1000,
                 grid_size=100):

        self.μ, self.s, self.d,  = μ, s, d,
        self.ρ, self.σ, self.β, self.c = ρ, σ, β, c

        # Set up grid
        z_mean = d / (1 - ρ)
        z_sd = σ / np.sqrt(1 - ρ**2)
        k = 3  # std devs from mean
        a, b = z_mean - k * z_sd, z_mean + k * z_sd
        self.z_grid = np.linspace(a, b, grid_size)

        # Draw and store shocks
        np.random.seed(1234)
        self.e_draws = randn(2, mc_size)

    def parameters(self):
        """
        Return all parameters as a tuple.
        """
        return self.μ, self.s, self.d, \
                self.ρ, self.σ, self.β, self.c
```

Next we implement the Q operator.

```
@njit(parallel=True)
def Q(js, f_in, f_out):
    """
    Apply the operator Q.

        * js is an instance of JobSearch
        * f_in and f_out are arrays that represent f and Qf respectively

    """

    μ, s, d, ρ, σ, β, c = js.parameters()
    M = js.e_draws.shape[1]

    for i in prange(len(js.z_grid)):
        z = js.z_grid[i]
        expectation = 0.0
        for m in range(M):
            e1, e2 = js.e_draws[:, m]
            z_next = d + ρ * z + σ * e1
            go_val = interp(js.z_grid, f_in, z_next)      # f(z')
            y_next = np.exp(μ + s * e2)                   # y' draw
            w_next = np.exp(z_next) + y_next              # w' draw
            stop_val = np.log(w_next) / (1 - β)
            expectation += max(stop_val, go_val)
```

(continues on next page)

(continued from previous page)

```
        expectation = expectation / M
        f_out[i] = np.log(c) + β * expectation
```

Here's a function to compute an approximation to the fixed point of Q.

```
def compute_fixed_point(js,
                        use_parallel=True,
                        tol=1e-4,
                        max_iter=1000,
                        verbose=True,
                        print_skip=25):

    f_init = np.full(len(js.z_grid), np.log(js.c))
    f_out = np.empty_like(f_init)

    # Set up loop
    f_in = f_init
    i = 0
    error = tol + 1

    while i < max_iter and error > tol:
        Q(js, f_in, f_out)
        error = np.max(np.abs(f_in - f_out))
        i += 1
        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")
        f_in[:] = f_out

    if error > tol:
        print("Failed to converge!")
    elif verbose:
        print(f"\nConverged in {i} iterations.")

    return f_out
```

Let's try generating an instance and solving the model.

```
js = JobSearch()

qe.tic()
f_star = compute_fixed_point(js, verbose=True)
qe.toc()
```

```
Error at iteration 25 is 0.576247783958749.
```

```
Error at iteration 50 is 0.11808817939665062.
```

```
Error at iteration 75 is 0.0285774413852522.
```

```
Error at iteration 100 is 0.007158336385160169.
```

```
Error at iteration 125 is 0.0018027870994501427.
```

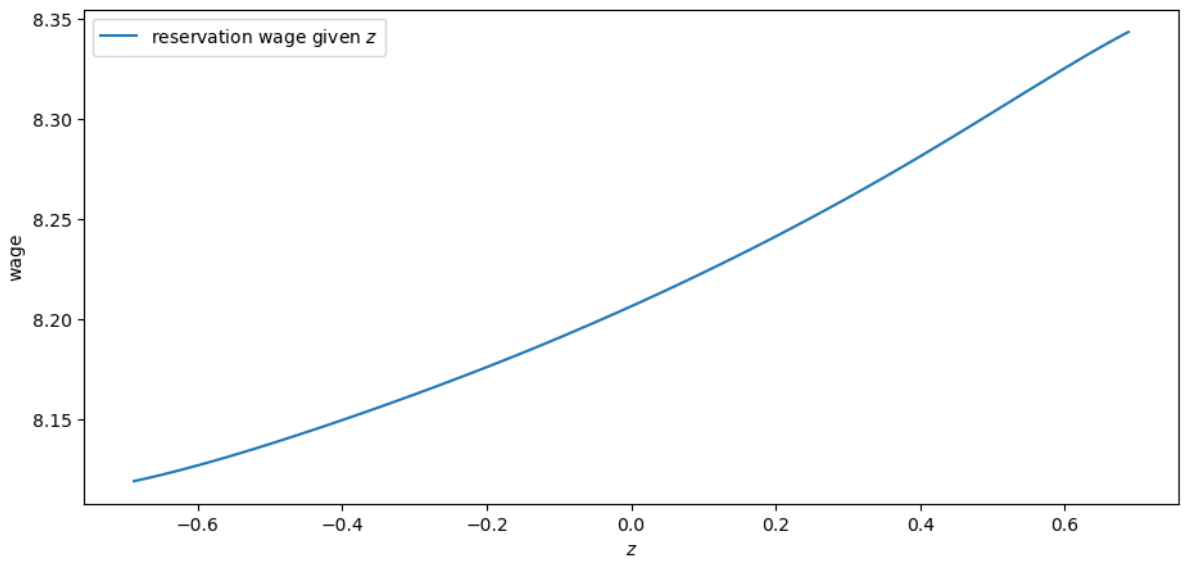
```
Error at iteration 150 is 0.00045489087412420304.
```

```
Error at iteration 175 is 0.00011479050300522431.

Converged in 178 iterations.
TOC: Elapsed: 0:00:3.94
```

```
3.9419326782226562
```

Next we will compute and plot the reservation wage function defined in (18.1).

```
res_wage_function = np.exp(f_star * (1 - js.β))

fig, ax = plt.subplots()
ax.plot(js.z_grid, res_wage_function, label="reservation wage given $z$")
ax.set(xlabel="$z$", ylabel="wage")
ax.legend()
plt.show()
```



Notice that the reservation wage is increasing in the current state z.

This is because a higher state leads the agent to predict higher future wages, increasing the option value of waiting.

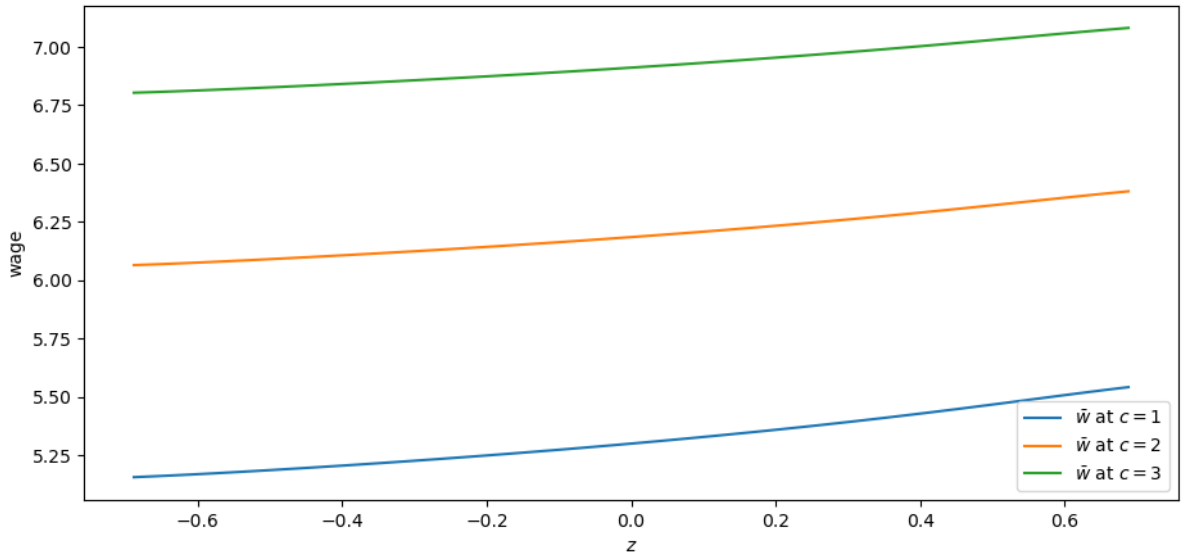
Let's try changing unemployment compensation and look at its impact on the reservation wage:

```
c_vals = 1, 2, 3

fig, ax = plt.subplots()

for c in c_vals:
    js = JobSearch(c=c)
    f_star = compute_fixed_point(js, verbose=False)
```

(continues on next page)

(continued from previous page)

```
    res_wage_function = np.exp(f_star * (1 - js.β))
    ax.plot(js.z_grid, res_wage_function, label=rf"$\bar w$ at $c = {c}$")

ax.set(xlabel="$z$", ylabel="wage")
ax.legend()
plt.show()
```



As expected, higher unemployment compensation shifts the reservation wage up at all state values.

18.4 Unemployment Duration

Next we study how mean unemployment duration varies with unemployment compensation.

For simplicity we'll fix the initial state at $z_t = 0$.

```
def compute_unemployment_duration(js, seed=1234):

    f_star = compute_fixed_point(js, verbose=False)
    μ, s, d, ρ, σ, β, c = js.parameters()
    z_grid = js.z_grid
    np.random.seed(seed)

    @njit
    def f_star_function(z):
        return interp(z_grid, f_star, z)

    @njit
    def draw_tau(t_max=10_000):
        z = 0
        t = 0

        unemployed = True
        while unemployed and t < t_max:
            # draw current wage
```

(continues on next page)

(continued from previous page)

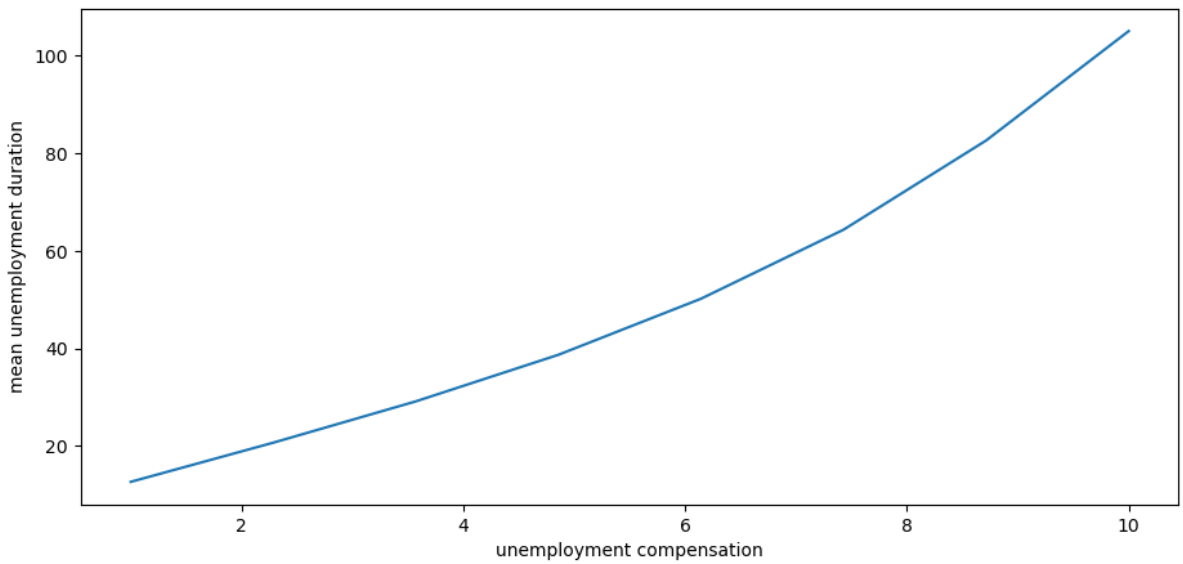
```
            y = np.exp(μ + s * np.random.randn())
            w = np.exp(z) + y
            res_wage = np.exp(f_star_function(z) * (1 - β))
            # if optimal to stop, record t
            if w >= res_wage:
                unemployed = False
                τ = t
            # else increment data and state
            else:
                z = ρ * z + d + σ * np.random.randn()
                t += 1
        return τ

    @njit(parallel=True)
    def compute_expected_tau(num_reps=100_000):
        sum_value = 0
        for i in prange(num_reps):
            sum_value += draw_tau()
        return sum_value / num_reps

    return compute_expected_tau()
```

Let's test this out with some possible values for unemployment compensation.

```
c_vals = np.linspace(1.0, 10.0, 8)
durations = np.empty_like(c_vals)
for i, c in enumerate(c_vals):
    js = JobSearch(c=c)
    τ = compute_unemployment_duration(js)
    durations[i] = τ
```

Here is a plot of the results.

```
fig, ax = plt.subplots()
ax.plot(c_vals, durations)
ax.set_xlabel("unemployment compensation")
ax.set_ylabel("mean unemployment duration")
plt.show()
```



Not surprisingly, unemployment duration increases when unemployment compensation is higher.

This is because the value of waiting increases with unemployment compensation.

18.5 Exercises

Exercise 18.5.1

Investigate how mean unemployment duration varies with the discount factor β.

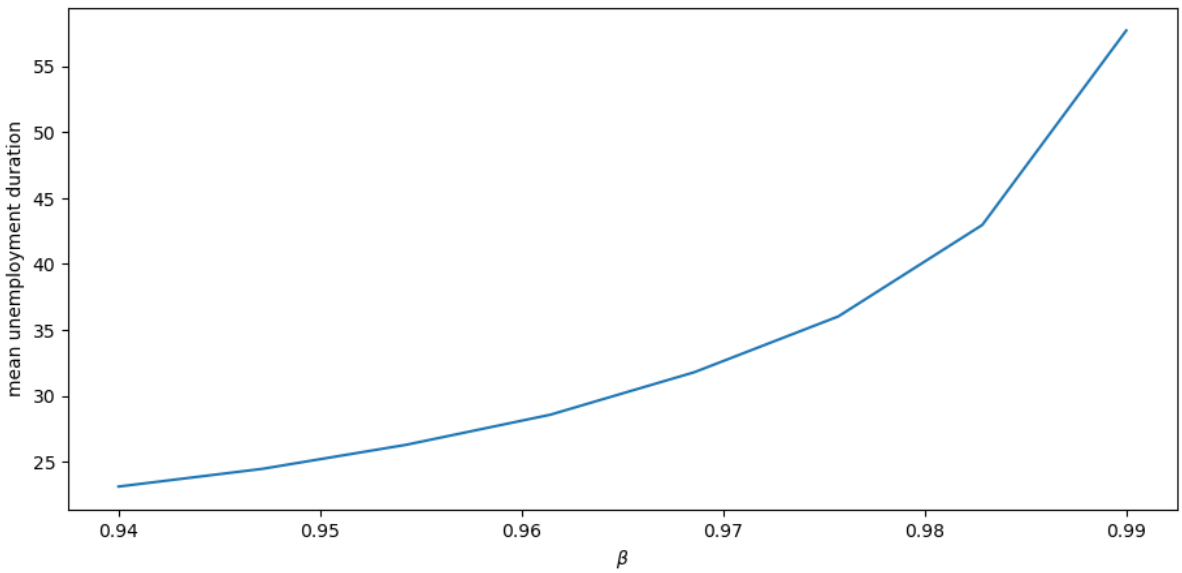
- What is your prior expectation?
- Do your results match up?

Solution to Exercise 18.5.1

Here is one solution

```
beta_vals = np.linspace(0.94, 0.99, 8)
durations = np.empty_like(beta_vals)
for i, β in enumerate(beta_vals):
    js = JobSearch(β=β)
    τ = compute_unemployment_duration(js)
    durations[i] = τ
```

```
fig, ax = plt.subplots()
ax.plot(beta_vals, durations)
ax.set_xlabel(r"$\beta$")
ax.set_ylabel("mean unemployment duration")
plt.show()
```



The figure shows that more patient individuals tend to wait longer before accepting an offer.

CHAPTER

NINETEEN

JOB SEARCH V: MODELING CAREER CHOICE

Contents

- *Job Search V: Modeling Career Choice*
  - *Overview*
  - *Model*
  - *Implementation*
  - *Exercises*

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

```
!pip install quantecon
```

19.1 Overview

Next, we study a computational problem concerning career and job choices.

The model is originally due to Derek Neal [Nea99].

This exposition draws on the presentation in [LS18], section 6.5.

We begin with some imports:

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
import quantecon as qe
from numba import njit, prange
from quantecon.distributions import BetaBinomial
from scipy.special import binom, beta
from mpl_toolkits.mplot3d.axes3d import Axes3D
from matplotlib import cm
```

19.1.1 Model Features

- Career and job within career both chosen to maximize expected discounted wage flow.
- Infinite horizon dynamic programming with two state variables.

19.2 Model

In what follows we distinguish between a career and a job, where

- a *career* is understood to be a general field encompassing many possible jobs, and
- a *job* is understood to be a position with a particular firm

For workers, wages can be decomposed into the contribution of job and career

- $w_t = \theta_t + \epsilon_t$, where
  - θ_t is the contribution of career at time t
  - ϵ_t is the contribution of the job at time t

At the start of time t, a worker has the following options

- retain a current (career, job) pair (θ_t, ϵ_t) — referred to hereafter as "stay put"
- retain a current career θ_t but redraw a job ϵ_t — referred to hereafter as "new job"
- redraw both a career θ_t and a job ϵ_t — referred to hereafter as "new life"

Draws of θ and ϵ are independent of each other and past values, with

- $\theta_t \sim F$
- $\epsilon_t \sim G$

Notice that the worker does not have the option to retain a job but redraw a career — starting a new career always requires starting a new job.

A young worker aims to maximize the expected sum of discounted wages

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t w_t \tag{19.1}$$

subject to the choice restrictions specified above.

Let $v(\theta, \epsilon)$ denote the value function, which is the maximum of (19.1) overall feasible (career, job) policies, given the initial state (θ, ϵ).

The value function obeys

$$v(\theta, \epsilon) = \max\{I, II, III\}$$

where

$$\begin{aligned}
I &= \theta + \epsilon + \beta v(\theta, \epsilon) \\
II &= \theta + \int \epsilon' G(d\epsilon') + \beta \int v(\theta, \epsilon') G(d\epsilon') \\
III &= \int \theta' F(d\theta') + \int \epsilon' G(d\epsilon') + \beta \int \int v(\theta', \epsilon') G(d\epsilon') F(d\theta')
\end{aligned}$$

Evidently I, II and III correspond to "stay put", "new job" and "new life", respectively.

19.2.1 Parameterization

As in [LS18], section 6.5, we will focus on a discrete version of the model, parameterized as follows:

- both θ and ϵ take values in the set `np.linspace(0, B, grid_size)` — an even grid of points between 0 and B inclusive
- `grid_size = 50`
- `B = 5`
- `β = 0.95`

The distributions F and G are discrete distributions generating draws from the grid points `np.linspace(0, B, grid_size)`.

A very useful family of discrete distributions is the Beta-binomial family, with probability mass function

$$p(k \mid n, a, b) = \binom{n}{k} \frac{B(k+a, n-k+b)}{B(a,b)}, \qquad k = 0, \dots, n$$

Interpretation:

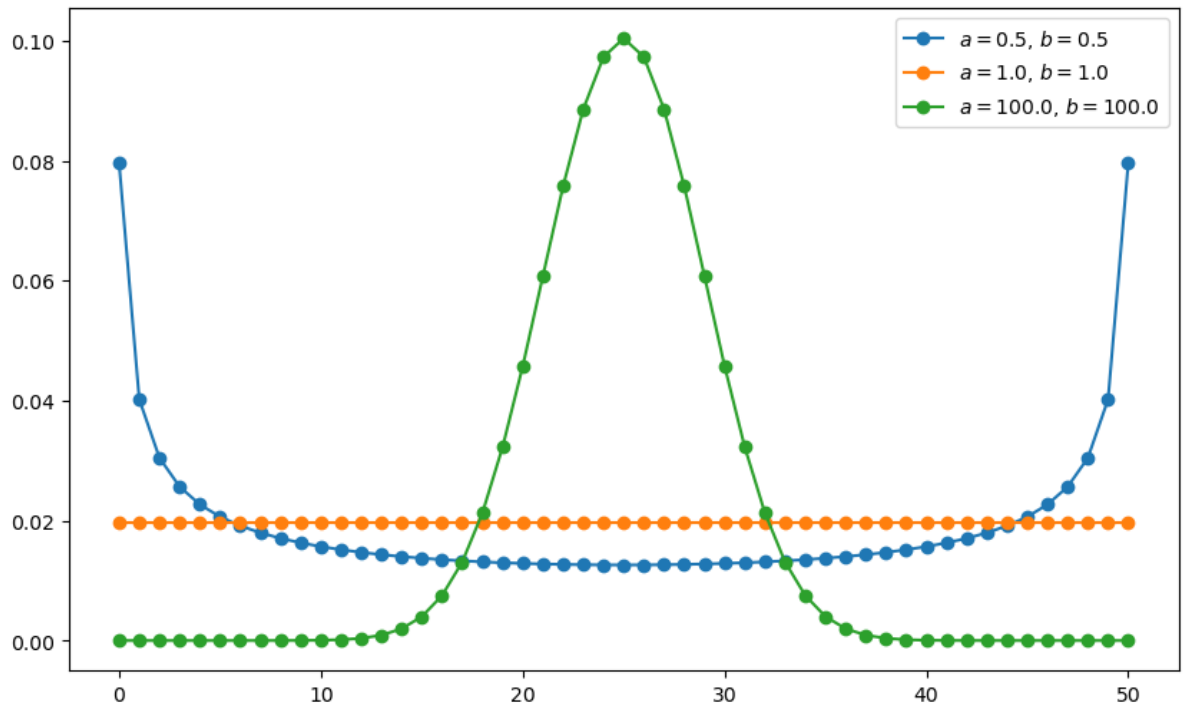
- draw q from a Beta distribution with shape parameters (a, b)
- run n independent binary trials, each with success probability q
- $p(k \mid n, a, b)$ is the probability of k successes in these n trials

Nice properties:

- very flexible class of distributions, including uniform, symmetric unimodal, etc.
- only three parameters

Here's a figure showing the effect on the pmf of different shape parameters when $n = 50$.

```
def gen_probs(n, a, b):
    probs = np.zeros(n+1)
    for k in range(n+1):
        probs[k] = binom(n, k) * beta(k + a, n - k + b) / beta(a, b)
    return probs

n = 50
a_vals = [0.5, 1, 100]
b_vals = [0.5, 1, 100]
fig, ax = plt.subplots(figsize=(10, 6))
for a, b in zip(a_vals, b_vals):
    ab_label = f'$a = {a:.1f}$, $b = {b:.1f}$'
    ax.plot(list(range(0, n+1)), gen_probs(n, a, b), '-o', label=ab_label)
ax.legend()
plt.show()
```



19.3 Implementation

We will first create a class `CareerWorkerProblem` which will hold the default parameterizations of the model and an initial guess for the value function.

```
class CareerWorkerProblem:

    def __init__(self,
                 B=5.0,           # Upper bound
                 β=0.95,          # Discount factor
                 grid_size=50,    # Grid size
                 F_a=1,
                 F_b=1,
                 G_a=1,
                 G_b=1):

        self.β, self.grid_size, self.B = β, grid_size, B

        self.θ = np.linspace(0, B, grid_size)      # Set of θ values
        self.ϵ = np.linspace(0, B, grid_size)      # Set of ϵ values

        self.F_probs = BetaBinomial(grid_size - 1, F_a, F_b).pdf()
        self.G_probs = BetaBinomial(grid_size - 1, G_a, G_b).pdf()
        self.F_mean = np.sum(self.θ * self.F_probs)
        self.G_mean = np.sum(self.ϵ * self.G_probs)

        # Store these parameters for str and repr methods
        self._F_a, self._F_b = F_a, F_b
        self._G_a, self._G_b = G_a, G_b
```

The following function takes an instance of `CareerWorkerProblem` and returns the corresponding Bellman operator T and the greedy policy function.

In this model, T is defined by $Tv(\theta, \epsilon) = \max\{I, II, III\}$, where I, II and III are as given in (19.2).

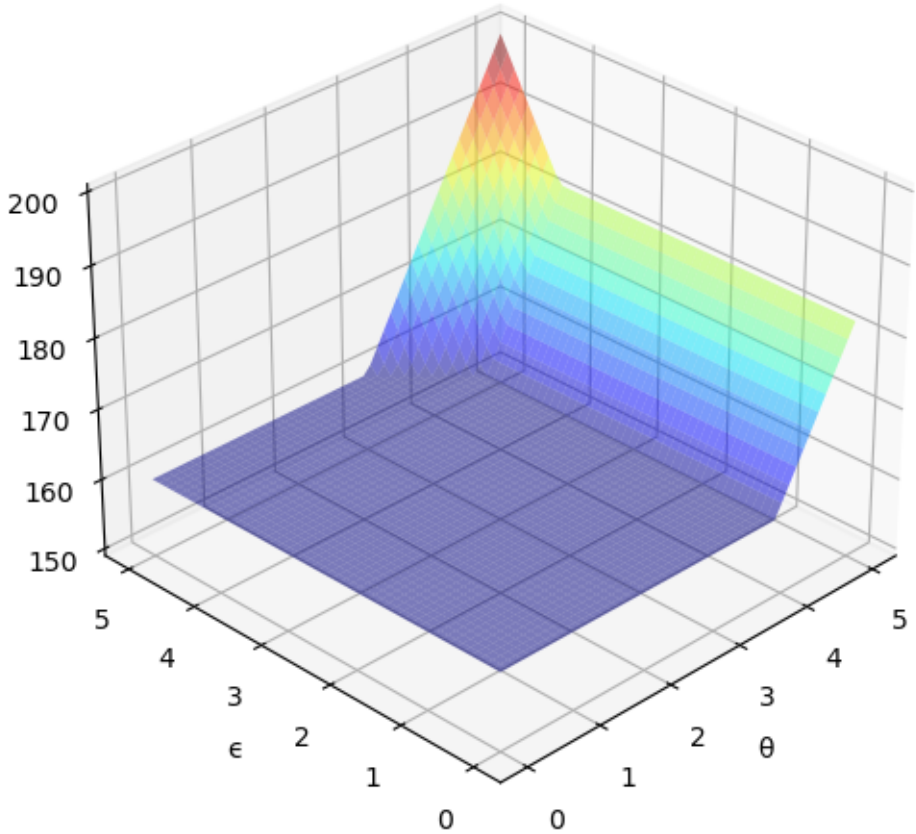
```
def operator_factory(cw, parallel_flag=True):

    """
    Returns jitted versions of the Bellman operator and the
    greedy policy function

    cw is an instance of ``CareerWorkerProblem``
    """

    θ, ϵ, β = cw.θ, cw.ϵ, cw.β
    F_probs, G_probs = cw.F_probs, cw.G_probs
    F_mean, G_mean = cw.F_mean, cw.G_mean

    @njit(parallel=parallel_flag)
    def T(v):
        "The Bellman operator"

        v_new = np.empty_like(v)

        for i in prange(len(v)):
            for j in prange(len(v)):
                v1 = θ[i] + ϵ[j] + β * v[i, j]                    # Stay put
                v2 = θ[i] + G_mean + β * v[i, :] @ G_probs        # New job
                v3 = G_mean + F_mean + β * F_probs @ v @ G_probs  # New life
                v_new[i, j] = max(v1, v2, v3)

        return v_new

    @njit
    def get_greedy(v):
        "Computes the v-greedy policy"

        σ = np.empty(v.shape)

        for i in range(len(v)):
            for j in range(len(v)):
                v1 = θ[i] + ϵ[j] + β * v[i, j]
                v2 = θ[i] + G_mean + β * v[i, :] @ G_probs
                v3 = G_mean + F_mean + β * F_probs @ v @ G_probs
                if v1 > max(v2, v3):
                    action = 1
                elif v2 > max(v1, v3):
                    action = 2
                else:
                    action = 3
                σ[i, j] = action

        return σ

    return T, get_greedy
```

Lastly, `solve_model` will take an instance of `CareerWorkerProblem` and iterate using the Bellman operator to find the fixed point of the Bellman equation.

```
def solve_model(cw,
                use_parallel=True,
                tol=1e-4,
                max_iter=1000,
                verbose=True,
                print_skip=25):

    T, _ = operator_factory(cw, parallel_flag=use_parallel)

    # Set up loop
    v = np.full((cw.grid_size, cw.grid_size), 100.)  # Initial guess
    i = 0
    error = tol + 1

    while i < max_iter and error > tol:
        v_new = T(v)
        error = np.max(np.abs(v - v_new))
        i += 1
        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")
        v = v_new

    if error > tol:
        print("Failed to converge!")

    elif verbose:
        print(f"\nConverged in {i} iterations.")

    return v_new
```

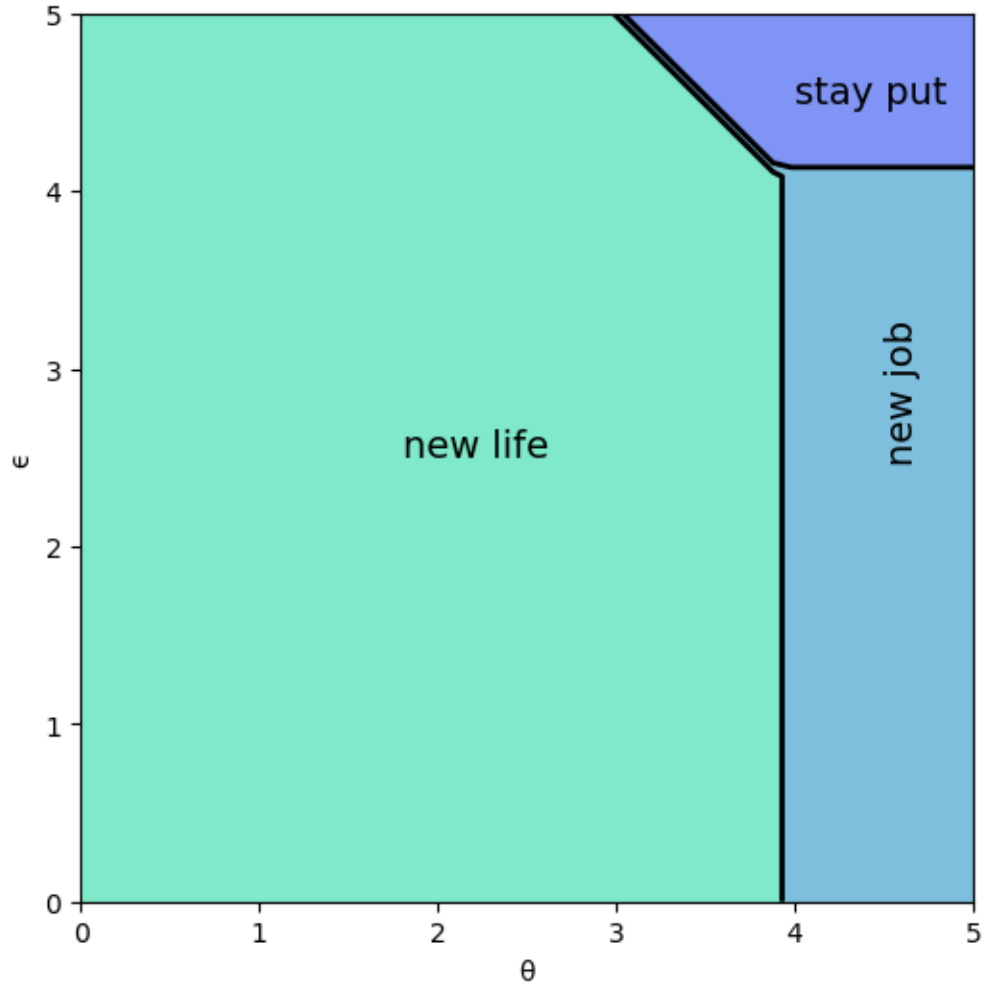
Here's the solution to the model – an approximate value function

```
cw = CareerWorkerProblem()
T, get_greedy = operator_factory(cw)
v_star = solve_model(cw, verbose=False)
greedy_star = get_greedy(v_star)

fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')
tg, eg = np.meshgrid(cw.θ, cw.ϵ)
ax.plot_surface(tg,
                eg,
                v_star.T,
                cmap=cm.jet,
                alpha=0.5,
                linewidth=0.25)
ax.set(xlabel='θ', ylabel='ϵ', zlim=(150, 200))
ax.view_init(ax.elev, 225)
plt.show()
```



And here is the optimal policy

```
fig, ax = plt.subplots(figsize=(6, 6))
tg, eg = np.meshgrid(cw.θ, cw.ϵ)
lvls = (0.5, 1.5, 2.5, 3.5)
ax.contourf(tg, eg, greedy_star.T, levels=lvls, cmap=cm.winter, alpha=0.5)
ax.contour(tg, eg, greedy_star.T, colors='k', levels=lvls, linewidths=2)
ax.set(xlabel='θ', ylabel='ϵ')
ax.text(1.8, 2.5, 'new life', fontsize=14)
ax.text(4.5, 2.5, 'new job', fontsize=14, rotation='vertical')
ax.text(4.0, 4.5, 'stay put', fontsize=14)
plt.show()
```



Interpretation:

- If both job and career are poor or mediocre, the worker will experiment with a new job and new career.
- If career is sufficiently good, the worker will hold it and experiment with new jobs until a sufficiently good one is found.
- If both job and career are good, the worker will stay put.

Notice that the worker will always hold on to a sufficiently good career, but not necessarily hold on to even the best paying job.

The reason is that high lifetime wages require both variables to be large, and the worker cannot change careers without changing jobs.

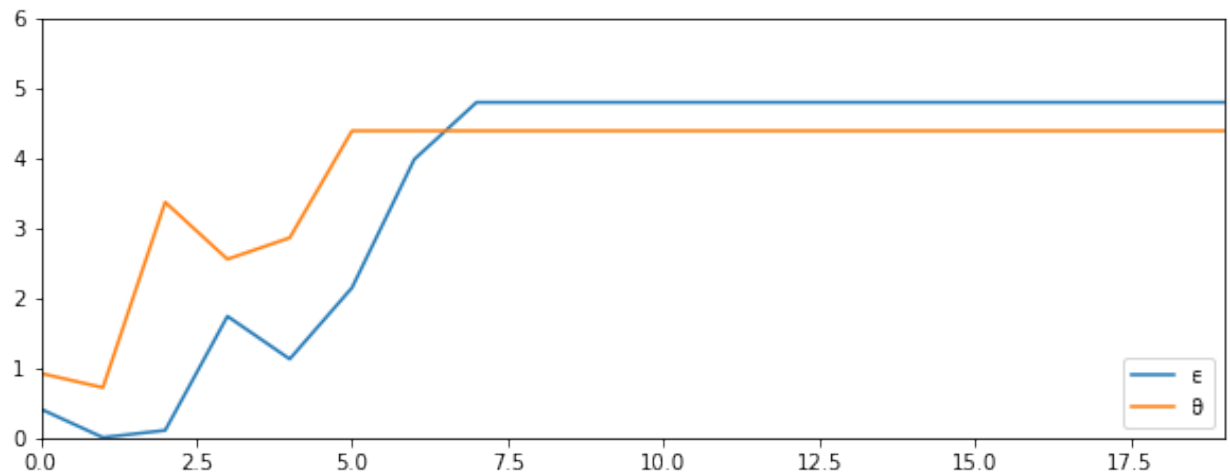
- Sometimes a good job must be sacrificed in order to change to a better career.

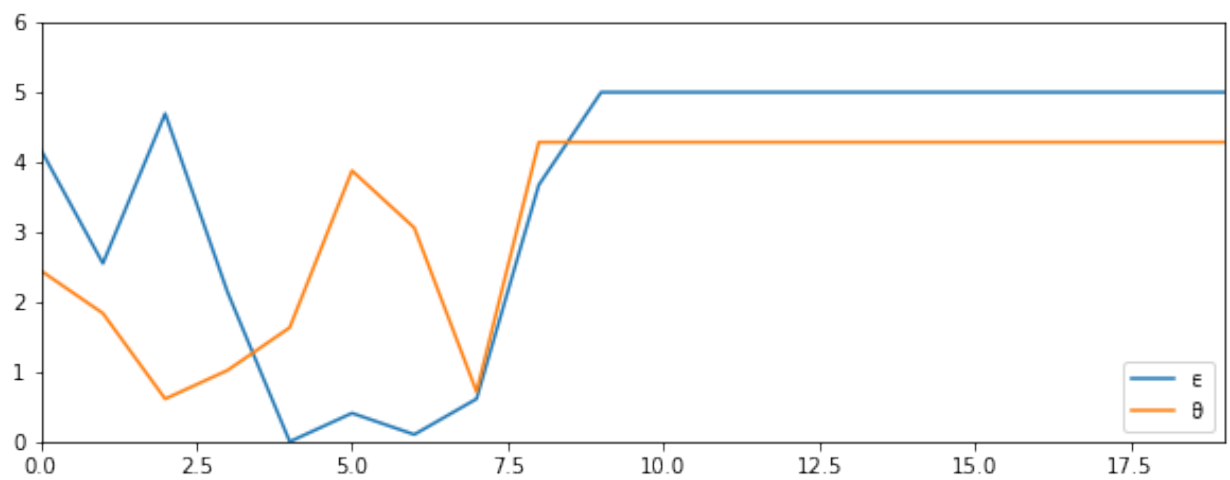
19.4 Exercises

Exercise 19.4.1

Using the default parameterization in the class `CareerWorkerProblem`, generate and plot typical sample paths for θ and ϵ when the worker follows the optimal policy.

In particular, modulo randomness, reproduce the following figure (where the horizontal axis represents time)





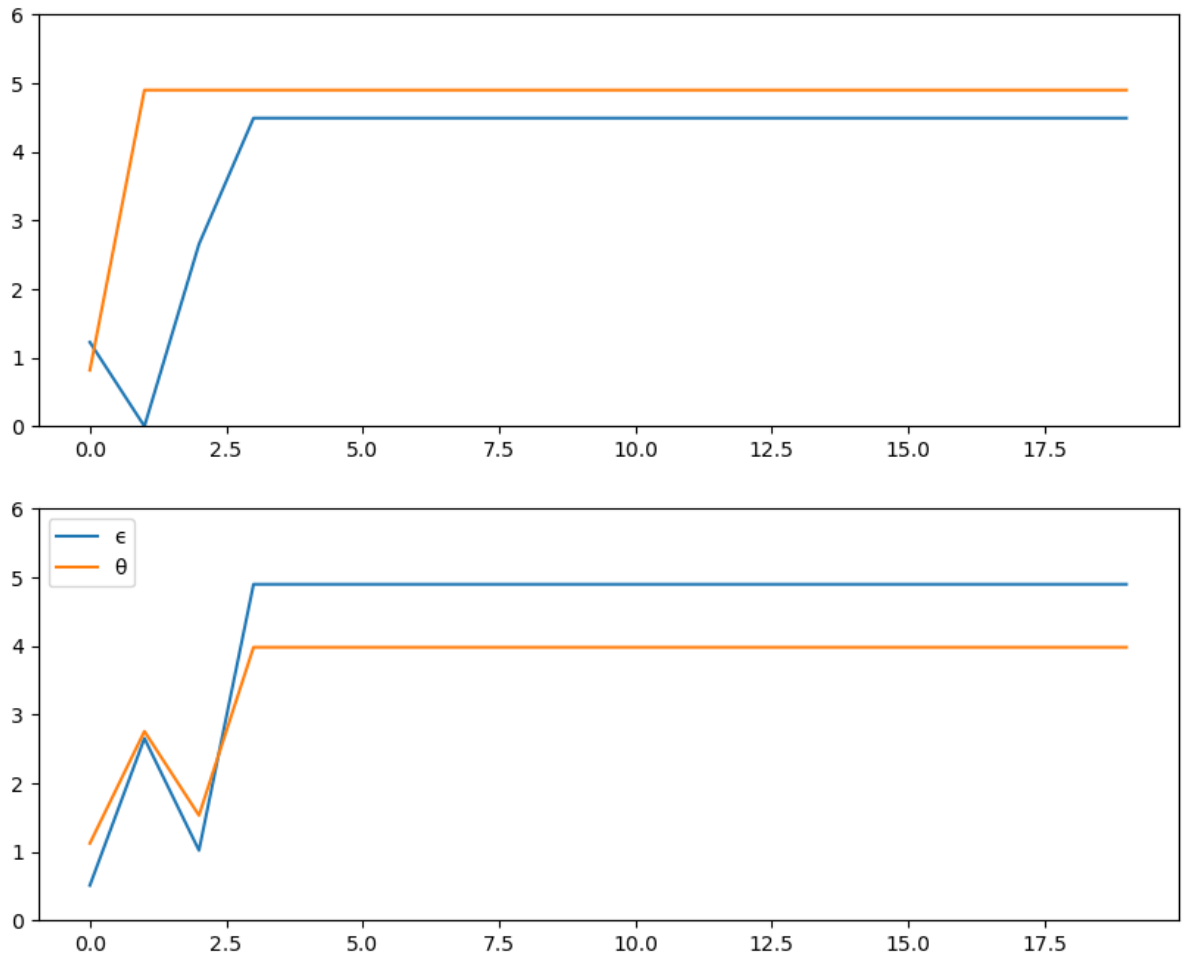
Hint: To generate the draws from the distributions F and G, use `quantecon.random.draw()`.

Solution to Exercise 19.4.1

Simulate job/career paths.

In reading the code, recall that `optimal_policy[i, j]` = policy at (θ_i, ϵ_j) = either 1, 2 or 3; meaning 'stay put', 'new job' and 'new life'.

```
F = np.cumsum(cw.F_probs)
G = np.cumsum(cw.G_probs)
v_star = solve_model(cw, verbose=False)
T, get_greedy = operator_factory(cw)
greedy_star = get_greedy(v_star)

def gen_path(optimal_policy, F, G, t=20):
    i = j = 0
    θ_index = []
    ϵ_index = []
    for t in range(t):
        if optimal_policy[i, j] == 1:       # Stay put
            pass

        elif greedy_star[i, j] == 2:     # New job
            j = qe.random.draw(G)

        else:                            # New life
            i, j = qe.random.draw(F), qe.random.draw(G)
        θ_index.append(i)
        ϵ_index.append(j)
    return cw.θ[θ_index], cw.ϵ[ϵ_index]


fig, axes = plt.subplots(2, 1, figsize=(10, 8))
for ax in axes:
    θ_path, ϵ_path = gen_path(greedy_star, F, G)
    ax.plot(ϵ_path, label='ϵ')
    ax.plot(θ_path, label='θ')
    ax.set_ylim(0, 6)

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



Exercise 19.4.2

Let's now consider how long it takes for the worker to settle down to a permanent job, given a starting point of $(\theta, \epsilon) = (0, 0)$.

In other words, we want to study the distribution of the random variable

$$T^* := \text{the first point in time from which the worker's job no longer changes}$$

Evidently, the worker's job becomes permanent if and only if (θ_t, ϵ_t) enters the "stay put" region of (θ, ϵ) space.

Letting S denote this region, T^* can be expressed as the first passage time to S under the optimal policy:

$$T^* := \inf\{t \geq 0 \,|\, (\theta_t, \epsilon_t) \in S\}$$

Collect 25,000 draws of this random variable and compute the median (which should be about 7).

Repeat the exercise with $\beta = 0.99$ and interpret the change.

Solution to Exercise 19.4.2

The median for the original parameterization can be computed as follows

```
cw = CareerWorkerProblem()
F = np.cumsum(cw.F_probs)
G = np.cumsum(cw.G_probs)
T, get_greedy = operator_factory(cw)
v_star = solve_model(cw, verbose=False)
greedy_star = get_greedy(v_star)

@njit
def passage_time(optimal_policy, F, G):
    t = 0
    i = j = 0
    while True:
        if optimal_policy[i, j] == 1:     # Stay put
            return t
        elif optimal_policy[i, j] == 2:   # New job
            j = qe.random.draw(G)
        else:                             # New life
            i, j  = qe.random.draw(F), qe.random.draw(G)
        t += 1

@njit(parallel=True)
def median_time(optimal_policy, F, G, M=25000):
    samples = np.empty(M)
    for i in prange(M):
        samples[i] = passage_time(optimal_policy, F, G)
    return np.median(samples)

median_time(greedy_star, F, G)
```

```
7.0
```

To compute the median with $\beta = 0.99$ instead of the default value $\beta = 0.95$, replace `cw = CareerWorkerProblem()` with `cw = CareerWorkerProblem(β=0.99)`.

The medians are subject to randomness but should be about 7 and 14 respectively.

Not surprisingly, more patient workers will wait longer to settle down to their final job.

Exercise 19.4.3

Set the parameterization to `G_a = G_b = 100` and generate a new optimal policy figure – interpret.

Solution to Exercise 19.4.3

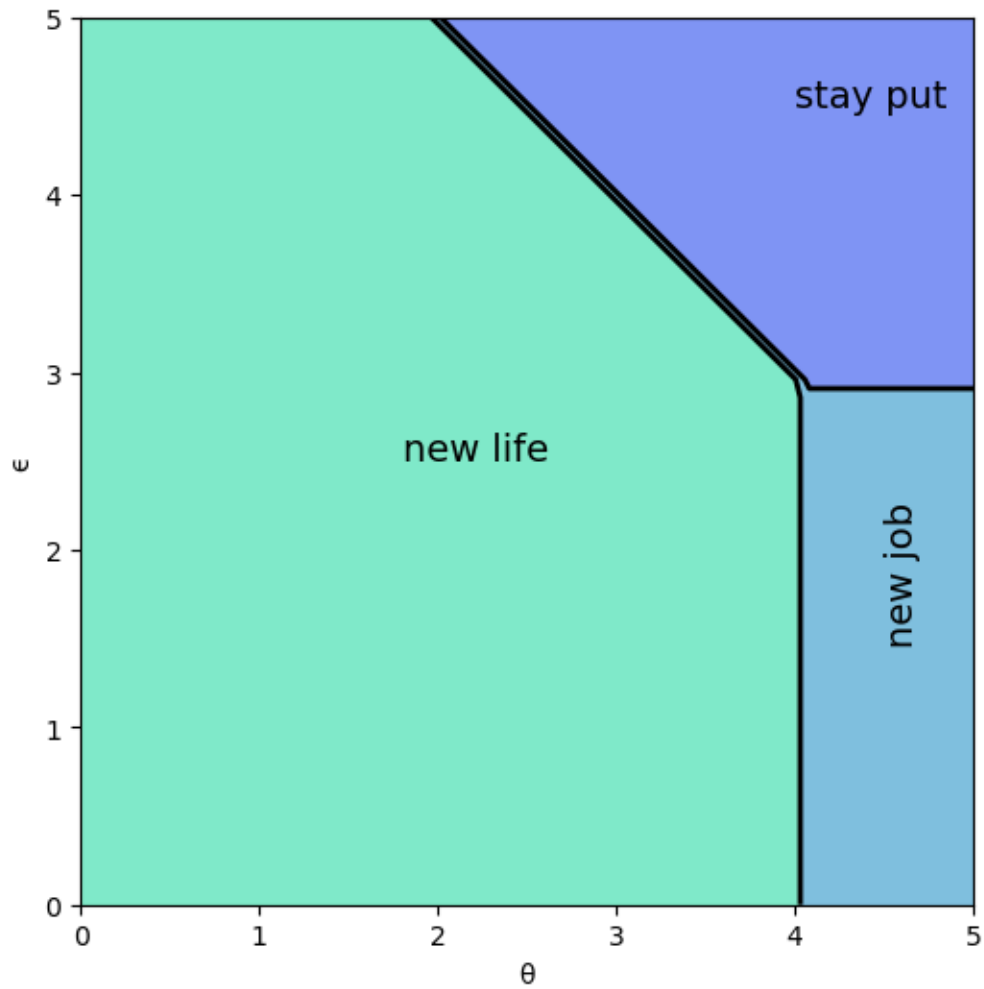
Here is one solution

```
cw = CareerWorkerProblem(G_a=100, G_b=100)
T, get_greedy = operator_factory(cw)
v_star = solve_model(cw, verbose=False)
greedy_star = get_greedy(v_star)

fig, ax = plt.subplots(figsize=(6, 6))
tg, eg = np.meshgrid(cw.θ, cw.ϵ)
lvls = (0.5, 1.5, 2.5, 3.5)
```

(continues on next page)

(continued from previous page)

```
ax.contourf(tg, eg, greedy_star.T, levels=lvls, cmap=cm.winter, alpha=0.5)
ax.contour(tg, eg, greedy_star.T, colors='k', levels=lvls, linewidths=2)
ax.set(xlabel='θ', ylabel='ϵ')
ax.text(1.8, 2.5, 'new life', fontsize=14)
ax.text(4.5, 1.5, 'new job', fontsize=14, rotation='vertical')
ax.text(4.0, 4.5, 'stay put', fontsize=14)
plt.show()
```



In the new figure, you see that the region for which the worker stays put has grown because the distribution for ϵ has become more concentrated around the mean, making high-paying jobs less realistic.

CHAPTER

TWENTY

JOB SEARCH VI: ON-THE-JOB SEARCH

Contents

- *Job Search VI: On-the-Job Search*
  - *Overview*
  - *Model*
  - *Implementation*
  - *Solving for Policies*
  - *Exercises*

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

```
!pip install interpolation
```

20.1 Overview

In this section, we solve a simple on-the-job search model

- based on [LS18], exercise 6.18, and [Jov79]

Let's start with some imports:

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
import scipy.stats as stats
from interpolation import interp
from numba import njit, prange
```

```
/home/runner/miniconda3/envs/quantecon/lib/python3.11/site-packages/numba/core/
↪decorators.py:262: NumbaDeprecationWarning: numba.generated_jit is deprecated.␣
↪Please see the documentation at: https://numba.readthedocs.io/en/stable/
↪reference/deprecation.html#deprecation-of-generated-jit for more information and␣
↪advice on a suitable replacement.
  warnings.warn(msg, NumbaDeprecationWarning)
```

20.1.1 Model Features

- job-specific human capital accumulation combined with on-the-job search
- infinite-horizon dynamic programming with one state variable and two controls

20.2 Model

Let x_t denote the time-t job-specific human capital of a worker employed at a given firm and let w_t denote current wages.

Let $w_t = x_t(1 - s_t - \phi_t)$, where

- ϕ_t is investment in job-specific human capital for the current role and
- s_t is search effort, devoted to obtaining new offers from other firms.

For as long as the worker remains in the current job, evolution of $\{x_t\}$ is given by $x_{t+1} = g(x_t, \phi_t)$.

When search effort at t is s_t, the worker receives a new job offer with probability $\pi(s_t) \in [0, 1]$.

The value of the offer, measured in job-specific human capital, is u_{t+1}, where $\{u_t\}$ is IID with common distribution f.

The worker can reject the current offer and continue with existing job.

Hence $x_{t+1} = u_{t+1}$ if he/she accepts and $x_{t+1} = g(x_t, \phi_t)$ otherwise.

Let $b_{t+1} \in \{0, 1\}$ be a binary random variable, where $b_{t+1} = 1$ indicates that the worker receives an offer at the end of time t.

We can write

$$x_{t+1} = (1 - b_{t+1})g(x_t, \phi_t) + b_{t+1} \max\{g(x_t, \phi_t), u_{t+1}\} \tag{20.1}$$

Agent's objective: maximize expected discounted sum of wages via controls $\{s_t\}$ and $\{\phi_t\}$.

Taking the expectation of $v(x_{t+1})$ and using (20.1), the Bellman equation for this problem can be written as

$$v(x) = \max_{s+\phi \leq 1} \left\{ x(1 - s - \phi) + \beta(1 - \pi(s))v[g(x, \phi)] + \beta\pi(s) \int v[g(x, \phi) \vee u] f(du) \right\} \tag{20.2}$$

Here nonnegativity of s and ϕ is understood, while $a \vee b := \max\{a, b\}$.

20.2.1 Parameterization

In the implementation below, we will focus on the parameterization

$$g(x, \phi) = A(x\phi)^\alpha, \quad \pi(s) = \sqrt{s} \quad \text{and} \quad f = \text{Beta}(2, 2)$$

with default parameter values

- $A = 1.4$
- $\alpha = 0.6$
- $\beta = 0.96$

The Beta$(2, 2)$ distribution is supported on $(0, 1)$ - it has a unimodal, symmetric density peaked at 0.5.

20.2.2 Back-of-the-Envelope Calculations

Before we solve the model, let's make some quick calculations that provide intuition on what the solution should look like.

To begin, observe that the worker has two instruments to build capital and hence wages:

1. invest in capital specific to the current job via ϕ
2. search for a new job with better job-specific capital match via s

Since wages are $x(1 - s - \phi)$, marginal cost of investment via either ϕ or s is identical.

Our risk-neutral worker should focus on whatever instrument has the highest expected return.

The relative expected return will depend on x.

For example, suppose first that $x = 0.05$

- If $s = 1$ and $\phi = 0$, then since $g(x, \phi) = 0$, taking expectations of (20.1) gives expected next period capital equal to $\pi(s)\mathbb{E}u = \mathbb{E}u = 0.5$.
- If $s = 0$ and $\phi = 1$, then next period capital is $g(x, \phi) = g(0.05, 1) \approx 0.23$.

Both rates of return are good, but the return from search is better.

Next, suppose that $x = 0.4$

- If $s = 1$ and $\phi = 0$, then expected next period capital is again 0.5
- If $s = 0$ and $\phi = 1$, then $g(x, \phi) = g(0.4, 1) \approx 0.8$

Return from investment via ϕ dominates expected return from search.

Combining these observations gives us two informal predictions:

1. At any given state x, the two controls ϕ and s will function primarily as substitutes — worker will focus on whichever instrument has the higher expected return.
2. For sufficiently small x, search will be preferable to investment in job-specific human capital. For larger x, the reverse will be true.

Now let's turn to implementation, and see if we can match our predictions.

20.3 Implementation

We will set up a class `JVWorker` that holds the parameters of the model described above

```
class JVWorker:
    r"""
    A Jovanovic-type model of employment with on-the-job search.

    """

    def __init__(self,
                 A=1.4,
                 α=0.6,
                 β=0.96,          # Discount factor
                 π=np.sqrt,       # Search effort function
                 a=2,             # Parameter of f
                 b=2,             # Parameter of f
                 grid_size=50,
```

(continues on next page)

(continued from previous page)

```
                 mc_size=100,
                 ε=1e-4):

        self.A, self.α, self.β, self.π = A, α, β, π
        self.mc_size, self.ε = mc_size, ε

        self.g = njit(lambda x, ϕ: A * (x * ϕ)**α)     # Transition function
        self.f_rvs = np.random.beta(a, b, mc_size)

        # Max of grid is the max of a large quantile value for f and the
        # fixed point y = g(y, 1)
        ε = 1e-4
        grid_max = max(A**(1 / (1 - α)), stats.beta(a, b).ppf(1 - ε))

        # Human capital
        self.x_grid = np.linspace(ε, grid_max, grid_size)
```

The function `operator_factory` takes an instance of this class and returns a jitted version of the Bellman operator `T`, i.e.

$$Tv(x) = \max_{s+\phi \leq 1} w(s, \phi)$$

where

$$w(s, \phi) := x(1 - s - \phi) + \beta(1 - \pi(s))v[g(x, \phi)] + \beta\pi(s) \int v[g(x, \phi) \vee u] f(du) \tag{20.3}$$

When we represent v, it will be with a NumPy array `v` giving values on grid `x_grid`.

But to evaluate the right-hand side of (20.3), we need a function, so we replace the arrays `v` and `x_grid` with a function `v_func` that gives linear interpolation of `v` on `x_grid`.

Inside the `for` loop, for each `x` in the grid over the state space, we set up the function $w(z) = w(s, \phi)$ defined in (20.3).

The function is maximized over all feasible (s, ϕ) pairs.

Another function, `get_greedy` returns the optimal choice of s and ϕ at each x, given a value function.

```
def operator_factory(jv, parallel_flag=True):

    """
    Returns a jitted version of the Bellman operator T

    jv is an instance of JVWorker

    """

    π, β = jv.π, jv.β
    x_grid, ε, mc_size = jv.x_grid, jv.ε, jv.mc_size
    f_rvs, g = jv.f_rvs, jv.g

    @njit
    def state_action_values(z, x, v):
        s, ϕ = z
        v_func = lambda x: interp(x_grid, v, x)

        integral = 0
        for m in range(mc_size):
```

(continues on next page)

(continued from previous page)

```
            u = f_rvs[m]
            integral += v_func(max(g(x, ϕ), u))
        integral = integral / mc_size

        q = π(s) * integral + (1 - π(s)) * v_func(g(x, ϕ))
        return x * (1 - ϕ - s) + β * q

    @njit(parallel=parallel_flag)
    def T(v):
        """
        The Bellman operator
        """

        v_new = np.empty_like(v)
        for i in prange(len(x_grid)):
            x = x_grid[i]

            # Search on a grid
            search_grid = np.linspace(ε, 1, 15)
            max_val = -1
            for s in search_grid:
                for ϕ in search_grid:
                    current_val = state_action_values((s, ϕ), x, v) if s + ϕ <= 1␣
↪else -1
                    if current_val > max_val:
                        max_val = current_val
            v_new[i] = max_val

        return v_new

    @njit
    def get_greedy(v):
        """
        Computes the v-greedy policy of a given function v
        """
        s_policy, ϕ_policy = np.empty_like(v), np.empty_like(v)

        for i in range(len(x_grid)):
            x = x_grid[i]
            # Search on a grid
            search_grid = np.linspace(ε, 1, 15)
            max_val = -1
            for s in search_grid:
                for ϕ in search_grid:
                    current_val = state_action_values((s, ϕ), x, v) if s + ϕ <= 1␣
↪else -1
                    if current_val > max_val:
                        max_val = current_val
                        max_s, max_ϕ = s, ϕ
                        s_policy[i], ϕ_policy[i] = max_s, max_ϕ
        return s_policy, ϕ_policy

    return T, get_greedy
```

To solve the model, we will write a function that uses the Bellman operator and iterates to find a fixed point.

```
def solve_model(jv,
                use_parallel=True,
                tol=1e-4,
                max_iter=1000,
                verbose=True,
                print_skip=25):

    """
    Solves the model by value function iteration

    * jv is an instance of JVWorker

    """

    T, _ = operator_factory(jv, parallel_flag=use_parallel)

    # Set up loop
    v = jv.x_grid * 0.5  # Initial condition
    i = 0
    error = tol + 1

    while i < max_iter and error > tol:
        v_new = T(v)
        error = np.max(np.abs(v - v_new))
        i += 1
        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")
        v = v_new

    if error > tol:
        print("Failed to converge!")
    elif verbose:
        print(f"\nConverged in {i} iterations.")

    return v_new
```

20.4 Solving for Policies

Let's generate the optimal policies and see what they look like.

```
jv = JVWorker()
T, get_greedy = operator_factory(jv)
v_star = solve_model(jv)
s_star, ϕ_star = get_greedy(v_star)
```

```
Error at iteration 25 is 0.15111173563338465.
Error at iteration 50 is 0.054460173375225907.
```

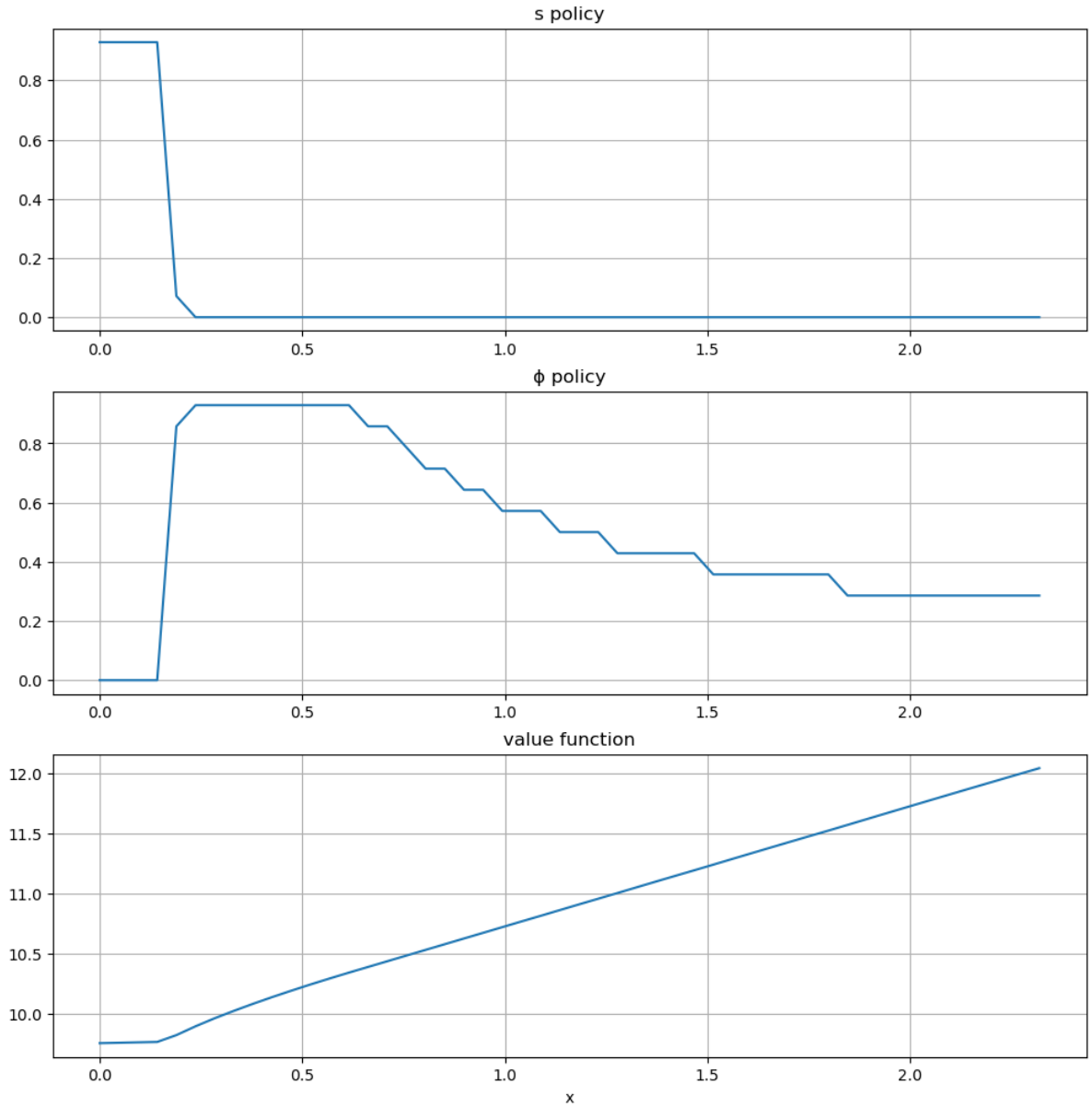
```
Error at iteration 75 is 0.019627267683953065.
Error at iteration 100 is 0.007073602834191561.
```

```
Error at iteration 125 is 0.002549303237801226.
Error at iteration 150 is 0.0009187605171820934.
```

```
Error at iteration 175 is 0.0003311182739746954.
Error at iteration 200 is 0.00011933393883545307.

Converged in 205 iterations.
```

Here are the plots:

```
plots = [s_star, ϕ_star, v_star]
titles = ["s policy", "ϕ policy",  "value function"]

fig, axes = plt.subplots(3, 1, figsize=(12, 12))

for ax, plot, title in zip(axes, plots, titles):
    ax.plot(jv.x_grid, plot)
    ax.set(title=title)
    ax.grid()

axes[-1].set_xlabel("x")
plt.show()
```



The horizontal axis is the state x, while the vertical axis gives $s(x)$ and $\phi(x)$.

Overall, the policies match well with our predictions from *above*

- Worker switches from one investment strategy to the other depending on relative return.
- For low values of x, the best option is to search for a new job.
- Once x is larger, worker does better by investing in human capital specific to the current position.

20.5 Exercises

Exercise 20.5.1

Let's look at the dynamics for the state process $\{x_t\}$ associated with these policies.

The dynamics are given by (20.1) when ϕ_t and s_t are chosen according to the optimal policies, and $\mathbb{P}\{b_{t+1} = 1\} = \pi(s_t)$.

Since the dynamics are random, analysis is a bit subtle.

One way to do it is to plot, for each x in a relatively fine grid called `plot_grid`, a large number K of realizations of x_{t+1} given $x_t = x$.

Plot this with one dot for each realization, in the form of a 45 degree diagram, setting

```
jv = JVWorker(grid_size=25, mc_size=50)
plot_grid_max, plot_grid_size = 1.2, 100
plot_grid = np.linspace(0, plot_grid_max, plot_grid_size)
fig, ax = plt.subplots()
ax.set_xlim(0, plot_grid_max)
ax.set_ylim(0, plot_grid_max)
```

By examining the plot, argue that under the optimal policies, the state x_t will converge to a constant value $\bar{x}$ close to unity.

Argue that at the steady state, $s_t \approx 0$ and $\phi_t \approx 0.6$.

Solution to Exercise 20.5.1

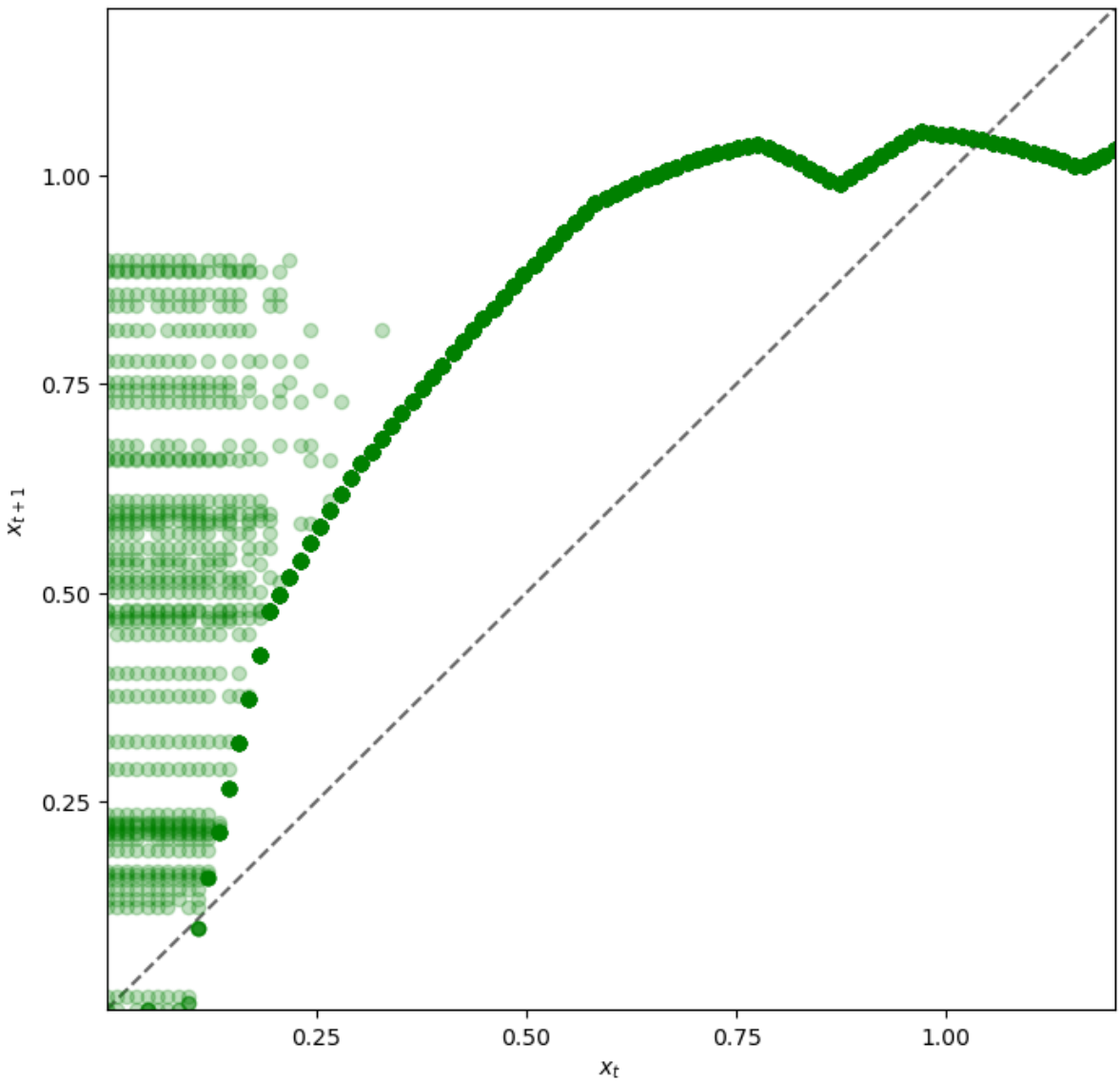
Here's code to produce the 45 degree diagram

```
jv = JVWorker(grid_size=25, mc_size=50)
π, g, f_rvs, x_grid = jv.π, jv.g, jv.f_rvs, jv.x_grid
T, get_greedy = operator_factory(jv)
v_star = solve_model(jv, verbose=False)
s_policy, ϕ_policy = get_greedy(v_star)

# Turn the policy function arrays into actual functions
s = lambda y: interp(x_grid, s_policy, y)
ϕ = lambda y: interp(x_grid, ϕ_policy, y)

def h(x, b, u):
    return (1 - b) * g(x, ϕ(x)) + b * max(g(x, ϕ(x)), u)


plot_grid_max, plot_grid_size = 1.2, 100
plot_grid = np.linspace(0, plot_grid_max, plot_grid_size)
fig, ax = plt.subplots(figsize=(8, 8))
ticks = (0.25, 0.5, 0.75, 1.0)
ax.set(xticks=ticks, yticks=ticks,
       xlim=(0, plot_grid_max),
       ylim=(0, plot_grid_max),
       xlabel='$x_t$', ylabel='$x_{t+1}$')

ax.plot(plot_grid, plot_grid, 'k--', alpha=0.6)  # 45 degree line
for x in plot_grid:
    for i in range(jv.mc_size):
```

(continues on next page)

(continued from previous page)

```
        b = 1 if np.random.uniform(0, 1) < π(s(x)) else 0
        u = f_rvs[i]
        y = h(x, b, u)
        ax.plot(x, y, 'go', alpha=0.25)

plt.show()
```



Looking at the dynamics, we can see that

- If x_t is below about 0.2 the dynamics are random, but $x_{t+1} > x_t$ is very likely.
- As x_t increases the dynamics become deterministic, and x_t converges to a steady state value close to 1.

Referring back to the figure *here* we see that $x_t \approx 1$ means that $s_t = s(x_t) \approx 0$ and $\phi_t = \phi(x_t) \approx 0.6$.

Exercise 20.5.2

In Exercise 20.5.1, we found that s_t converges to zero and ϕ_t converges to about 0.6.

Since these results were calculated at a value of β close to one, let's compare them to the best choice for an *infinitely* patient worker.

Intuitively, an infinitely patient worker would like to maximize steady state wages, which are a function of steady state capital.

You can take it as given—it's certainly true—that the infinitely patient worker does not search in the long run (i.e., $s_t = 0$ for large t).

Thus, given ϕ, steady state capital is the positive fixed point $x^*(\phi)$ of the map $x \mapsto g(x, \phi)$.

Steady state wages can be written as $w^*(\phi) = x^*(\phi)(1 - \phi)$.

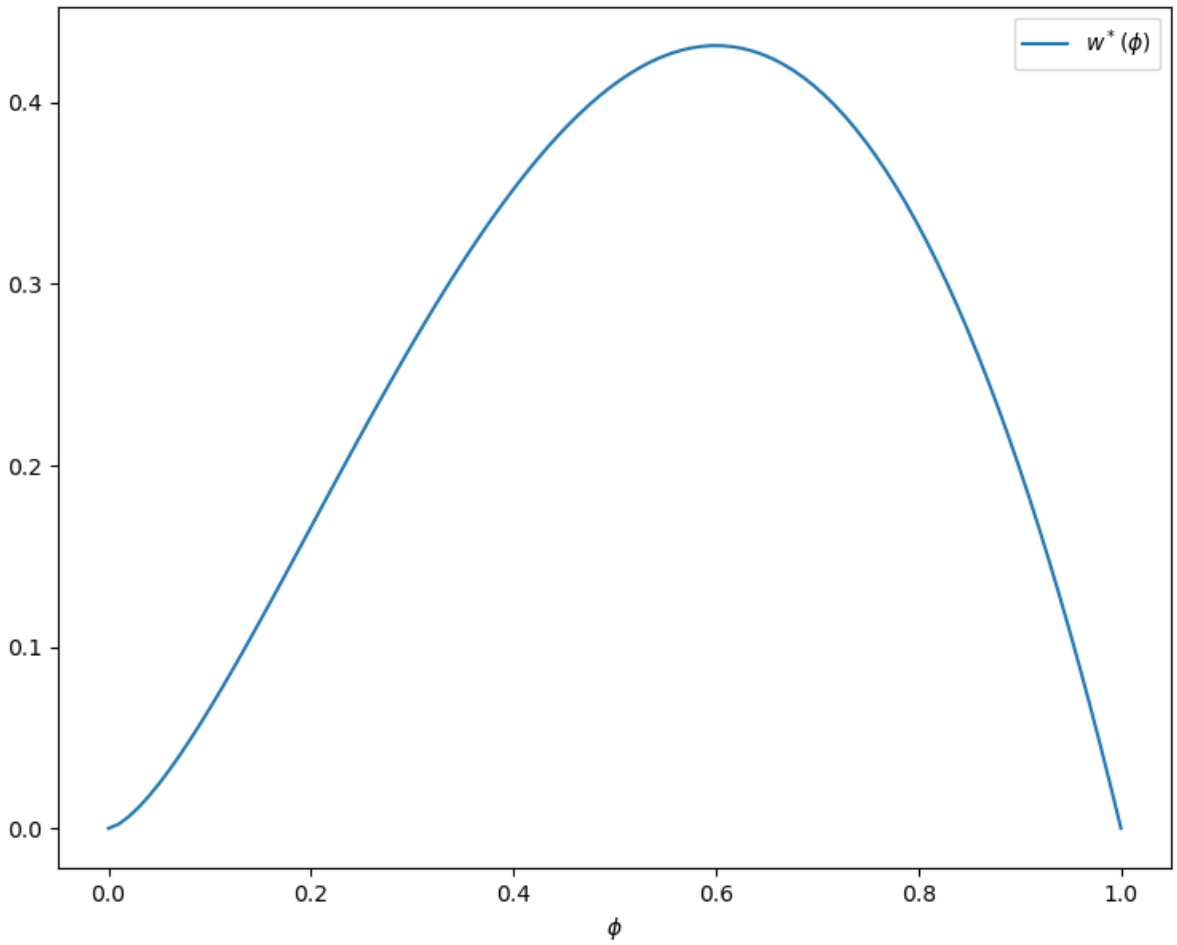
Graph $w^*(\phi)$ with respect to ϕ, and examine the best choice of ϕ.

Can you give a rough interpretation for the value that you see?

Solution to Exercise 20.5.2

The figure can be produced as follows

```
jv = JVWorker()

def xbar(ϕ):
    A, α = jv.A, jv.α
    return (A * ϕ**α)**(1 / (1 - α))

ϕ_grid = np.linspace(0, 1, 100)
fig, ax = plt.subplots(figsize=(9, 7))
ax.set(xlabel='$\phi$')
ax.plot(ϕ_grid, [xbar(ϕ) * (1 - ϕ) for ϕ in ϕ_grid], label='$w^*(\phi)$')
ax.legend()

plt.show()
```



Observe that the maximizer is around 0.6.

This is similar to the long-run value for ϕ obtained in Exercise 20.5.1.

Hence the behavior of the infinitely patent worker is similar to that of the worker with $\beta = 0.96$.

This seems reasonable and helps us confirm that our dynamic programming solutions are probably correct.

CHAPTER

TWENTYONE

JOB SEARCH VII: SEARCH WITH LEARNING

Contents

- *Job Search VII: Search with Learning*
  - *Overview*
  - *Model*
  - *Take 1: Solution by VFI*
  - *Take 2: A More Efficient Method*
  - *Another Functional Equation*
  - *Solving the RWFE*
  - *Implementation*
  - *Exercises*
  - *Solutions*
  - *Appendix A*
  - *Appendix B*
  - *Examples*

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

```
!pip install interpolation
```

21.1 Overview

In this lecture, we consider an extension of the previously studied job search model of McCall [McC70].

We'll build on a model of Bayesian learning discussed in this lecture on the topic of exchangeability and its relationship to the concept of IID (identically and independently distributed) random variables and to Bayesian updating.

In the McCall model, an unemployed worker decides when to accept a permanent job at a specific fixed wage, given

- his or her discount factor
- the level of unemployment compensation
- the distribution from which wage offers are drawn

In the version considered below, the wage distribution is unknown and must be learned.

- The following is based on the presentation in [LS18], section 6.6.

Let's start with some imports

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
from numba import njit, prange, vectorize
from interpolation import mlinterp, interp
from math import gamma
import numpy as np
from matplotlib import cm
import scipy.optimize as op
from scipy.stats import cumfreq, beta
```

```
/home/runner/miniconda3/envs/quantecon/lib/python3.11/site-packages/numba/core/
↪decorators.py:262: NumbaDeprecationWarning: numba.generated_jit is deprecated.␣
↪Please see the documentation at: https://numba.readthedocs.io/en/stable/
↪reference/deprecation.html#deprecation-of-generated-jit for more information and␣
↪advice on a suitable replacement.
  warnings.warn(msg, NumbaDeprecationWarning)
```

21.1.1 Model Features

- Infinite horizon dynamic programming with two states and one binary control.
- Bayesian updating to learn the unknown distribution.

21.2 Model

Let's first review the basic McCall model [McC70] and then add the variation we want to consider.

21.2.1 The Basic McCall Model

Recall that, *in the baseline model*, an unemployed worker is presented in each period with a permanent job offer at wage W_t.

At time t, our worker either

1. accepts the offer and works permanently at constant wage W_t
2. rejects the offer, receives unemployment compensation c and reconsiders next period

The wage sequence W_t is IID and generated from known density q.

The worker aims to maximize the expected discounted sum of earnings $\mathbb{E}\sum_{t=0}^{\infty}\beta^t y_t$.

Let $v(w)$ be the optimal value of the problem for a previously unemployed worker who has just received offer w and is yet to decide whether to accept or reject the offer.

The value function v satisfies the recursion

$$v(w) = \max\left\{\frac{w}{1-\beta}, c + \beta\int v(w')q(w')dw'\right\} \tag{21.1}$$

The optimal policy has the form $\mathbf{1}\{w \geq \bar{w}\}$, where $\bar{w}$ is a constant called the *reservation wage*.

21.2.2 Offer Distribution Unknown

Now let's extend the model by considering the variation presented in [LS18], section 6.6.

The model is as above, apart from the fact that

- the density q is unknown
- the worker learns about q by starting with a prior and updating based on wage offers that he/she observes

The worker knows there are two possible distributions F and G.

These two distributions have densities f and g, repectively.

Just before time starts, "nature" selects q to be either f or g.

This is then the wage distribution from which the entire sequence W_t will be drawn.

The worker does not know which distribution nature has drawn, but the worker does know the two possible distributions f and g.

The worker puts a (subjective) prior probability π_0 on f having been chosen.

The worker's time 0 subjective distribution for the distribution of W_0 is

$$\pi_0 f + (1 - \pi_0) g$$

The worker's time t subjective belief about the the distribution of W_t is

$$\pi_t f + (1 - \pi_t) g,$$

where π_t updates via

$$\pi_{t+1} = \frac{\pi_t f(w_{t+1})}{\pi_t f(w_{t+1}) + (1 - \pi_t) g(w_{t+1})} \tag{21.2}$$

This last expression follows from Bayes' rule, which tells us that

$$\mathbb{P}\{q = f \,|\, W = w\} = \frac{\mathbb{P}\{W = w \,|\, q = f\}\mathbb{P}\{q = f\}}{\mathbb{P}\{W = w\}} \quad \text{and} \quad \mathbb{P}\{W = w\} = \sum_{\omega \in \{f, g\}} \mathbb{P}\{W = w \,|\, q = \omega\}\mathbb{P}\{q = \omega\}$$

The fact that (21.2) is recursive allows us to progress to a recursive solution method.

Letting

$$q_\pi(w) := \pi f(w) + (1 - \pi) g(w) \quad \text{and} \quad \kappa(w, \pi) := \frac{\pi f(w)}{\pi f(w) + (1 - \pi) g(w)}$$

we can express the value function for the unemployed worker recursively as follows

$$v(w, \pi) = \max \left\{ \frac{w}{1 - \beta}, \, c + \beta \int v(w', \pi') \, q_\pi(w') \, dw' \right\} \quad \text{where} \quad \pi' = \kappa(w', \pi) \tag{21.3}$$

Notice that the current guess π is a state variable, since it affects the worker's perception of probabilities for future rewards.

21.2.3 Parameterization

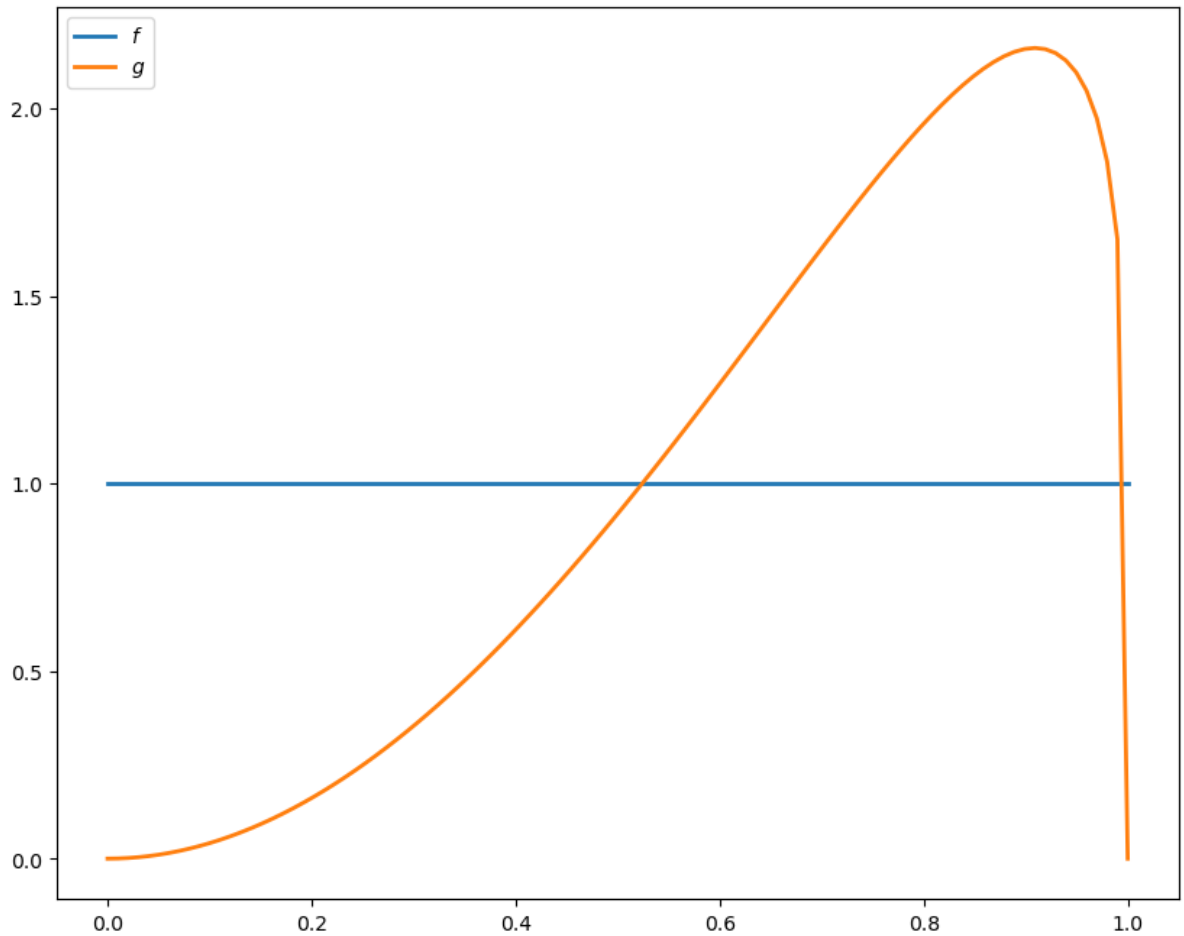
Following section 6.6 of [LS18], our baseline parameterization will be

- f is Beta$(1, 1)$
- g is Beta$(3, 1.2)$

- $\beta = 0.95$ and $c = 0.3$

The densities f and g have the following shape

```
@vectorize
def p(x, a, b):
    r = gamma(a + b) / (gamma(a) * gamma(b))
    return r * x**(a-1) * (1 - x)**(b-1)


x_grid = np.linspace(0, 1, 100)
f = lambda x: p(x, 1, 1)
g = lambda x: p(x, 3, 1.2)

fig, ax = plt.subplots(figsize=(10, 8))
ax.plot(x_grid, f(x_grid), label='$f$', lw=2)
ax.plot(x_grid, g(x_grid), label='$g$', lw=2)

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



21.2.4 Looking Forward

What kind of optimal policy might result from (21.3) and the parameterization specified above?

Intuitively, if we accept at w_a and $w_a \leq w_b$, then — all other things being given — we should also accept at w_b.

This suggests a policy of accepting whenever w exceeds some threshold value $\bar{w}$.

But $\bar{w}$ should depend on π — in fact, it should be decreasing in π because

- f is a less attractive offer distribution than g
- larger π means more weight on f and less on g

Thus, larger π depresses the worker's assessment of her future prospects, so relatively low current offers become more attractive.

Summary: We conjecture that the optimal policy is of the form $\mathbb{1}w \geq \bar{w}(\pi)$ for some decreasing function $\bar{w}$.

21.3 Take 1: Solution by VFI

Let's set about solving the model and see how our results match with our intuition.

We begin by solving via value function iteration (VFI), which is natural but ultimately turns out to be second best.

The class `SearchProblem` is used to store parameters and methods needed to compute optimal actions.

```
class SearchProblem:
    """
    A class to store a given parameterization of the "offer distribution
    unknown" model.

    """

    def __init__(self,
                 β=0.95,            # Discount factor
                 c=0.3,             # Unemployment compensation
                 F_a=1,
                 F_b=1,
                 G_a=3,
                 G_b=1.2,
                 w_max=1,           # Maximum wage possible
                 w_grid_size=100,
                 π_grid_size=100,
                 mc_size=500):

        self.β, self.c, self.w_max = β, c, w_max

        self.f = njit(lambda x: p(x, F_a, F_b))
        self.g = njit(lambda x: p(x, G_a, G_b))

        self.π_min, self.π_max = 1e-3, 1-1e-3    # Avoids instability
        self.w_grid = np.linspace(0, w_max, w_grid_size)
        self.π_grid = np.linspace(self.π_min, self.π_max, π_grid_size)

        self.mc_size = mc_size

        self.w_f = np.random.beta(F_a, F_b, mc_size)
        self.w_g = np.random.beta(G_a, G_b, mc_size)
```

The following function takes an instance of this class and returns jitted versions of the Bellman operator `T`, and a `get_greedy()` function to compute the approximate optimal policy from a guess `v` of the value function

```
def operator_factory(sp, parallel_flag=True):

    f, g = sp.f, sp.g
    w_f, w_g = sp.w_f, sp.w_g
    β, c = sp.β, sp.c
    mc_size = sp.mc_size
    w_grid, π_grid = sp.w_grid, sp.π_grid

    @njit
    def v_func(x, y, v):
        return mlinterp((w_grid, π_grid), v, (x, y))

    @njit
    def κ(w, π):
        """
        Updates π using Bayes' rule and the current wage observation w.
        """
        pf, pg = π * f(w), (1 - π) * g(w)
        π_new = pf / (pf + pg)

        return π_new

    @njit(parallel=parallel_flag)
    def T(v):
        """
        The Bellman operator.

        """
        v_new = np.empty_like(v)

        for i in prange(len(w_grid)):
            for j in prange(len(π_grid)):
                w = w_grid[i]
                π = π_grid[j]

                v_1 = w / (1 - β)

                integral_f, integral_g = 0, 0
                for m in prange(mc_size):
                    integral_f += v_func(w_f[m], κ(w_f[m], π), v)
                    integral_g += v_func(w_g[m], κ(w_g[m], π), v)
                integral = (π * integral_f + (1 - π) * integral_g) / mc_size

                v_2 = c + β * integral
                v_new[i, j] = max(v_1, v_2)

        return v_new

    @njit(parallel=parallel_flag)
    def get_greedy(v):
        """"
        Compute optimal actions taking v as the value function.

        """
        σ = np.empty_like(v)
```

(continues on next page)

(continued from previous page)

```
        for i in prange(len(w_grid)):
            for j in prange(len(π_grid)):
                w = w_grid[i]
                π = π_grid[j]

                v_1 = w / (1 - β)

                integral_f, integral_g = 0, 0
                for m in prange(mc_size):
                    integral_f += v_func(w_f[m], κ(w_f[m], π), v)
                    integral_g += v_func(w_g[m], κ(w_g[m], π), v)
                integral = (π * integral_f + (1 - π) * integral_g) / mc_size

                v_2 = c + β * integral

                σ[i, j] = v_1 > v_2  # Evaluates to 1 or 0

        return σ

    return T, get_greedy
```

We will omit a detailed discussion of the code because there is a more efficient solution method that we will use later.

To solve the model we will use the following function that iterates using T to find a fixed point

```
def solve_model(sp,
                use_parallel=True,
                tol=1e-4,
                max_iter=1000,
                verbose=True,
                print_skip=5):

    """
    Solves for the value function

    * sp is an instance of SearchProblem
    """

    T, _ = operator_factory(sp, use_parallel)

    # Set up loop
    i = 0
    error = tol + 1
    m, n = len(sp.w_grid), len(sp.π_grid)

    # Initialize v
    v = np.zeros((m, n)) + sp.c / (1 - sp.β)

    while i < max_iter and error > tol:
        v_new = T(v)
        error = np.max(np.abs(v - v_new))
        i += 1
        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")
        v = v_new
```

(continues on next page)

(continued from previous page)

```
    if error > tol:
        print("Failed to converge!")
    elif verbose:
        print(f"\nConverged in {i} iterations.")


    return v_new
```

Let's look at solutions computed from value function iteration

```
sp = SearchProblem()
v_star = solve_model(sp)
fig, ax = plt.subplots(figsize=(6, 6))
ax.contourf(sp.π_grid, sp.w_grid, v_star, 12, alpha=0.6, cmap=cm.jet)
cs = ax.contour(sp.π_grid, sp.w_grid, v_star, 12, colors="black")
ax.clabel(cs, inline=1, fontsize=10)
ax.set(xlabel='$\pi$', ylabel='$w$')

plt.show()
```

```
Error at iteration 5 is 0.6249662408731584.
```

```
Error at iteration 10 is 0.0977944474922996.
```

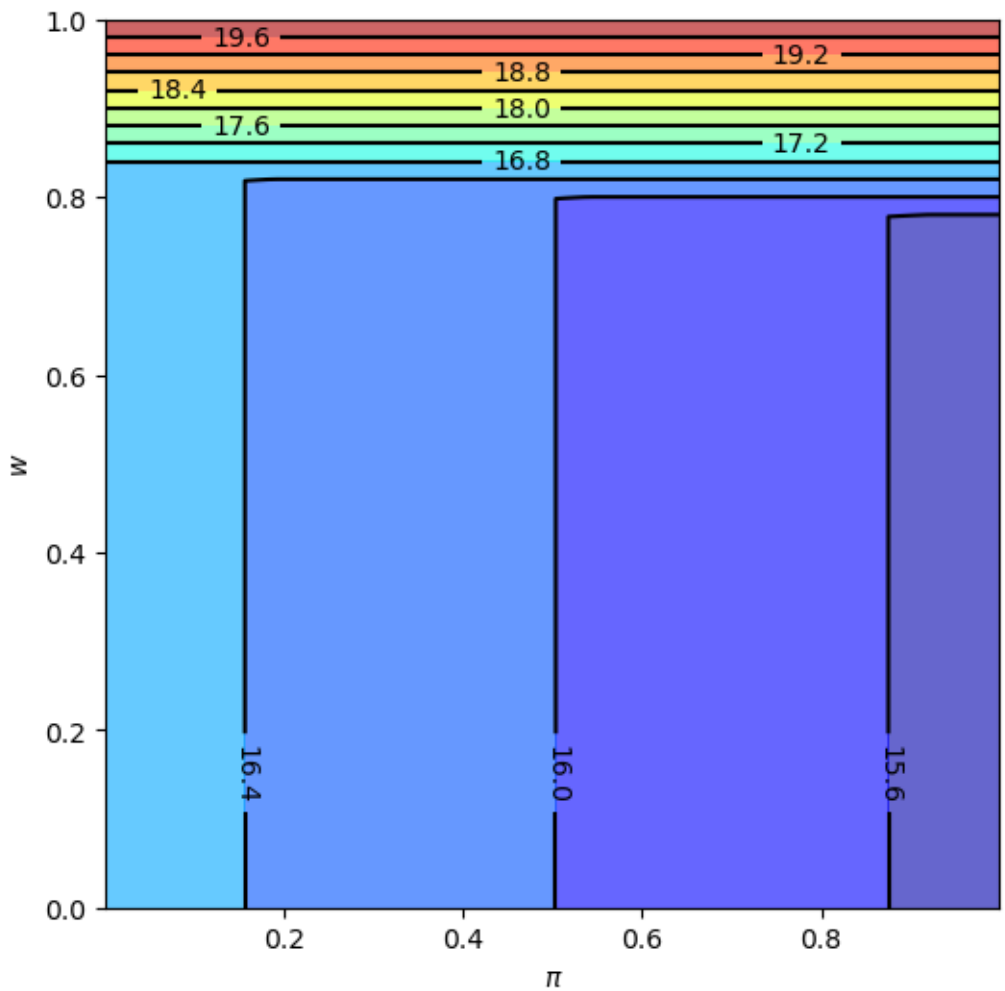
```
Error at iteration 15 is 0.01888978145071185.
```

```
Error at iteration 20 is 0.0038720231919242565.
```

```
Error at iteration 25 is 0.0007939536739804964.
```

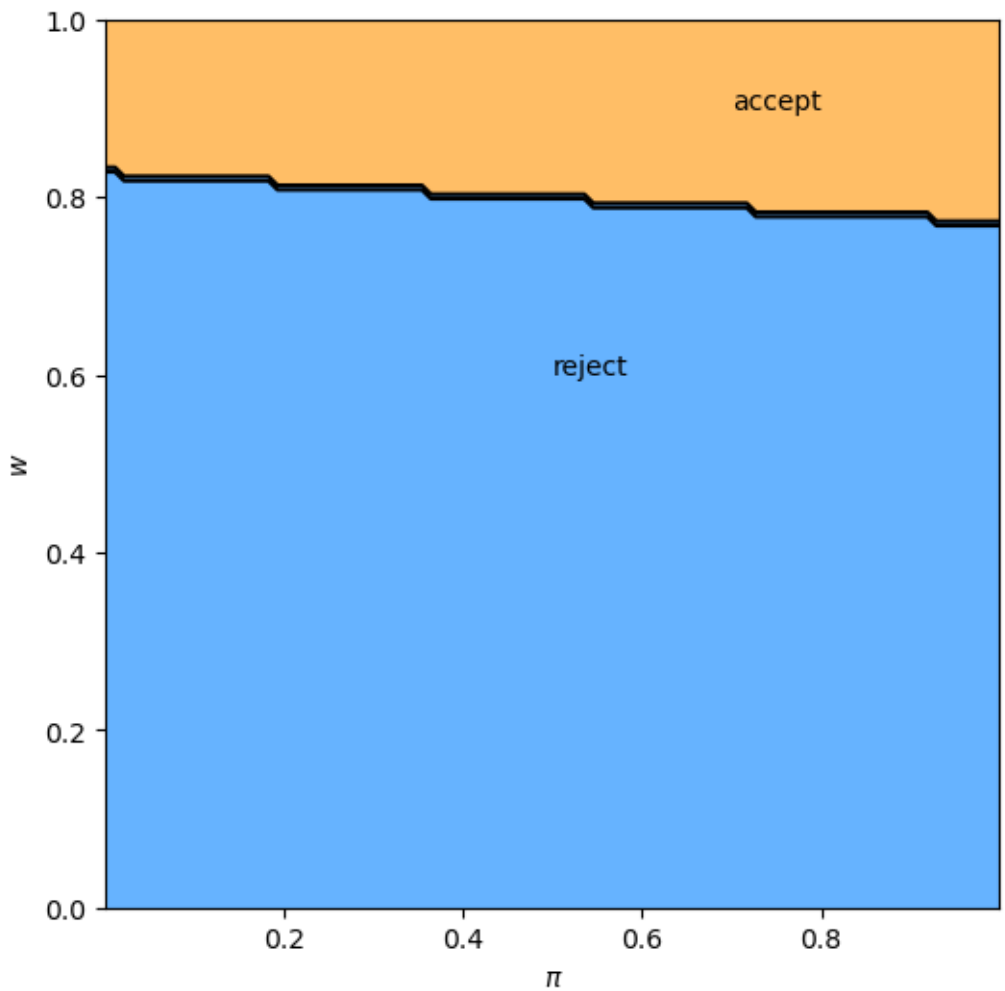
```
Error at iteration 30 is 0.00016280401116119947.
```

```
Converged in 32 iterations.
```



We will also plot the optimal policy

```
T, get_greedy = operator_factory(sp)
σ_star = get_greedy(v_star)

fig, ax = plt.subplots(figsize=(6, 6))
ax.contourf(sp.π_grid, sp.w_grid, σ_star, 1, alpha=0.6, cmap=cm.jet)
ax.contour(sp.π_grid, sp.w_grid, σ_star, 1, colors="black")
ax.set(xlabel='$\pi$', ylabel='$w$')

ax.text(0.5, 0.6, 'reject')
ax.text(0.7, 0.9, 'accept')

plt.show()
```



The results fit well with our intuition from section *looking forward*.

- The black line in the figure above corresponds to the function $\bar{w}(\pi)$ introduced there.
- It is decreasing as expected.

21.4 Take 2: A More Efficient Method

Let's consider another method to solve for the optimal policy.

We will use iteration with an operator that has the same contraction rate as the Bellman operator, but

- one dimensional rather than two dimensional
- no maximization step

As a consequence, the algorithm is orders of magnitude faster than VFI.

This section illustrates the point that when it comes to programming, a bit of mathematical analysis goes a long way.

21.5 Another Functional Equation

To begin, note that when $w = \bar{w}(\pi)$, the worker is indifferent between accepting and rejecting.

Hence the two choices on the right-hand side of (21.3) have equal value:

$$\frac{\bar{w}(\pi)}{1-\beta} = c + \beta \int v(w', \pi')\, q_\pi(w')\, dw' \tag{21.4}$$

Together, (21.3) and (21.4) give

$$v(w, \pi) = \max\left\{\frac{w}{1-\beta}, \frac{\bar{w}(\pi)}{1-\beta}\right\} \tag{21.5}$$

Combining (21.4) and (21.5), we obtain

$$\frac{\bar{w}(\pi)}{1-\beta} = c + \beta \int \max\left\{\frac{w'}{1-\beta}, \frac{\bar{w}(\pi')}{1-\beta}\right\} q_\pi(w')\, dw'$$

Multiplying by $1-\beta$, substituting in $\pi' = \kappa(w', \pi)$ and using $\circ$ for composition of functions yields

$$\bar{w}(\pi) = (1-\beta)c + \beta \int \max\left\{w', \bar{w} \circ \kappa(w', \pi)\right\}\, q_\pi(w')\, dw' \tag{21.6}$$

Equation (21.6) can be understood as a functional equation, where $\bar{w}$ is the unknown function.

- Let's call it the *reservation wage functional equation* (RWFE).
- The solution $\bar{w}$ to the RWFE is the object that we wish to compute.

21.6 Solving the RWFE

To solve the RWFE, we will first show that its solution is the fixed point of a contraction mapping.

To this end, let

- $b[0,1]$ be the bounded real-valued functions on $[0,1]$
- $\|\omega\| := \sup_{x \in [0,1]} |\omega(x)|$

Consider the operator Q mapping $\omega \in b[0,1]$ into $Q\omega \in b[0,1]$ via

$$(Q\omega)(\pi) = (1-\beta)c + \beta \int \max\left\{w', \omega \circ \kappa(w', \pi)\right\}\, q_\pi(w')\, dw' \tag{21.7}$$

Comparing (21.6) and (21.7), we see that the set of fixed points of Q exactly coincides with the set of solutions to the RWFE.

- If $Q\bar{w} = \bar{w}$ then $\bar{w}$ solves (21.6) and vice versa.

Moreover, for any $\omega, \omega' \in b[0,1]$, basic algebra and the triangle inequality for integrals tells us that

$$|(Q\omega)(\pi) - (Q\omega')(\pi)| \leq \beta \int |\max\left\{w', \omega \circ \kappa(w', \pi)\right\} - \max\left\{w', \omega' \circ \kappa(w', \pi)\right\}|\, q_\pi(w')\, dw' \tag{21.8}$$

Working case by case, it is easy to check that for real numbers a, b, c we always have

$$|\max\{a, b\} - \max\{a, c\}| \leq |b - c| \tag{21.9}$$

Combining (21.8) and (21.9) yields

$$|(Q\omega)(\pi) - (Q\omega')(\pi)| \leq \beta \int |\omega \circ \kappa(w', \pi) - \omega' \circ \kappa(w', \pi)| \; q_\pi(w') \, dw' \leq \beta \|\omega - \omega'\| \tag{21.10}$$

Taking the supremum over π now gives us

$$\|Q\omega - Q\omega'\| \leq \beta \|\omega - \omega'\| \tag{21.11}$$

In other words, Q is a contraction of modulus β on the complete metric space $(b[0,1], \|\cdot\|)$.

Hence

- A unique solution $\bar{w}$ to the RWFE exists in $b[0,1]$.
- $Q^k \omega \to \bar{w}$ uniformly as $k \to \infty$, for any $\omega \in b[0,1]$.

21.7 Implementation

The following function takes an instance of `SearchProblem` and returns the operator `Q`

```
def Q_factory(sp, parallel_flag=True):

    f, g = sp.f, sp.g
    w_f, w_g = sp.w_f, sp.w_g
    β, c = sp.β, sp.c
    mc_size = sp.mc_size
    w_grid, π_grid = sp.w_grid, sp.π_grid

    @njit
    def ω_func(p, ω):
        return interp(π_grid, ω, p)

    @njit
    def κ(w, π):
        """
        Updates π using Bayes' rule and the current wage observation w.
        """
        pf, pg = π * f(w), (1 - π) * g(w)
        π_new = pf / (pf + pg)

        return π_new

    @njit(parallel=parallel_flag)
    def Q(ω):
        """

        Updates the reservation wage function guess ω via the operator
        Q.

        """
        ω_new = np.empty_like(ω)

        for i in prange(len(π_grid)):
            π = π_grid[i]
            integral_f, integral_g = 0, 0
```

(continues on next page)

(continued from previous page)

```
            for m in prange(mc_size):
                integral_f += max(w_f[m], ω_func(κ(w_f[m], π), ω))
                integral_g += max(w_g[m], ω_func(κ(w_g[m], π), ω))
            integral = (π * integral_f + (1 - π) * integral_g) / mc_size

            ω_new[i] = (1 - β) * c + β * integral

        return ω_new

    return Q
```

In the next exercise, you are asked to compute an approximation to $\bar{w}$.

21.8 Exercises

Exercise 21.8.1

Use the default parameters and `Q_factory` to compute an optimal policy.

Your result should coincide closely with the figure for the optimal policy *shown above*.

Try experimenting with different parameters, and confirm that the change in the optimal policy coincides with your intuition.

21.9 Solutions

Solution to Exercise 21.8.1

This code solves the "Offer Distribution Unknown" model by iterating on a guess of the reservation wage function.

You should find that the run time is shorter than that of the value function approach.

Similar to above, we set up a function to iterate with `Q` to find the fixed point

```
def solve_wbar(sp,
               use_parallel=True,
               tol=1e-4,
               max_iter=1000,
               verbose=True,
               print_skip=5):

    Q = Q_factory(sp, use_parallel)

    # Set up loop
    i = 0
    error = tol + 1
    m, n = len(sp.w_grid), len(sp.π_grid)

    # Initialize w
    w = np.ones_like(sp.π_grid)
```

(continues on next page)

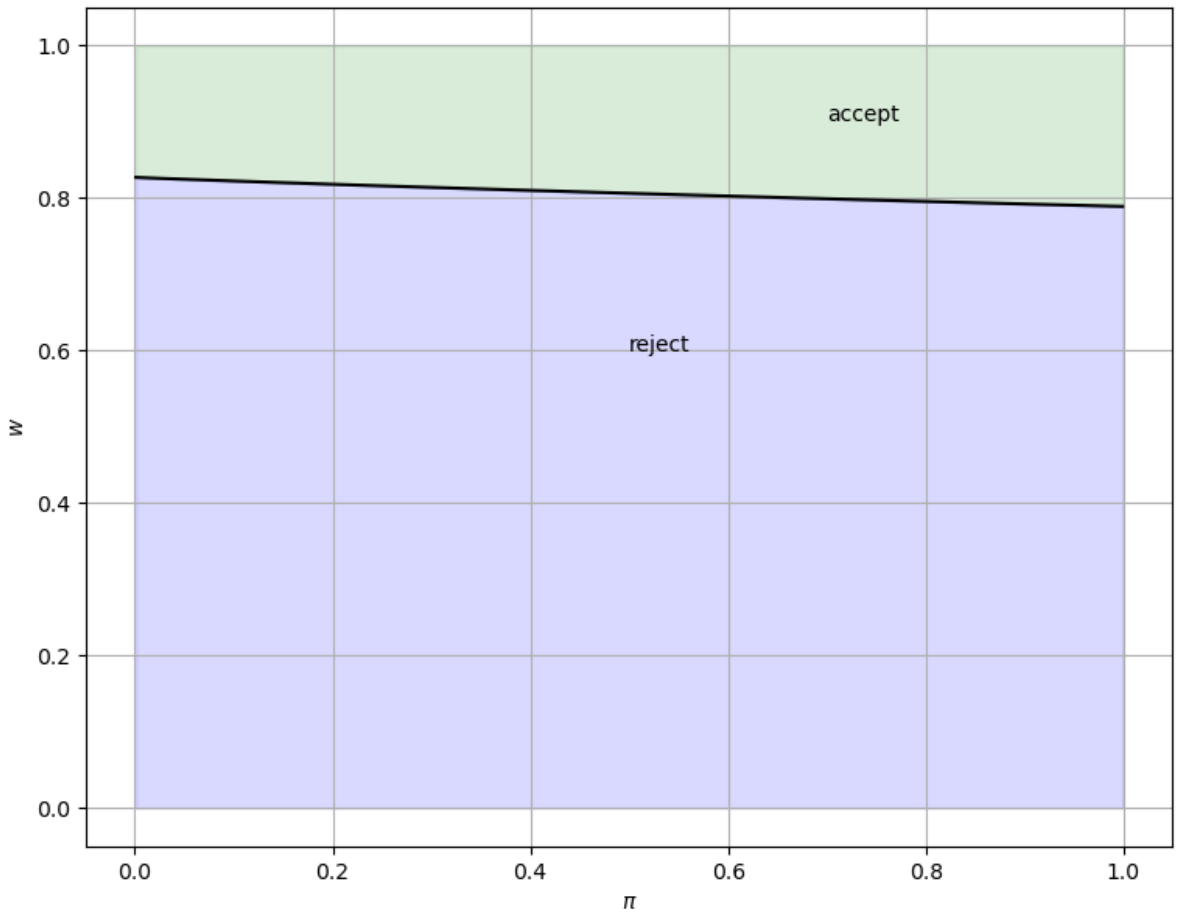
(continued from previous page)

```
    while i < max_iter and error > tol:
        w_new = Q(w)
        error = np.max(np.abs(w - w_new))
        i += 1
        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")
        w = w_new

    if error > tol:
        print("Failed to converge!")
    elif verbose:
        print(f"\nConverged in {i} iterations.")

    return w_new
```

The solution can be plotted as follows

```
sp = SearchProblem()
w_bar = solve_wbar(sp)

fig, ax = plt.subplots(figsize=(9, 7))

ax.plot(sp.π_grid, w_bar, color='k')
ax.fill_between(sp.π_grid, 0, w_bar, color='blue', alpha=0.15)
ax.fill_between(sp.π_grid, w_bar, sp.w_max, color='green', alpha=0.15)
ax.text(0.5, 0.6, 'reject')
ax.text(0.7, 0.9, 'accept')
ax.set(xlabel='$\pi$', ylabel='$w$')
ax.grid()
plt.show()
```

```
Error at iteration 5 is 0.02085314429269003.
Error at iteration 10 is 0.005718363925303449.
Error at iteration 15 is 0.001196238891997159.
Error at iteration 20 is 0.00023760790615423577.

Converged in 23 iterations.
```



21.10 Appendix A

The next piece of code generates a fun simulation to see what the effect of a change in the underlying distribution on the unemployment rate is.

At a point in the simulation, the distribution becomes significantly worse.

It takes a while for agents to learn this, and in the meantime, they are too optimistic and turn down too many jobs.

As a result, the unemployment rate spikes

```
F_a, F_b, G_a, G_b = 1, 1, 3, 1.2

sp = SearchProblem(F_a=F_a, F_b=F_b, G_a=G_a, G_b=G_b)
f, g = sp.f, sp.g

# Solve for reservation wage
w_bar = solve_wbar(sp, verbose=False)

# Interpolate reservation wage function
π_grid = sp.π_grid
w_func = njit(lambda x: interp(π_grid, w_bar, x))
```

(continues on next page)

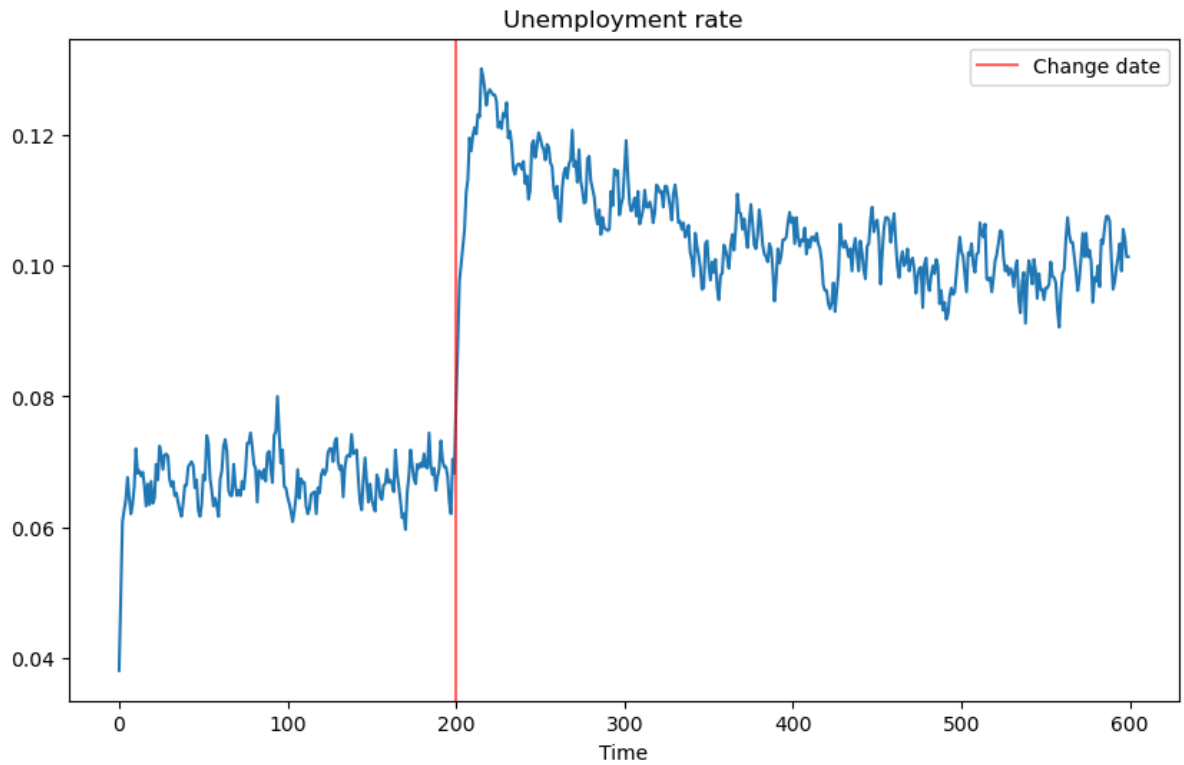
(continued from previous page)

```
@njit
def update(a, b, e, π):
    "Update e and π by drawing wage offer from beta distribution with parameters a␣
 ↪and b"

    if e == False:
        w = np.random.beta(a, b)       # Draw random wage
        if w >= w_func(π):
            e = True                   # Take new job
        else:
            π = 1 / (1 + ((1 - π) * g(w)) / (π * f(w)))

    return e, π

@njit
def simulate_path(F_a=F_a,
                  F_b=F_b,
                  G_a=G_a,
                  G_b=G_b,
                  N=5000,        # Number of agents
                  T=600,         # Simulation length
                  d=200,         # Change date
                  s=0.025):      # Separation rate

    """Simulates path of employment for N number of works over T periods"""

    e = np.ones((N, T+1))
    π = np.full((N, T+1), 1e-3)

    a, b = G_a, G_b   # Initial distribution parameters

    for t in range(T+1):

        if t == d:
            a, b = F_a, F_b  # Change distribution parameters

        # Update each agent
        for n in range(N):
            if e[n, t] == 1:                    # If agent is currently employment
                p = np.random.uniform(0, 1)
                if p <= s:                      # Randomly separate with probability s
                    e[n, t] = 0

            new_e, new_π = update(a, b, e[n, t], π[n, t])
            e[n, t+1] = new_e
            π[n, t+1] = new_π

    return e[:, 1:]

d = 200  # Change distribution at time d
unemployment_rate = 1 - simulate_path(d=d).mean(axis=0)

fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(unemployment_rate)
ax.axvline(d, color='r', alpha=0.6, label='Change date')
```

(continues on next page)

(continued from previous page)

```
ax.set_xlabel('Time')
ax.set_title('Unemployment rate')
ax.legend()
plt.show()
```



21.11 Appendix B

In this appendix we provide more details about how Bayes' Law contributes to the workings of the model.

We present some graphs that bring out additional insights about how learning works.

We build on graphs proposed in this lecture.

In particular, we'll add actions of our searching worker to a key graph presented in that lecture.

To begin, we first define two functions for computing the empirical distributions of unemployment duration and π at the time of employment.

```
@njit
def empirical_dist(F_a, F_b, G_a, G_b, w_bar, π_grid,
                   N=10000, T=600):
    """
    Simulates population for computing empirical cumulative
    distribution of unemployment duration and π at time when
    the worker accepts the wage offer. For each job searching
    problem, we simulate for two cases that either f or g is
    the true offer distribution.
```

(continues on next page)

(continued from previous page)

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

    F_a, F_b, G_a, G_b : parameters of beta distributions F and G.
    w_bar : the reservation wage
    π_grid : grid points of π, for interpolation
    N : number of workers for simulation, optional
    T : maximum of time periods for simulation, optional

    Returns
    -------
    accpet_t : 2 by N ndarray. the empirical distribution of
               unemployment duration when f or g generates offers.
    accept_π : 2 by N ndarray. the empirical distribution of
               π at the time of employment when f or g generates offers.
    """

    accept_t = np.empty((2, N))
    accept_π = np.empty((2, N))

    # f or g generates offers
    for i, (a, b) in enumerate([(F_a, F_b), (G_a, G_b)]):
        # update each agent
        for n in range(N):

            # initial priori
            π = 0.5

            for t in range(T+1):

                # Draw random wage
                w = np.random.beta(a, b)
                lw = p(w, F_a, F_b) / p(w, G_a, G_b)
                π = π * lw / (π * lw + 1 - π)

                # move to next agent if accepts
                if w >= interp(π_grid, w_bar, π):
                    break

            # record the unemployment duration
            # and π at the time of acceptance
            accept_t[i, n] = t
            accept_π[i, n] = π

    return accept_t, accept_π

def cumfreq_x(res):
    """
    A helper function for calculating the x grids of
    the cumulative frequency histogram.
    """

    cumcount = res.cumcount
    lowerlimit, binsize = res.lowerlimit, res.binsize
```

(continues on next page)

(continued from previous page)

```
    x = lowerlimit + np.linspace(0, binsize*cumcount.size, cumcount.size)

    return x
```

Now we define a wrapper function for analyzing job search models with learning under different parameterizations.

The wrapper takes parameters of beta distributions and unemployment compensation as inputs and then displays various things we want to know to interpret the solution of our search model.

In addition, it computes empirical cumulative distributions of two key objects.

```
def job_search_example(F_a=1, F_b=1, G_a=3, G_b=1.2, c=0.3):
    """
    Given the parameters that specify F and G distributions,
    calculate and display the rejection and acceptance area,
    the evolution of belief π, and the probability of accepting
    an offer at different π level, and simulate and calculate
    the empirical cumulative distribution of the duration of
    unemployment and π at the time the worker accepts the offer.
    """

    # construct a search problem
    sp = SearchProblem(F_a=F_a, F_b=F_b, G_a=G_a, G_b=G_b, c=c)
    f, g = sp.f, sp.g
    π_grid = sp.π_grid

    # Solve for reservation wage
    w_bar = solve_wbar(sp, verbose=False)

    # l(w) = f(w) / g(w)
    l = lambda w: f(w) / g(w)
    # objective function for solving l(w) = 1
    obj = lambda w: l(w) - 1.

    # the mode of beta distribution
    # use this to divide w into two intervals for root finding
    G_mode = (G_a - 1) / (G_a + G_b - 2)
    roots = np.empty(2)
    roots[0] = op.root_scalar(obj, bracket=[1e-10, G_mode]).root
    roots[1] = op.root_scalar(obj, bracket=[G_mode, 1-1e-10]).root

    fig, axs = plt.subplots(2, 2, figsize=(12, 9))

    # part 1: display the details of the model settings and some results
    w_grid = np.linspace(1e-12, 1-1e-12, 100)

    axs[0, 0].plot(l(w_grid), w_grid, label='$l$', lw=2)
    axs[0, 0].vlines(1., 0., 1., linestyle="--")
    axs[0, 0].hlines(roots, 0., 2., linestyle="--")
    axs[0, 0].set_xlim([0., 2.])
    axs[0, 0].legend(loc=4)
    axs[0, 0].set(xlabel='$l(w)=f(w)/g(w)$', ylabel='$w$')

    axs[0, 1].plot(sp.π_grid, w_bar, color='k')
    axs[0, 1].fill_between(sp.π_grid, 0, w_bar, color='blue', alpha=0.15)
    axs[0, 1].fill_between(sp.π_grid, w_bar, sp.w_max, color='green', alpha=0.15)
```

(continues on next page)

(continued from previous page)

```
    axs[0, 1].text(0.5, 0.6, 'reject')
    axs[0, 1].text(0.7, 0.9, 'accept')

    W = np.arange(0.01, 0.99, 0.08)
    Π = np.arange(0.01, 0.99, 0.08)

    ΔW = np.zeros((len(W), len(Π)))
    ΔΠ = np.empty((len(W), len(Π)))
    for i, w in enumerate(W):
        for j, π in enumerate(Π):
            lw = l(w)
            ΔΠ[i, j] = π * (lw / (π * lw + 1 - π) - 1)

    q = axs[0, 1].quiver(Π, W, ΔΠ, ΔW, scale=2, color='r', alpha=0.8)

    axs[0, 1].hlines(roots, 0., 1., linestyle="--")
    axs[0, 1].set(xlabel='$\pi$', ylabel='$w$')
    axs[0, 1].grid()

    axs[1, 0].plot(f(x_grid), x_grid, label='$f$', lw=2)
    axs[1, 0].plot(g(x_grid), x_grid, label='$g$', lw=2)
    axs[1, 0].vlines(1., 0., 1., linestyle="--")
    axs[1, 0].hlines(roots, 0., 2., linestyle="--")
    axs[1, 0].legend(loc=4)
    axs[1, 0].set(xlabel='$f(w), g(w)$', ylabel='$w$')

    axs[1, 1].plot(sp.π_grid, 1 - beta.cdf(w_bar, F_a, F_b), label='$f$')
    axs[1, 1].plot(sp.π_grid, 1 - beta.cdf(w_bar, G_a, G_b), label='$g$')
    axs[1, 1].set_ylim([0., 1.])
    axs[1, 1].grid()
    axs[1, 1].legend(loc=4)
    axs[1, 1].set(xlabel='$\pi$', ylabel='$\mathbb{P}\{w > \overline{w} (\pi)\}$')

    plt.show()

    # part 2: simulate empirical cumulative distribution
    accept_t, accept_π = empirical_dist(F_a, F_b, G_a, G_b, w_bar, π_grid)
    N = accept_t.shape[1]

    cfq_t_F = cumfreq(accept_t[0, :], numbins=100)
    cfq_π_F = cumfreq(accept_π[0, :], numbins=100)

    cfq_t_G = cumfreq(accept_t[1, :], numbins=100)
    cfq_π_G = cumfreq(accept_π[1, :], numbins=100)

    fig, axs = plt.subplots(2, 1, figsize=(12, 9))

    axs[0].plot(cumfreq_x(cfq_t_F), cfq_t_F.cumcount/N, label="f generates")
    axs[0].plot(cumfreq_x(cfq_t_G), cfq_t_G.cumcount/N, label="g generates")
    axs[0].grid(linestyle='--')
    axs[0].legend(loc=4)
    axs[0].title.set_text('CDF of duration of unemployment')
    axs[0].set(xlabel='time', ylabel='Prob(time)')

    axs[1].plot(cumfreq_x(cfq_π_F), cfq_π_F.cumcount/N, label="f generates")
    axs[1].plot(cumfreq_x(cfq_π_G), cfq_π_G.cumcount/N, label="g generates")
```

(continues on next page)

(continued from previous page)

```
    axs[1].grid(linestyle='--')
    axs[1].legend(loc=4)
    axs[1].title.set_text('CDF of π at time worker accepts wage and leaves␣
↪unemployment')
    axs[1].set(xlabel='π', ylabel='Prob(π)')

    plt.show()
```

We now provide some examples that provide insights about how the model works.

21.12 Examples

21.12.1 Example 1 (Baseline)

F ~ Beta(1, 1), G ~ Beta(3, 1.2), c=0.3.

In the graphs below, the red arrows in the upper right figure show how π_t is updated in response to the new information w_t.

Recall the following formula from this lecture

$$\frac{\pi_{t+1}}{\pi_t} = \frac{l\left(w_{t+1}\right)}{\pi_t l\left(w_{t+1}\right) + \left(1 - \pi_t\right)} \begin{cases} > 1 & \text{if } l\left(w_{t+1}\right) > 1 \\ \leq 1 & \text{if } l\left(w_{t+1}\right) \leq 1 \end{cases}$$

The formula implies that the direction of motion of π_t is determined by the relationship between $l(w_t)$ and 1.

The magnitude is small if

- $l(w)$ is close to 1, which means the new w is not very informative for distinguishing two distributions,
- π_{t-1} is close to either 0 or 1, which means the priori is strong.

Will an unemployed worker accept an offer earlier or not, when the actual ruling distribution is g instead of f?

Two countervailing effects are at work.

- if f generates successive wage offers, then w is more likely to be low, but π is moving up toward to 1, which lowers the reservation wage, i.e., the worker becomes less selective the longer he or she remains unemployed.
- if g generates wage offers, then w is more likely to be high, but π is moving downward toward 0, increasing the reservation wage, i.e., the worker becomes more selective the longer he or she remains unemployed.

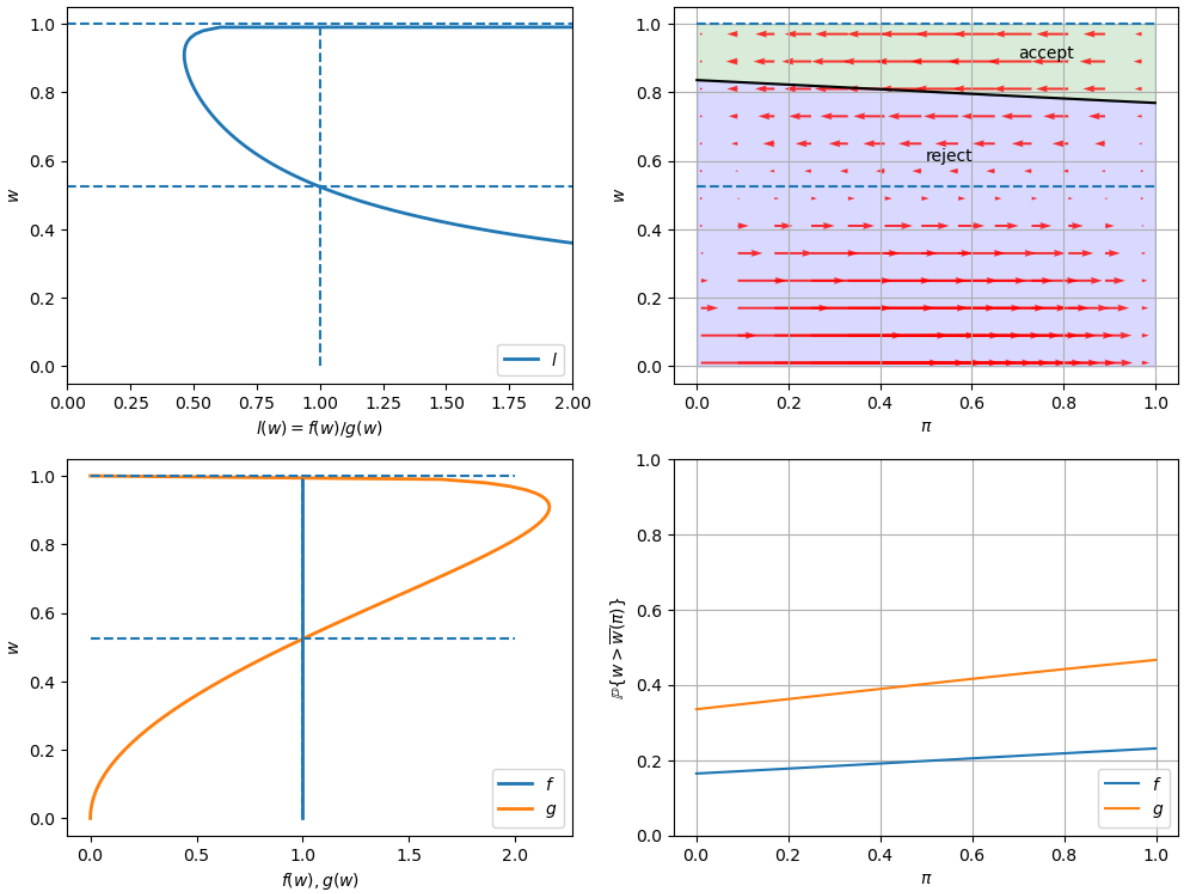
Quantitatively, the lower right figure sheds light on which effect dominates in this example.

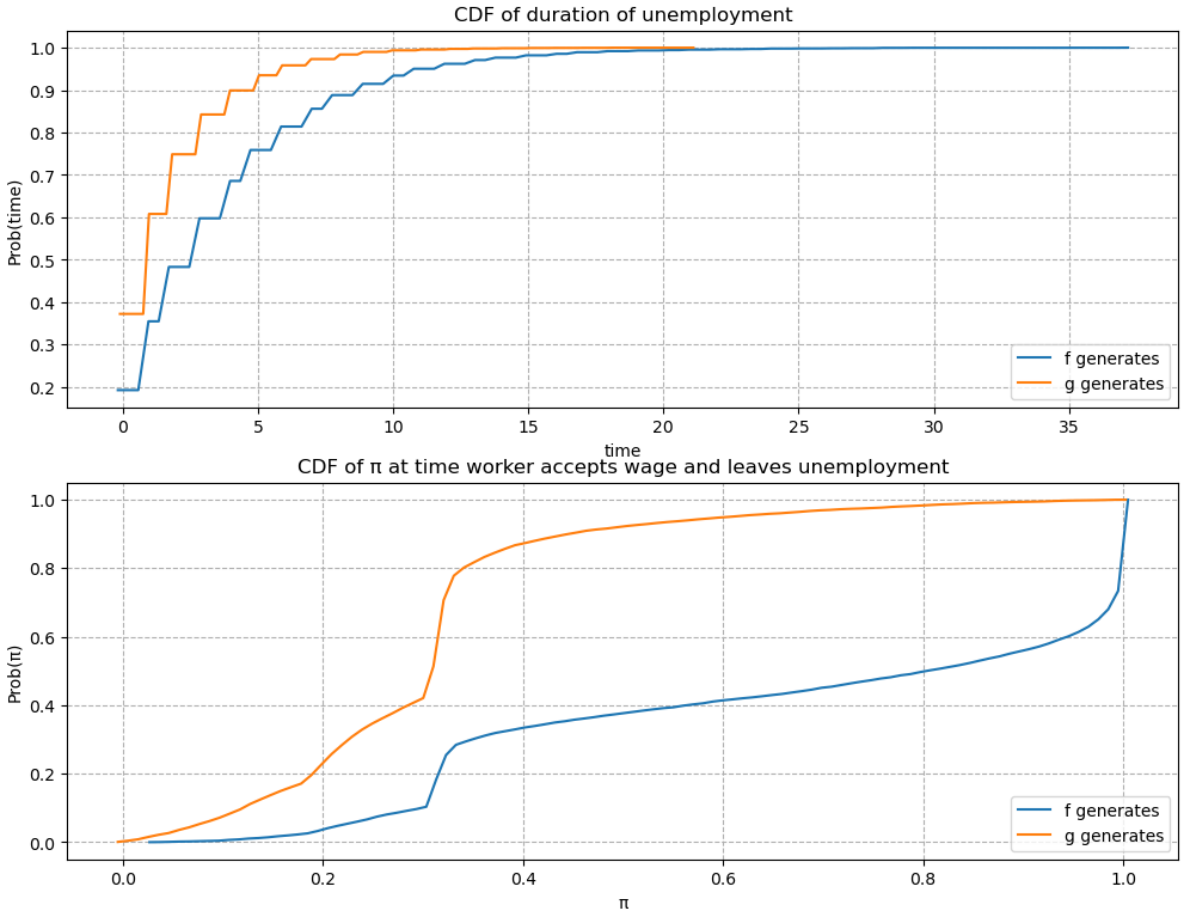
It shows the probability that a previously unemployed worker accepts an offer at different values of π when f or g generates wage offers.

That graph shows that for the particular f and g in this example, the worker is always more likely to accept an offer when f generates the data even when π is close to zero so that the worker believes the true distribution is g and therefore is relatively more selective.

The empirical cumulative distribution of the duration of unemployment verifies our conjecture.

```
job_search_example()
```


w
l
$l(w) = f(w)/g(w)$
accept
reject
π
$f(w), g(w)$
f
g
$\mathbb{P}\{w > \overline{w}(\pi)\}$



21.12.2 Example 2

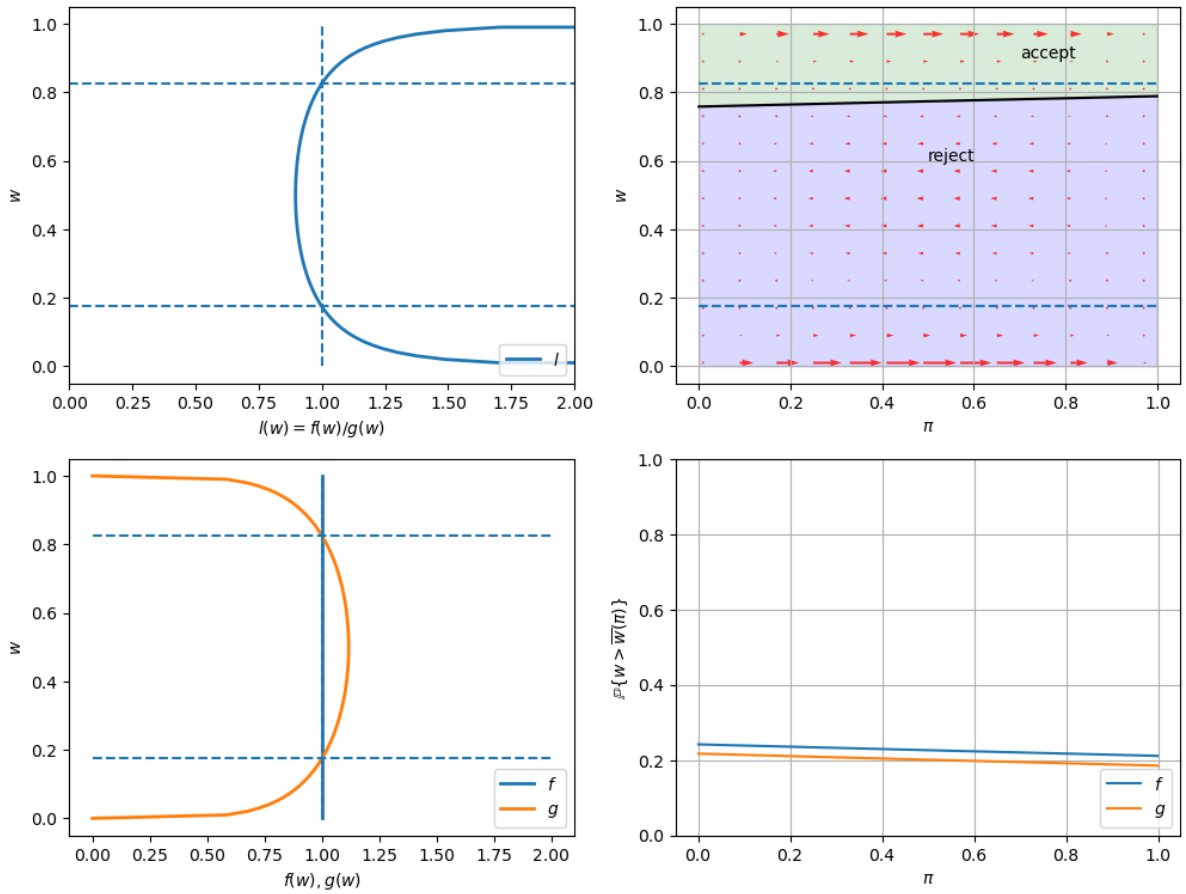
F ~ Beta(1, 1), G ~ Beta(1.2, 1.2), c=0.3.

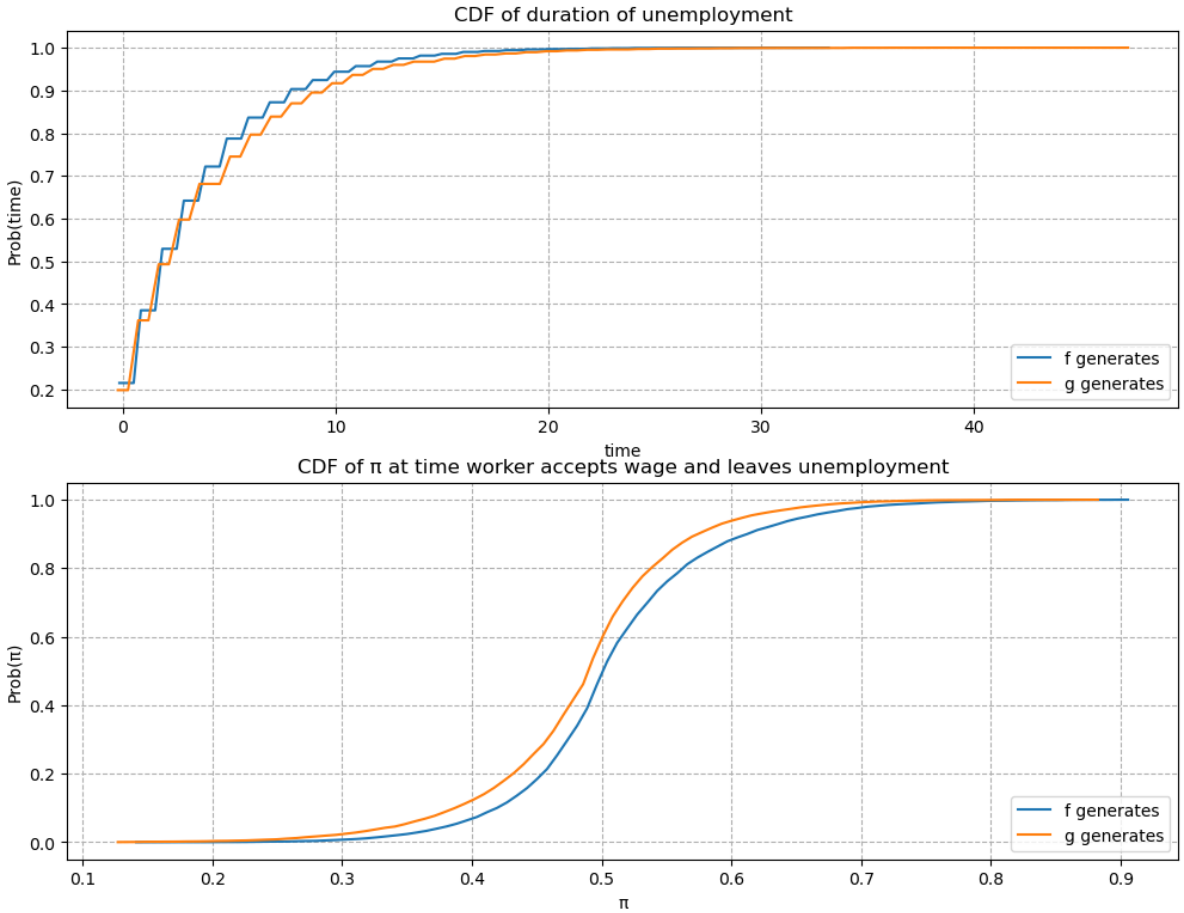
Now G has the same mean as F with a smaller variance.

Since the unemployment compensation c serves as a lower bound for bad wage offers, G is now an "inferior" distribution to F.

Consequently, we observe that the optimal policy $\overline{w}(\pi)$ is increasing in π.

```
job_search_example(1, 1, 1.2, 1.2, 0.3)
```


$l(w) = f(w)/g(w)$
l
accept
reject
π
$f(w), g(w)$
f
g
$\mathbb{P}\{w > \overline{w}(\pi)\}$

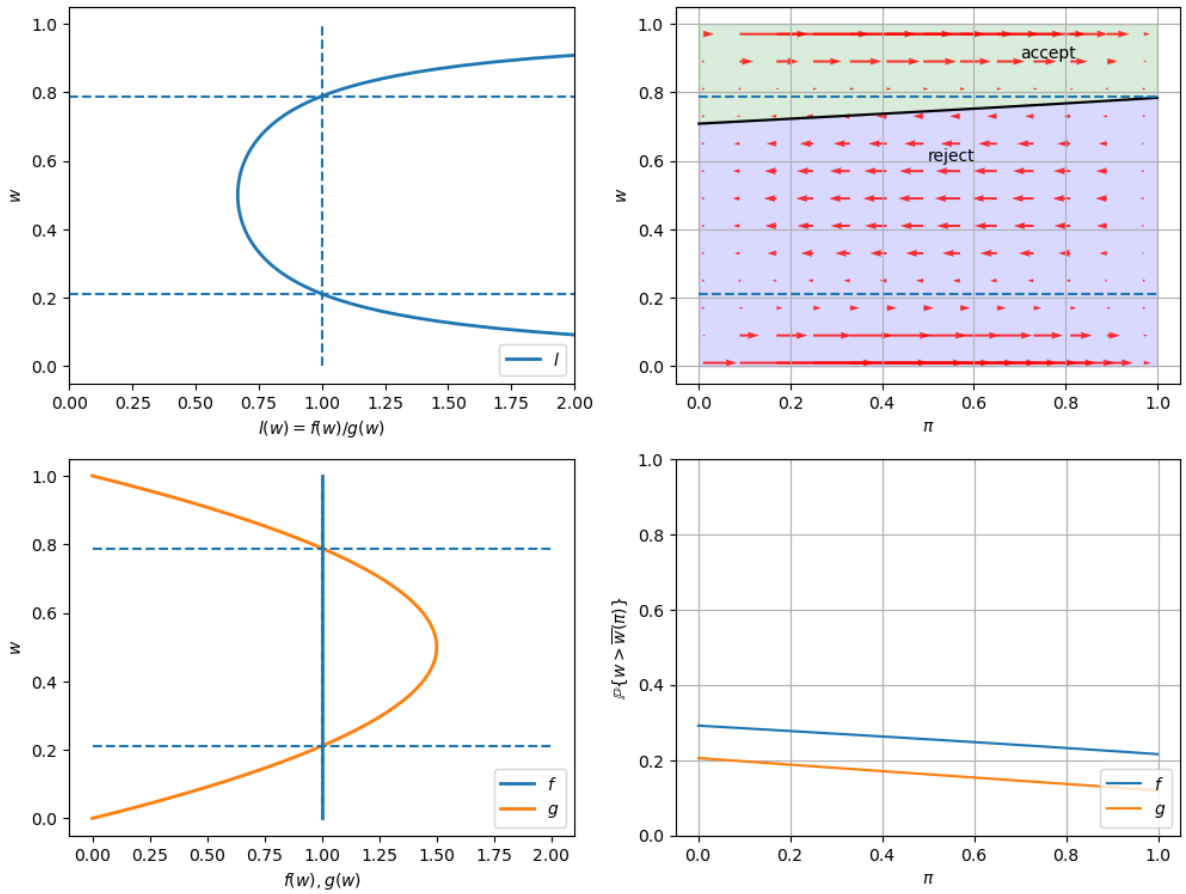


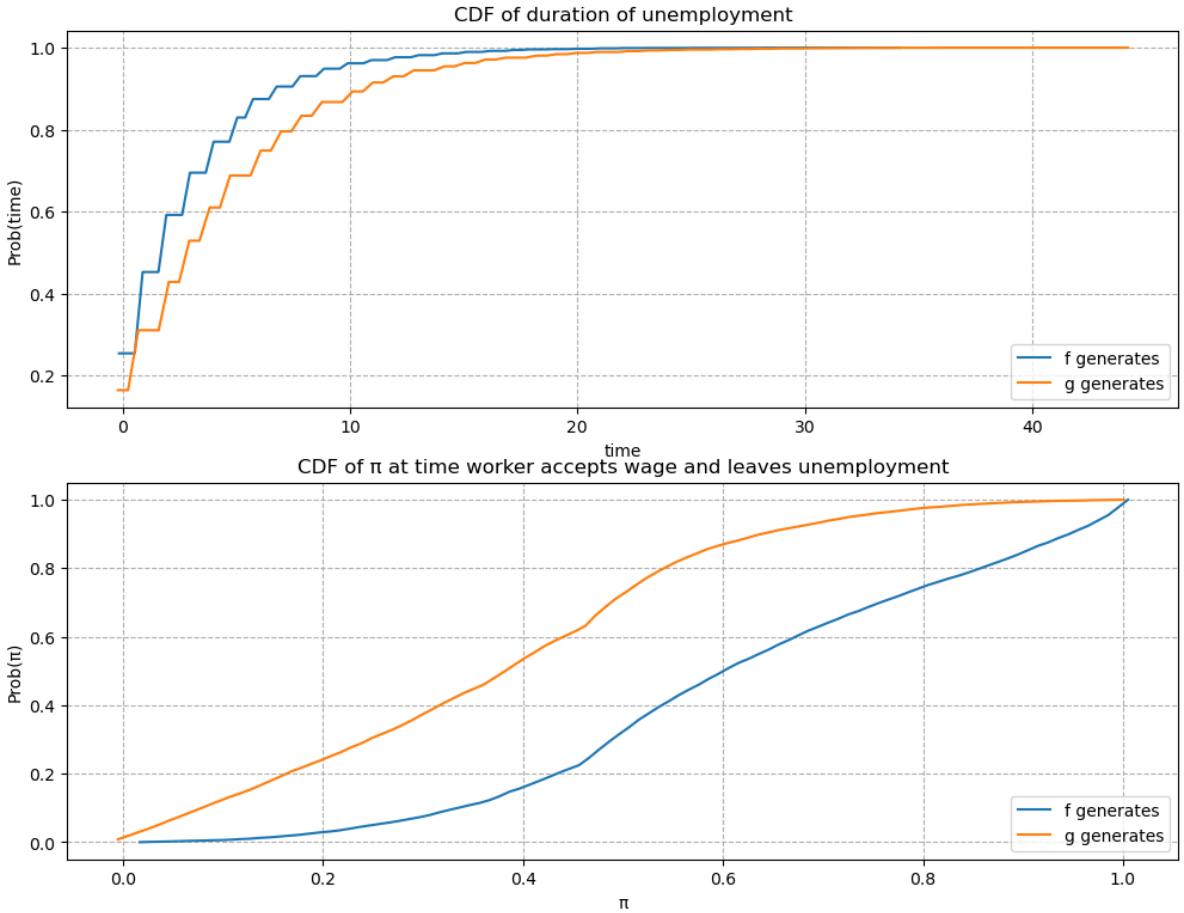
21.12.3 Example 3

F ~ Beta(1, 1), G ~ Beta(2, 2), c=0.3.

If the variance of G is smaller, we observe in the result that G is even more "inferior" and the slope of $\overline{w}(\pi)$ is larger.

```
job_search_example(1, 1, 2, 2, 0.3)
```


w
l(w) = f(w)/g(w)
l
accept
reject
π
f(w), g(w)
f
g
$\mathbb{P}\{w > \overline{w}(\pi)\}$



21.12.4 Example 4

F ~ Beta(1, 1), G ~ Beta(3, 1.2), and c=0.8.

In this example, we keep the parameters of beta distributions to be the same with the baseline case but increase the unemployment compensation c.

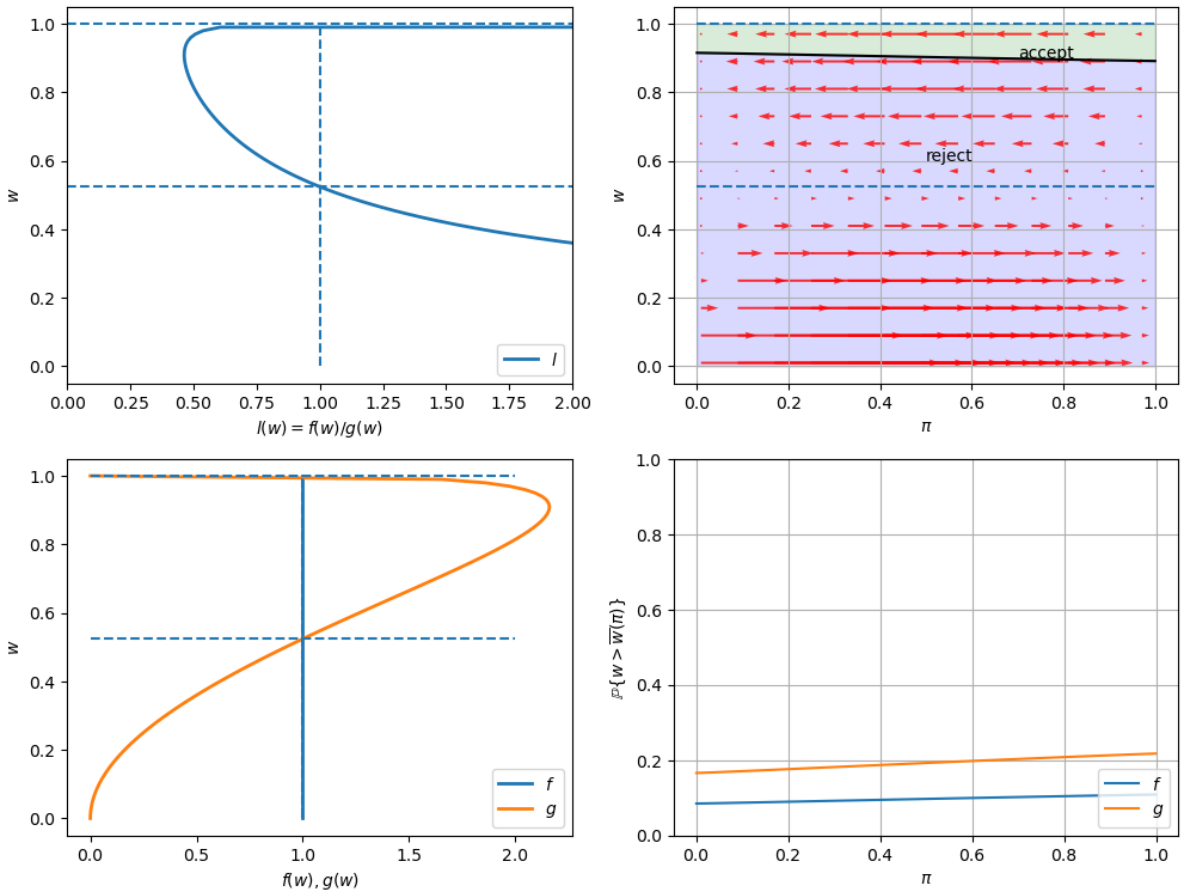
Comparing outcomes to the baseline case (example 1) in which unemployment compensation if low (c=0.3), now the worker can afford a longer learning period.

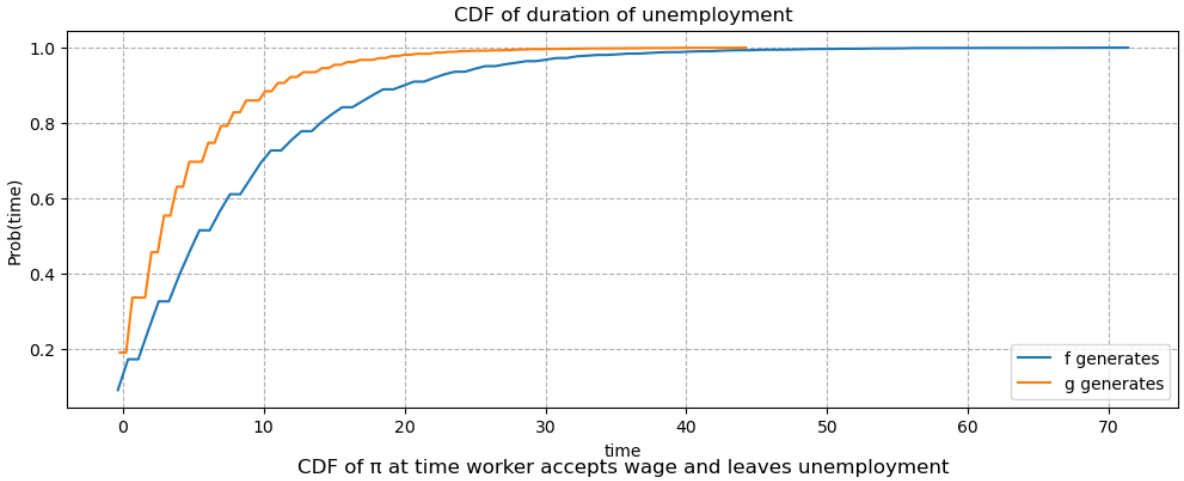
As a result, the worker tends to accept wage offers much later.

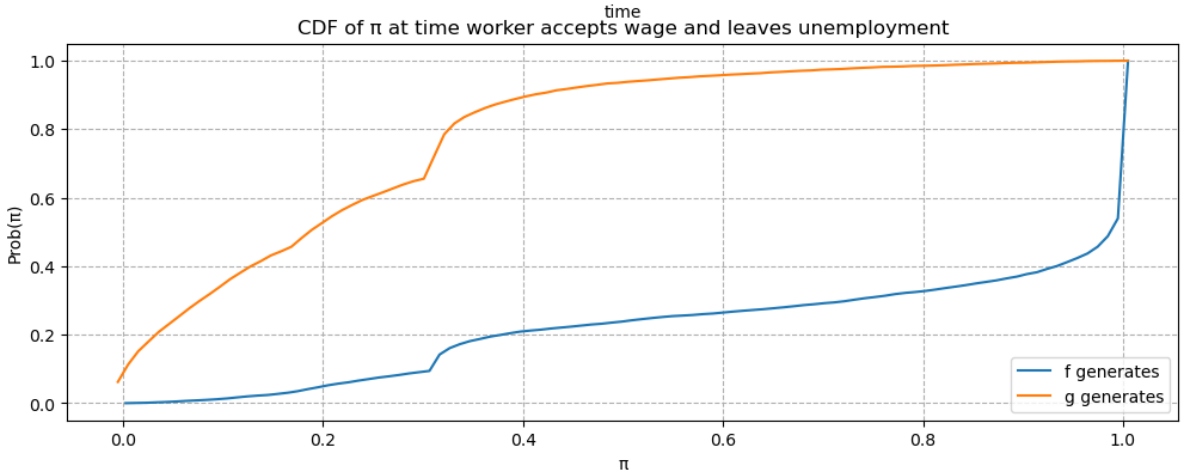
Furthermore, at the time of accepting employment, the belief π is closer to either 0 or 1.

That means that the worker has a better idea about what the true distribution is when he eventually chooses to accept a wage offer.

```
job_search_example(1, 1, 3, 1.2, c=0.8)
```


w
l(w) = f(w)/g(w)
l
accept
reject
π
f(w), g(w)
f
g
$\mathbb{P}^{\pi}\{w > \overline{w}(\pi)\}$



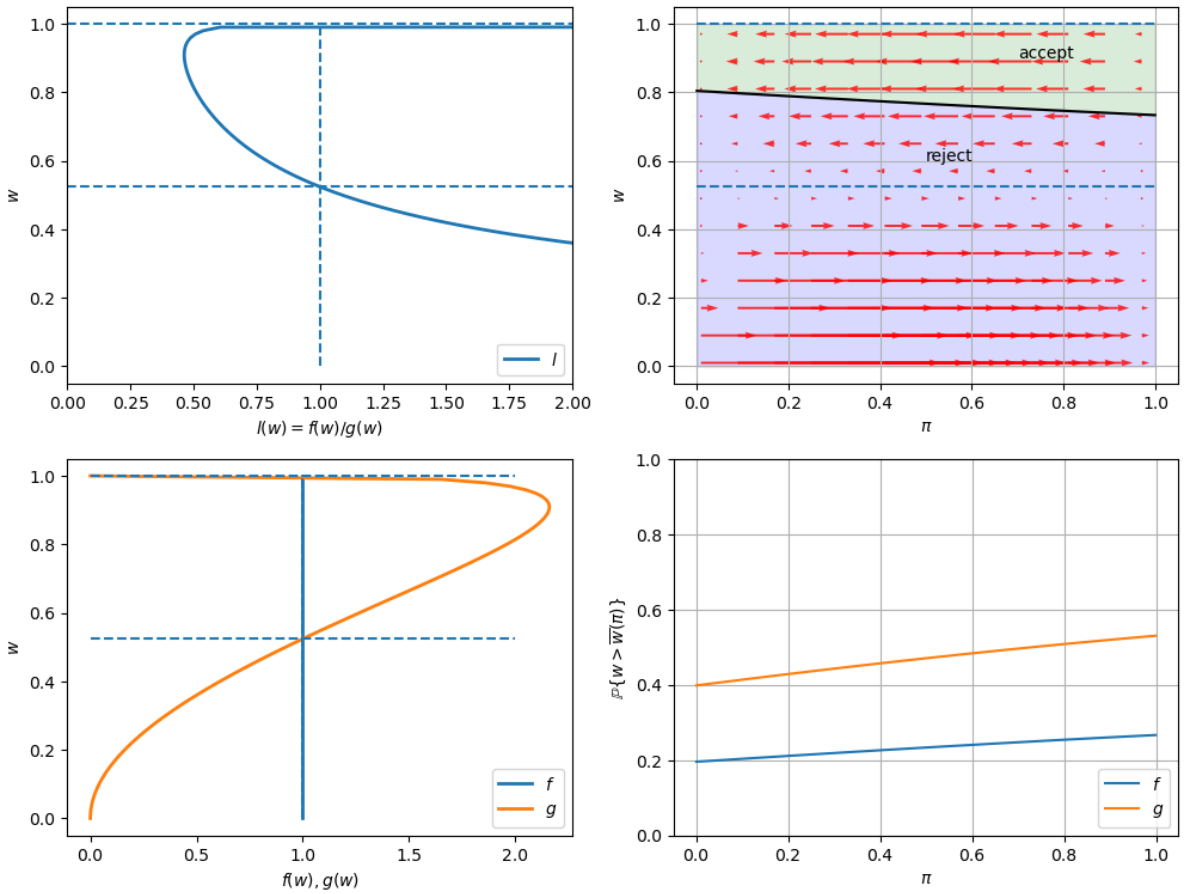


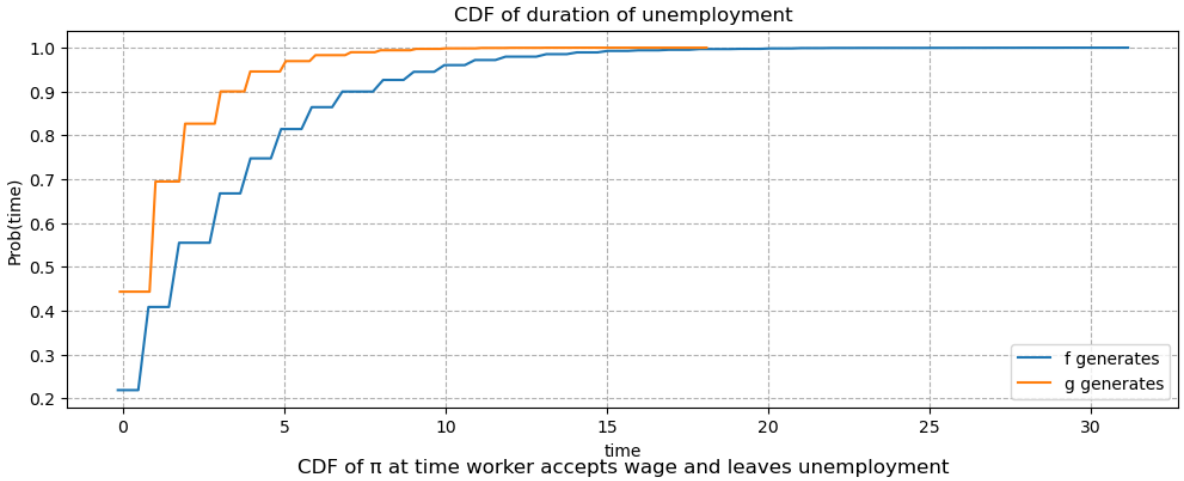
21.12.5 Example 5

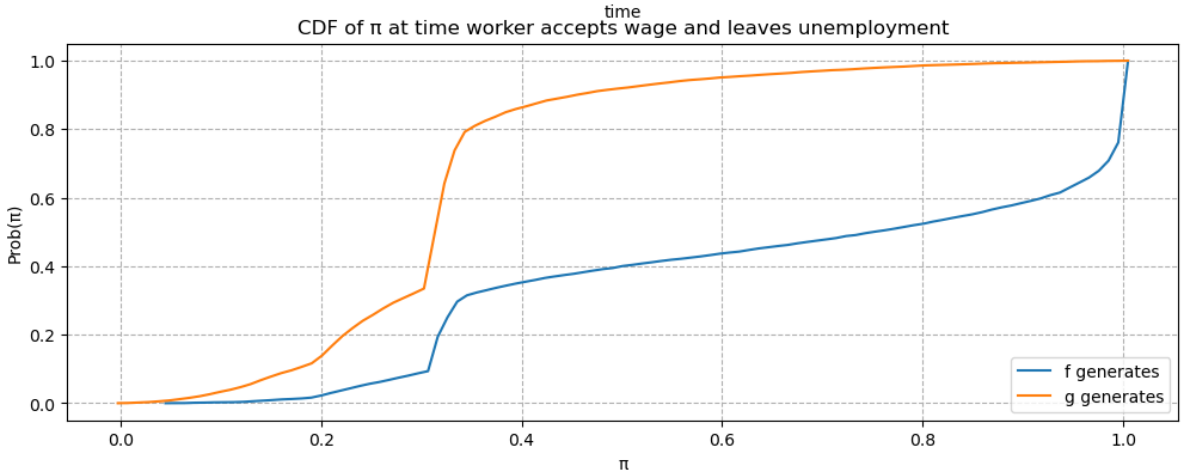
F ~ Beta(1, 1), G ~ Beta(3, 1.2), and c=0.1.

As expected, a smaller c makes an unemployed worker accept wage offers earlier after having acquired less information about the wage distribution.

```
job_search_example(1, 1, 3, 1.2, c=0.1)
```


w
l(w) = f(w)/g(w)
l
accept
reject
π
f(w), g(w)
f
g
$\mathbb{P}\{w > \overline{w}(\pi)\}$


CDF of duration of unemployment
Prob(time)
time
f generates
g generates


CDF of π at time worker accepts wage and leaves unemployment
Prob(π)
π
f generates
g generates

CHAPTER

TWENTYTWO

JOB SEARCH VIII: A MCCALL WORKER Q-LEARNS

22.1 Overview

This lecture illustrates a powerful machine learning technique called Q-learning.

[SB18] presents Q-learning and a variety of other statistical learning procedures.

The Q-learning algorithm combines ideas from

- dynamic programming
- a recursive version of least squares known as temporal difference learning.

This lecture applies a Q-learning algorithm to the situation faced by a McCall worker.

This lecture also considers the case where a McCall worker is given an option to quit the current job.

Relative to the dynamic programming formulation of the McCall worker model that we studied in *quantecon lecture*, a Q-learning algorithm gives the worker less knowledge about

- the random process that generates a sequence of wages
- the reward function that tells consequences of accepting or rejecting a job

The Q-learning algorithm invokes a statistical learning model to learn about these things.

Statistical learning often comes down to some version of least squares, and it will be here too.

Any time we say **statistical learning**, we have to say what object is being learned.

For Q-learning, the object that is learned is not the **value function** that is a focus of dynamic programming.

But it is something that is closely affiliated with it.

In the finite-action, finite state context studied in this lecture, the object to be learned statistically is a **Q-table**, an instance of a **Q-function** for finite sets.

Sometimes a Q-function or Q-table is called a quality-function or quality-table.

The rows and columns of a Q-table correspond to possible states that an agent might encounter, and possible actions that he can take in each state.

An equation that resembles a Bellman equation plays an important role in the algorithm.

It differs from the Bellman equation for the McCall model that we have seen in *this quantecon lecture*

In this lecture, we'll learn a little about

- the **Q-function** or **quality function** that is affiliated with any Markov decision problem whose optimal value function satisfies a Bellman equation
- **temporal difference learning**, a key component of a Q-learning algorithm

As usual, let's import some Python modules.

```
!pip install quantecon
```

```
import numpy as np

from numba import jit, float64, int64
from numba.experimental import jitclass
from quantecon.distributions import BetaBinomial

import matplotlib.pyplot as plt

np.random.seed(123)
```

22.2 Review of McCall Model

We begin by reviewing the McCall model described in *this quantecon lecture*.

We'll compute an optimal value function and a policy that attains it.

We'll eventually compare that optimal policy to what the Q-learning McCall worker learns.

The McCall model is characterized by parameters β, c and a known distribution of wage offers F.

A McCall worker wants to maximize an expected discounted sum of lifetime incomes

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t y_t$$

The worker's income y_t equals his wage w if he is employed, and unemployment compensation c if he is unemployed.

An optimal value $V(w)$ for a McCall worker who has just received a wage offer w and is deciding whether to accept or reject it satisfies the Bellman equation

$$V(w) = \max_{\text{accept, reject}} \left\{ \frac{w}{1-\beta}, c + \beta \int V(w')\, dF(w') \right\} \tag{22.1}$$

To form a benchmark to compare with results from Q-learning, we first approximate the optimal value function.

With possible states residing in a finite discrete state space indexed by $\{1, 2, ..., n\}$, we make an initial guess for the value function of $v \in \mathbb{R}^n$ and then iterate on the Bellman equation:

$$v'(i) = \max \left\{ \frac{w(i)}{1-\beta}, c + \beta \sum_{1 \leq j \leq n} v(j) q(j) \right\} \quad \text{for } i = 1, \dots, n$$

Let's use Python code from *this quantecon lecture*.

We use a Python method called `VFI` to compute the optimal value function using value function iterations.

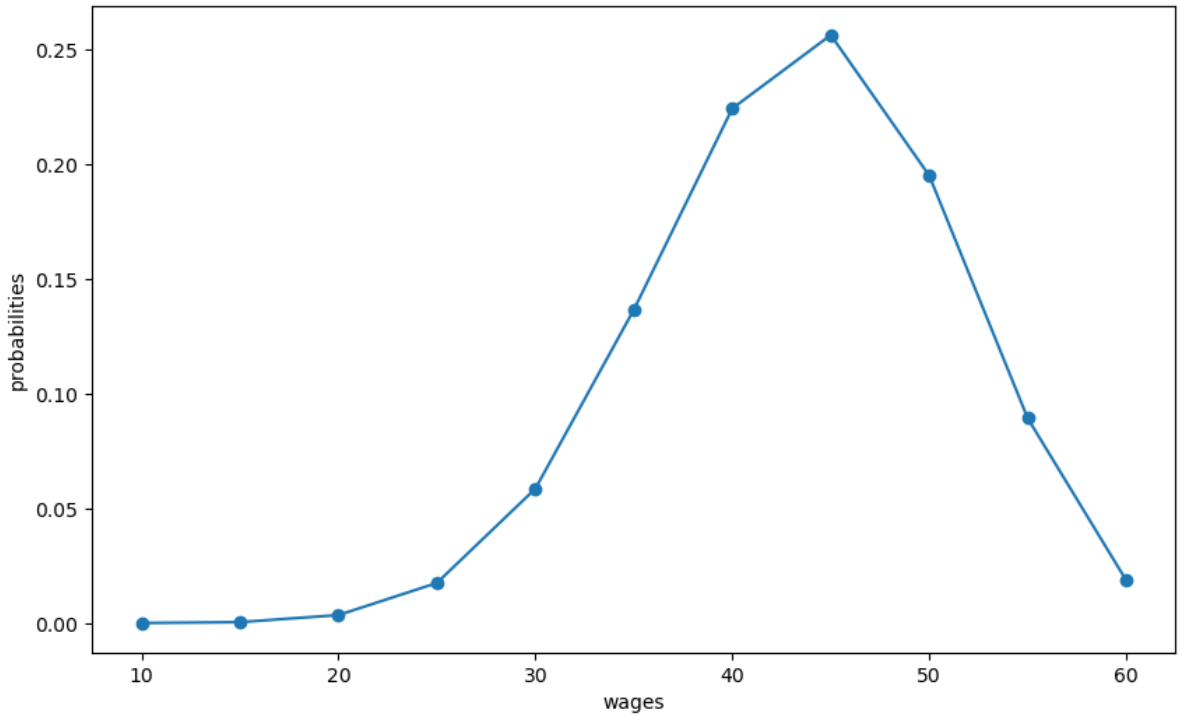
We construct an assumed distribution of wages and plot it with the following Python code

```
n, a, b = 10, 200, 100                        # default parameters
q_default = BetaBinomial(n, a, b).pdf()       # default choice of q

w_min, w_max = 10, 60
w_default = np.linspace(w_min, w_max, n+1)
```

(continues on next page)

(continued from previous page)

```
# plot distribution of wage offer
fig, ax = plt.subplots(figsize=(10,6))
ax.plot(w_default, q_default, '-o', label='$q(w(i))$')
ax.set_xlabel('wages')
ax.set_ylabel('probabilities')

plt.show()
```



Next we'll compute the worker's optimal value function by iterating to convergence on the Bellman equation.

Then we'll plot various iterates on the Bellman operator.

```
mccall_data = [
    ('c', float64),       # unemployment compensation
    ('β', float64),       # discount factor
    ('w', float64[:]),    # array of wage values, w[i] = wage at state i
    ('q', float64[:]),     # array of probabilities
]


@jitclass(mccall_data)
class McCallModel:

    def __init__(self, c=25, β=0.99, w=w_default, q=q_default):

        self.c, self.β = c, β
        self.w, self.q = w, q

    def state_action_values(self, i, v):
```

(continues on next page)

(continued from previous page)

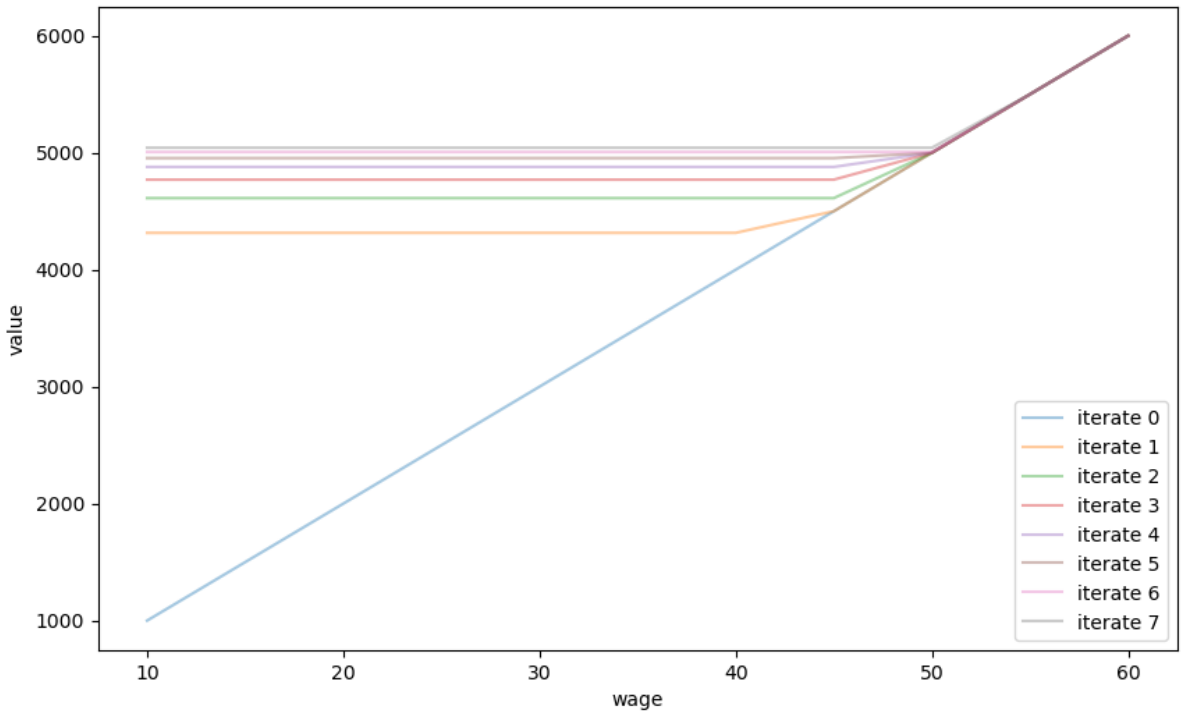
```
        """
        The values of state-action pairs.
        """
        # Simplify names
        c, β, w, q = self.c, self.β, self.w, self.q
        # Evaluate value for each state-action pair
        # Consider action = accept or reject the current offer
        accept = w[i] / (1 - β)
        reject = c + β * np.sum(v * q)

        return np.array([accept, reject])

    def VFI(self, eps=1e-5, max_iter=500):
        """
        Find the optimal value function.
        """

        n = len(self.w)
        v = self.w / (1 - self.β)
        v_next = np.empty_like(v)
        flag=0

        for i in range(max_iter):
            for j in range(n):
                v_next[j] = np.max(self.state_action_values(j, v))

            if np.max(np.abs(v_next - v))<=eps:
                flag=1
                break
            v[:] = v_next

        return v, flag

def plot_value_function_seq(mcm, ax, num_plots=8):
    """
    Plot a sequence of value functions.

        * mcm is an instance of McCallModel
        * ax is an axes object that implements a plot method.

    """

    n = len(mcm.w)
    v = mcm.w / (1 - mcm.β)
    v_next = np.empty_like(v)
    for i in range(num_plots):
        ax.plot(mcm.w, v, '-', alpha=0.4, label=f"iterate {i}")
        # Update guess
        for i in range(n):
            v_next[i] = np.max(mcm.state_action_values(i, v))
        v[:] = v_next  # copy contents into v

    ax.legend(loc='lower right')
```

```
mcm = McCallModel()
```

(continues on next page)

(continued from previous page)

```
valfunc_VFI, flag = mcm.VFI()

fig, ax = plt.subplots(figsize=(10,6))
ax.set_xlabel('wage')
ax.set_ylabel('value')
plot_value_function_seq(mcm, ax)
plt.show()
```



Next we'll print out the limit of the sequence of iterates.

This is the approximation to the McCall worker's value function that is produced by value function iteration.

We'll use this value function as a benchmark later after we have done some Q-learning.

```
print(valfunc_VFI)
```

```
[5322.27935875 5322.27935875 5322.27935875 5322.27935875 5322.27935875
 5322.27935875 5322.27935875 5322.27935875 5322.27935875 5500.
 6000.        ]
```

22.3 Implied Quality Function Q

A **quality function** Q map state-action pairs into optimal values.

They are tightly linked to optimal value functions.

But value functions are functions just of states, and not actions.

For each given state, the quality function gives a list of optimal values that can be attained starting from that state, with each component of the list indicating one of the possible actions that is taken.

For our McCall worker with a finite set of possible wages

- the state space $\mathcal{W} = \{w_1, w_2, ..., w_n\}$ is indexed by integers $1, 2, ..., n$
- the action space is $\mathcal{A} = \{\text{accept}, \text{reject}\}$

Let $a \in \mathcal{A}$ be one of the two possible actions, i.e., accept or reject.

For our McCall worker, an optimal Q-function $Q(w, a)$ equals the maximum value of that a previously unemployed worker who has offer w in hand can attain if he takes action a.

This definition of $Q(w, a)$ presumes that in subsequent periods the worker takes optimal actions.

An optimal Q-function for our McCall worker satisfies

$$\begin{aligned} Q\left(w, \text{accept}\right) &= \frac{w}{1-\beta} \\ Q\left(w, \text{reject}\right) &= c + \beta \int \max_{\text{accept, reject}} \left\{ \frac{w'}{1-\beta}, Q\left(w', \text{reject}\right) \right\} dF\left(w'\right) \end{aligned} \tag{22.2}$$

Note that the first equation of system (22.2) presumes that after the agent has accepted an offer, he will not have the objection to reject that same offer in the future.

These equations are aligned with the Bellman equation for the worker's optimal value function that we studied in *this quantecon lecture*.

Evidently, the optimal value function $V(w)$ described in that lecture is related to our Q-function by

$$V(w) = \max_{\text{accept,reject}} \left\{ Q(w, \text{accept}), Q\left(w, \text{reject}\right) \right\}$$

If we stare at the second equation of system (22.2), we notice that since the wage process is identically and independently distributed over time, $Q\left(w, \text{reject}\right)$, the right side of the equation is independent of the current state w.

So we can denote it as a scalar

$$Q_r := Q\left(w, \text{reject}\right) \quad \forall\, w \in \mathcal{W}.$$

This fact provides us with an an alternative, and as it turns out in this case, a faster way to compute an optimal value function and associated optimal policy for the McCall worker.

Instead of using the value function iterations that we deployed above, we can instead iterate to convergence on a version of the second equation in system (22.2) that maps an estimate of Q_r into an improved estimate Q_r':

$$Q_r' = c + \beta \int \max \left\{ \frac{w'}{1-\beta}, Q_r \right\} dF\left(w'\right)$$

After a Q_r sequence has converged, we can recover the optimal value function $V(w)$ for the McCall worker from

$$V\left(w\right) = \max \left\{ \frac{w}{1-\beta}, Q_r \right\}$$

22.4 From Probabilities to Samples

We noted above that the optimal Q function for our McCall worker satisfies the Bellman equations

$$\begin{aligned} w+\beta \max_{\text{accept, reject}}\{Q(w,\text{accept}),Q(w,\text{reject})\}-Q(w,\text{accept})=0 \\ c+\beta\int \max_{\text{accept, reject}}\{Q(w',\text{accept}),Q\left(w',\text{reject}\right)\}\,dF\left(w'\right)-Q\left(w,\text{reject}\right)=0 \end{aligned} \tag{22.3}$$

Notice the integral over $F(w')$ on the second line.

Erasing the integral sign sets the stage for an illegitmate argument that can get us started thinking about Q-learning.

Thus, construct a difference equation system that keeps the first equation of (22.3) but replaces the second by removing integration over $F(w')$:

$$\begin{aligned} w+\beta \max_{\text{accept, reject}}\{Q(w,\text{accept}),Q(w,\text{reject})\}-Q(w,\text{accept})=0 \\ c+\beta \max_{\text{accept, reject}}\{Q(w',\text{accept}),Q\left(w',\text{reject}\right)\}-Q\left(w,\text{reject}\right)\approx 0 \end{aligned} \tag{22.4}$$

The second equation can't hold for all w, w' pairs in the appropriate Cartesian product of our state space.

But maybe an appeal to a Law of Large numbers could let us hope that it would hold **on average** for a long time series sequence of draws of w_t, w_{t+1} pairs, where we are thinking of w_t as w and w_{t+1} as w'.

The basic idea of Q-learning is to draw a long sample of wage offers from F (we know F though we assume that the worker doesn't) and iterate on a recursion that maps an estimate $\hat{Q}_t$ of a Q-function at date t into an improved estimate $\hat{Q}_{t+1}$ at date $t+1$.

To set up such an algorithm, we first define some errors or "differences"

$$\begin{aligned} w+\beta \max_{\text{accept, reject}}\left\{\hat{Q}_t(w_t,\text{accept}),\hat{Q}_t(w_t,\text{reject})\right\}-\hat{Q}_t(w_t,\text{accept})=\text{diff}_{\text{accept},t} \\ c+\beta \max_{\text{accept, reject}}\left\{\hat{Q}_t(w_{t+1},\text{accept}),\hat{Q}_t\left(w_{t+1},\text{reject}\right)\right\}-\hat{Q}_t\left(w_t,\text{reject}\right)=\text{diff}_{\text{reject},t} \end{aligned} \tag{22.5}$$

The adaptive learning scheme would then be some version of

$$\hat{Q}_{t+1}=\hat{Q}_t+\alpha\,\text{diff}_t \tag{22.6}$$

where $\alpha\in(0,1)$ is a small **gain** parameter that governs the rate of learning and $\hat{Q}_t$ and diff_t are 2×1 vectors corresponding to objects in equation system (22.5).

This informal argument takes us to the threshold of Q-learning.

22.5 Q-Learning

Let's first describe a Q-learning algorithm precisely.

Then we'll implement it.

The algorithm works by using a Monte Carlo method to update estimates of a Q-function.

We begin with an initial guess for a Q-function.

In the example studied in this lecture, we have a finite action space and also a finite state space.

That means that we can represent a Q-function as a matrix or Q-table, $\widetilde{Q}(w,a)$.

Q-learning proceeds by updating the Q-function as the decision maker acquires experience along a path of wage draws generated by simulation.

During the learning process, our McCall worker takes actions and experiences rewards that are consequences of those actions.

He learns simultaneously about the environment, in this case the distribution of wages, and the reward function, in this case the unemployment compensation c and the present value of wages.

The updating algorithm is based on a slight modification (to be described soon) of a recursion like

$$\widetilde{Q}^{new}(w,a) = \widetilde{Q}^{old}(w,a) + \alpha \widetilde{TD}(w,a) \tag{22.7}$$

where

$$\begin{aligned} \widetilde{TD}(w,\text{accept}) &= \left[w + \beta \max_{a' \in \mathcal{A}} \widetilde{Q}^{old}(w,a')\right] - \widetilde{Q}^{old}(w,\text{accept}) \\ \widetilde{TD}(w,\text{reject}) &= \left[c + \beta \max_{a' \in \mathcal{A}} \widetilde{Q}^{old}(w',a')\right] - \widetilde{Q}^{old}(w,\text{reject}),\ w' \sim F \end{aligned} \tag{22.8}$$

The terms $\widetilde{TD}(w,a)$ for $a = \{\text{accept,reject}\}$ are the **temporal difference errors** that drive the updates.

This system is thus a version of the adaptive system that we sketched informally in equation (22.6).

An aspect of the algorithm not yet captured by equation system (22.8) is random **experimentation** that we add by occasionally randomly replacing

$$\text{argmax}_{a' \in \mathcal{A}} \widetilde{Q}^{old}(w,a')$$

with

$$\text{argmin}_{a' \in \mathcal{A}} \widetilde{Q}^{old}(w,a')$$

and occasionally replacing

$$\text{argmax}_{a' \in \mathcal{A}} \widetilde{Q}^{old}(w',a')$$

with

$$\text{argmin}_{a' \in \mathcal{A}} \widetilde{Q}^{old}(w',a')$$

We activate such experimentation with probability ϵ in step 3 of the following pseudo-code for our McCall worker to do Q-learning:

1. Set an arbitrary initial Q-table.
2. Draw an initial wage offer w from F.
3. From the appropriate row in the Q-table, choose an action using the following ϵ-greedy algorithm:
   - with probability $1-\epsilon$, choose the action that maximizes the value, and
   - with probability ϵ, choose the alternative action.
4. Update the state associated with the chosen action and compute $\widetilde{TD}$ according to (22.8) and update $\widetilde{Q}$ according to (22.7).
5. Either draw a new state w' if required or else take existing wage if and update the Q-table again according to (22.7).
6. Stop when the old and new Q-tables are close enough, i.e., $\|\tilde{Q}^{new} - \tilde{Q}^{old}\|_\infty \leq \delta$ for given δ or if the worker keeps accepting for T periods for a prescribed T.

7. Return to step 2 with the updated Q-table.

Repeat this procedure for N episodes or until the updated Q-table has converged.

We call one pass through steps 2 to 7 an "episode" or "epoch" of temporal difference learning.

In our context, each episode starts with an agent drawing an initial wage offer, i.e., a new state.

The agent then takes actions based on the preset Q-table, receives rewards, and then enters a new state implied by this period's actions.

The Q-table is updated via temporal difference learning.

We iterate this until convergence of the Q-table or the maximum length of an episode is reached.

Multiple episodes allow the agent to start afresh and visit states that she was less likely to visit from the terminal state of a previos episode.

For example, an agent who has accepted a wage offer based on her Q-table will be less likely to draw a new offer from other parts of the wage distribution.

By using the ϵ-greedy method and also by increasing the number of episodes, the Q-learning algorithm balances gains from exploration and from exploitation.

Remark: Notice that $\widetilde{TD}$ associated with an optimal Q-table defined in (22.7) automatically above satisfies $\widetilde{TD} = 0$ for all state action pairs. Whether a limit of our Q-learning algorithm converges to an optimal Q-table depends on whether the algorithm visits all state-action pairs often enough.

We implement this pseudo code in a Python class.

For simplicity and convenience, we let `s` represent the state index between 0 and $n = 50$ and $w_s = w[s]$.

The first column of the Q-table represents the value associated with rejecting the wage and the second represents accepting the wage.

We use `numba` compilation to accelerate computations.

```
params=[
    ('c', float64),             # unemployment compensation
    ('β', float64),             # discount factor
    ('w', float64[:]),          # array of wage values, w[i] = wage at state i
    ('q', float64[:]),          # array of probabilities
    ('eps', float64),           # for epsilon greedy algorithm
    ('δ', float64),             # Q-table threshold
    ('lr', float64),            # the learning rate α
    ('T', int64),               # maximum periods of accepting
    ('quit_allowed', int64)     # whether quit is allowed after accepting the wage␣
 ↪offer
]

@jitclass(params)
class Qlearning_McCall:
    def __init__(self, c=25, β=0.99, w=w_default, q=q_default, eps=0.1,
                 δ=1e-5, lr=0.5, T=10000, quit_allowed=0):

        self.c, self.β = c, β
        self.w, self.q = w, q
        self.eps, self.δ, self.lr, self.T = eps, δ, lr, T
        self.quit_allowed = quit_allowed


    def draw_offer_index(self):
```

(continues on next page)

(continued from previous page)

```
        """
        Draw a state index from the wage distribution.
        """

        q = self.q
        return np.searchsorted(np.cumsum(q), np.random.random(), side="right")

    def temp_diff(self, qtable, state, accept):
        """
        Compute the TD associated with state and action.
        """

        c, β, w = self.c, self.β, self.w

        if accept==0:
            state_next = self.draw_offer_index()
            TD = c + β*np.max(qtable[state_next, :]) - qtable[state, accept]
        else:
            state_next = state
            if self.quit_allowed == 0:
                TD = w[state_next] + β*np.max(qtable[state_next, :]) - qtable[state,␣
↪accept]
            else:
                TD = w[state_next] + β*qtable[state_next, 1] - qtable[state, accept]

        return TD, state_next

    def run_one_epoch(self, qtable, max_times=20000):
        """
        Run an "epoch".
        """

        c, β, w = self.c, self.β, self.w
        eps, δ, lr, T = self.eps, self.δ, self.lr, self.T

        s0 = self.draw_offer_index()
        s = s0
        accept_count = 0

        for t in range(max_times):

            # choose action
            accept = np.argmax(qtable[s, :])
            if np.random.random()<=eps:
                accept = 1 - accept

            if accept == 1:
                accept_count += 1
            else:
                accept_count = 0

            TD, s_next = self.temp_diff(qtable, s, accept)

            # update qtable
            qtable_new = qtable.copy()
            qtable_new[s, accept] = qtable[s, accept] + lr*TD
```

(continues on next page)

(continued from previous page)

```
            if np.max(np.abs(qtable_new-qtable))<=δ:
                break

            if accept_count == T:
                break

            s, qtable = s_next, qtable_new

        return qtable_new

@jit(nopython=True)
def run_epochs(N, qlmc, qtable):
    """
    Run epochs N times with qtable from the last iteration each time.
    """

    for n in range(N):
        if n%(N/10)==0:
            print(f"Progress: EPOCHs = {n}")
        new_qtable = qlmc.run_one_epoch(qtable)
        qtable = new_qtable

    return qtable

def valfunc_from_qtable(qtable):
    return np.max(qtable, axis=1)

def compute_error(valfunc, valfunc_VFI):
    return np.mean(np.abs(valfunc-valfunc_VFI))
```

```
# create an instance of Qlearning_McCall
qlmc = Qlearning_McCall()

# run
qtable0 = np.zeros((len(w_default), 2))
qtable = run_epochs(20000, qlmc, qtable0)
```

```
Progress: EPOCHs = 0
Progress: EPOCHs = 2000
Progress: EPOCHs = 4000
Progress: EPOCHs = 6000
Progress: EPOCHs = 8000
Progress: EPOCHs = 10000
Progress: EPOCHs = 12000
Progress: EPOCHs = 14000
Progress: EPOCHs = 16000
Progress: EPOCHs = 18000
```

```
print(qtable)
```

```
[[2750.66171732   58.37719309]
 [5343.62039433 5184.12761459]
```

(continues on next page)

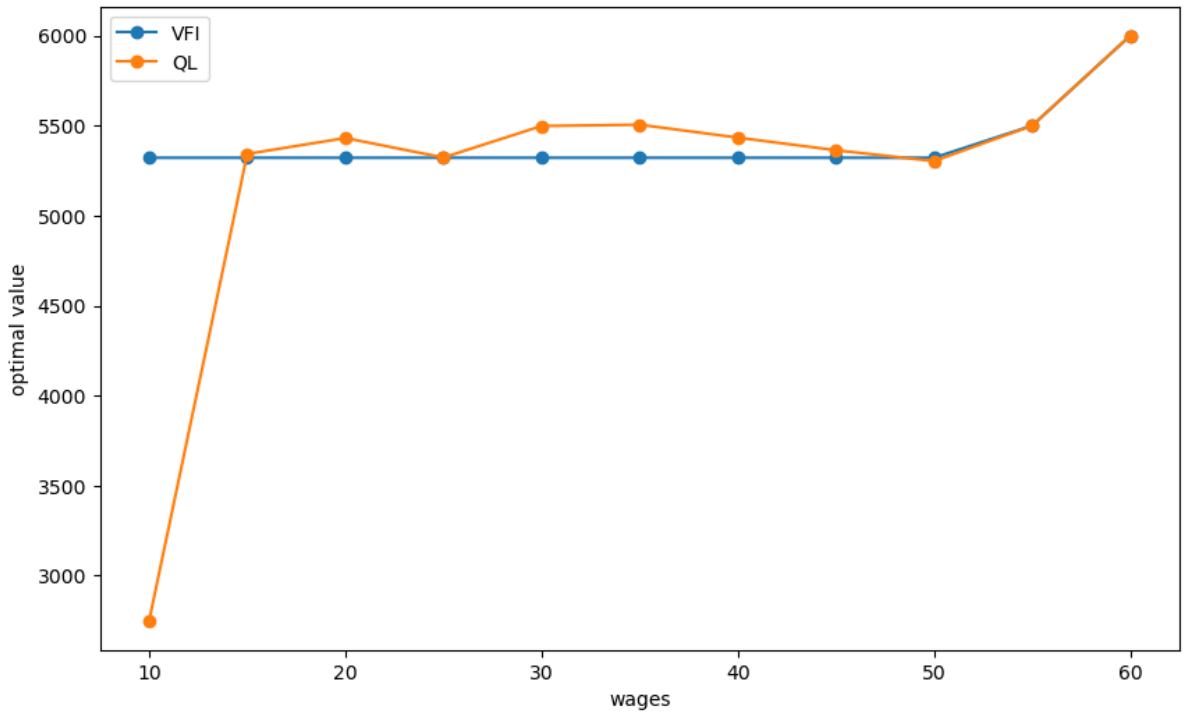
(continued from previous page)

```
 [5431.99315537 5270.18631235]
 [5323.6432479  5225.73867082]
 [5498.79221079 5492.44460336]
 [5505.92838719 5279.3135874 ]
 [5434.24327996 5296.10600802]
 [5364.21345504 5273.41067615]
 [5303.63028844 5290.23691017]
 [5327.48856577 5500.00000273]
 [5332.49759811 6000.        ]]
```

```
# inspect value function
valfunc_qlr = valfunc_from_qtable(qtable)

print(valfunc_qlr)
```

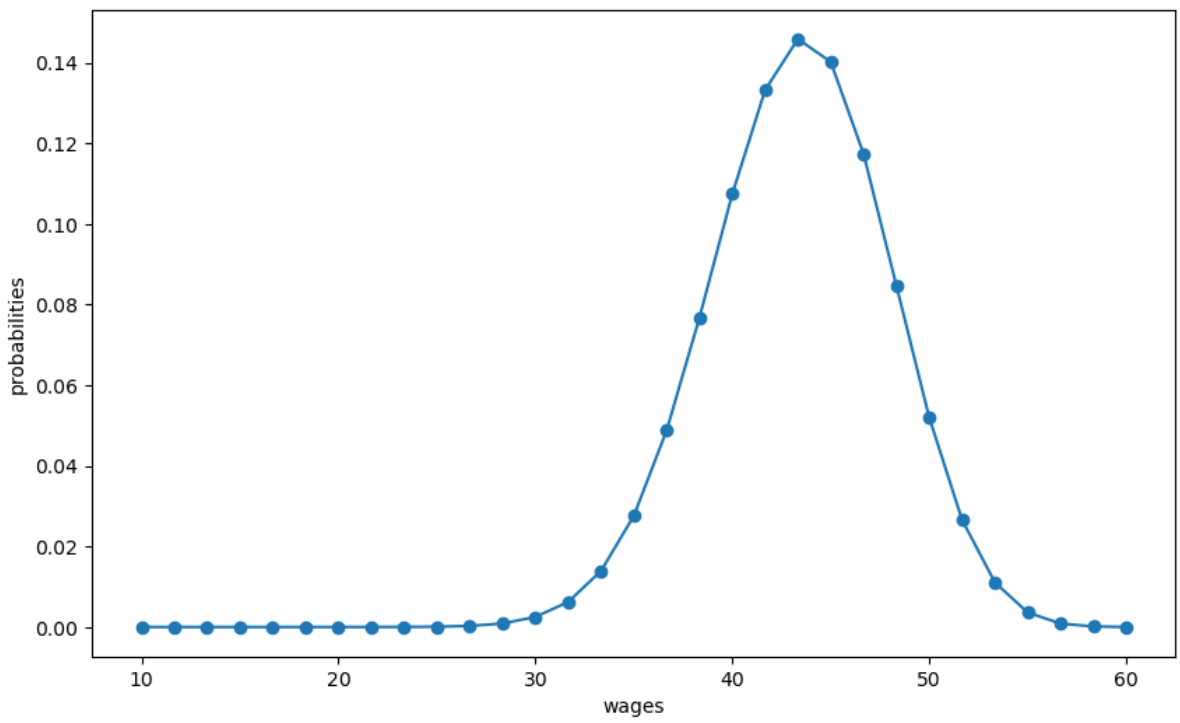
```
[2750.66171732 5343.62039433 5431.99315537 5323.6432479  5498.79221079
 5505.92838719 5434.24327996 5364.21345504 5303.63028844 5500.00000273
 6000.        ]
```

```
# plot
fig, ax = plt.subplots(figsize=(10,6))
ax.plot(w_default, valfunc_VFI, '-o', label='VFI')
ax.plot(w_default, valfunc_qlr, '-o', label='QL')
ax.set_xlabel('wages')
ax.set_ylabel('optimal value')
ax.legend()

plt.show()
```



Now, let us compute the case with a larger state space: $n = 30$ instead of $n = 10$.

```
n, a, b = 30, 200, 100                          # default parameters
q_new = BetaBinomial(n, a, b).pdf()             # default choice of q

w_min, w_max = 10, 60
w_new = np.linspace(w_min, w_max, n+1)


# plot distribution of wage offer
fig, ax = plt.subplots(figsize=(10,6))
ax.plot(w_new, q_new, '-o', label='$q(w(i))$')
ax.set_xlabel('wages')
ax.set_ylabel('probabilities')

plt.show()

# VFI
mcm = McCallModel(w=w_new, q=q_new)
valfunc_VFI, flag = mcm.VFI()
```




```
mcm = McCallModel(w=w_new, q=q_new)
valfunc_VFI, flag = mcm.VFI()
valfunc_VFI
```

```
array([4859.77015703, 4859.77015703, 4859.77015703, 4859.77015703,
       4859.77015703, 4859.77015703, 4859.77015703, 4859.77015703,
       4859.77015703, 4859.77015703, 4859.77015703, 4859.77015703,
       4859.77015703, 4859.77015703, 4859.77015703, 4859.77015703,
       4859.77015703, 4859.77015703, 4859.77015703, 4859.77015703,
```

(continues on next page)

(continued from previous page)

```
       4859.77015703, 4859.77015703, 4859.77015703, 4859.77015703,
       5000.        , 5166.66666667, 5333.33333333, 5500.        ,
       5666.66666667, 5833.33333333, 6000.        ])
```

```
def plot_epochs(epochs_to_plot, quit_allowed=1):
    "Plot value function implied by outcomes of an increasing number of epochs."
    qlmc_new = Qlearning_McCall(w=w_new, q=q_new, quit_allowed=quit_allowed)
    qtable = np.zeros((len(w_new),2))
    epochs_to_plot = np.asarray(epochs_to_plot)
    # plot
    fig, ax = plt.subplots(figsize=(10,6))
    ax.plot(w_new, valfunc_VFI, '-o', label='VFI')

    max_epochs = np.max(epochs_to_plot)
    # iterate on epoch numbers
    for n in range(max_epochs + 1):
        if n%(max_epochs/10)==0:
            print(f"Progress: EPOCHs = {n}")
        if n in epochs_to_plot:
            valfunc_qlr = valfunc_from_qtable(qtable)
            error = compute_error(valfunc_qlr, valfunc_VFI)

            ax.plot(w_new, valfunc_qlr, '-o', label=f'QL:epochs={n}, mean error=
↪{error}')


        new_qtable = qlmc_new.run_one_epoch(qtable)
        qtable = new_qtable

    ax.set_xlabel('wages')
    ax.set_ylabel('optimal value')
    ax.legend(loc='lower right')
    plt.show()
```

```
plot_epochs(epochs_to_plot=[100, 1000, 10000, 100000, 200000])
```

```
Progress: EPOCHs = 0
```

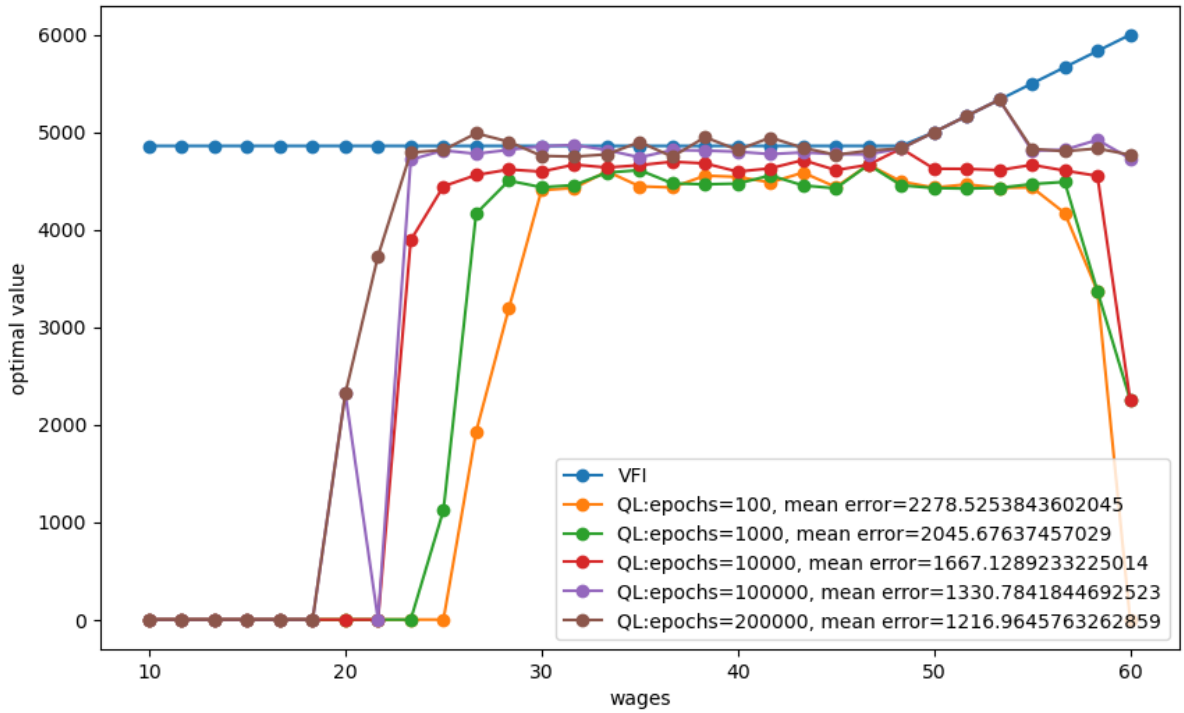
```
Progress: EPOCHs = 20000
Progress: EPOCHs = 40000
```

```
Progress: EPOCHs = 60000
Progress: EPOCHs = 80000
```

```
Progress: EPOCHs = 100000
Progress: EPOCHs = 120000
```

```
Progress: EPOCHs = 140000
Progress: EPOCHs = 160000
```

```
Progress: EPOCHs = 180000
Progress: EPOCHs = 200000
```



The above graphs indicates that

- the Q-learning algorithm has trouble learning the Q-table well for wages that are rarely drawn
- the quality of approximation to the "true" value function computed by value function iteration improves for longer epochs

22.6 Employed Worker Can't Quit

The preceding version of temporal difference Q-learning described in equation system (22.8) lets an employed worker quit, i.e., reject her wage as an incumbent and instead receive unemployment compensation this period and draw a new offer next period.

This is an option that the McCall worker described in *this quantecon lecture* would not take.

See [LS18], chapter 6 on search, for a proof.

But in the context of Q-learning, giving the worker the option to quit and get unemployment compensation while unemployed turns out to accelerate the learning process by promoting experimentation vis a vis premature exploitation only.

To illustrate this, we'll amend our formulas for temporal differences to forbid an employed worker from quitting a job she had accepted earlier.

With this understanding about available choices, we obtain the following temporal difference values:

$$
\begin{aligned}
\widetilde{TD}\left(w,\text{accept}\right) &= \left[w+\beta\widetilde{Q}^{old}\left(w,\text{accept}\right)\right]-\widetilde{Q}^{old}\left(w,\text{accept}\right)\\
\widetilde{TD}\left(w,\text{reject}\right) &= \left[c+\beta\max_{a'\in\mathcal{A}}\widetilde{Q}^{old}\left(w',a'\right)\right]-\widetilde{Q}^{old}\left(w,\text{reject}\right),\ w'\sim F
\end{aligned}
\tag{22.9}
$$

It turns out that formulas (22.9) combined with our Q-learning recursion (22.7) can lead our agent to eventually learn the optimal value function as well as in the case where an option to redraw can be exercised.

But learning is slower because an agent who ends up accepting a wage offer prematurally loses the option to explore new states in the same episode and to adjust the value associated with that state.

This can lead to inferior outcomes when the number of epochs/episodes is low.

But if we increase the number of epochs/episodes, we can observe that the error decreases and the outcomes get better.

We illustrate these possibilities with the following code and graph.

```
plot_epochs(epochs_to_plot=[100, 1000, 10000, 100000, 200000], quit_allowed=0)
```

```
Progress: EPOCHs = 0
```

```
Progress: EPOCHs = 20000
```

```
Progress: EPOCHs = 40000
```

```
Progress: EPOCHs = 60000
```

```
Progress: EPOCHs = 80000
```

```
Progress: EPOCHs = 100000
```

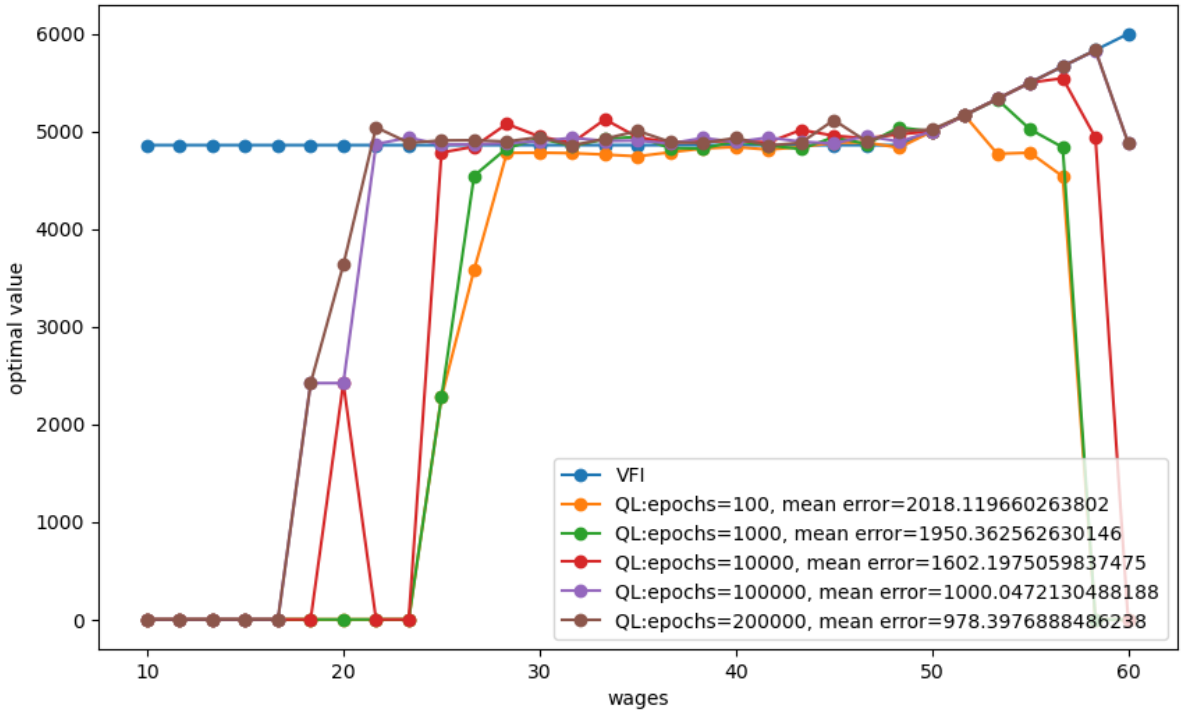
```
Progress: EPOCHs = 120000
```

```
Progress: EPOCHs = 140000
```

```
Progress: EPOCHs = 160000
```

```
Progress: EPOCHs = 180000
```

```
Progress: EPOCHs = 200000
```



22.7 Possible Extensions

To extend the algorthm to handle problems with continuous state spaces, a typical approach is to restrict Q-functions and policy functions to take particular functional forms.

This is the approach in **deep Q-learning** where the idea is to use a multilayer neural network as a good function approximator.

We will take up this topic in a subsequent quantecon lecture.

CHAPTER

TWENTYTHREE

A LAKE MODEL OF EMPLOYMENT AND UNEMPLOYMENT

Contents

- *A Lake Model of Employment and Unemployment*
  - *Overview*
  - *The Model*
  - *Implementation*
  - *Dynamics of an Individual Worker*
  - *Endogenous Job Finding Rate*
  - *Exercises*

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

```
!pip install quantecon
```

23.1 Overview

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

The lake model is a basic tool for modeling unemployment.

It allows us to analyze

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

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

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

The "flows" between the lakes are caused by

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

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

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

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

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

Let's start with some imports:

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

23.1.1 Prerequisites

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

You will also need some basic linear algebra and probability.

23.2 The Model

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

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

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

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

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

23.2.1 Aggregate Variables

We want to derive the dynamics of the following aggregates

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

We also want to know the values of the following objects

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

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

23.2.2 Laws of Motion for Stock Variables

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

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

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

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

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

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

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

A similar analysis implies

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

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

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

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

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

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

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

23.2.3 Laws of Motion for Rates

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

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

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

Letting

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

we can also write this as

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

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

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

23.3 Implementation

Let's code up these equations.

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

This class will

1. store the primitives α, λ, b, d
2. compute and store the implied objects $g, A, \hat{A}$
3. provide methods to simulate dynamics of the stocks and rates
4. provide a method to compute the steady state vector $\bar{x}$ of employment and unemployment rates using *a technique* we previously introduced for computing stationary distributions of Markov chains

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

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

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

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

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

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

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

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


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

        Returns
        --------
        xbar : steady state vector of employment and unemployment rates
        """
```

(continues on next page)

(continued from previous page)

```
        x = np.array([self.A_hat[0, 1], self.A_hat[1, 0]])
        x /= x.sum()
        return x

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

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

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

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

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

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

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

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

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

```
lm = LakeModel()
lm.α
```

```
0.013
```

```
lm.A
```

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

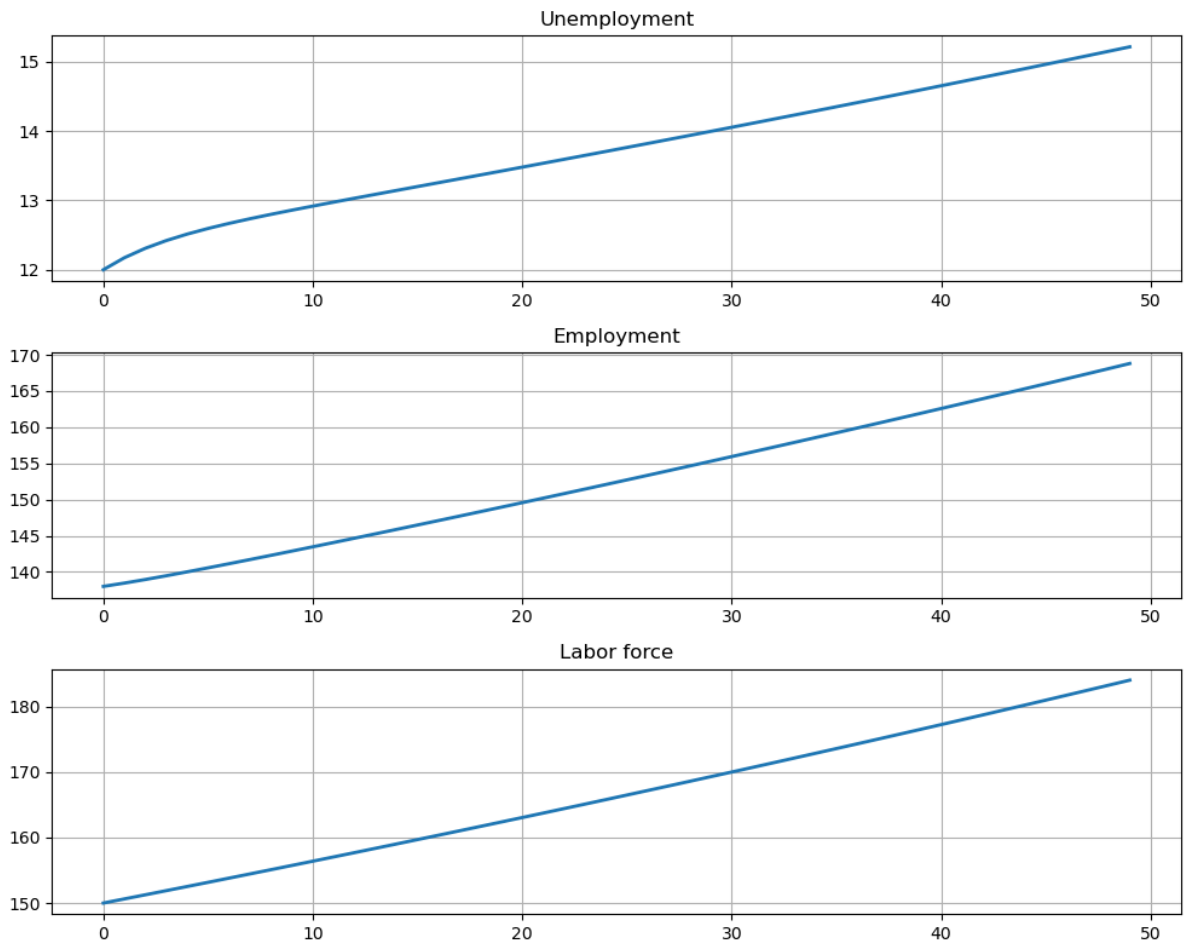
```
lm = LakeModel(α = 0.03)
lm.A
```

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

23.3.1 Aggregate Dynamics

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

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

U_0 = u_0 * N_0
E_0 = e_0 * N_0

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

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

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

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

for ax in axes:
    ax.grid()

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



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

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

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

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

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

This is the case for our default parameters:

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

```
(0.6953067378358462, 1.0)
```

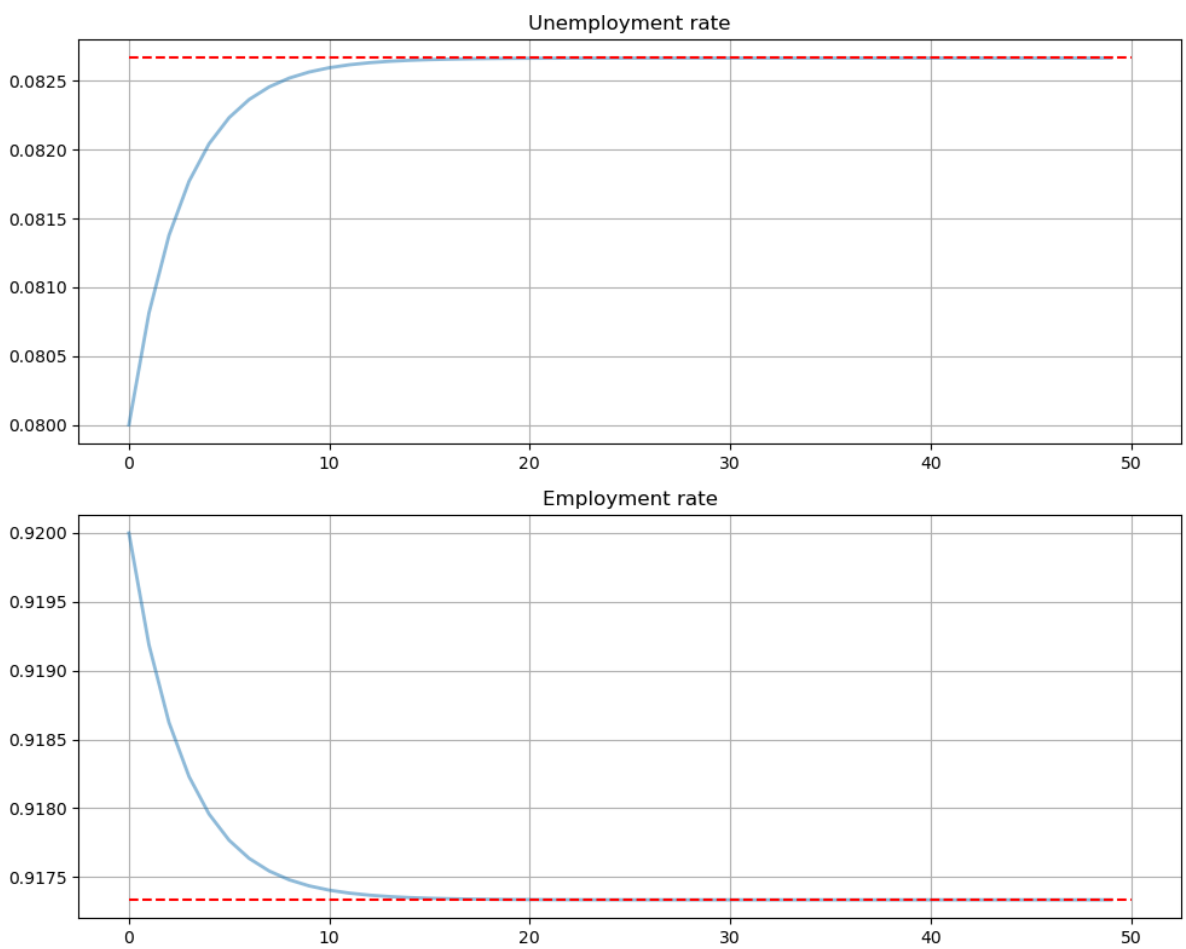
Let's look at the convergence of the unemployment and employment rate to steady state levels (dashed red line)

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

(continues on next page)

(continued from previous page)

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

xbar = lm.rate_steady_state()

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

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

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

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



23.4 Dynamics of an Individual Worker

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

The worker can be in one of two states:

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

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

The associated transition matrix is then

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

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

As usual, we regard it as a row vector.

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

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

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

The unique stationary distribution satisfies

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

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

23.4.1 Ergodicity

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

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

Let

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

and

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

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

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

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

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

with probability one.

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

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

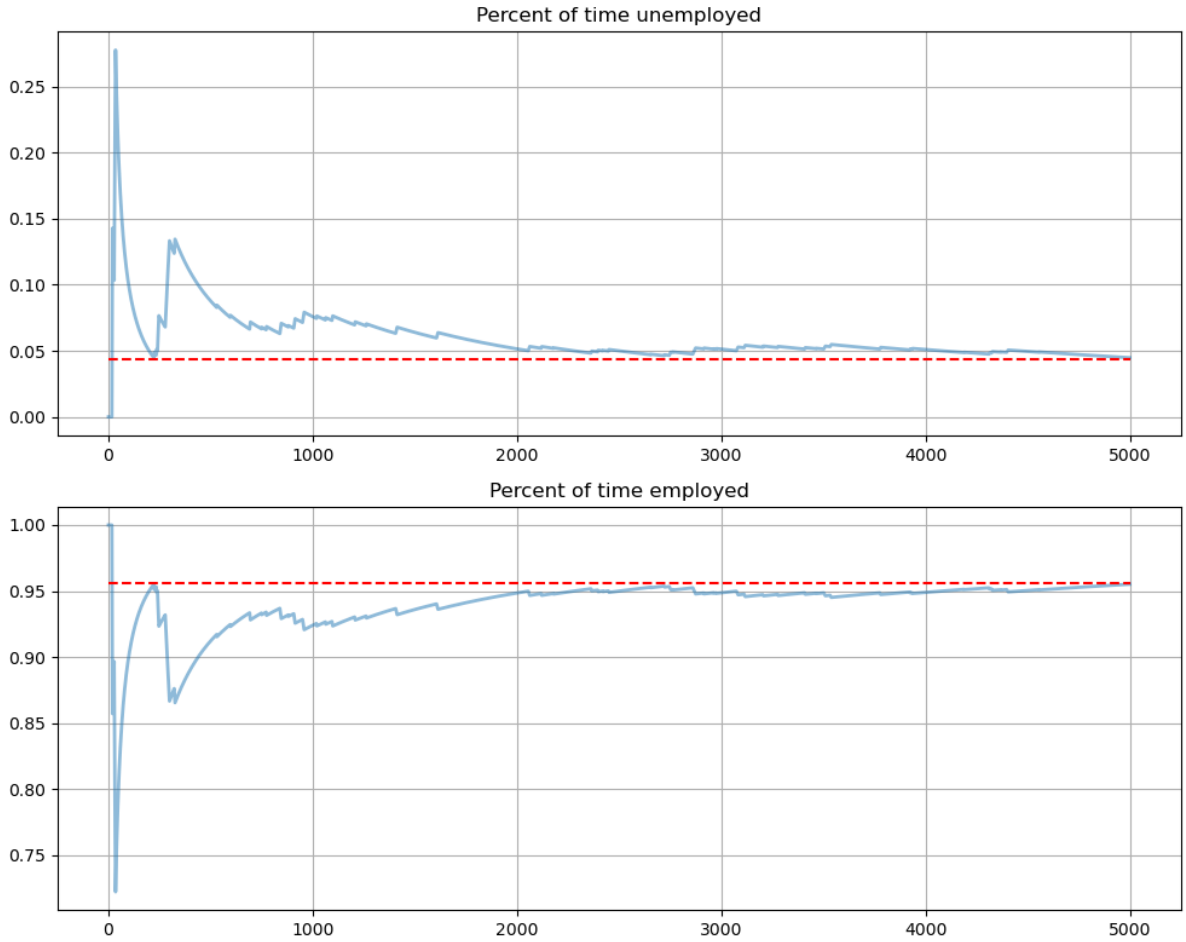
23.4.2 Convergence Rate

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

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

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

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

α, λ = lm.α, lm.λ

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

mc = MarkovChain(P)

xbar = lm.rate_steady_state()

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

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

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

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



The stationary probabilities are given by the dashed red line.

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

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

23.5 Endogenous Job Finding Rate

We now make the hiring rate endogenous.

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

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

23.5.1 Reservation Wage

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

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

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

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

23.5.2 Linking the McCall Search Model to the Lake Model

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

The exogenous probability of leaving employment remains α.

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

This is now

$$\lambda = \gamma \mathbb{P}\{w_t \geq \bar{w}\} = \gamma \sum_{w' \geq \bar{w}} p(w') \tag{23.1}$$

23.5.3 Fiscal Policy

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

We assume that the government sets unemployment compensation c.

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

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

$$\tau = uc$$

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

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

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

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

This determines λ via (23.1) evaluated at post tax wages, which in turn determines a steady state unemployment rate $u(c, \tau)$.

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

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

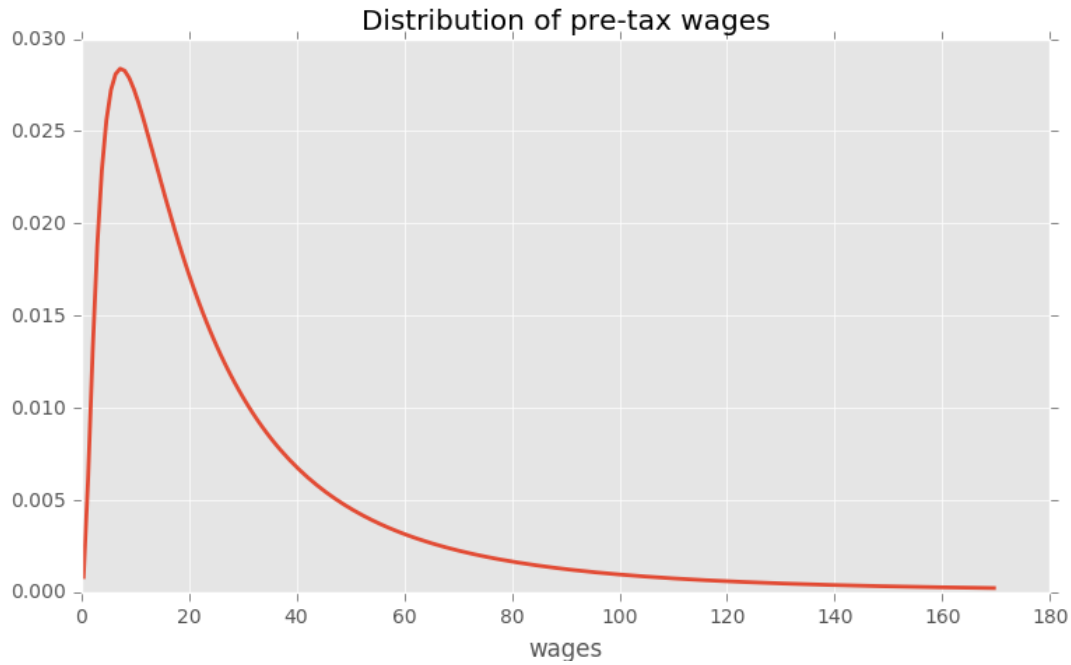
To evaluate alternative government tax-unemployment compensation pairs, we require a welfare criterion.

We use a steady state welfare criterion

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

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

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



We take a period to be a month.

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

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

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

- $\alpha = 0.013$

23.5.4 Fiscal Policy Code

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

The first piece of code implements value function iteration

```
# A default utility function

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

(continues on next page)

(continued from previous page)

```
        return -10e6


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

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

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

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

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

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

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

    return U_new


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

    Parameters
```

(continues on next page)

(continued from previous page)

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

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

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

    return V, U
```

```
/tmp/ipykernel_6848/1748977302.py:3: NumbaDeprecationWarning: The 'nopython'␣
↪keyword argument was not supplied to the 'numba.jit' decorator. The implicit␣
↪default value for this argument is currently False, but it will be changed to␣
↪True in Numba 0.59.0. See https://numba.readthedocs.io/en/stable/reference/
↪deprecation.html#deprecation-of-object-mode-fall-back-behaviour-when-using-jit␣
↪for details.
  @jit
/tmp/ipykernel_6848/1748977302.py:41: NumbaDeprecationWarning: The 'nopython'␣
↪keyword argument was not supplied to the 'numba.jit' decorator. The implicit␣
↪default value for this argument is currently False, but it will be changed to␣
↪True in Numba 0.59.0. See https://numba.readthedocs.io/en/stable/reference/
↪deprecation.html#deprecation-of-object-mode-fall-back-behaviour-when-using-jit␣
↪for details.
  @jit
```

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

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

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

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

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

(continues on next page)

(continued from previous page)

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

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

    """

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

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

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

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

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

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


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

    """

    mcm = McCallModel(α=α_q,
                      β=β,
                      γ=γ,
```

(continues on next page)

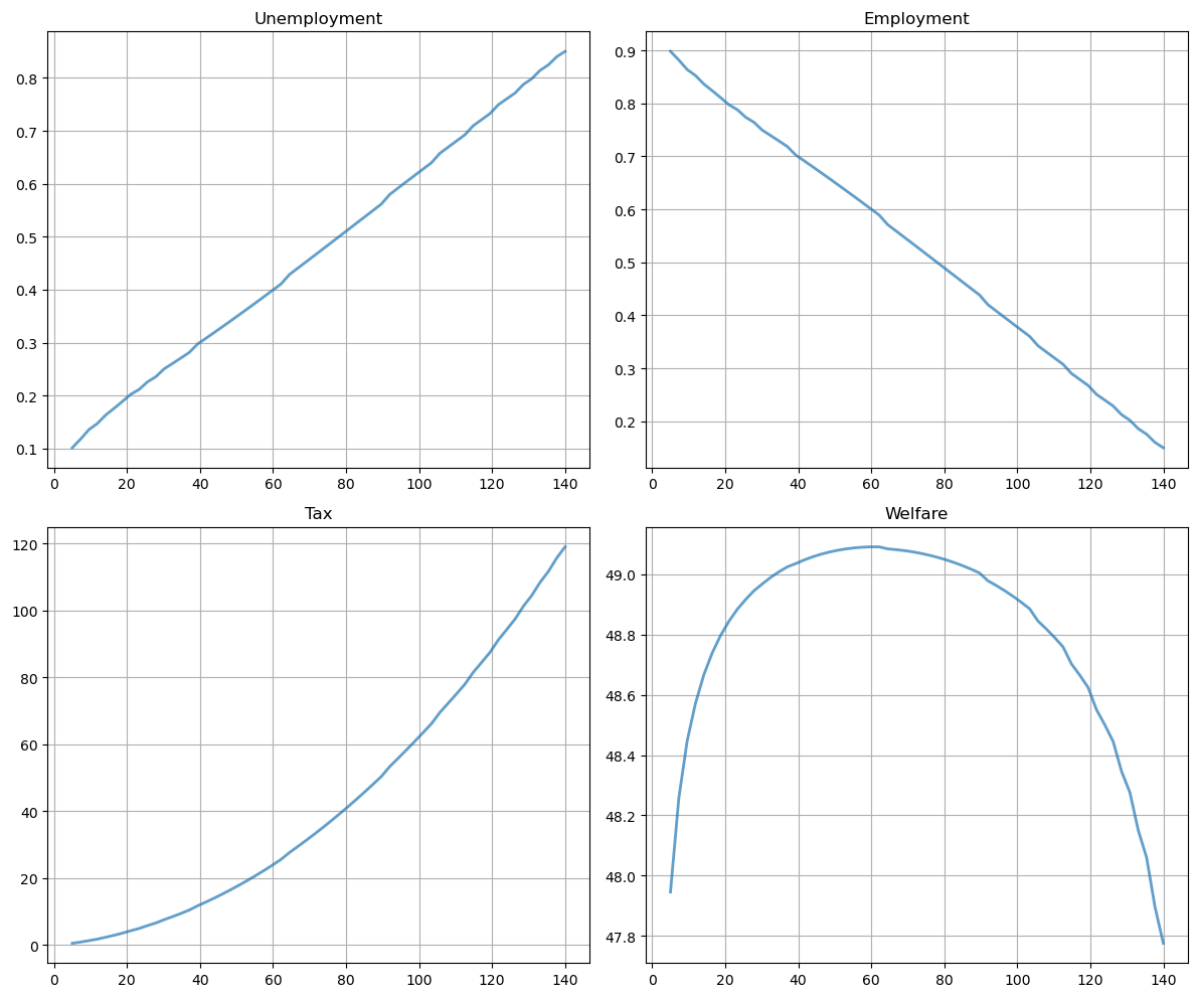
(continued from previous page)

```
                    c=c-τ,          # Post tax compensation
                    σ=σ,
                    w_vec=w_vec-τ,  # Post tax wages
                    p_vec=p_vec)

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

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

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

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

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

    return e, u, welfare


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

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

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


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

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

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

(continues on next page)

(continued from previous page)

```
    unempl_vec.append(u_rate)
    empl_vec.append(e_rate)
    welfare_vec.append(welfare)

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

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

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

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



Welfare first increases and then decreases as unemployment benefits rise.

The level that maximizes steady state welfare is approximately 62.

23.6 Exercises

Exercise 23.6.1

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

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

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

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

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

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

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

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

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

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

Solution to Exercise 23.6.1

Here is one solution

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

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

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

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

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

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

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

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

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

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

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

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


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

        Returns
```

(continues on next page)

(continued from previous page)

```
        --------
        xbar : steady state vector of employment and unemployment rates
        """
        x = np.array([self.A_hat[0, 1], self.A_hat[1, 0]])
        x /= x.sum()
        return x

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

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

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

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

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

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

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

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

Exercise 23.6.2

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

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

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

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

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

Plot the transition dynamics for the rates.

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

What is the new steady state level of employment?

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

Solution to Exercise 23.6.2

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

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

```
Initial Steady State: [0.08266627 0.91733373]
```

Initialize the simulation values

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

New legislation changes λ to 0.2

```
lm.λ = 0.2

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

```
New Steady State: [0.11309295 0.88690705]
```

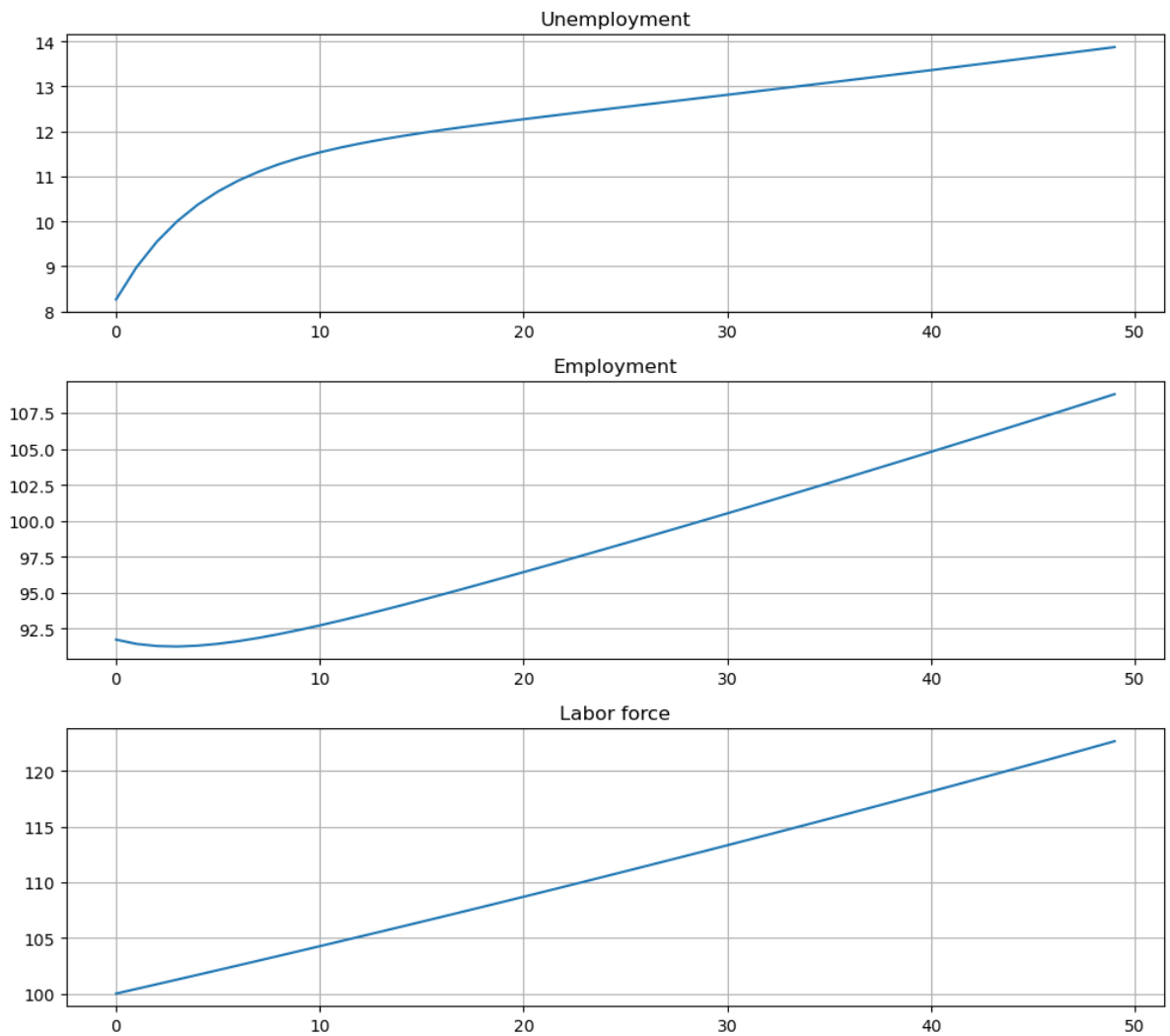
Now plot stocks

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

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

(continues on next page)

(continued from previous page)

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

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

for ax in axes:
    ax.grid()

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



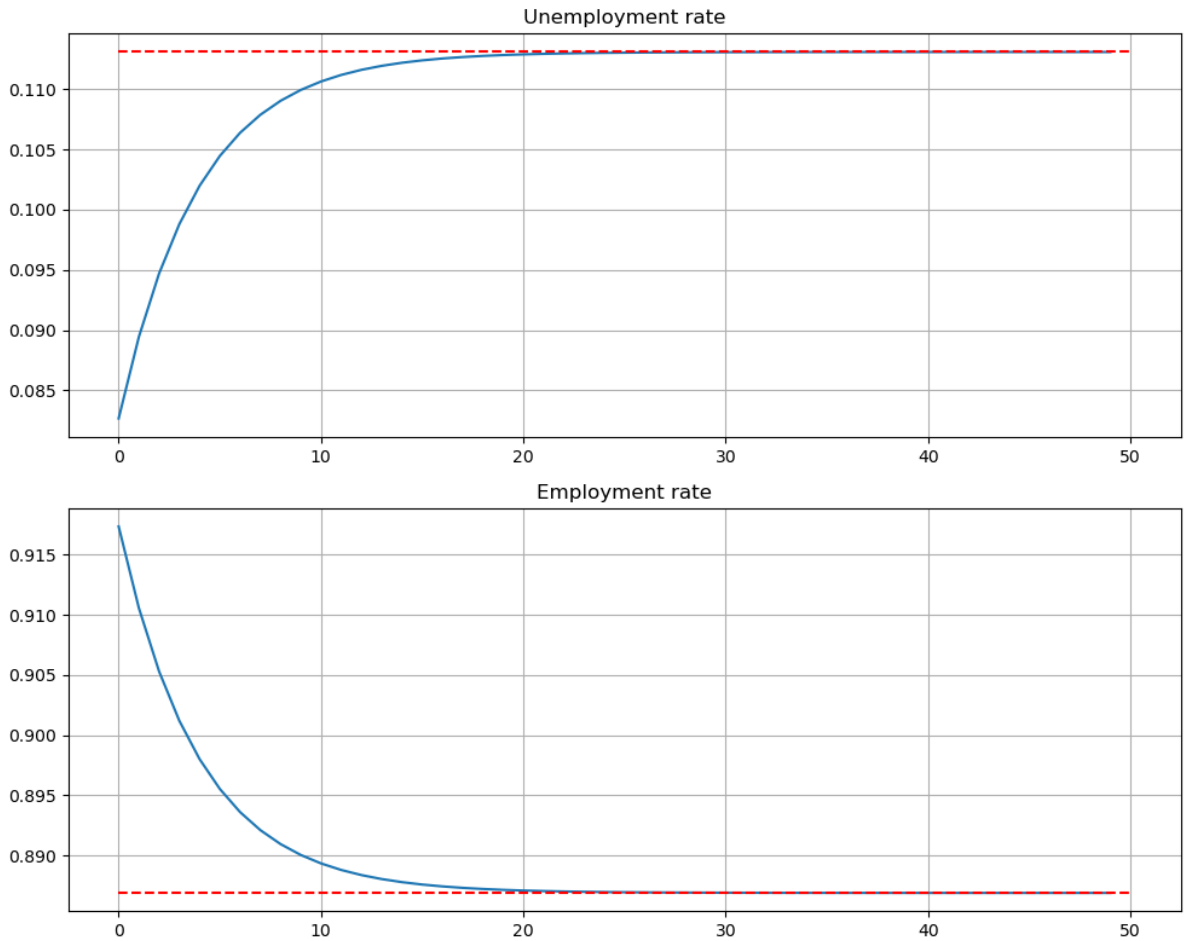
And how the rates evolve

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

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

for i, title in enumerate(titles):
```

(continues on next page)

(continued from previous page)

```
    axes[i].plot(x_path[:, i])
    axes[i].hlines(xbar[i], 0, T, 'r', '--')
    axes[i].set_title(title)
    axes[i].grid()

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



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

Exercise 23.6.3

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

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

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

Plot the transition dynamics for the rates.

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

Solution to Exercise 23.6.3

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

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

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

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

Here are the other parameters:

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

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

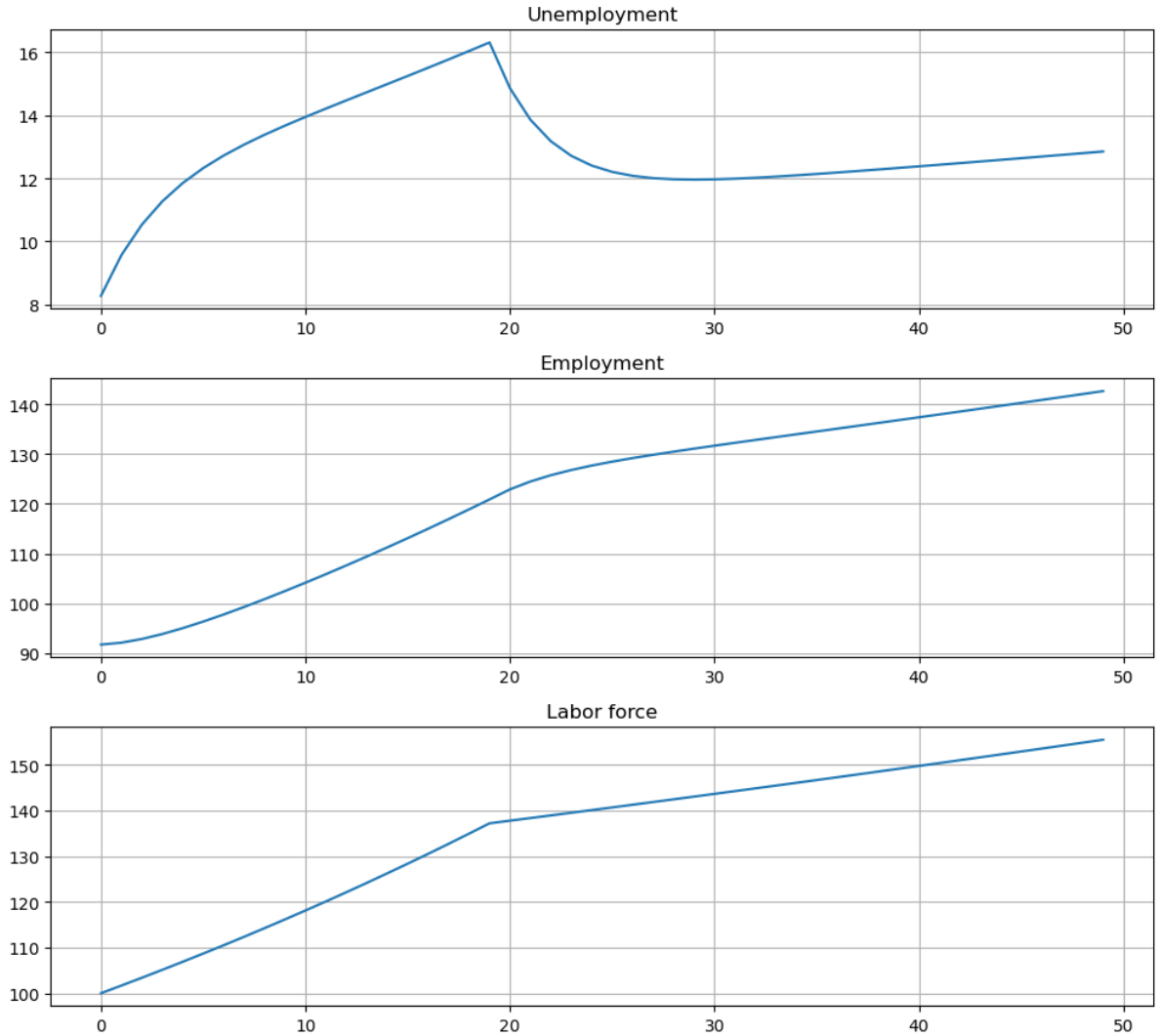
```
lm.b = b_hat
# Simulate stocks
X_path1 = np.vstack(tuple(lm.simulate_stock_path(x0 * N0, T_hat)))
# Simulate rates
x_path1 = np.vstack(tuple(lm.simulate_rate_path(x0, T_hat)))
```

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

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

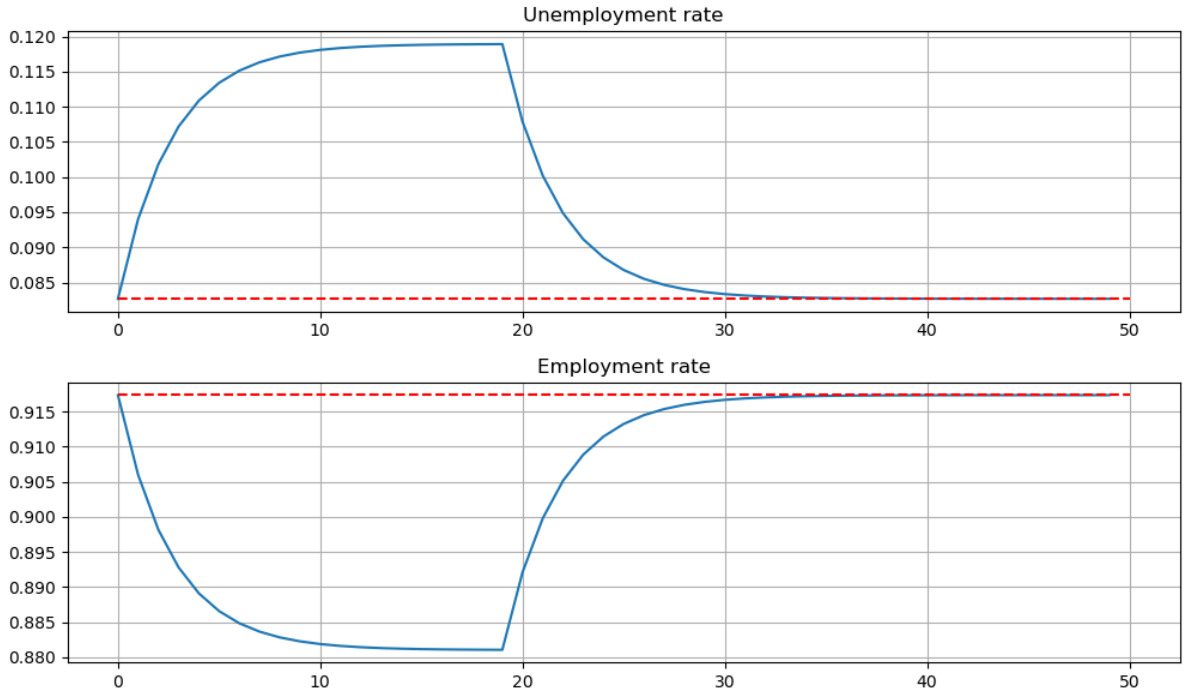
Finally, we combine these two paths and plot

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

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

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

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

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

for ax in axes:
    ax.grid()

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



And the rates

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

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

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

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


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

Part IV

Optimal Savings

CHAPTER

TWENTYFOUR

CAKE EATING I: INTRODUCTION TO OPTIMAL SAVING

Contents

- *Cake Eating I: Introduction to Optimal Saving*
  - *Overview*
  - *The Model*
  - *The Value Function*
  - *The Optimal Policy*
  - *The Euler Equation*
  - *Exercises*

24.1 Overview

In this lecture we introduce a simple "cake eating" problem.

The intertemporal problem is: how much to enjoy today and how much to leave for the future?

Although the topic sounds trivial, this kind of trade-off between current and future utility is at the heart of many savings and consumption problems.

Once we master the ideas in this simple environment, we will apply them to progressively more challenging—and useful—problems.

The main tool we will use to solve the cake eating problem is dynamic programming.

Readers might find it helpful to review the following lectures before reading this one:

- The shortest paths lecture
- The *basic McCall model*
- The *McCall model with separation*
- The *McCall model with separation and a continuous wage distribution*

In what follows, we require the following imports:

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

24.2 The Model

We consider an infinite time horizon $t = 0, 1, 2, 3..$

At $t = 0$ the agent is given a complete cake with size $\bar{x}$.

Let x_t denote the size of the cake at the beginning of each period, so that, in particular, $x_0 = \bar{x}$.

We choose how much of the cake to eat in any given period t.

After choosing to consume c_t of the cake in period t there is

$$x_{t+1} = x_t - c_t$$

left in period $t + 1$.

Consuming quantity c of the cake gives current utility $u(c)$.

We adopt the CRRA utility function

$$u(c) = \frac{c^{1-\gamma}}{1-\gamma} \qquad (\gamma > 0, \, \gamma \neq 1) \tag{24.1}$$

In Python this is

```
def u(c, γ):

    return c**(1 - γ) / (1 - γ)
```

Future cake consumption utility is discounted according to $\beta \in (0, 1)$.

In particular, consumption of c units t periods hence has present value $\beta^t u(c)$

The agent's problem can be written as

$$\max_{\{c_t\}} \sum_{t=0}^{\infty} \beta^t u(c_t) \tag{24.2}$$

subject to

$$x_{t+1} = x_t - c_t \quad \text{and} \quad 0 \leq c_t \leq x_t \tag{24.3}$$

for all t.

A consumption path $\{c_t\}$ satisfying (24.3) where $x_0 = \bar{x}$ is called **feasible**.

In this problem, the following terminology is standard:

- x_t is called the **state variable**
- c_t is called the **control variable** or the **action**
- β and γ are **parameters**

24.2.1 Trade-Off

The key trade-off in the cake-eating problem is this:

- Delaying consumption is costly because of the discount factor.
- But delaying some consumption is also attractive because u is concave.

The concavity of u implies that the consumer gains value from *consumption smoothing*, which means spreading consumption out over time.

This is because concavity implies diminishing marginal utility—a progressively smaller gain in utility for each additional spoonful of cake consumed within one period.

24.2.2 Intuition

The reasoning given above suggests that the discount factor β and the curvature parameter γ will play a key role in determining the rate of consumption.

Here's an educated guess as to what impact these parameters will have.

First, higher β implies less discounting, and hence the agent is more patient, which should reduce the rate of consumption.

Second, higher γ implies that marginal utility $u'(c) = c^{-\gamma}$ falls faster with c.

This suggests more smoothing, and hence a lower rate of consumption.

In summary, we expect the rate of consumption to be *decreasing in both parameters*.

Let's see if this is true.

24.3 The Value Function

The first step of our dynamic programming treatment is to obtain the Bellman equation.

The next step is to use it to calculate the solution.

24.3.1 The Bellman Equation

To this end, we let $v(x)$ be maximum lifetime utility attainable from the current time when x units of cake are left.

That is,

$$v(x) = \max \sum_{t=0}^{\infty} \beta^t u(c_t) \tag{24.4}$$

where the maximization is over all paths $\{c_t\}$ that are feasible from $x_0 = x$.

At this point, we do not have an expression for v, but we can still make inferences about it.

For example, as was the case with the *McCall model*, the value function will satisfy a version of the *Bellman equation*.

In the present case, this equation states that v satisfies

$$v(x) = \max_{0 \leq c \leq x} \{u(c) + \beta v(x - c)\} \quad \text{for any given } x \geq 0. \tag{24.5}$$

The intuition here is essentially the same it was for the McCall model.

Choosing c optimally means trading off current vs future rewards.

Current rewards from choice c are just $u(c)$.

Future rewards given current cake size x, measured from next period and assuming optimal behavior, are $v(x-c)$.

These are the two terms on the right hand side of (24.5), after suitable discounting.

If c is chosen optimally using this trade off strategy, then we obtain maximal lifetime rewards from our current state x.

Hence, $v(x)$ equals the right hand side of (24.5), as claimed.

24.3.2 An Analytical Solution

It has been shown that, with u as the CRRA utility function in (24.1), the function

$$v^*(x_t) = \left(1 - \beta^{1/\gamma}\right)^{-\gamma} u(x_t) \tag{24.6}$$

solves the Bellman equation and hence is equal to the value function.

You are asked to confirm that this is true in the exercises below.

The solution (24.6) depends heavily on the CRRA utility function.

In fact, if we move away from CRRA utility, usually there is no analytical solution at all.

In other words, beyond CRRA utility, we know that the value function still satisfies the Bellman equation, but we do not have a way of writing it explicitly, as a function of the state variable and the parameters.

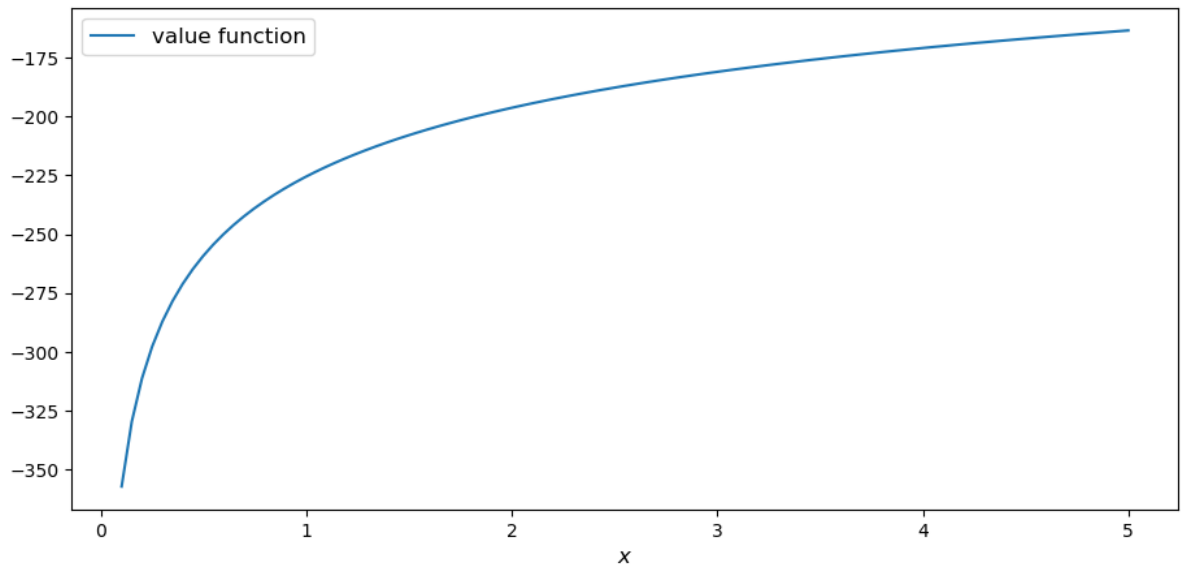
We will deal with that situation numerically when the time comes.

Here is a Python representation of the value function:

```
def v_star(x, β, γ):

    return (1 - β**(1 / γ))**(-γ) * u(x, γ)
```

And here's a figure showing the function for fixed parameters:

```
β, γ = 0.95, 1.2
x_grid = np.linspace(0.1, 5, 100)

fig, ax = plt.subplots()

ax.plot(x_grid, v_star(x_grid, β, γ), label='value function')

ax.set_xlabel('$x$', fontsize=12)
ax.legend(fontsize=12)

plt.show()
```



24.4 The Optimal Policy

Now that we have the value function, it is straightforward to calculate the optimal action at each state.

We should choose consumption to maximize the right hand side of the Bellman equation (24.5).

$$c^* = \arg\max_c \{u(c) + \beta v(x - c)\}$$

We can think of this optimal choice as a function of the state x, in which case we call it the **optimal policy**.

We denote the optimal policy by σ^*, so that

$$\sigma^*(x) := \arg\max_c \{u(c) + \beta v(x - c)\} \quad \text{for all } x$$

If we plug the analytical expression (24.6) for the value function into the right hand side and compute the optimum, we find that

$$\sigma^*(x) = \left(1 - \beta^{1/\gamma}\right) x \tag{24.7}$$

Now let's recall our intuition on the impact of parameters.

We guessed that the consumption rate would be decreasing in both parameters.

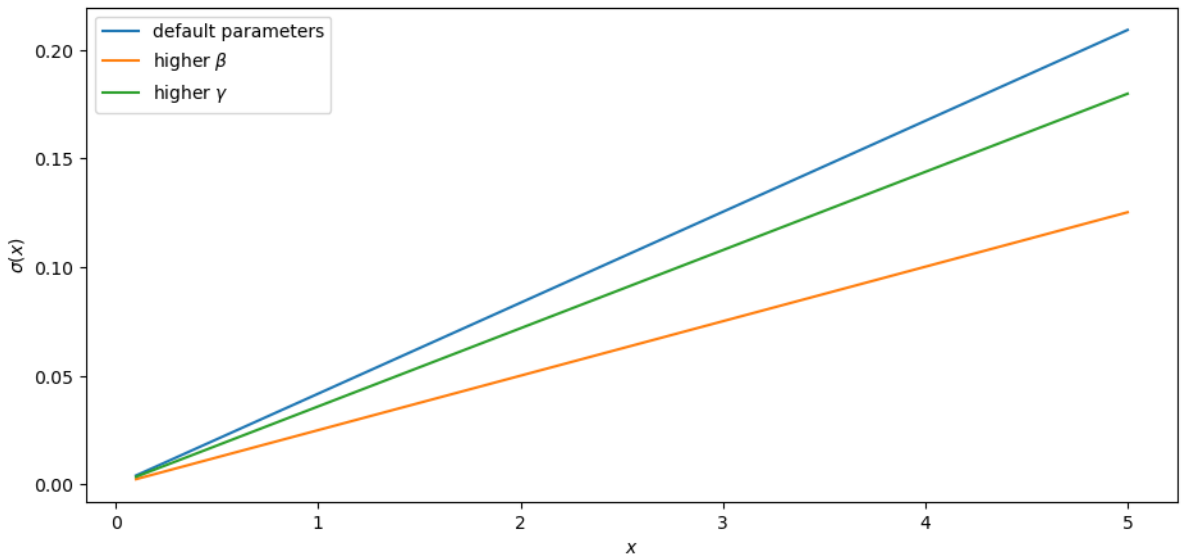
This is in fact the case, as can be seen from (24.7).

Here's some plots that illustrate.

```
def c_star(x, β, γ):

    return (1 - β ** (1/γ)) * x
```

Continuing with the values for β and γ used above, the plot is

```
fig, ax = plt.subplots()
ax.plot(x_grid, c_star(x_grid, β, γ), label='default parameters')
ax.plot(x_grid, c_star(x_grid, β + 0.02, γ), label=r'higher $\beta$')
ax.plot(x_grid, c_star(x_grid, β, γ + 0.2), label=r'higher $\gamma$')
ax.set_ylabel(r'$\sigma(x)$')
ax.set_xlabel('$x$')
ax.legend()

plt.show()
```



24.5 The Euler Equation

In the discussion above we have provided a complete solution to the cake eating problem in the case of CRRA utility.

There is in fact another way to solve for the optimal policy, based on the so-called **Euler equation**.

Although we already have a complete solution, now is a good time to study the Euler equation.

This is because, for more difficult problems, this equation provides key insights that are hard to obtain by other methods.

24.5.1 Statement and Implications

The Euler equation for the present problem can be stated as

$$u'(c_t^*) = \beta u'(c_{t+1}^*) \tag{24.8}$$

This is necessary condition for the optimal path.

It says that, along the optimal path, marginal rewards are equalized across time, after appropriate discounting.

This makes sense: optimality is obtained by smoothing consumption up to the point where no marginal gains remain.

We can also state the Euler equation in terms of the policy function.

A **feasible consumption policy** is a map $x \mapsto \sigma(x)$ satisfying $0 \leq \sigma(x) \leq x$.

The last restriction says that we cannot consume more than the remaining quantity of cake.

A feasible consumption policy σ is said to **satisfy the Euler equation** if, for all $x > 0$,

$$u'(\sigma(x)) = \beta u'(\sigma(x - \sigma(x))) \tag{24.9}$$

Evidently (24.9) is just the policy equivalent of (24.8).

It turns out that a feasible policy is optimal if and only if it satisfies the Euler equation.

In the exercises, you are asked to verify that the optimal policy (24.7) does indeed satisfy this functional equation.

Note: A **functional equation** is an equation where the unknown object is a function.

For a proof of sufficiency of the Euler equation in a very general setting, see proposition 2.2 of [MST20].

The following arguments focus on necessity, explaining why an optimal path or policy should satisfy the Euler equation.

24.5.2 Derivation I: A Perturbation Approach

Let's write c as a shorthand for consumption path $\{c_t\}_{t=0}^{\infty}$.

The overall cake-eating maximization problem can be written as

$$\max_{c \in F} U(c) \quad \text{where } U(c) := \sum_{t=0}^{\infty} \beta^t u(c_t)$$

and F is the set of feasible consumption paths.

We know that differentiable functions have a zero gradient at a maximizer.

So the optimal path $c^* := \{c_t^*\}_{t=0}^{\infty}$ must satisfy $U'(c^*) = 0$.

Note: If you want to know exactly how the derivative $U'(c^*)$ is defined, given that the argument c^* is a vector of infinite length, you can start by learning about Gateaux derivatives. However, such knowledge is not assumed in what follows.

In other words, the rate of change in U must be zero for any infinitesimally small (and feasible) perturbation away from the optimal path.

So consider a feasible perturbation that reduces consumption at time t to $c_t^* - h$ and increases it in the next period to $c_{t+1}^* + h$.

Consumption does not change in any other period.

We call this perturbed path c^h.

By the preceding argument about zero gradients, we have

$$\lim_{h \to 0} \frac{U(c^h) - U(c^*)}{h} = U'(c^*) = 0$$

Recalling that consumption only changes at t and $t+1$, this becomes

$$\lim_{h \to 0} \frac{\beta^t u(c_t^* - h) + \beta^{t+1} u(c_{t+1}^* + h) - \beta^t u(c_t^*) - \beta^{t+1} u(c_{t+1}^*)}{h} = 0$$

After rearranging, the same expression can be written as

$$\lim_{h \to 0} \frac{u(c_t^* - h) - u(c_t^*)}{h} + \beta \lim_{h \to 0} \frac{u(c_{t+1}^* + h) - u(c_{t+1}^*)}{h} = 0$$

or, taking the limit,

$$-u'(c_t^*) + \beta u'(c_{t+1}^*) = 0$$

This is just the Euler equation.

24.5.3 Derivation II: Using the Bellman Equation

Another way to derive the Euler equation is to use the Bellman equation (24.5).

Taking the derivative on the right hand side of the Bellman equation with respect to c and setting it to zero, we get

$$u'(c) = \beta v'(x - c) \tag{24.10}$$

To obtain $v'(x - c)$, we set $g(c, x) = u(c) + \beta v(x - c)$, so that, at the optimal choice of consumption,

$$v(x) = g(c, x) \tag{24.11}$$

Differentiating both sides while acknowledging that the maximizing consumption will depend on x, we get

$$v'(x) = \frac{\partial}{\partial c} g(c, x) \frac{\partial c}{\partial x} + \frac{\partial}{\partial x} g(c, x)$$

When $g(c, x)$ is maximized at c, we have $\frac{\partial}{\partial c} g(c, x) = 0$.

Hence the derivative simplifies to

$$v'(x) = \frac{\partial g(c, x)}{\partial x} = \frac{\partial}{\partial x} \beta v(x - c) = \beta v'(x - c) \tag{24.12}$$

(This argument is an example of the Envelope Theorem.)

But now an application of (24.10) gives

$$u'(c) = v'(x) \tag{24.13}$$

Thus, the derivative of the value function is equal to marginal utility.

Combining this fact with (24.12) recovers the Euler equation.

24.6 Exercises

Exercise 24.6.1

How does one obtain the expressions for the value function and optimal policy given in (24.6) and (24.7) respectively?

The first step is to make a guess of the functional form for the consumption policy.

So suppose that we do not know the solutions and start with a guess that the optimal policy is linear.

In other words, we conjecture that there exists a positive θ such that setting $c_t^* = \theta x_t$ for all t produces an optimal path.

Starting from this conjecture, try to obtain the solutions (24.6) and (24.7).

In doing so, you will need to use the definition of the value function and the Bellman equation.

Solution to Exercise 24.6.1

We start with the conjecture $c_t^* = \theta x_t$, which leads to a path for the state variable (cake size) given by

$$x_{t+1} = x_t(1-\theta)$$

Then $x_t = x_0(1-\theta)^t$ and hence

$$\begin{aligned} v(x_0) &= \sum_{t=0}^{\infty} \beta^t u(\theta x_t) \\ &= \sum_{t=0}^{\infty} \beta^t u(\theta x_0 (1-\theta)^t) \\ &= \sum_{t=0}^{\infty} \theta^{1-\gamma} \beta^t (1-\theta)^{t(1-\gamma)} u(x_0) \\ &= \frac{\theta^{1-\gamma}}{1-\beta(1-\theta)^{1-\gamma}} u(x_0) \end{aligned}$$

From the Bellman equation, then,

$$\begin{aligned} v(x) &= \max_{0 \leq c \leq x} \left\{ u(c) + \beta \frac{\theta^{1-\gamma}}{1-\beta(1-\theta)^{1-\gamma}} \cdot u(x-c) \right\} \\ &= \max_{0 \leq c \leq x} \left\{ \frac{c^{1-\gamma}}{1-\gamma} + \beta \frac{\theta^{1-\gamma}}{1-\beta(1-\theta)^{1-\gamma}} \cdot \frac{(x-c)^{1-\gamma}}{1-\gamma} \right\} \end{aligned}$$

From the first order condition, we obtain

$$c^{-\gamma} + \beta \frac{\theta^{1-\gamma}}{1-\beta(1-\theta)^{1-\gamma}} \cdot (x-c)^{-\gamma}(-1) = 0$$

or

$$c^{-\gamma} = \beta \frac{\theta^{1-\gamma}}{1-\beta(1-\theta)^{1-\gamma}} \cdot (x-c)^{-\gamma}$$

With $c = \theta x$ we get

$$(\theta x)^{-\gamma} = \beta \frac{\theta^{1-\gamma}}{1-\beta(1-\theta)^{1-\gamma}} \cdot (x(1-\theta))^{-\gamma}$$

Some rearrangement produces

$$\theta = 1 - \beta^{\frac{1}{\gamma}}$$

This confirms our earlier expression for the optimal policy:

$$c_t^* = \left(1 - \beta^{\frac{1}{\gamma}}\right) x_t$$

Substituting θ into the value function above gives

$$v^*(x_t) = \frac{\left(1-\beta^{\frac{1}{\gamma}}\right)^{1-\gamma}}{1-\beta\left(\beta^{\frac{1-\gamma}{\gamma}}\right)} u(x_t)$$

Rearranging gives

$$v^*(x_t) = \left(1-\beta^{\frac{1}{\gamma}}\right)^{-\gamma} u(x_t)$$

Our claims are now verified.

CHAPTER

TWENTYFIVE

CAKE EATING II: NUMERICAL METHODS

Contents

- *Cake Eating II: Numerical Methods*
  - *Overview*
  - *Reviewing the Model*
  - *Value Function Iteration*
  - *Time Iteration*
  - *Exercises*

In addition to what's in Anaconda, this lecture will require the following library:

```
!pip install interpolation
```

25.1 Overview

In this lecture we continue the study of *the cake eating problem*.

The aim of this lecture is to solve the problem using numerical methods.

At first this might appear unnecessary, since we already obtained the optimal policy analytically.

However, the cake eating problem is too simple to be useful without modifications, and once we start modifying the problem, numerical methods become essential.

Hence it makes sense to introduce numerical methods now, and test them on this simple problem.

Since we know the analytical solution, this will allow us to assess the accuracy of alternative numerical methods.

We will use the following imports:

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from interpolation import interp
from scipy.optimize import minimize_scalar, bisect
```

```
/home/runner/miniconda3/envs/quantecon/lib/python3.11/site-packages/numba/core/
↪decorators.py:262: NumbaDeprecationWarning: numba.generated_jit is deprecated.␣
↪Please see the documentation at: https://numba.readthedocs.io/en/stable/
↪reference/deprecation.html#deprecation-of-generated-jit for more information and␣
↪advice on a suitable replacement.
  warnings.warn(msg, NumbaDeprecationWarning)
```

25.2 Reviewing the Model

You might like to *review the details* before we start.

Recall in particular that the Bellman equation is

$$v(x) = \max_{0 \leq c \leq x} \{u(c) + \beta v(x - c)\} \quad \text{for all } x \geq 0. \tag{25.1}$$

where u is the CRRA utility function.

The analytical solutions for the value function and optimal policy were found to be as follows.

```
def c_star(x, β, γ):

    return (1 - β ** (1/γ)) * x


def v_star(x, β, γ):

    return (1 - β**(1 / γ))**(-γ) * (x**(1-γ) / (1-γ))
```

Our first aim is to obtain these analytical solutions numerically.

25.3 Value Function Iteration

The first approach we will take is **value function iteration**.

This is a form of **successive approximation**, and was discussed in our *lecture on job search*.

The basic idea is:

1. Take an arbitary intial guess of v.
2. Obtain an update w defined by

$$w(x) = \max_{0 \leq c \leq x} \{u(c) + \beta v(x - c)\}$$

3. Stop if w is approximately equal to v, otherwise set $v = w$ and go back to step 2.

Let's write this a bit more mathematically.

25.3.1 The Bellman Operator

We introduce the **Bellman operator** T that takes a function v as an argument and returns a new function Tv defined by

$$Tv(x) = \max_{0 \leq c \leq x} \{u(c) + \beta v(x - c)\}$$

From v we get Tv, and applying T to this yields $T^2 v := T(Tv)$ and so on.

This is called **iterating with the Bellman operator** from initial guess v.

As we discuss in more detail in later lectures, one can use Banach's contraction mapping theorem to prove that the sequence of functions $T^n v$ converges to the solution to the Bellman equation.

25.3.2 Fitted Value Function Iteration

Both consumption c and the state variable x are continuous.

This causes complications when it comes to numerical work.

For example, we need to store each function $T^n v$ in order to compute the next iterate $T^{n+1} v$.

But this means we have to store $T^n v(x)$ at infinitely many x, which is, in general, impossible.

To circumvent this issue we will use fitted value function iteration, as discussed previously in *one of the lectures* on job search.

The process looks like this:

1. Begin with an array of values $\{v_0, \dots, v_I\}$ representing the values of some initial function v on the grid points $\{x_0, \dots, x_I\}$.
2. Build a function $\hat{v}$ on the state space $\mathbb{R}_+$ by linear interpolation, based on these data points.
3. Obtain and record the value $T\hat{v}(x_i)$ on each grid point x_i by repeatedly solving the maximization problem in the Bellman equation.
4. Unless some stopping condition is satisfied, set $\{v_0, \dots, v_I\} = \{T\hat{v}(x_0), \dots, T\hat{v}(x_I)\}$ and go to step 2.

In step 2 we'll use continuous piecewise linear interpolation.

25.3.3 Implementation

The `maximize` function below is a small helper function that converts a SciPy minimization routine into a maximization routine.

```
def maximize(g, a, b, args):
    """
    Maximize the function g over the interval [a, b].

    We use the fact that the maximizer of g on any interval is
    also the minimizer of -g.  The tuple args collects any extra
    arguments to g.

    Returns the maximal value and the maximizer.
    """

    objective = lambda x: -g(x, *args)
    result = minimize_scalar(objective, bounds=(a, b), method='bounded')
```

(continues on next page)

(continued from previous page)

```
    maximizer, maximum = result.x, -result.fun
    return maximizer, maximum
```

We'll store the parameters β and γ in a class called `CakeEating`.

The same class will also provide a method called `state_action_value` that returns the value of a consumption choice given a particular state and guess of v.

```
class CakeEating:

    def __init__(self,
                 β=0.96,           # discount factor
                 γ=1.5,            # degree of relative risk aversion
                 x_grid_min=1e-3,  # exclude zero for numerical stability
                 x_grid_max=2.5,   # size of cake
                 x_grid_size=120):

        self.β, self.γ = β, γ

        # Set up grid
        self.x_grid = np.linspace(x_grid_min, x_grid_max, x_grid_size)

    # Utility function
    def u(self, c):

        γ = self.γ

        if γ == 1:
            return np.log(c)
        else:
            return (c ** (1 - γ)) / (1 - γ)

    # first derivative of utility function
    def u_prime(self, c):

        return c ** (-self.γ)

    def state_action_value(self, c, x, v_array):
        """
        Right hand side of the Bellman equation given x and c.
        """

        u, β = self.u, self.β
        v = lambda x: interp(self.x_grid, v_array, x)

        return u(c) + β * v(x - c)
```

We now define the Bellman operation:

```
def T(v, ce):
    """
    The Bellman operator.  Updates the guess of the value function.

    * ce is an instance of CakeEating
    * v is an array representing a guess of the value function
```

(continues on next page)

(continued from previous page)

```
    """
    v_new = np.empty_like(v)

    for i, x in enumerate(ce.x_grid):
        # Maximize RHS of Bellman equation at state x
        v_new[i] = maximize(ce.state_action_value, 1e-10, x, (x, v))[1]

    return v_new
```

After defining the Bellman operator, we are ready to solve the model.

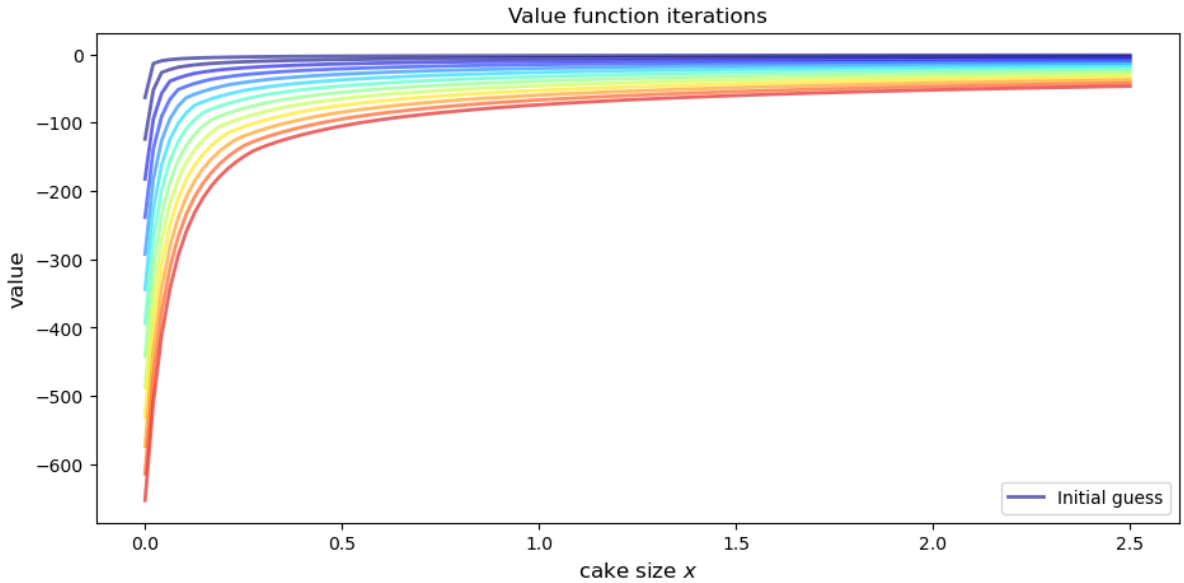
Let's start by creating a `CakeEating` instance using the default parameterization.

```
ce = CakeEating()
```

Now let's see the iteration of the value function in action.

We start from guess v given by $v(x) = u(x)$ for every x grid point.

```
x_grid = ce.x_grid
v = ce.u(x_grid)        # Initial guess
n = 12                  # Number of iterations

fig, ax = plt.subplots()

ax.plot(x_grid, v, color=plt.cm.jet(0),
        lw=2, alpha=0.6, label='Initial guess')

for i in range(n):
    v = T(v, ce)  # Apply the Bellman operator
    ax.plot(x_grid, v, color=plt.cm.jet(i / n), lw=2, alpha=0.6)

ax.legend()
ax.set_ylabel('value', fontsize=12)
ax.set_xlabel('cake size $x$', fontsize=12)
ax.set_title('Value function iterations')

plt.show()
```



To do this more systematically, we introduce a wrapper function called `compute_value_function` that iterates until some convergence conditions are satisfied.

```
def compute_value_function(ce,
                           tol=1e-4,
                           max_iter=1000,
                           verbose=True,
                           print_skip=25):

    # Set up loop
    v = np.zeros(len(ce.x_grid)) # Initial guess
    i = 0
    error = tol + 1

    while i < max_iter and error > tol:
        v_new = T(v, ce)

        error = np.max(np.abs(v - v_new))
        i += 1

        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")

        v = v_new

    if error > tol:
        print("Failed to converge!")
    elif verbose:
        print(f"\nConverged in {i} iterations.")

    return v_new
```

Now let's call it, noting that it takes a little while to run.

```
v = compute_value_function(ce)
```

```
Error at iteration 25 is 23.8003755134813.
```

```
Error at iteration 50 is 8.577577195046615.
```

```
Error at iteration 75 is 3.091330659691039.
```

```
Error at iteration 100 is 1.1141054204751981.
```

```
Error at iteration 125 is 0.4015199357729671.
```

```
Error at iteration 150 is 0.14470646660561215.
```

```
Error at iteration 175 is 0.052151735472762084.
```

```
Error at iteration 200 is 0.018795314242879613.
```

```
Error at iteration 225 is 0.006773769545588948.
```

```
Error at iteration 250 is 0.0024412443051460286.
```

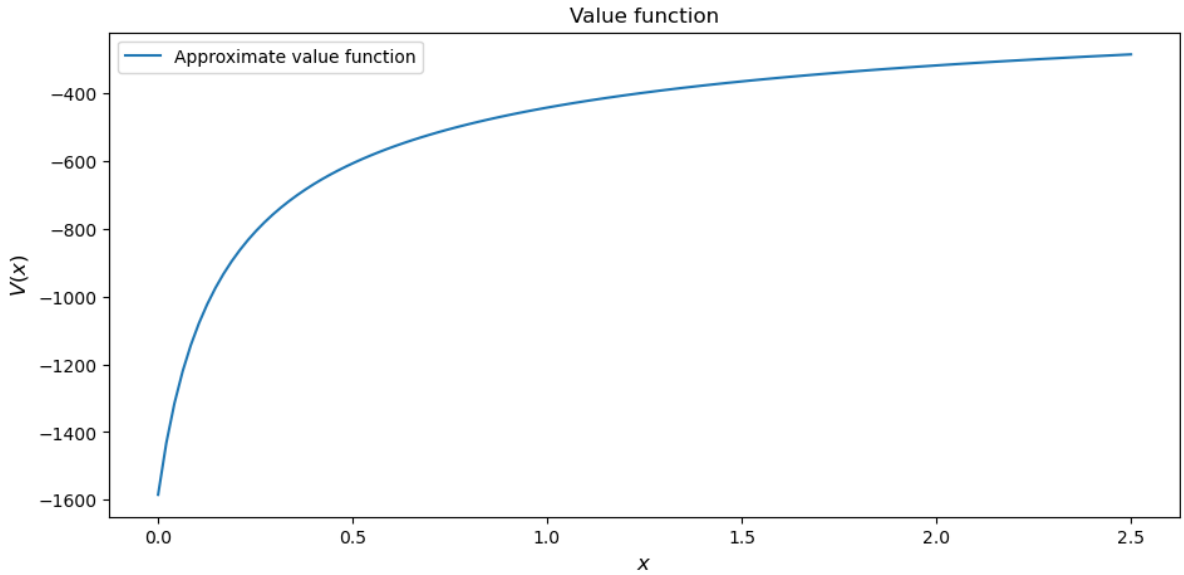
```
Error at iteration 275 is 0.000879816432870939.
```

```
Error at iteration 300 is 0.00031708295398402697.
```

```
Error at iteration 325 is 0.00011427565573285392.

Converged in 329 iterations.
```

Now we can plot and see what the converged value function looks like.

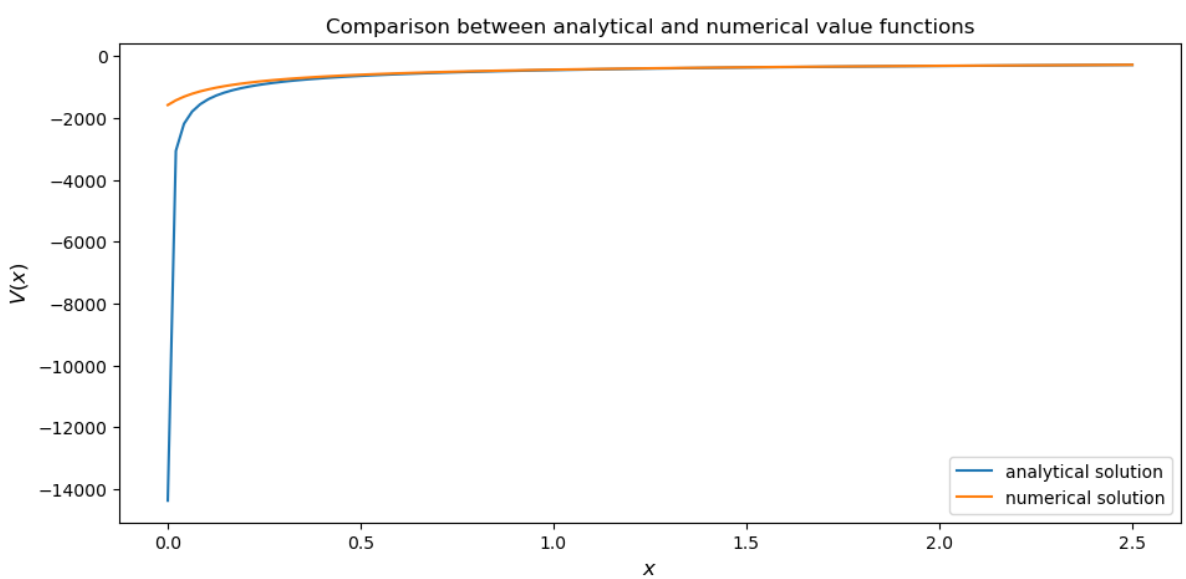
```
fig, ax = plt.subplots()

ax.plot(x_grid, v, label='Approximate value function')
ax.set_ylabel('$V(x)$', fontsize=12)
ax.set_xlabel('$x$', fontsize=12)
ax.set_title('Value function')
ax.legend()
plt.show()
```



Next let's compare it to the analytical solution.

```
v_analytical = v_star(ce.x_grid, ce.β, ce.γ)
```

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

ax.plot(x_grid, v_analytical, label='analytical solution')
ax.plot(x_grid, v, label='numerical solution')
ax.set_ylabel('$V(x)$', fontsize=12)
ax.set_xlabel('$x$', fontsize=12)
ax.legend()
ax.set_title('Comparison between analytical and numerical value functions')
plt.show()
```



The quality of approximation is reasonably good for large x, but less so near the lower boundary.

The reason is that the utility function and hence value function is very steep near the lower boundary, and hence hard to

approximate.

25.3.4 Policy Function

Let's see how this plays out in terms of computing the optimal policy.

In the *first lecture on cake eating*, the optimal consumption policy was shown to be

$$\sigma^*(x) = \left(1 - \beta^{1/\gamma}\right) x$$

Let's see if our numerical results lead to something similar.

Our numerical strategy will be to compute

$$\sigma(x) = \arg\max_{0 \leq c \leq x} \{u(c) + \beta v(x - c)\}$$

on a grid of x points and then interpolate.

For v we will use the approximation of the value function we obtained above.

Here's the function:

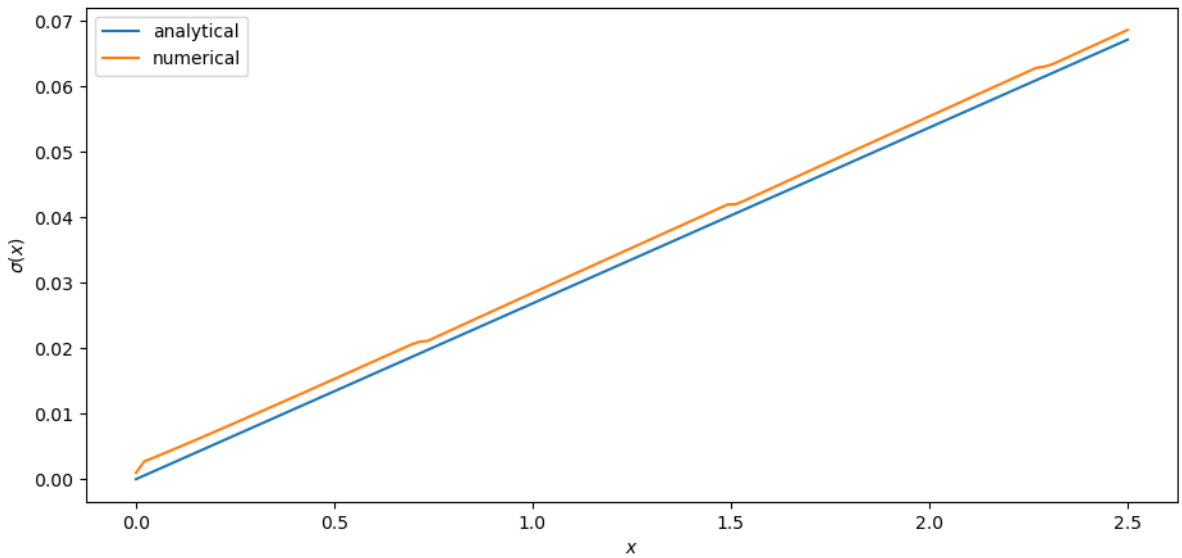
```
def σ(ce, v):
    """
    The optimal policy function. Given the value function,
    it finds optimal consumption in each state.

    * ce is an instance of CakeEating
    * v is a value function array

    """
    c = np.empty_like(v)

    for i in range(len(ce.x_grid)):
        x = ce.x_grid[i]
        # Maximize RHS of Bellman equation at state x
        c[i] = maximize(ce.state_action_value, 1e-10, x, (x, v))[0]

    return c
```

Now let's pass the approximate value function and compute optimal consumption:

```
c = σ(ce, v)
```

Let's plot this next to the true analytical solution

```
c_analytical = c_star(ce.x_grid, ce.β, ce.γ)

fig, ax = plt.subplots()

ax.plot(ce.x_grid, c_analytical, label='analytical')
ax.plot(ce.x_grid, c, label='numerical')
ax.set_ylabel(r'$\sigma(x)$')
ax.set_xlabel('$x$')
ax.legend()

plt.show()
```



The fit is reasonable but not perfect.

We can improve it by increasing the grid size or reducing the error tolerance in the value function iteration routine.

However, both changes will lead to a longer compute time.

Another possibility is to use an alternative algorithm, which offers the possibility of faster compute time and, at the same time, more accuracy.

We explore this next.

25.4 Time Iteration

Now let's look at a different strategy to compute the optimal policy.

Recall that the optimal policy satisfies the Euler equation

$$u'(\sigma(x)) = \beta u'(\sigma(x - \sigma(x))) \quad \text{for all } x > 0 \tag{25.2}$$

Computationally, we can start with any initial guess of σ_0 and now choose c to solve

$$u'(c) = \beta u'(\sigma_0(x - c))$$

Choosing c to satisfy this equation at all $x > 0$ produces a function of x.

Call this new function σ_1, treat it as the new guess and repeat.

This is called **time iteration**.

As with value function iteration, we can view the update step as action of an operator, this time denoted by K.

- In particular, $K\sigma$ is the policy updated from σ using the procedure just described.
- We will use this terminology in the exercises below.

The main advantage of time iteration relative to value function iteration is that it operates in policy space rather than value function space.

This is helpful because the policy function has less curvature, and hence is easier to approximate.

In the exercises you are asked to implement time iteration and compare it to value function iteration.

You should find that the method is faster and more accurate.

This is due to

1. the curvature issue mentioned just above and
2. the fact that we are using more information — in this case, the first order conditions.

25.5 Exercises

Exercise 25.5.1

Try the following modification of the problem.

Instead of the cake size changing according to $x_{t+1} = x_t - c_t$, let it change according to

$$x_{t+1} = (x_t - c_t)^\alpha$$

where α is a parameter satisfying $0 < \alpha < 1$.

(We will see this kind of update rule when we study optimal growth models.)

Make the required changes to value function iteration code and plot the value and policy functions.

Try to reuse as much code as possible.

Solution to Exercise 25.5.1

We need to create a class to hold our primitives and return the right hand side of the Bellman equation.

We will use inheritance to maximize code reuse.

```
class OptimalGrowth(CakeEating):
    """
    A subclass of CakeEating that adds the parameter α and overrides
    the state_action_value method.
    """

    def __init__(self,
                 β=0.96,           # discount factor
                 γ=1.5,            # degree of relative risk aversion
                 α=0.4,            # productivity parameter
                 x_grid_min=1e-3,  # exclude zero for numerical stability
                 x_grid_max=2.5,   # size of cake
                 x_grid_size=120):

        self.α = α
        CakeEating.__init__(self, β, γ, x_grid_min, x_grid_max, x_grid_size)

    def state_action_value(self, c, x, v_array):
        """
        Right hand side of the Bellman equation given x and c.
        """

        u, β, α = self.u, self.β, self.α
        v = lambda x: interp(self.x_grid, v_array, x)
```

(continues on next page)

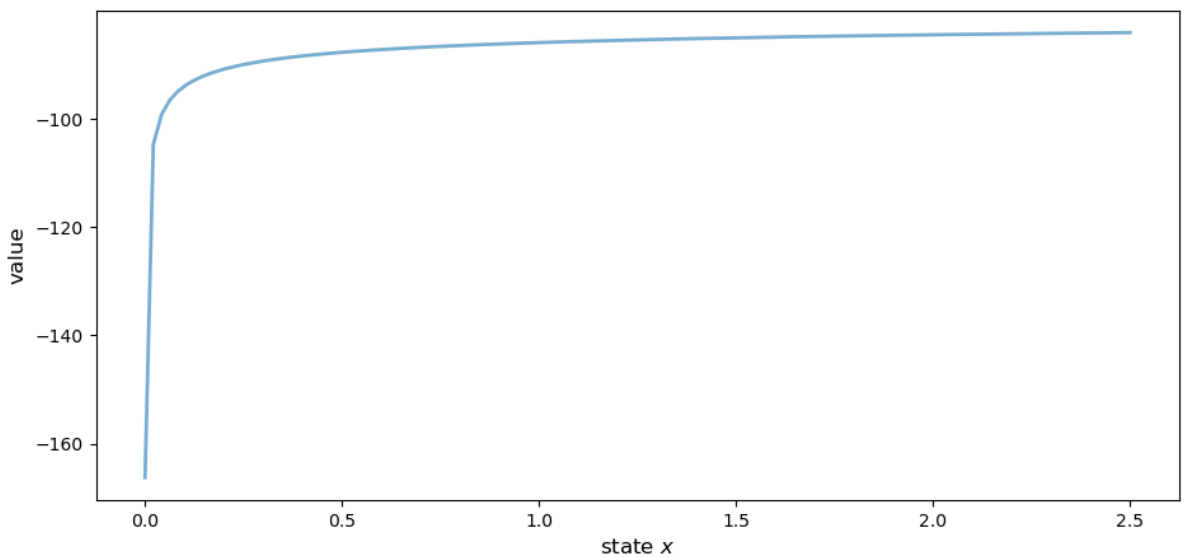
(continued from previous page)

```
        return u(c) + β * v((x - c)**α)
```

```
og = OptimalGrowth()
```

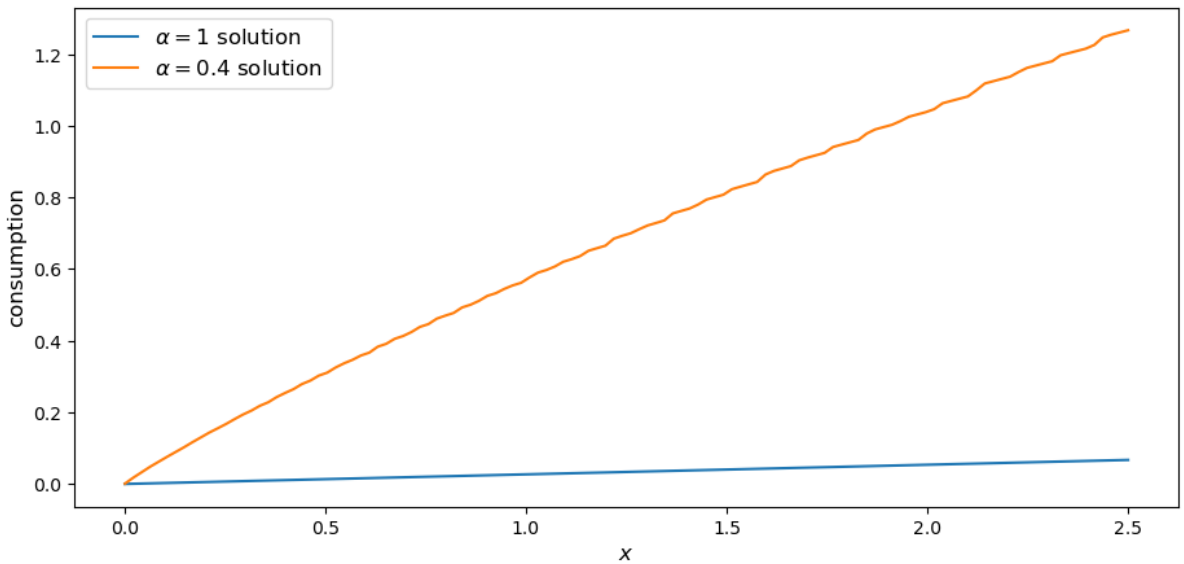
Here's the computed value function.

```
v = compute_value_function(og, verbose=False)

fig, ax = plt.subplots()

ax.plot(x_grid, v, lw=2, alpha=0.6)
ax.set_ylabel('value', fontsize=12)
ax.set_xlabel('state $x$', fontsize=12)

plt.show()
```



Here's the computed policy, combined with the solution we derived above for the standard cake eating case $\alpha = 1$.

```
c_new = σ(og, v)

fig, ax = plt.subplots()

ax.plot(ce.x_grid, c_analytical, label=r'$\alpha=1$ solution')
ax.plot(ce.x_grid, c_new, label=fr'$\alpha={og.α}$ solution')

ax.set_ylabel('consumption', fontsize=12)
ax.set_xlabel('$x$', fontsize=12)

ax.legend(fontsize=12)

plt.show()
```



Consumption is higher when $\alpha < 1$ because, at least for large x, the return to savings is lower.

Exercise 25.5.2

Implement time iteration, returning to the original case (i.e., dropping the modification in the exercise above).

Solution to Exercise 25.5.2

Here's one way to implement time iteration.

```
def K(σ_array, ce):
    """
    The policy function operator. Given the policy function,
    it updates the optimal consumption using Euler equation.

    * σ_array is an array of policy function values on the grid
    * ce is an instance of CakeEating

    """

    u_prime, β, x_grid = ce.u_prime, ce.β, ce.x_grid
    σ_new = np.empty_like(σ_array)

    σ = lambda x: interp(x_grid, σ_array, x)

    def euler_diff(c, x):
        return u_prime(c) - β * u_prime(σ(x - c))

    for i, x in enumerate(x_grid):

        # handle small x separately --- helps numerical stability
        if x < 1e-12:
            σ_new[i] = 0.0
```

(continues on next page)

(continued from previous page)

```
        # handle other x
        else:
            σ_new[i] = bisect(euler_diff, 1e-10, x - 1e-10, x)

    return σ_new
```

```
def iterate_euler_equation(ce,
                           max_iter=500,
                           tol=1e-5,
                           verbose=True,
                           print_skip=25):

    x_grid = ce.x_grid

    σ = np.copy(x_grid)         # initial guess

    i = 0
    error = tol + 1
    while i < max_iter and error > tol:

        σ_new = K(σ, ce)

        error = np.max(np.abs(σ_new - σ))
        i += 1

        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")

        σ = σ_new

    if error > tol:
        print("Failed to converge!")
    elif verbose:
        print(f"\nConverged in {i} iterations.")

    return σ
```

```
ce = CakeEating(x_grid_min=0.0)
c_euler = iterate_euler_equation(ce)
```

```
Error at iteration 25 is 0.0036456675931543225.
```

```
Error at iteration 50 is 0.0008283185047067848.
```

```
Error at iteration 75 is 0.00030791132300957147.
```

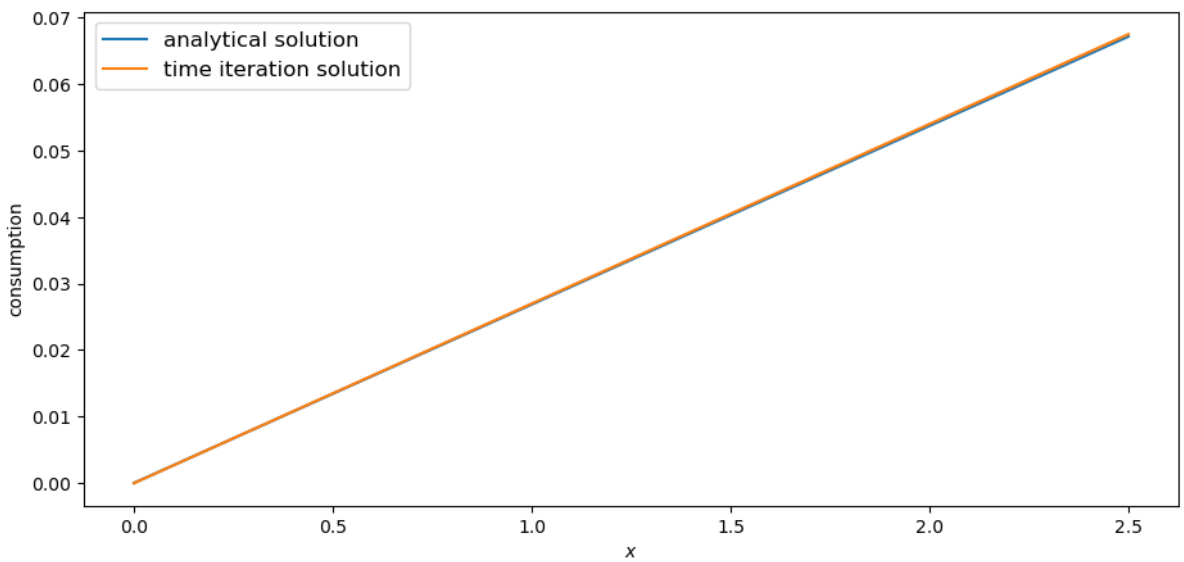
```
Error at iteration 100 is 0.00013555502390599772.
```

```
Error at iteration 125 is 6.417740905302616e-05.
```

```
Error at iteration 150 is 3.1438019047758115e-05.
```

```
Error at iteration 175 is 1.5658492883291464e-05.

Converged in 192 iterations.
```

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

ax.plot(ce.x_grid, c_analytical, label='analytical solution')
ax.plot(ce.x_grid, c_euler, label='time iteration solution')

ax.set_ylabel('consumption')
ax.set_xlabel('$x$')
ax.legend(fontsize=12)

plt.show()
```



CHAPTER

TWENTYSIX

OPTIMAL GROWTH I: THE STOCHASTIC OPTIMAL GROWTH MODEL

Contents

- *Optimal Growth I: The Stochastic Optimal Growth Model*
  - *Overview*
  - *The Model*
  - *Computation*
  - *Exercises*

26.1 Overview

In this lecture, we're going to study a simple optimal growth model with one agent.

The model is a version of the standard one sector infinite horizon growth model studied in

- [SLP89], chapter 2
- [LS18], section 3.1
- EDTC, chapter 1
- [Sun96], chapter 12

It is an extension of the simple *cake eating problem* we looked at earlier.

The extension involves

- nonlinear returns to saving, through a production function, and
- stochastic returns, due to shocks to production.

Despite these additions, the model is still relatively simple.

We regard it as a stepping stone to more sophisticated models.

We solve the model using dynamic programming and a range of numerical techniques.

In this first lecture on optimal growth, the solution method will be value function iteration (VFI).

While the code in this first lecture runs slowly, we will use a variety of techniques to drastically improve execution time over the next few lectures.

Let's start with some imports:

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from scipy.interpolate import interp1d
from scipy.optimize import minimize_scalar
```

26.2 The Model

Consider an agent who owns an amount $y_t \in \mathbb{R}_+ := [0, \infty)$ of a consumption good at time t.

This output can either be consumed or invested.

When the good is invested, it is transformed one-for-one into capital.

The resulting capital stock, denoted here by k_{t+1}, will then be used for production.

Production is stochastic, in that it also depends on a shock ξ_{t+1} realized at the end of the current period.

Next period output is

$$y_{t+1} := f(k_{t+1})\xi_{t+1}$$

where $f \colon \mathbb{R}_+ \to \mathbb{R}_+$ is called the production function.

The resource constraint is

$$k_{t+1} + c_t \leq y_t \tag{26.1}$$

and all variables are required to be nonnegative.

26.2.1 Assumptions and Comments

In what follows,

- The sequence $\{\xi_t\}$ is assumed to be IID.
- The common distribution of each ξ_t will be denoted by ϕ.
- The production function f is assumed to be increasing and continuous.
- Depreciation of capital is not made explicit but can be incorporated into the production function.

While many other treatments of the stochastic growth model use k_t as the state variable, we will use y_t.

This will allow us to treat a stochastic model while maintaining only one state variable.

We consider alternative states and timing specifications in some of our other lectures.

26.2.2 Optimization

Taking y_0 as given, the agent wishes to maximize

$$\mathbb{E}\left[\sum_{t=0}^{\infty}\beta^t u(c_t)\right] \tag{26.2}$$

subject to

$$y_{t+1} = f(y_t - c_t)\xi_{t+1} \quad \text{and} \quad 0 \leq c_t \leq y_t \quad \text{for all } t \tag{26.3}$$

where

- u is a bounded, continuous and strictly increasing utility function and
- $\beta \in (0,1)$ is a discount factor.

In (26.3) we are assuming that the resource constraint (26.1) holds with equality — which is reasonable because u is strictly increasing and no output will be wasted at the optimum.

In summary, the agent's aim is to select a path $c_0, c_1, c_2, \ldots$ for consumption that is

1. nonnegative,
2. feasible in the sense of (26.1),
3. optimal, in the sense that it maximizes (26.2) relative to all other feasible consumption sequences, and
4. *adapted*, in the sense that the action c_t depends only on observable outcomes, not on future outcomes such as ξ_{t+1}.

In the present context

- y_t is called the *state* variable — it summarizes the "state of the world" at the start of each period.
- c_t is called the *control* variable — a value chosen by the agent each period after observing the state.

26.2.3 The Policy Function Approach

One way to think about solving this problem is to look for the best **policy function**.

A policy function is a map from past and present observables into current action.

We'll be particularly interested in **Markov policies**, which are maps from the current state y_t into a current action c_t.

For dynamic programming problems such as this one (in fact for any Markov decision process), the optimal policy is always a Markov policy.

In other words, the current state y_t provides a sufficient statistic for the history in terms of making an optimal decision today.

This is quite intuitive, but if you wish you can find proofs in texts such as [SLP89] (section 4.1).

Hereafter we focus on finding the best Markov policy.

In our context, a Markov policy is a function $\sigma \colon \mathbb{R}_+ \to \mathbb{R}_+$, with the understanding that states are mapped to actions via

$$c_t = \sigma(y_t) \quad \text{for all } t$$

In what follows, we will call σ a *feasible consumption policy* if it satisfies

$$0 \leq \sigma(y) \leq y \quad \text{for all} \quad y \in \mathbb{R}_+ \tag{26.4}$$

In other words, a feasible consumption policy is a Markov policy that respects the resource constraint.

The set of all feasible consumption policies will be denoted by Σ.

Each $\sigma \in \Sigma$ determines a continuous state Markov process $\{y_t\}$ for output via

$$y_{t+1} = f(y_t - \sigma(y_t))\xi_{t+1}, \quad y_0 \text{ given} \tag{26.5}$$

This is the time path for output when we choose and stick with the policy σ.

We insert this process into the objective function to get

$$\mathbb{E}\left[\sum_{t=0}^{\infty} \beta^t u(c_t)\right] = \mathbb{E}\left[\sum_{t=0}^{\infty} \beta^t u(\sigma(y_t))\right] \tag{26.6}$$

This is the total expected present value of following policy σ forever, given initial income y_0.

The aim is to select a policy that makes this number as large as possible.

The next section covers these ideas more formally.

26.2.4 Optimality

The σ associated with a given policy σ is the mapping defined by

$$v_\sigma(y) = \mathbb{E}\left[\sum_{t=0}^{\infty} \beta^t u(\sigma(y_t))\right] \tag{26.7}$$

when $\{y_t\}$ is given by (26.5) with $y_0 = y$.

In other words, it is the lifetime value of following policy σ starting at initial condition y.

The **value function** is then defined as

$$v^*(y) := \sup_{\sigma \in \Sigma} \; v_\sigma(y) \tag{26.8}$$

The value function gives the maximal value that can be obtained from state y, after considering all feasible policies.

A policy $\sigma \in \Sigma$ is called **optimal** if it attains the supremum in (26.8) for all $y \in \mathbb{R}_+$.

26.2.5 The Bellman Equation

With our assumptions on utility and production functions, the value function as defined in (26.8) also satisfies a **Bellman equation**.

For this problem, the Bellman equation takes the form

$$v(y) = \max_{0 \leq c \leq y} \left\{ u(c) + \beta \int v(f(y-c)z)\phi(dz) \right\} \qquad (y \in \mathbb{R}_+) \tag{26.9}$$

This is a *functional equation in* v.

The term $\int v(f(y-c)z)\phi(dz)$ can be understood as the expected next period value when

- v is used to measure value
- the state is y
- consumption is set to c

As shown in EDTC, theorem 10.1.11 and a range of other texts

The value function v^* satisfies the Bellman equation

In other words, (26.9) holds when $v = v^*$.

The intuition is that maximal value from a given state can be obtained by optimally trading off

- current reward from a given action, vs
- expected discounted future value of the state resulting from that action

The Bellman equation is important because it gives us more information about the value function.

It also suggests a way of computing the value function, which we discuss below.

26.2.6 Greedy Policies

The primary importance of the value function is that we can use it to compute optimal policies.

The details are as follows.

Given a continuous function v on $\mathbb{R}_+$, we say that $\sigma \in \Sigma$ is v-**greedy** if $\sigma(y)$ is a solution to

$$\max_{0 \leq c \leq y} \left\{ u(c) + \beta \int v(f(y-c)z)\phi(dz) \right\} \tag{26.10}$$

for every $y \in \mathbb{R}_+$.

In other words, $\sigma \in \Sigma$ is v-greedy if it optimally trades off current and future rewards when v is taken to be the value function.

In our setting, we have the following key result

- A feasible consumption policy is optimal if and only if it is v^*-greedy.

The intuition is similar to the intuition for the Bellman equation, which was provided after (26.9).

See, for example, theorem 10.1.11 of EDTC.

Hence, once we have a good approximation to v^*, we can compute the (approximately) optimal policy by computing the corresponding greedy policy.

The advantage is that we are now solving a much lower dimensional optimization problem.

26.2.7 The Bellman Operator

How, then, should we compute the value function?

One way is to use the so-called **Bellman operator**.

(An operator is a map that sends functions into functions.)

The Bellman operator is denoted by T and defined by

$$Tv(y) := \max_{0 \leq c \leq y} \left\{ u(c) + \beta \int v(f(y-c)z)\phi(dz) \right\} \qquad (y \in \mathbb{R}_+) \tag{26.11}$$

In other words, T sends the function v into the new function Tv defined by (26.11).

By construction, the set of solutions to the Bellman equation (26.9) *exactly coincides with* the set of fixed points of T.

For example, if $Tv = v$, then, for any $y \geq 0$,

$$v(y) = Tv(y) = \max_{0 \leq c \leq y} \left\{ u(c) + \beta \int v^*(f(y-c)z)\phi(dz) \right\}$$

which says precisely that v is a solution to the Bellman equation.

It follows that v^* is a fixed point of T.

26.2.8 Review of Theoretical Results

One can also show that T is a contraction mapping on the set of continuous bounded functions on $\mathbb{R}_+$ under the supremum distance

$$\rho(g, h) = \sup_{y \geq 0} |g(y) - h(y)|$$

See EDTC, lemma 10.1.18.

Hence, it has exactly one fixed point in this set, which we know is equal to the value function.

It follows that

- The value function v^* is bounded and continuous.
- Starting from any bounded and continuous v, the sequence $v, Tv, T^2v, \ldots$ generated by iteratively applying T converges uniformly to v^*.

This iterative method is called **value function iteration**.

We also know that a feasible policy is optimal if and only if it is v^*-greedy.

It's not too hard to show that a v^*-greedy policy exists (see EDTC, theorem 10.1.11 if you get stuck).

Hence, at least one optimal policy exists.

Our problem now is how to compute it.

26.2.9 Unbounded Utility

The results stated above assume that the utility function is bounded.

In practice economists often work with unbounded utility functions — and so will we.

In the unbounded setting, various optimality theories exist.

Unfortunately, they tend to be case-specific, as opposed to valid for a large range of applications.

Nevertheless, their main conclusions are usually in line with those stated for the bounded case just above (as long as we drop the word "bounded").

Consult, for example, section 12.2 of EDTC, [Kam12] or [MdRV10].

26.3 Computation

Let's now look at computing the value function and the optimal policy.

Our implementation in this lecture will focus on clarity and flexibility.

Both of these things are helpful, but they do cost us some speed — as you will see when you run the code.

Later we will sacrifice some of this clarity and flexibility in order to accelerate our code with just-in-time (JIT) compilation.

The algorithm we will use is fitted value function iteration, which was described in earlier lectures *the McCall model* and *cake eating*.

The algorithm will be

1. Begin with an array of values $\{v_1, \dots, v_I\}$ representing the values of some initial function v on the grid points $\{y_1, \dots, y_I\}$.
2. Build a function $\hat{v}$ on the state space $\mathbb{R}_+$ by linear interpolation, based on these data points.
3. Obtain and record the value $T\hat{v}(y_i)$ on each grid point y_i by repeatedly solving (26.11).
4. Unless some stopping condition is satisfied, set $\{v_1, \dots, v_I\} = \{T\hat{v}(y_1), \dots, T\hat{v}(y_I)\}$ and go to step 2.

26.3.1 Scalar Maximization

To maximize the right hand side of the Bellman equation (26.9), we are going to use the `minimize_scalar` routine from SciPy.

Since we are maximizing rather than minimizing, we will use the fact that the maximizer of g on the interval $[a, b]$ is the minimizer of $-g$ on the same interval.

To this end, and to keep the interface tidy, we will wrap `minimize_scalar` in an outer function as follows:

```
def maximize(g, a, b, args):
    """
    Maximize the function g over the interval [a, b].

    We use the fact that the maximizer of g on any interval is
    also the minimizer of -g.  The tuple args collects any extra
    arguments to g.

    Returns the maximal value and the maximizer.
    """

    objective = lambda x: -g(x, *args)
    result = minimize_scalar(objective, bounds=(a, b), method='bounded')
    maximizer, maximum = result.x, -result.fun
    return maximizer, maximum
```

26.3.2 Optimal Growth Model

We will assume for now that ϕ is the distribution of $\xi := \exp(\mu + s\zeta)$ where

- ζ is standard normal,
- μ is a shock location parameter and
- s is a shock scale parameter.

We will store this and other primitives of the optimal growth model in a class.

The class, defined below, combines both parameters and a method that realizes the right hand side of the Bellman equation (26.9).

```
class OptimalGrowthModel:

    def __init__(self,
                 u,             # utility function
                 f,             # production function
                 β=0.96,        # discount factor
                 μ=0,           # shock location parameter
```

(continues on next page)

(continued from previous page)

```
                 s=0.1,         # shock scale parameter
                 grid_max=4,
                 grid_size=120,
                 shock_size=250,
                 seed=1234):

        self.u, self.f, self.β, self.μ, self.s = u, f, β, μ, s

        # Set up grid
        self.grid = np.linspace(1e-4, grid_max, grid_size)

        # Store shocks (with a seed, so results are reproducible)
        np.random.seed(seed)
        self.shocks = np.exp(μ + s * np.random.randn(shock_size))

    def state_action_value(self, c, y, v_array):
        """
        Right hand side of the Bellman equation.
        """

        u, f, β, shocks = self.u, self.f, self.β, self.shocks

        v = interp1d(self.grid, v_array)

        return u(c) + β * np.mean(v(f(y - c) * shocks))
```

In the second last line we are using linear interpolation.

In the last line, the expectation in (26.11) is computed via Monte Carlo, using the approximation

$$\int v(f(y-c)z)\phi(dz) \approx \frac{1}{n}\sum_{i=1}^{n} v(f(y-c)\xi_i)$$

where $\{\xi_i\}_{i=1}^n$ are IID draws from ϕ.

Monte Carlo is not always the most efficient way to compute integrals numerically but it does have some theoretical advantages in the present setting.

(For example, it preserves the contraction mapping property of the Bellman operator — see, e.g., [PalS13].)

26.3.3 The Bellman Operator

The next function implements the Bellman operator.

(We could have added it as a method to the `OptimalGrowthModel` class, but we prefer small classes rather than monolithic ones for this kind of numerical work.)

```
def T(v, og):
    """
    The Bellman operator.  Updates the guess of the value function
    and also computes a v-greedy policy.

      * og is an instance of OptimalGrowthModel
      * v is an array representing a guess of the value function
```

(continues on next page)

(continued from previous page)

```
    """
    v_new = np.empty_like(v)
    v_greedy = np.empty_like(v)

    for i in range(len(grid)):
        y = grid[i]

        # Maximize RHS of Bellman equation at state y
        c_star, v_max = maximize(og.state_action_value, 1e-10, y, (y, v))
        v_new[i] = v_max
        v_greedy[i] = c_star

    return v_greedy, v_new
```

26.3.4 An Example

Let's suppose now that

$$f(k) = k^\alpha \quad \text{and} \quad u(c) = \ln c$$

For this particular problem, an exact analytical solution is available (see [LS18], section 3.1.2), with

$$v^*(y) = \frac{\ln(1-\alpha\beta)}{1-\beta} + \frac{(\mu + \alpha \ln(\alpha\beta))}{1-\alpha}\left[\frac{1}{1-\beta} - \frac{1}{1-\alpha\beta}\right] + \frac{1}{1-\alpha\beta}\ln y \tag{26.12}$$

and optimal consumption policy

$$\sigma^*(y) = (1-\alpha\beta)y$$

It is valuable to have these closed-form solutions because it lets us check whether our code works for this particular case.

In Python, the functions above can be expressed as:

```
def v_star(y, α, β, μ):
    """
    True value function
    """
    c1 = np.log(1 - α * β) / (1 - β)
    c2 = (μ + α * np.log(α * β)) / (1 - α)
    c3 = 1 / (1 - β)
    c4 = 1 / (1 - α * β)
    return c1 + c2 * (c3 - c4) + c4 * np.log(y)

def σ_star(y, α, β):
    """
    True optimal policy
    """
    return (1 - α * β) * y
```

Next let's create an instance of the model with the above primitives and assign it to the variable `og`.

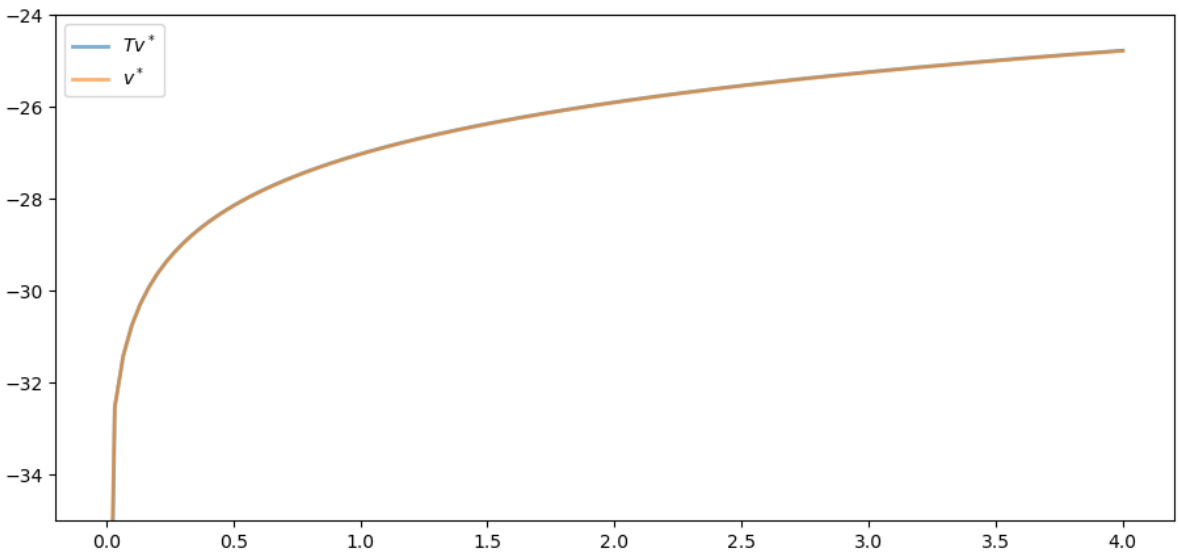
```
α = 0.4
def fcd(k):
    return k**α

og = OptimalGrowthModel(u=np.log, f=fcd)
```

Now let's see what happens when we apply our Bellman operator to the exact solution v^* in this case.

In theory, since v^* is a fixed point, the resulting function should again be v^*.

In practice, we expect some small numerical error.

```
grid = og.grid

v_init = v_star(grid, α, og.β, og.μ)     # Start at the solution
v_greedy, v = T(v_init, og)              # Apply T once

fig, ax = plt.subplots()
ax.set_ylim(-35, -24)
ax.plot(grid, v, lw=2, alpha=0.6, label='$Tv^*$')
ax.plot(grid, v_init, lw=2, alpha=0.6, label='$v^*$')
ax.legend()
plt.show()
```



The two functions are essentially indistinguishable, so we are off to a good start.

Now let's have a look at iterating with the Bellman operator, starting from an arbitrary initial condition.

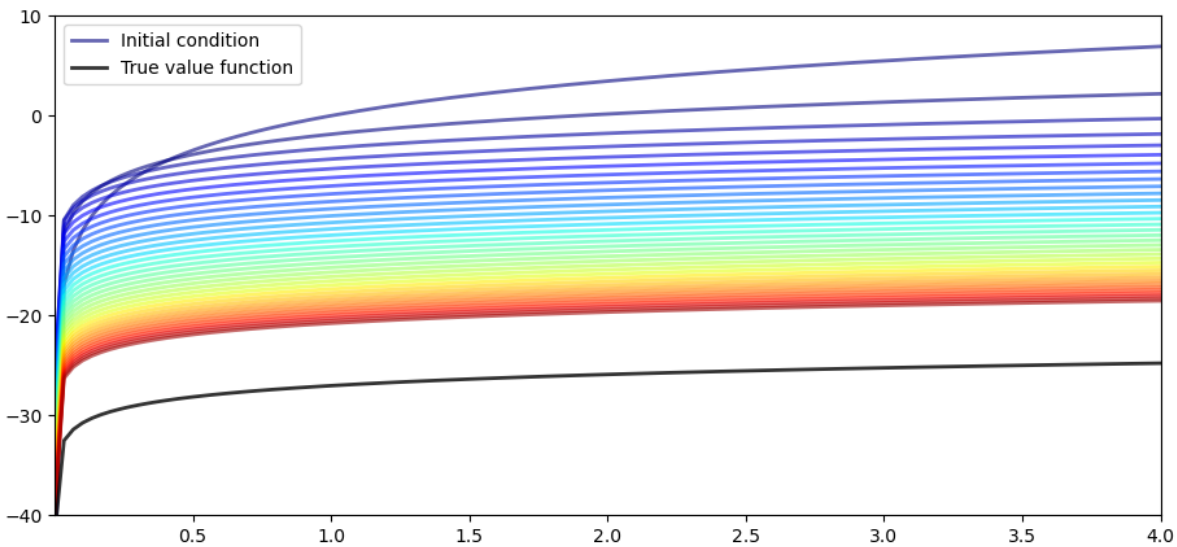
The initial condition we'll start with is, somewhat arbitrarily, $v(y) = 5\ln(y)$.

```
v = 5 * np.log(grid)  # An initial condition
n = 35

fig, ax = plt.subplots()

ax.plot(grid, v, color=plt.cm.jet(0),
        lw=2, alpha=0.6, label='Initial condition')

for i in range(n):
    v_greedy, v = T(v, og)  # Apply the Bellman operator
    ax.plot(grid, v, color=plt.cm.jet(i / n), lw=2, alpha=0.6)

ax.plot(grid, v_star(grid, α, og.β, og.μ), 'k-', lw=2,
        alpha=0.8, label='True value function')
```

(continues on next page)

(continued from previous page)

```
ax.legend()
ax.set(ylim=(-40, 10), xlim=(np.min(grid), np.max(grid)))
plt.show()
```



The figure shows

1. the first 36 functions generated by the fitted value function iteration algorithm, with hotter colors given to higher iterates
2. the true value function v^* drawn in black

The sequence of iterates converges towards v^*.

We are clearly getting closer.

26.3.5 Iterating to Convergence

We can write a function that iterates until the difference is below a particular tolerance level.

```
def solve_model(og,
                tol=1e-4,
                max_iter=1000,
                verbose=True,
                print_skip=25):
    """
    Solve model by iterating with the Bellman operator.

    """

    # Set up loop
    v = og.u(og.grid)  # Initial condition
    i = 0
    error = tol + 1

    while i < max_iter and error > tol:
        v_greedy, v_new = T(v, og)
```

(continues on next page)

(continued from previous page)

```
        error = np.max(np.abs(v - v_new))
        i += 1
        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")
        v = v_new

    if error > tol:
        print("Failed to converge!")
    elif verbose:
        print(f"\nConverged in {i} iterations.")

    return v_greedy, v_new
```

Let's use this function to compute an approximate solution at the defaults.

```
v_greedy, v_solution = solve_model(og)
```

```
Error at iteration 25 is 0.40975776844490497.
```

```
Error at iteration 50 is 0.1476753540823772.
```

```
Error at iteration 75 is 0.05322171277213883.
```

```
Error at iteration 100 is 0.019180930548646558.
```

```
Error at iteration 125 is 0.006912744396029069.
```

```
Error at iteration 150 is 0.002491330384817303.
```

```
Error at iteration 175 is 0.000897867291303811.
```

```
Error at iteration 200 is 0.00032358842396718046.
```

```
Error at iteration 225 is 0.00011662020561331587.
```

```
Converged in 229 iterations.
```

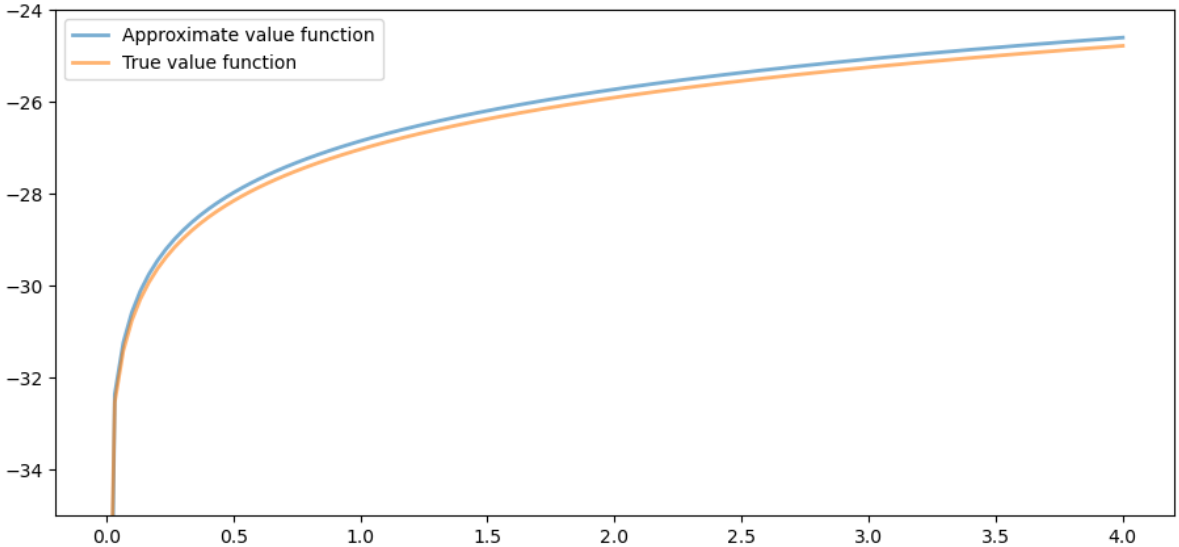
Now we check our result by plotting it against the true value:

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

ax.plot(grid, v_solution, lw=2, alpha=0.6,
        label='Approximate value function')

ax.plot(grid, v_star(grid, α, og.β, og.μ), lw=2,
        alpha=0.6, label='True value function')

ax.legend()
```

(continues on next page)

(continued from previous page)

```
ax.set_ylim(-35, -24)
plt.show()
```



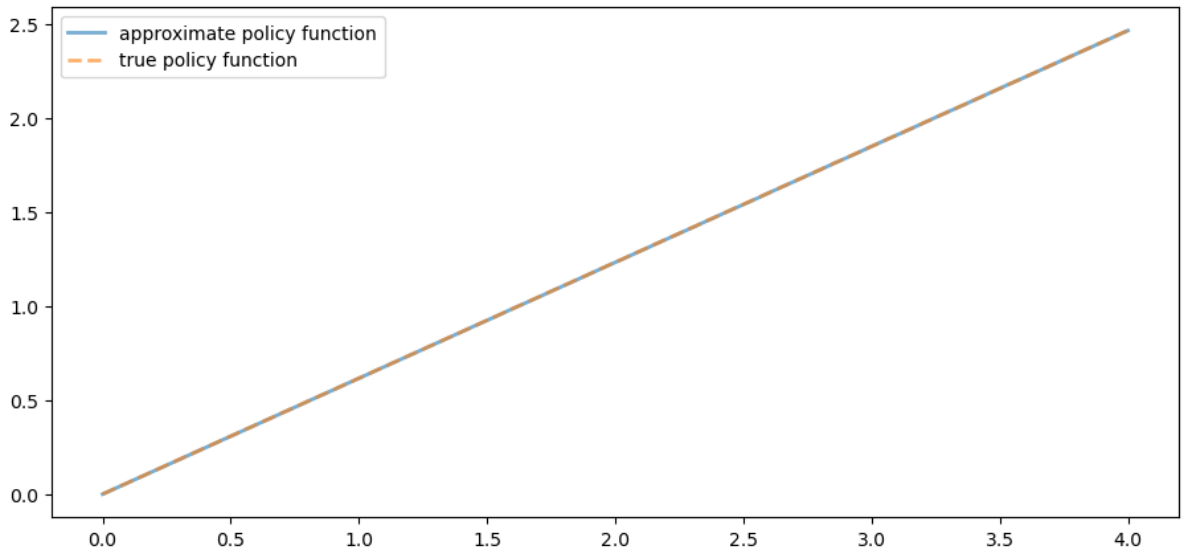
The figure shows that we are pretty much on the money.

26.3.6 The Policy Function

The policy `v_greedy` computed above corresponds to an approximate optimal policy.

The next figure compares it to the exact solution, which, as mentioned above, is $\sigma(y) = (1 - \alpha\beta)y$

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

ax.plot(grid, v_greedy, lw=2,
        alpha=0.6, label='approximate policy function')

ax.plot(grid, σ_star(grid, α, og.β), '--',
        lw=2, alpha=0.6, label='true policy function')

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



The figure shows that we've done a good job in this instance of approximating the true policy.

26.4 Exercises

Exercise 26.4.1

A common choice for utility function in this kind of work is the CRRA specification

$$u(c) = \frac{c^{1-\gamma}}{1-\gamma}$$

Maintaining the other defaults, including the Cobb-Douglas production function, solve the optimal growth model with this utility specification.

Setting $\gamma = 1.5$, compute and plot an estimate of the optimal policy.

Time how long this function takes to run, so you can compare it to faster code developed in the *next lecture*.

Solution to Exercise 26.4.1

Here we set up the model.

```
γ = 1.5   # Preference parameter

def u_crra(c):
    return (c**(1 - γ) - 1) / (1 - γ)

og = OptimalGrowthModel(u=u_crra, f=fcd)
```

Now let's run it, with a timer.

```
%%time
v_greedy, v_solution = solve_model(og)
```

```
Error at iteration 25 is 0.5528151810417512.
```

```
Error at iteration 50 is 0.19923228425590978.
```

```
Error at iteration 75 is 0.07180266113800826.
```

```
Error at iteration 100 is 0.025877443335843964.
```

```
Error at iteration 125 is 0.009326145618970827.
```

```
Error at iteration 150 is 0.003361112262005861.
```

```
Error at iteration 175 is 0.0012113338243295857.
```

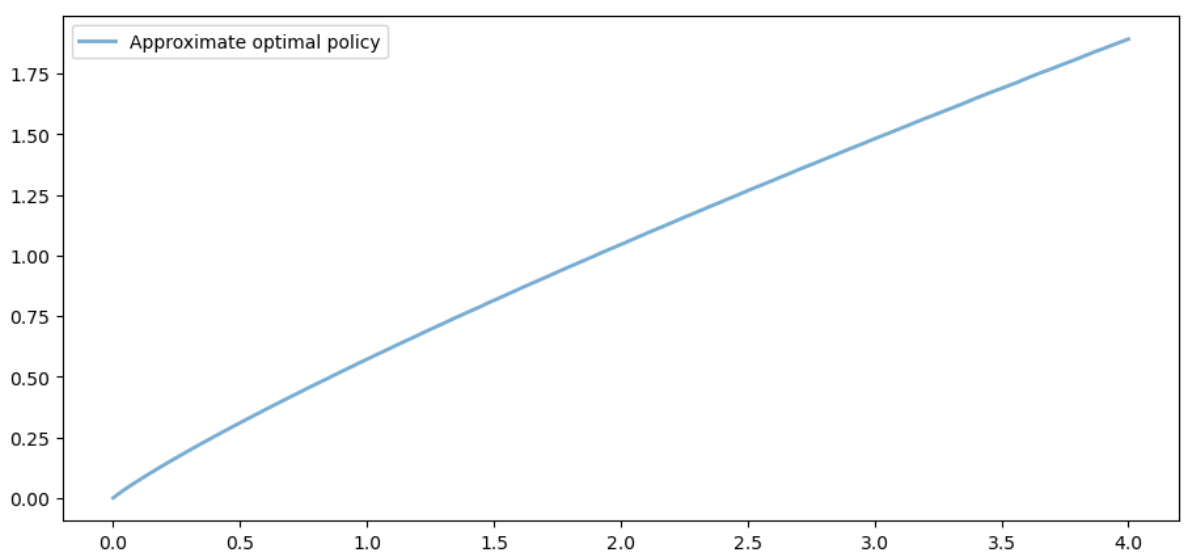
```
Error at iteration 200 is 0.0004365607333056687.
```

```
Error at iteration 225 is 0.00015733505506432266.
```

```
Converged in 237 iterations.
CPU times: user 20.6 s, sys: 19.9 ms, total: 20.7 s
Wall time: 20.7 s
```

Let's plot the policy function just to see what it looks like:

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

ax.plot(grid, v_greedy, lw=2,
        alpha=0.6, label='Approximate optimal policy')

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



Exercise 26.4.2

Time how long it takes to iterate with the Bellman operator 20 times, starting from initial condition $v(y) = u(y)$.

Use the model specification in the previous exercise.

(As before, we will compare this number with that for the faster code developed in the *next lecture*.)

Solution to Exercise 26.4.2

Let's set up:

```
og = OptimalGrowthModel(u=u_crra, f=fcd)
v = og.u(og.grid)
```

Here's the timing:

```
%%time

for i in range(20):
    v_greedy, v_new = T(v, og)
    v = v_new
```

```
CPU times: user 1.76 s, sys: 0 ns, total: 1.76 s
Wall time: 1.76 s
```

CHAPTER

TWENTYSEVEN

OPTIMAL GROWTH II: ACCELERATING THE CODE WITH NUMBA

Contents

- *Optimal Growth II: Accelerating the Code with Numba*
  - *Overview*
  - *The Model*
  - *Computation*
  - *Exercises*

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

```
!pip install quantecon
!pip install interpolation
```

27.1 Overview

Previously, we studied a stochastic optimal growth model with one representative agent.

We solved the model using dynamic programming.

In writing our code, we focused on clarity and flexibility.

These are important, but there's often a trade-off between flexibility and speed.

The reason is that, when code is less flexible, we can exploit structure more easily.

(This is true about algorithms and mathematical problems more generally: more specific problems have more structure, which, with some thought, can be exploited for better results.)

So, in this lecture, we are going to accept less flexibility while gaining speed, using just-in-time (JIT) compilation to accelerate our code.

Let's start with some imports:

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from interpolation import interp
from numba import jit, njit
from quantecon.optimize.scalar_maximization import brent_max
```

```
/home/runner/miniconda3/envs/quantecon/lib/python3.11/site-packages/numba/core/
↪decorators.py:262: NumbaDeprecationWarning: numba.generated_jit is deprecated.␣
↪Please see the documentation at: https://numba.readthedocs.io/en/stable/
↪reference/deprecation.html#deprecation-of-generated-jit for more information and␣
↪advice on a suitable replacement.
  warnings.warn(msg, NumbaDeprecationWarning)
```

We are using an interpolation function from interpolation.py because it helps us JIT-compile our code.

The function `brent_max` is also designed for embedding in JIT-compiled code.

These are alternatives to similar functions in SciPy (which, unfortunately, are not JIT-aware).

27.2 The Model

The model is the same as discussed in our *previous lecture* on optimal growth.

We will start with log utility:

$$u(c) = \ln(c)$$

We continue to assume that

- $f(k) = k^\alpha$
- ϕ is the distribution of $\xi := \exp(\mu + s\zeta)$ when ζ is standard normal

We will once again use value function iteration to solve the model.

In particular, the algorithm is unchanged, and the only difference is in the implementation itself.

As before, we will be able to compare with the true solutions

```
def v_star(y, α, β, μ):
    """
    True value function
    """
    c1 = np.log(1 - α * β) / (1 - β)
    c2 = (μ + α * np.log(α * β)) / (1 - α)
    c3 = 1 / (1 - β)
    c4 = 1 / (1 - α * β)
    return c1 + c2 * (c3 - c4) + c4 * np.log(y)

def σ_star(y, α, β):
    """
    True optimal policy
    """
    return (1 - α * β) * y
```

27.3 Computation

We will again store the primitives of the optimal growth model in a class.

But now we are going to use Numba's `@jitclass` decorator to target our class for JIT compilation.

Because we are going to use Numba to compile our class, we need to specify the data types.

You will see this as a list called `opt_growth_data` above our class.

Unlike in the *previous lecture*, we hardwire the production and utility specifications into the class.

This is where we sacrifice flexibility in order to gain more speed.

```
from numba import float64
from numba.experimental import jitclass

opt_growth_data = [
    ('α', float64),          # Production parameter
    ('β', float64),          # Discount factor
    ('μ', float64),          # Shock location parameter
    ('s', float64),          # Shock scale parameter
    ('grid', float64[:]),    # Grid (array)
    ('shocks', float64[:])   # Shock draws (array)
]

@jitclass(opt_growth_data)
class OptimalGrowthModel:

    def __init__(self,
                 α=0.4,
                 β=0.96,
                 μ=0,
                 s=0.1,
                 grid_max=4,
                 grid_size=120,
                 shock_size=250,
                 seed=1234):

        self.α, self.β, self.μ, self.s = α, β, μ, s

        # Set up grid
        self.grid = np.linspace(1e-5, grid_max, grid_size)

        # Store shocks (with a seed, so results are reproducible)
        np.random.seed(seed)
        self.shocks = np.exp(μ + s * np.random.randn(shock_size))


    def f(self, k):
        "The production function"
        return k**self.α


    def u(self, c):
        "The utility function"
        return np.log(c)

    def f_prime(self, k):
```

(continues on next page)

(continued from previous page)

```
        "Derivative of f"
        return self.α * (k**(self.α - 1))


    def u_prime(self, c):
        "Derivative of u"
        return 1/c

    def u_prime_inv(self, c):
        "Inverse of u'"
        return 1/c
```

The class includes some methods such as `u_prime` that we do not need now but will use in later lectures.

27.3.1 The Bellman Operator

We will use JIT compilation to accelerate the Bellman operator.

First, here's a function that returns the value of a particular consumption choice `c`, given state `y`, as per the Bellman equation (26.9).

```
@njit
def state_action_value(c, y, v_array, og):
    """
    Right hand side of the Bellman equation.

     * c is consumption
     * y is income
     * og is an instance of OptimalGrowthModel
     * v_array represents a guess of the value function on the grid

    """

    u, f, β, shocks = og.u, og.f, og.β, og.shocks

    v = lambda x: interp(og.grid, v_array, x)

    return u(c) + β * np.mean(v(f(y - c) * shocks))
```

Now we can implement the Bellman operator, which maximizes the right hand side of the Bellman equation:

```
@jit(nopython=True)
def T(v, og):
    """
    The Bellman operator.

     * og is an instance of OptimalGrowthModel
     * v is an array representing a guess of the value function

    """

    v_new = np.empty_like(v)
    v_greedy = np.empty_like(v)
```

(continues on next page)

(continued from previous page)

```
    for i in range(len(og.grid)):
        y = og.grid[i]

        # Maximize RHS of Bellman equation at state y
        result = brent_max(state_action_value, 1e-10, y, args=(y, v, og))
        v_greedy[i], v_new[i] = result[0], result[1]

    return v_greedy, v_new
```

We use the `solve_model` function to perform iteration until convergence.

```
def solve_model(og,
                tol=1e-4,
                max_iter=1000,
                verbose=True,
                print_skip=25):
    """
    Solve model by iterating with the Bellman operator.

    """

    # Set up loop
    v = og.u(og.grid)  # Initial condition
    i = 0
    error = tol + 1

    while i < max_iter and error > tol:
        v_greedy, v_new = T(v, og)
        error = np.max(np.abs(v - v_new))
        i += 1
        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")
        v = v_new

    if error > tol:
        print("Failed to converge!")
    elif verbose:
        print(f"\nConverged in {i} iterations.")

    return v_greedy, v_new
```

Let's compute the approximate solution at the default parameters.

First we create an instance:

```
og = OptimalGrowthModel()
```

Now we call `solve_model`, using the `%%time` magic to check how long it takes.

```
%%time
v_greedy, v_solution = solve_model(og)
```

```
Error at iteration 25 is 0.41372668361363196.
```

```
Error at iteration 50 is 0.14767653072604503.
```

```
Error at iteration 75 is 0.053221715530355596.
```

```
Error at iteration 100 is 0.019180931418503633.
```

```
Error at iteration 125 is 0.006912744709538288.
```

```
Error at iteration 150 is 0.002491330497818467.
```

```
Error at iteration 175 is 0.0008978673320712005.
```

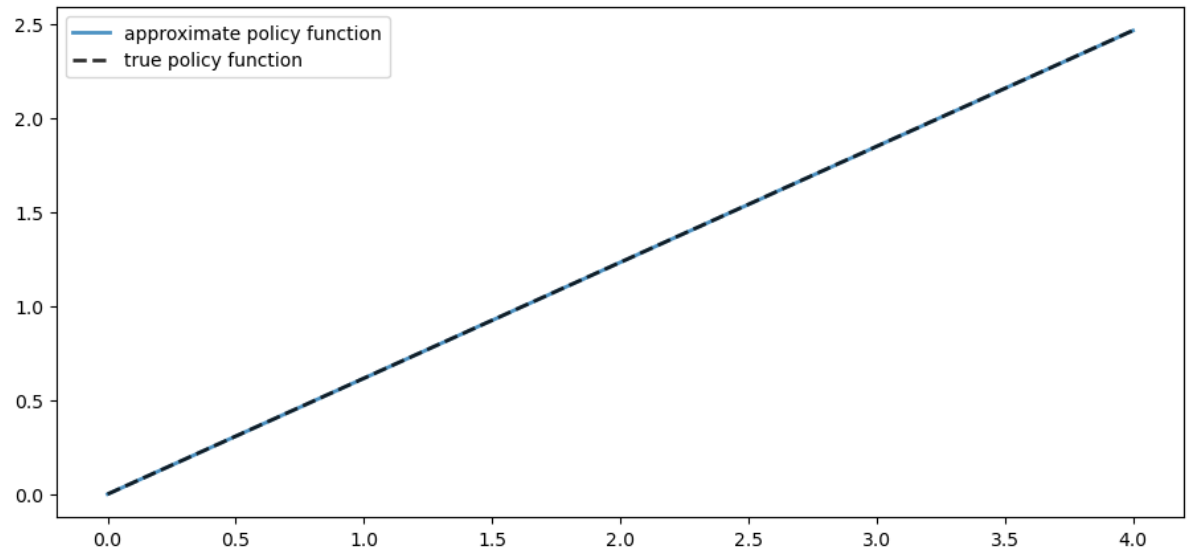
```
Error at iteration 200 is 0.0003235884386754151.
```

```
Error at iteration 225 is 0.00011662021095304453.

Converged in 229 iterations.
CPU times: user 4.14 s, sys: 241 ms, total: 4.38 s
Wall time: 4.38 s
```

You will notice that this is *much* faster than our *original implementation*.

Here is a plot of the resulting policy, compared with the true policy:

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

ax.plot(og.grid, v_greedy, lw=2,
        alpha=0.8, label='approximate policy function')

ax.plot(og.grid, σ_star(og.grid, og.α, og.β), 'k--',
        lw=2, alpha=0.8, label='true policy function')

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



Again, the fit is excellent — this is as expected since we have not changed the algorithm.

The maximal absolute deviation between the two policies is

```
np.max(np.abs(v_greedy - σ_star(og.grid, og.α, og.β)))
```

```
0.0010480539639137199
```

27.4 Exercises

Exercise 27.4.1

Time how long it takes to iterate with the Bellman operator 20 times, starting from initial condition $v(y) = u(y)$.

Use the default parameterization.

Solution to Exercise 27.4.1

Let's set up the initial condition.

```
v = og.u(og.grid)
```

Here's the timing:

```
%%time

for i in range(20):
    v_greedy, v_new = T(v, og)
    v = v_new
```

```
CPU times: user 219 ms, sys: 0 ns, total: 219 ms
Wall time: 219 ms
```

Compared with our *timing* for the non-compiled version of value function iteration, the JIT-compiled code is usually an order of magnitude faster.

Exercise 27.4.2

Modify the optimal growth model to use the CRRA utility specification.

$$u(c) = \frac{c^{1-\gamma}}{1-\gamma}$$

Set `γ = 1.5` as the default value and maintaining other specifications.

(Note that `jitclass` currently does not support inheritance, so you will have to copy the class and change the relevant parameters and methods.)

Compute an estimate of the optimal policy, plot it and compare visually with the same plot from the *analogous exercise* in the first optimal growth lecture.

Compare execution time as well.

Solution to Exercise 27.4.2

Here's our CRRA version of `OptimalGrowthModel`:

```
from numba import float64
from numba.experimental import jitclass

opt_growth_data = [
    ('α', float64),          # Production parameter
    ('β', float64),          # Discount factor
    ('μ', float64),          # Shock location parameter
    ('γ', float64),          # Preference parameter
    ('s', float64),          # Shock scale parameter
    ('grid', float64[:]),    # Grid (array)
    ('shocks', float64[:])   # Shock draws (array)
]

@jitclass(opt_growth_data)
class OptimalGrowthModel_CRRA:

    def __init__(self,
                 α=0.4,
                 β=0.96,
                 μ=0,
                 s=0.1,
                 γ=1.5,
                 grid_max=4,
                 grid_size=120,
                 shock_size=250,
                 seed=1234):

        self.α, self.β, self.γ, self.μ, self.s = α, β, γ, μ, s

        # Set up grid
        self.grid = np.linspace(1e-5, grid_max, grid_size)
```

(continues on next page)

(continued from previous page)

```
        # Store shocks (with a seed, so results are reproducible)
        np.random.seed(seed)
        self.shocks = np.exp(μ + s * np.random.randn(shock_size))


    def f(self, k):
        "The production function."
        return k**self.α

    def u(self, c):
        "The utility function."
        return c**(1 - self.γ) / (1 - self.γ)

    def f_prime(self, k):
        "Derivative of f."
        return self.α * (k**(self.α - 1))

    def u_prime(self, c):
        "Derivative of u."
        return c**(-self.γ)

    def u_prime_inv(c):
        return c**(-1 / self.γ)
```

Let's create an instance:

```
og_crra = OptimalGrowthModel_CRRA()
```

Now we call `solve_model`, using the `%%time` magic to check how long it takes.

```
%%time
v_greedy, v_solution = solve_model(og_crra)
```

```
Error at iteration 25 is 1.6201897527234905.
```

```
Error at iteration 50 is 0.459106047057503.
```

```
Error at iteration 75 is 0.165423522162655.
```

```
Error at iteration 100 is 0.05961808343499797.
```

```
Error at iteration 125 is 0.021486161531569792.
```

```
Error at iteration 150 is 0.007743542074422294.
```

```
Error at iteration 175 is 0.002790747140650751.
```

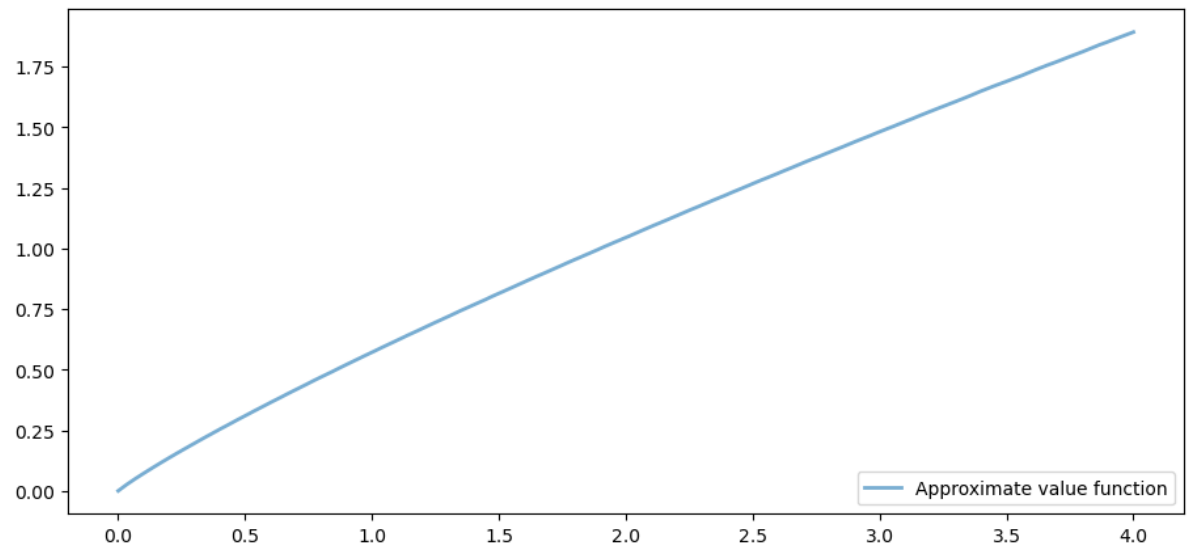
```
Error at iteration 200 is 0.001005776107120937.
```

```
Error at iteration 225 is 0.0003624784085332067.
```

```
Error at iteration 250 is 0.00013063602793295104.

Converged in 257 iterations.
CPU times: user 3.99 s, sys: 144 ms, total: 4.13 s
Wall time: 4.13 s
```

Here is a plot of the resulting policy:

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

ax.plot(og.grid, v_greedy, lw=2,
        alpha=0.6, label='Approximate value function')

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



This matches the solution that we obtained in our non-jitted code, *in the exercises*.

Execution time is an order of magnitude faster.

Exercise 27.4.3

In this exercise we return to the original log utility specification.

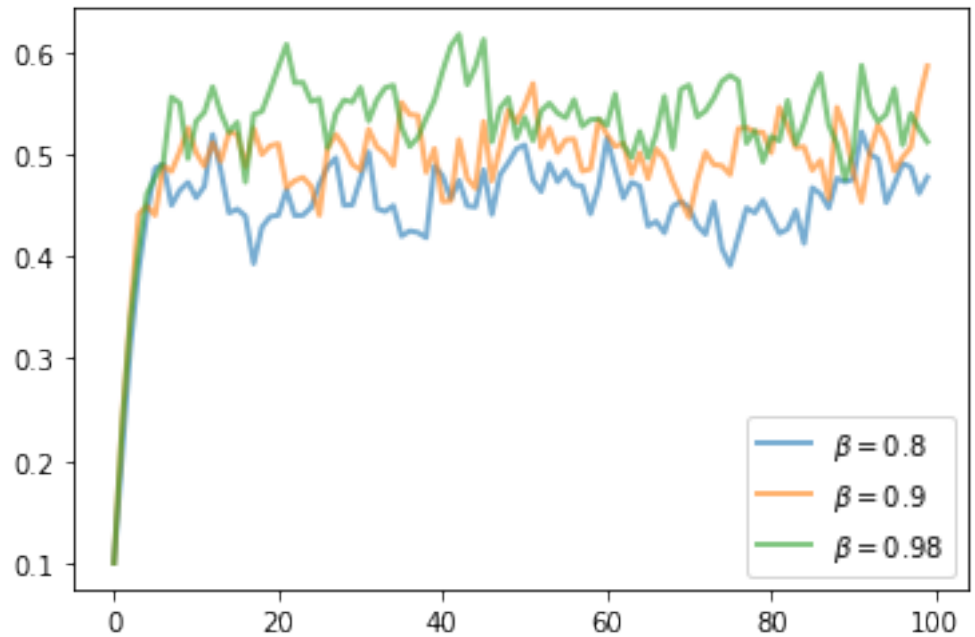
Once an optimal consumption policy σ is given, income follows

$$y_{t+1} = f(y_t - \sigma(y_t))\xi_{t+1}$$

The next figure shows a simulation of 100 elements of this sequence for three different discount factors (and hence three different policies).

In each sequence, the initial condition is $y_0 = 0.1$.

The discount factors are `discount_factors = (0.8, 0.9, 0.98)`.



We have also dialed down the shocks a bit with `s = 0.05`.

Otherwise, the parameters and primitives are the same as the log-linear model discussed earlier in the lecture.

Notice that more patient agents typically have higher wealth.

Replicate the figure modulo randomness.

Solution to Exercise 27.4.3

Here's one solution:

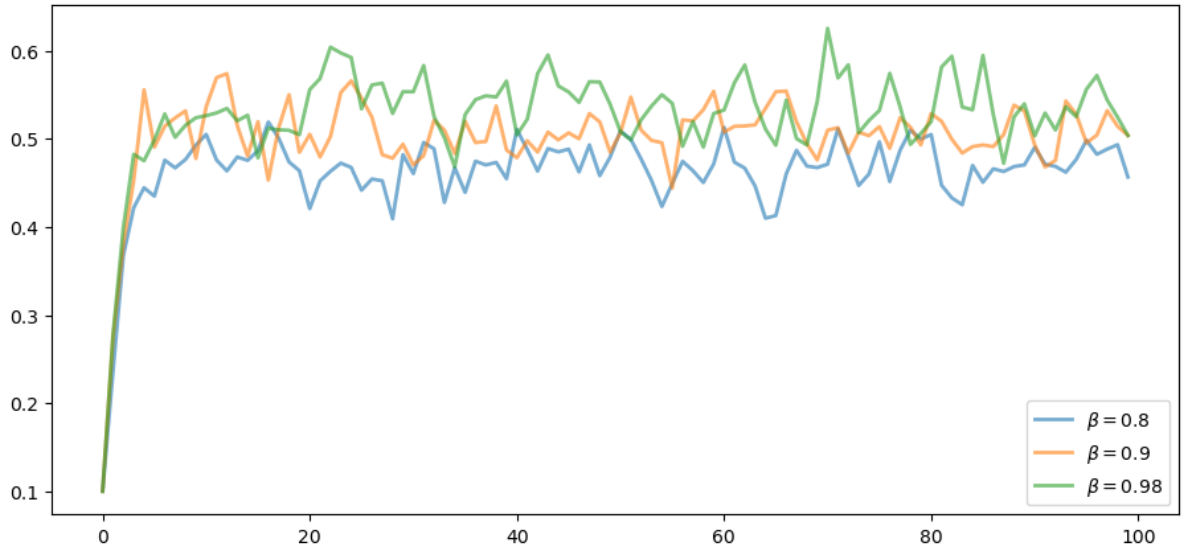
```
def simulate_og(σ_func, og, y0=0.1, ts_length=100):
    '''
    Compute a time series given consumption policy σ.
    '''
    y = np.empty(ts_length)
    ξ = np.random.randn(ts_length-1)
    y[0] = y0
    for t in range(ts_length-1):
        y[t+1] = (y[t] - σ_func(y[t]))**og.α * np.exp(og.μ + og.s * ξ[t])
    return y
```

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

for β in (0.8, 0.9, 0.98):

    og = OptimalGrowthModel(β=β, s=0.05)

    v_greedy, v_solution = solve_model(og, verbose=False)

    # Define an optimal policy function
```

(continues on next page)

(continued from previous page)

```
    σ_func = lambda x: interp(og.grid, v_greedy, x)
    y = simulate_og(σ_func, og)
    ax.plot(y, lw=2, alpha=0.6, label=rf'$\beta = {β}$')

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



CHAPTER

TWENTYEIGHT

OPTIMAL GROWTH III: TIME ITERATION

Contents

- *Optimal Growth III: Time Iteration*
  - *Overview*
  - *The Euler Equation*
  - *Implementation*
  - *Exercises*

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

```
!pip install quantecon
!pip install interpolation
```

28.1 Overview

In this lecture, we'll continue our *earlier* study of the stochastic optimal growth model.

In that lecture, we solved the associated dynamic programming problem using value function iteration.

The beauty of this technique is its broad applicability.

With numerical problems, however, we can often attain higher efficiency in specific applications by deriving methods that are carefully tailored to the application at hand.

The stochastic optimal growth model has plenty of structure to exploit for this purpose, especially when we adopt some concavity and smoothness assumptions over primitives.

We'll use this structure to obtain an Euler equation based method.

This will be an extension of the time iteration method considered in our elementary lecture on *cake eating*.

In a *subsequent lecture*, we'll see that time iteration can be further adjusted to obtain even more efficiency.

Let's start with some imports:

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

(continues on next page)

(continued from previous page)

```
from quantecon.optimize import brentq
from numba import njit
```

```
/home/runner/miniconda3/envs/quantecon/lib/python3.11/site-packages/numba/core/
↪decorators.py:262: NumbaDeprecationWarning: numba.generated_jit is deprecated.␣
↪Please see the documentation at: https://numba.readthedocs.io/en/stable/
↪reference/deprecation.html#deprecation-of-generated-jit for more information and␣
↪advice on a suitable replacement.
  warnings.warn(msg, NumbaDeprecationWarning)
```

28.2 The Euler Equation

Our first step is to derive the Euler equation, which is a generalization of the Euler equation we obtained in the *lecture on cake eating*.

We take the model set out in *the stochastic growth model lecture* and add the following assumptions:

1. u and f are continuously differentiable and strictly concave
2. $f(0) = 0$
3. $\lim_{c \to 0} u'(c) = \infty$ and $\lim_{c \to \infty} u'(c) = 0$
4. $\lim_{k \to 0} f'(k) = \infty$ and $\lim_{k \to \infty} f'(k) = 0$

The last two conditions are usually called **Inada conditions**.

Recall the Bellman equation

$$v^*(y) = \max_{0 \leq c \leq y} \left\{ u(c) + \beta \int v^*(f(y-c)z)\phi(dz) \right\} \quad \text{for all} \quad y \in \mathbb{R}_+ \tag{28.1}$$

Let the optimal consumption policy be denoted by σ^*.

We know that σ^* is a v^*-greedy policy so that $\sigma^*(y)$ is the maximizer in (28.1).

The conditions above imply that

- σ^* is the unique optimal policy for the stochastic optimal growth model
- the optimal policy is continuous, strictly increasing and also **interior**, in the sense that $0 < \sigma^*(y) < y$ for all strictly positive y, and
- the value function is strictly concave and continuously differentiable, with

$$(v^*)'(y) = u'(\sigma^*(y)) := (u' \circ \sigma^*)(y) \tag{28.2}$$

The last result is called the **envelope condition** due to its relationship with the envelope theorem.

To see why (28.2) holds, write the Bellman equation in the equivalent form

$$v^*(y) = \max_{0 \leq k \leq y} \left\{ u(y-k) + \beta \int v^*(f(k)z)\phi(dz) \right\},$$

Differentiating with respect to y, and then evaluating at the optimum yields (28.2).

(Section 12.1 of EDTC contains full proofs of these results, and closely related discussions can be found in many other texts.)

Differentiability of the value function and interiority of the optimal policy imply that optimal consumption satisfies the first order condition associated with (28.1), which is

$$u'(\sigma^*(y)) = \beta \int (v^*)'(f(y - \sigma^*(y))z) f'(y - \sigma^*(y)) z \phi(dz) \tag{28.3}$$

Combining (28.2) and the first-order condition (28.3) gives the **Euler equation**

$$(u' \circ \sigma^*)(y) = \beta \int (u' \circ \sigma^*)(f(y - \sigma^*(y))z) f'(y - \sigma^*(y)) z \phi(dz) \tag{28.4}$$

We can think of the Euler equation as a functional equation

$$(u' \circ \sigma)(y) = \beta \int (u' \circ \sigma)(f(y - \sigma(y))z) f'(y - \sigma(y)) z \phi(dz) \tag{28.5}$$

over interior consumption policies σ, one solution of which is the optimal policy σ^*.

Our aim is to solve the functional equation (28.5) and hence obtain σ^*.

28.2.1 The Coleman-Reffett Operator

Recall the Bellman operator

$$Tv(y) := \max_{0 \leq c \leq y} \left\{ u(c) + \beta \int v(f(y - c)z) \phi(dz) \right\} \tag{28.6}$$

Just as we introduced the Bellman operator to solve the Bellman equation, we will now introduce an operator over policies to help us solve the Euler equation.

This operator K will act on the set of all $\sigma \in \Sigma$ that are continuous, strictly increasing and interior.

Henceforth we denote this set of policies by $\mathcal{P}$

1. The operator K takes as its argument a $\sigma \in \mathcal{P}$ and
2. returns a new function $K\sigma$, where $K\sigma(y)$ is the $c \in (0, y)$ that solves.

$$u'(c) = \beta \int (u' \circ \sigma)(f(y - c)z) f'(y - c) z \phi(dz) \tag{28.7}$$

We call this operator the **Coleman-Reffett operator** to acknowledge the work of [Col90] and [Ref96].

In essence, $K\sigma$ is the consumption policy that the Euler equation tells you to choose today when your future consumption policy is σ.

The important thing to note about K is that, by construction, its fixed points coincide with solutions to the functional equation (28.5).

In particular, the optimal policy σ^* is a fixed point.

Indeed, for fixed y, the value $K\sigma^*(y)$ is the c that solves

$$u'(c) = \beta \int (u' \circ \sigma^*)(f(y - c)z) f'(y - c) z \phi(dz)$$

In view of the Euler equation, this is exactly $\sigma^*(y)$.

28.2.2 Is the Coleman-Reffett Operator Well Defined?

In particular, is there always a unique $c \in (0, y)$ that solves (28.7)?

The answer is yes, under our assumptions.

For any $\sigma \in \mathcal{P}$, the right side of (28.7)

- is continuous and strictly increasing in c on $(0, y)$
- diverges to $+\infty$ as $c \uparrow y$

The left side of (28.7)

- is continuous and strictly decreasing in c on $(0, y)$
- diverges to $+\infty$ as $c \downarrow 0$

Sketching these curves and using the information above will convince you that they cross exactly once as c ranges over $(0, y)$.

With a bit more analysis, one can show in addition that $K\sigma \in \mathcal{P}$ whenever $\sigma \in \mathcal{P}$.

28.2.3 Comparison with VFI (Theory)

It is possible to prove that there is a tight relationship between iterates of K and iterates of the Bellman operator.

Mathematically, the two operators are *topologically conjugate*.

Loosely speaking, this means that if iterates of one operator converge then so do iterates of the other, and vice versa.

Moreover, there is a sense in which they converge at the same rate, at least in theory.

However, it turns out that the operator K is more stable numerically and hence more efficient in the applications we consider.

Examples are given below.

28.3 Implementation

As in our *previous study*, we continue to assume that

- $u(c) = \ln c$
- $f(k) = k^\alpha$
- ϕ is the distribution of $\xi := \exp(\mu + s\zeta)$ when ζ is standard normal

This will allow us to compare our results to the analytical solutions

```
def v_star(y, α, β, μ):
    """
    True value function
    """
    c1 = np.log(1 - α * β) / (1 - β)
    c2 = (μ + α * np.log(α * β)) / (1 - α)
    c3 = 1 / (1 - β)
    c4 = 1 / (1 - α * β)
    return c1 + c2 * (c3 - c4) + c4 * np.log(y)
```

(continues on next page)

(continued from previous page)

```
def σ_star(y, α, β):
    """
    True optimal policy
    """
    return (1 - α * β) * y
```

As discussed above, our plan is to solve the model using time iteration, which means iterating with the operator K.

For this we need access to the functions u' and f, f'.

These are available in a class called `OptimalGrowthModel` that we constructed in an *earlier lecture*.

```
from numba import float64
from numba.experimental import jitclass

opt_growth_data = [
    ('α', float64),          # Production parameter
    ('β', float64),          # Discount factor
    ('μ', float64),          # Shock location parameter
    ('s', float64),          # Shock scale parameter
    ('grid', float64[:]),    # Grid (array)
    ('shocks', float64[:])   # Shock draws (array)
]

@jitclass(opt_growth_data)
class OptimalGrowthModel:

    def __init__(self,
                 α=0.4,
                 β=0.96,
                 μ=0,
                 s=0.1,
                 grid_max=4,
                 grid_size=120,
                 shock_size=250,
                 seed=1234):

        self.α, self.β, self.μ, self.s = α, β, μ, s

        # Set up grid
        self.grid = np.linspace(1e-5, grid_max, grid_size)

        # Store shocks (with a seed, so results are reproducible)
        np.random.seed(seed)
        self.shocks = np.exp(μ + s * np.random.randn(shock_size))


    def f(self, k):
        "The production function"
        return k**self.α


    def u(self, c):
        "The utility function"
        return np.log(c)

    def f_prime(self, k):
```

(continues on next page)

(continued from previous page)

```
        "Derivative of f"
        return self.α * (k**(self.α - 1))


    def u_prime(self, c):
        "Derivative of u"
        return 1/c

    def u_prime_inv(self, c):
        "Inverse of u'"
        return 1/c
```

Now we implement a method called `euler_diff`, which returns

$$u'(c) - \beta \int (u' \circ \sigma)(f(y-c)z) f'(y-c) z \phi(dz) \tag{28.8}$$

```
@njit
def euler_diff(c, σ, y, og):
    """
    Set up a function such that the root with respect to c,
    given y and σ, is equal to Kσ(y).

    """

    β, shocks, grid = og.β, og.shocks, og.grid
    f, f_prime, u_prime = og.f, og.f_prime, og.u_prime

    # First turn σ into a function via interpolation
    σ_func = lambda x: interp(grid, σ, x)

    # Now set up the function we need to find the root of.
    vals = u_prime(σ_func(f(y - c) * shocks)) * f_prime(y - c) * shocks
    return u_prime(c) - β * np.mean(vals)
```

The function `euler_diff` evaluates integrals by Monte Carlo and approximates functions using linear interpolation.

We will use a root-finding algorithm to solve (28.8) for c given state y and σ, the current guess of the policy.

Here's the operator K, that implements the root-finding step.

```
@njit
def K(σ, og):
    """
    The Coleman-Reffett operator

     Here og is an instance of OptimalGrowthModel.
    """

    β = og.β
    f, f_prime, u_prime = og.f, og.f_prime, og.u_prime
    grid, shocks = og.grid, og.shocks

    σ_new = np.empty_like(σ)
    for i, y in enumerate(grid):
        # Solve for optimal c at y
```

(continues on next page)

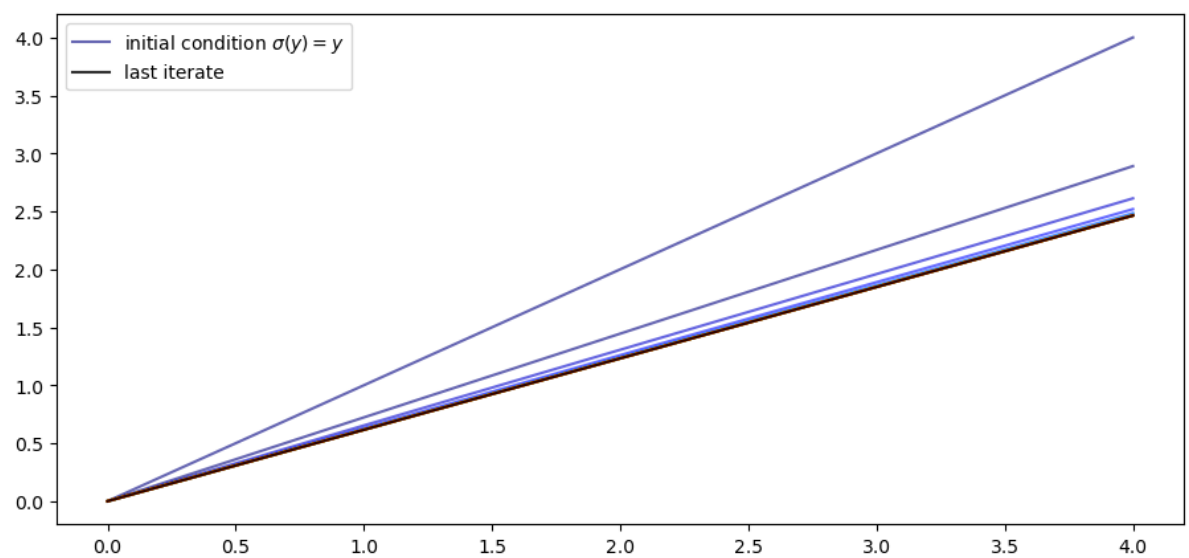
(continued from previous page)

```
        c_star = brentq(euler_diff, 1e-10, y-1e-10, args=(σ, y, og))[0]
        σ_new[i] = c_star

    return σ_new
```

28.3.1 Testing

Let's generate an instance and plot some iterates of K, starting from $\sigma(y) = y$.

```
og = OptimalGrowthModel()
grid = og.grid

n = 15
σ = grid.copy()  # Set initial condition

fig, ax = plt.subplots()
lb = 'initial condition $\sigma(y) = y$'
ax.plot(grid, σ, color=plt.cm.jet(0), alpha=0.6, label=lb)

for i in range(n):
    σ = K(σ, og)
    ax.plot(grid, σ, color=plt.cm.jet(i / n), alpha=0.6)

# Update one more time and plot the last iterate in black
σ = K(σ, og)
ax.plot(grid, σ, color='k', alpha=0.8, label='last iterate')

ax.legend()

plt.show()
```



We see that the iteration process converges quickly to a limit that resembles the solution we obtained in *the previous lecture*.

Here is a function called `solve_model_time_iter` that takes an instance of `OptimalGrowthModel` and returns

an approximation to the optimal policy, using time iteration.

```
def solve_model_time_iter(model,     # Class with model information
                          σ,         # Initial condition
                          tol=1e-4,
                          max_iter=1000,
                          verbose=True,
                          print_skip=25):

    # Set up loop
    i = 0
    error = tol + 1

    while i < max_iter and error > tol:
        σ_new = K(σ, model)
        error = np.max(np.abs(σ - σ_new))
        i += 1
        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")
        σ = σ_new

    if error > tol:
        print("Failed to converge!")
    elif verbose:
        print(f"\nConverged in {i} iterations.")

    return σ_new
```

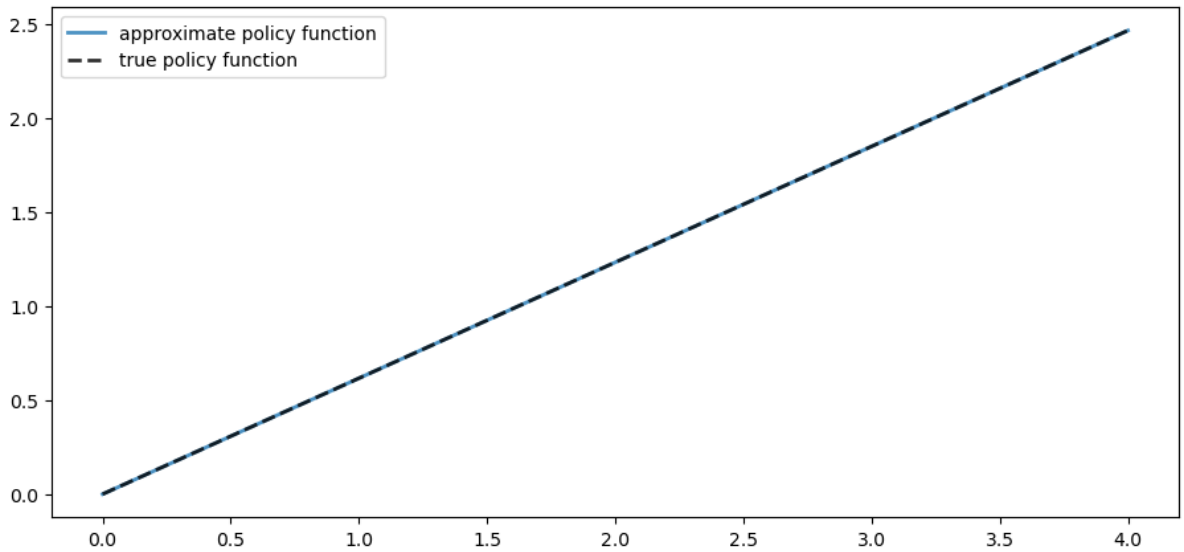
Let's call it:

```
σ_init = np.copy(og.grid)
σ = solve_model_time_iter(og, σ_init)
```

```
Converged in 11 iterations.
```

Here is a plot of the resulting policy, compared with the true policy:

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

ax.plot(og.grid, σ, lw=2,
        alpha=0.8, label='approximate policy function')

ax.plot(og.grid, σ_star(og.grid, og.α, og.β), 'k--',
        lw=2, alpha=0.8, label='true policy function')

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



Again, the fit is excellent.

The maximal absolute deviation between the two policies is

```
np.max(np.abs(σ - σ_star(og.grid, og.α, og.β)))
```

```
2.5329106132954138e-05
```

How long does it take to converge?

```
%%timeit -n 3 -r 1
σ = solve_model_time_iter(og, σ_init, verbose=False)
```

```
83.1 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 3 loops each)
```

Convergence is very fast, even compared to our *JIT-compiled value function iteration*.

Overall, we find that time iteration provides a very high degree of efficiency and accuracy, at least for this model.

28.4 Exercises

Exercise 28.4.1

Solve the model with CRRA utility

$$u(c) = \frac{c^{1-\gamma}}{1-\gamma}$$

Set `γ = 1.5`.

Compute and plot the optimal policy.

Solution to Exercise 28.4.1

We use the class `OptimalGrowthModel_CRRA` from our *VFI lecture*.

```
from numba import float64
from numba.experimental import jitclass

opt_growth_data = [
    ('α', float64),          # Production parameter
    ('β', float64),          # Discount factor
    ('μ', float64),          # Shock location parameter
    ('γ', float64),          # Preference parameter
    ('s', float64),          # Shock scale parameter
    ('grid', float64[:]),    # Grid (array)
    ('shocks', float64[:])   # Shock draws (array)
]

@jitclass(opt_growth_data)
class OptimalGrowthModel_CRRA:

    def __init__(self,
                 α=0.4,
                 β=0.96,
                 μ=0,
                 s=0.1,
                 γ=1.5,
                 grid_max=4,
                 grid_size=120,
                 shock_size=250,
                 seed=1234):

        self.α, self.β, self.γ, self.μ, self.s = α, β, γ, μ, s

        # Set up grid
        self.grid = np.linspace(1e-5, grid_max, grid_size)

        # Store shocks (with a seed, so results are reproducible)
        np.random.seed(seed)
        self.shocks = np.exp(μ + s * np.random.randn(shock_size))


    def f(self, k):
        "The production function."
        return k**self.α

    def u(self, c):
        "The utility function."
        return c**(1 - self.γ) / (1 - self.γ)

    def f_prime(self, k):
        "Derivative of f."
        return self.α * (k**(self.α - 1))

    def u_prime(self, c):
        "Derivative of u."
        return c**(-self.γ)

    def u_prime_inv(c):
        return c**(-1 / self.γ)
```

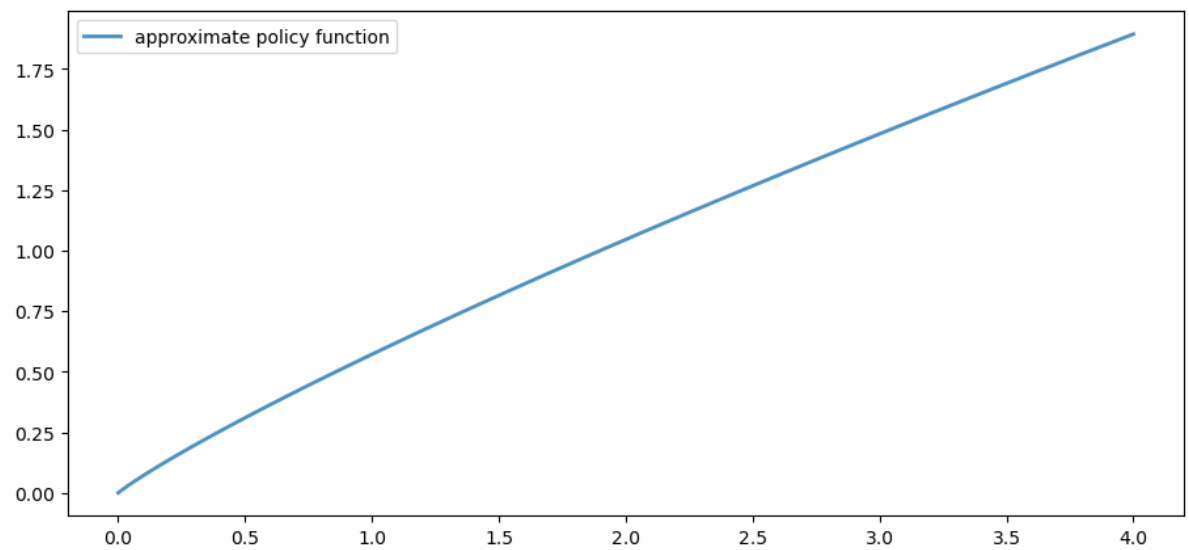
Let's create an instance:

```
og_crra = OptimalGrowthModel_CRRA()
```

Now we solve and plot the policy:

```
%%time
σ = solve_model_time_iter(og_crra, σ_init)


fig, ax = plt.subplots()

ax.plot(og.grid, σ, lw=2,
        alpha=0.8, label='approximate policy function')

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

```
Converged in 13 iterations.
```




```
CPU times: user 1.67 s, sys: 144 ms, total: 1.81 s
Wall time: 1.81 s
```

CHAPTER

TWENTYNINE

OPTIMAL GROWTH IV: THE ENDOGENOUS GRID METHOD

Contents

- *Optimal Growth IV: The Endogenous Grid Method*
  - *Overview*
  - *Key Idea*
  - *Implementation*

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

```
!pip install interpolation
```

29.1 Overview

Previously, we solved the stochastic optimal growth model using

1. *value function iteration*
2. *Euler equation based time iteration*

We found time iteration to be significantly more accurate and efficient.

In this lecture, we'll look at a clever twist on time iteration called the **endogenous grid method** (EGM).

EGM is a numerical method for implementing policy iteration invented by Chris Carroll.

The original reference is [Car06].

Let's start with some standard imports:

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

```
/home/runner/miniconda3/envs/quantecon/lib/python3.11/site-packages/numba/core/
↪decorators.py:262: NumbaDeprecationWarning: numba.generated_jit is deprecated.␣
↪Please see the documentation at: https://numba.readthedocs.io/en/stable/
↪reference/deprecation.html#deprecation-of-generated-jit for more information and␣
↪advice on a suitable replacement.
  warnings.warn(msg, NumbaDeprecationWarning)
```

29.2 Key Idea

Let's start by reminding ourselves of the theory and then see how the numerics fit in.

29.2.1 Theory

Take the model set out in *the time iteration lecture*, following the same terminology and notation.

The Euler equation is

$$(u' \circ \sigma^*)(y) = \beta \int (u' \circ \sigma^*)(f(y - \sigma^*(y))z) f'(y - \sigma^*(y)) z \phi(dz) \tag{29.1}$$

As we saw, the Coleman-Reffett operator is a nonlinear operator K engineered so that σ^* is a fixed point of K.

It takes as its argument a continuous strictly increasing consumption policy $\sigma \in \Sigma$.

It returns a new function $K\sigma$, where $(K\sigma)(y)$ is the $c \in (0, \infty)$ that solves

$$u'(c) = \beta \int (u' \circ \sigma)(f(y - c)z) f'(y - c) z \phi(dz) \tag{29.2}$$

29.2.2 Exogenous Grid

As discussed in *the lecture on time iteration*, to implement the method on a computer, we need a numerical approximation.

In particular, we represent a policy function by a set of values on a finite grid.

The function itself is reconstructed from this representation when necessary, using interpolation or some other method.

Previously, to obtain a finite representation of an updated consumption policy, we

- fixed a grid of income points $\{y_i\}$
- calculated the consumption value c_i corresponding to each y_i using (29.2) and a root-finding routine

Each c_i is then interpreted as the value of the function $K\sigma$ at y_i.

Thus, with the points $\{y_i, c_i\}$ in hand, we can reconstruct $K\sigma$ via approximation.

Iteration then continues…

29.2.3 Endogenous Grid

The method discussed above requires a root-finding routine to find the c_i corresponding to a given income value y_i.

Root-finding is costly because it typically involves a significant number of function evaluations.

As pointed out by Carroll [Car06], we can avoid this if y_i is chosen endogenously.

The only assumption required is that u' is invertible on $(0, \infty)$.

Let $(u')^{-1}$ be the inverse function of u'.

The idea is this:

- First, we fix an *exogenous* grid $\{k_i\}$ for capital ($k = y - c$).
- Then we obtain c_i via

$$c_i = (u')^{-1} \left\{ \beta \int (u' \circ \sigma)(f(k_i)z) \, f'(k_i) \, z \, \phi(dz) \right\} \tag{29.3}$$

- Finally, for each c_i we set $y_i = c_i + k_i$.

It is clear that each (y_i, c_i) pair constructed in this manner satisfies (29.2).

With the points $\{y_i, c_i\}$ in hand, we can reconstruct $K\sigma$ via approximation as before.

The name EGM comes from the fact that the grid $\{y_i\}$ is determined **endogenously**.

29.3 Implementation

As *before*, we will start with a simple setting where

- $u(c) = \ln c$,
- production is Cobb-Douglas, and
- the shocks are lognormal.

This will allow us to make comparisons with the analytical solutions

```
def v_star(y, α, β, μ):
    """
    True value function
    """
    c1 = np.log(1 - α * β) / (1 - β)
    c2 = (μ + α * np.log(α * β)) / (1 - α)
    c3 = 1 / (1 - β)
    c4 = 1 / (1 - α * β)
    return c1 + c2 * (c3 - c4) + c4 * np.log(y)

def σ_star(y, α, β):
    """
    True optimal policy
    """
    return (1 - α * β) * y
```

We reuse the `OptimalGrowthModel` class

```
from numba import float64
from numba.experimental import jitclass

opt_growth_data = [
    ('α', float64),          # Production parameter
    ('β', float64),          # Discount factor
    ('μ', float64),          # Shock location parameter
    ('s', float64),          # Shock scale parameter
    ('grid', float64[:]),    # Grid (array)
    ('shocks', float64[:])   # Shock draws (array)
]

@jitclass(opt_growth_data)
class OptimalGrowthModel:

    def __init__(self,
                 α=0.4,
                 β=0.96,
                 μ=0,
                 s=0.1,
                 grid_max=4,
                 grid_size=120,
                 shock_size=250,
                 seed=1234):

        self.α, self.β, self.μ, self.s = α, β, μ, s

        # Set up grid
        self.grid = np.linspace(1e-5, grid_max, grid_size)

        # Store shocks (with a seed, so results are reproducible)
        np.random.seed(seed)
        self.shocks = np.exp(μ + s * np.random.randn(shock_size))


    def f(self, k):
        "The production function"
        return k**self.α


    def u(self, c):
        "The utility function"
        return np.log(c)

    def f_prime(self, k):
        "Derivative of f"
        return self.α * (k**(self.α - 1))


    def u_prime(self, c):
        "Derivative of u"
        return 1/c

    def u_prime_inv(self, c):
        "Inverse of u'"
        return 1/c
```

29.3.1 The Operator

Here's an implementation of K using EGM as described above.

```
@njit
def K(σ_array, og):
    """
    The Coleman-Reffett operator using EGM

    """

    # Simplify names
    f, β = og.f, og.β
    f_prime, u_prime = og.f_prime, og.u_prime
    u_prime_inv = og.u_prime_inv
    grid, shocks = og.grid, og.shocks

    # Determine endogenous grid
    y = grid + σ_array  # y_i = k_i + c_i

    # Linear interpolation of policy using endogenous grid
    σ = lambda x: interp(y, σ_array, x)

    # Allocate memory for new consumption array
    c = np.empty_like(grid)

    # Solve for updated consumption value
    for i, k in enumerate(grid):
        vals = u_prime(σ(f(k) * shocks)) * f_prime(k) * shocks
        c[i] = u_prime_inv(β * np.mean(vals))

    return c
```

Note the lack of any root-finding algorithm.

29.3.2 Testing

First we create an instance.

```
og = OptimalGrowthModel()
grid = og.grid
```

Here's our solver routine:

```
def solve_model_time_iter(model,     # Class with model information
                          σ,         # Initial condition
                          tol=1e-4,
                          max_iter=1000,
                          verbose=True,
                          print_skip=25):

    # Set up loop
    i = 0
    error = tol + 1
```

(continues on next page)

(continued from previous page)

```
    while i < max_iter and error > tol:
        σ_new = K(σ, model)
        error = np.max(np.abs(σ - σ_new))
        i += 1
        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")
        σ = σ_new

    if error > tol:
        print("Failed to converge!")
    elif verbose:
        print(f"\nConverged in {i} iterations.")

    return σ_new
```

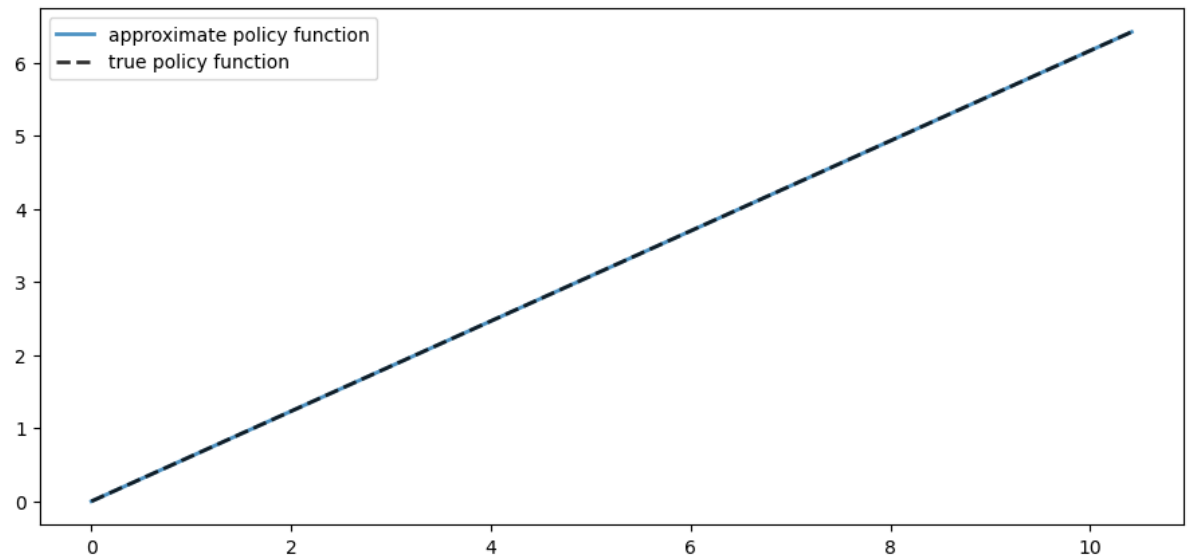
Let's call it:

```
σ_init = np.copy(grid)
σ = solve_model_time_iter(og, σ_init)
```

```
Converged in 12 iterations.
```

Here is a plot of the resulting policy, compared with the true policy:

```
y = grid + σ  # y_i = k_i + c_i

fig, ax = plt.subplots()

ax.plot(y, σ, lw=2,
        alpha=0.8, label='approximate policy function')

ax.plot(y, σ_star(y, og.α, og.β), 'k--',
        lw=2, alpha=0.8, label='true policy function')

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



The maximal absolute deviation between the two policies is

```
np.max(np.abs(σ - σ_star(y, og.α, og.β)))
```

```
1.530274914252061e-05
```

How long does it take to converge?

```
%%timeit -n 3 -r 1
σ = solve_model_time_iter(og, σ_init, verbose=False)
```

```
11.8 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 3 loops each)
```

Relative to time iteration, which as already found to be highly efficient, EGM has managed to shave off still more run time without compromising accuracy.

This is due to the lack of a numerical root-finding step.

We can now solve the optimal growth model at given parameters extremely fast.

CHAPTER

THIRTY

THE INCOME FLUCTUATION PROBLEM I: BASIC MODEL

Contents

- *The Income Fluctuation Problem I: Basic Model*
  - *Overview*
  - *The Optimal Savings Problem*
  - *Computation*
  - *Implementation*
  - *Exercises*

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

```
!pip install quantecon
!pip install interpolation
```

30.1 Overview

In this lecture, we study an optimal savings problem for an infinitely lived consumer—the "common ancestor" described in [LS18], section 1.3.

This is an essential sub-problem for many representative macroeconomic models

- [Aiy94]
- [Hug93]
- etc.

It is related to the decision problem in the *stochastic optimal growth model* and yet differs in important ways.

For example, the choice problem for the agent includes an additive income term that leads to an occasionally binding constraint.

Moreover, in this and the following lectures, we will inject more realistic features such as correlated shocks.

To solve the model we will use Euler equation based time iteration, which proved to be *fast and accurate* in our investigation of the *stochastic optimal growth model*.

Time iteration is globally convergent under mild assumptions, even when utility is unbounded (both above and below).

We'll need the following imports:

```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from quantecon.optimize import brentq
from interpolation import interp
from numba import njit, float64
from numba.experimental import jitclass
from quantecon import MarkovChain
```

```
/home/runner/miniconda3/envs/quantecon/lib/python3.11/site-packages/numba/core/
↪decorators.py:262: NumbaDeprecationWarning: numba.generated_jit is deprecated.␣
↪Please see the documentation at: https://numba.readthedocs.io/en/stable/
↪reference/deprecation.html#deprecation-of-generated-jit for more information and␣
↪advice on a suitable replacement.
  warnings.warn(msg, NumbaDeprecationWarning)
```

30.1.1 References

Our presentation is a simplified version of [MST20].

Other references include [Dea91], [DH10], [Kuh13], [Rab02], [Rei09] and [SE77].

30.2 The Optimal Savings Problem

Let's write down the model and then discuss how to solve it.

30.2.1 Set-Up

Consider a household that chooses a state-contingent consumption plan $\{c_t\}_{t \geq 0}$ to maximize

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

subject to

$$a_{t+1} \leq R(a_t - c_t) + Y_{t+1}, \quad c_t \geq 0, \quad a_t \geq 0 \quad t = 0, 1, \ldots \tag{30.1}$$

Here

- $\beta \in (0, 1)$ is the discount factor
- a_t is asset holdings at time t, with borrowing constraint $a_t \geq 0$
- c_t is consumption
- Y_t is non-capital income (wages, unemployment compensation, etc.)
- $R := 1 + r$, where $r > 0$ is the interest rate on savings

The timing here is as follows:

1. At the start of period t, the household chooses consumption c_t.
2. Labor is supplied by the household throughout the period and labor income Y_{t+1} is received at the end of period t.

3. Financial income $R(a_t - c_t)$ is received at the end of period t.
4. Time shifts to $t+1$ and the process repeats.

Non-capital income Y_t is given by $Y_t = y(Z_t)$, where $\{Z_t\}$ is an exogeneous state process.

As is common in the literature, we take $\{Z_t\}$ to be a finite state Markov chain taking values in Z with Markov matrix P.

We further assume that

1. $\beta R < 1$
2. u is smooth, strictly increasing and strictly concave with $\lim_{c \to 0} u'(c) = \infty$ and $\lim_{c \to \infty} u'(c) = 0$

The asset space is $\mathbb{R}_+$ and the state is the pair $(a, z) \in \mathsf{S} := \mathbb{R}_+ \times \mathsf{Z}$.

A *feasible consumption path* from $(a, z) \in \mathsf{S}$ is a consumption sequence $\{c_t\}$ such that $\{c_t\}$ and its induced asset path $\{a_t\}$ satisfy

1. $(a_0, z_0) = (a, z)$
2. the feasibility constraints in (30.1), and
3. measurability, which means that c_t is a function of random outcomes up to date t but not after.

The meaning of the third point is just that consumption at time t cannot be a function of outcomes are yet to be observed.

In fact, for this problem, consumption can be chosen optimally by taking it to be contingent only on the current state.

Optimality is defined below.

30.2.2 Value Function and Euler Equation

The *value function* $V \colon \mathsf{S} \to \mathbb{R}$ is defined by

$$V(a, z) := \max \mathbb{E} \left\{ \sum_{t=0}^{\infty} \beta^t u(c_t) \right\} \tag{30.2}$$

where the maximization is overall feasible consumption paths from (a, z).

An *optimal consumption path* from (a, z) is a feasible consumption path from (a, z) that attains the supremum in (30.2).

To pin down such paths we can use a version of the Euler equation, which in the present setting is

$$u'(c_t) \geq \beta R \, \mathbb{E}_t u'(c_{t+1}) \tag{30.3}$$

and

$$c_t < a_t \implies u'(c_t) = \beta R \, \mathbb{E}_t u'(c_{t+1}) \tag{30.4}$$

When $c_t = a_t$ we obviously have $u'(c_t) = u'(a_t)$,

When c_t hits the upper bound a_t, the strict inequality $u'(c_t) > \beta R \, \mathbb{E}_t u'(c_{t+1})$ can occur because c_t cannot increase sufficiently to attain equality.

(The lower boundary case $c_t = 0$ never arises at the optimum because $u'(0) = \infty$.)

With some thought, one can show that (30.3) and (30.4) are equivalent to

$$u'(c_t) = \max \left\{ \beta R \, \mathbb{E}_t u'(c_{t+1}) \, , \; u'(a_t) \right\} \tag{30.5}$$

30.2.3 Optimality Results

As shown in [MST20],

1. For each $(a, z) \in \mathsf{S}$, a unique optimal consumption path from (a, z) exists
2. This path is the unique feasible path from (a, z) satisfying the Euler equality (30.5) and the transversality condition

$$\lim_{t \to \infty} \beta^t \, \mathbb{E} \left[u'(c_t) a_{t+1} \right] = 0 \tag{30.6}$$

Moreover, there exists an *optimal consumption function* $\sigma^* \colon \mathsf{S} \to \mathbb{R}_+$ such that the path from (a, z) generated by

$$(a_0, z_0) = (a, z), \quad c_t = \sigma^*(a_t, Z_t) \quad \text{and} \quad a_{t+1} = R(a_t - c_t) + Y_{t+1}$$

satisfies both (30.5) and (30.6), and hence is the unique optimal path from (a, z).

Thus, to solve the optimization problem, we need to compute the policy σ^*.

30.3 Computation

There are two standard ways to solve for σ^*

1. time iteration using the Euler equality and
2. value function iteration.

Our investigation of the cake eating problem and stochastic optimal growth model suggests that time iteration will be faster and more accurate.

This is the approach that we apply below.

30.3.1 Time Iteration

We can rewrite (30.5) to make it a statement about functions rather than random variables.

In particular, consider the functional equation

$$(u' \circ \sigma)(a, z) = \max \left\{ \beta R \, \mathbb{E}_z (u' \circ \sigma)[R(a - \sigma(a, z)) + \hat{Y}, \, \hat{Z}] \, , \; u'(a) \right\} \tag{30.7}$$

where

- $(u' \circ \sigma)(s) := u'(\sigma(s))$.
- $\mathbb{E}_z$ conditions on current state z and $\hat{X}$ indicates next period value of random variable X and
- σ is the unknown function.

We need a suitable class of candidate solutions for the optimal consumption policy.

The right way to pick such a class is to consider what properties the solution is likely to have, in order to restrict the search space and ensure that iteration is well behaved.

To this end, let $\mathcal{C}$ be the space of continuous functions $\sigma \colon \mathbf{S} \to \mathbb{R}$ such that σ is increasing in the first argument, $0 < \sigma(a, z) \leq a$ for all $(a, z) \in \mathbf{S}$, and

$$\sup_{(a,z) \in \mathbf{S}} |(u' \circ \sigma)(a, z) - u'(a)| < \infty \tag{30.8}$$

This will be our candidate class.

In addition, let $K \colon \mathcal{C} \to \mathcal{C}$ be defined as follows.

For given $\sigma \in \mathcal{C}$, the value $K\sigma(a, z)$ is the unique $c \in [0, a]$ that solves

$$u'(c) = \max \left\{ \beta R \, \mathbb{E}_z (u' \circ \sigma) \left[R(a - c) + \hat{Y}, \, \hat{Z} \right], \; u'(a) \right\} \tag{30.9}$$

We refer to K as the Coleman–Reffett operator.

The operator K is constructed so that fixed points of K coincide with solutions to the functional equation (30.7).

It is shown in [MST20] that the unique optimal policy can be computed by picking any $\sigma \in \mathcal{C}$ and iterating with the operator K defined in (30.9).

30.3.2 Some Technical Details

The proof of the last statement is somewhat technical but here is a quick summary:

It is shown in [MST20] that K is a contraction mapping on $\mathcal{C}$ under the metric

$$\rho(c, d) := \| \, u' \circ \sigma_1 - u' \circ \sigma_2 \, \| := \sup_{s \in S} | \, u'(\sigma_1(s)) - u'(\sigma_2(s)) \, | \qquad (\sigma_1, \sigma_2 \in \mathcal{C})$$

which evaluates the maximal difference in terms of marginal utility.

(The benefit of this measure of distance is that, while elements of $\mathcal{C}$ are not generally bounded, ρ is always finite under our assumptions.)

It is also shown that the metric ρ is complete on $\mathcal{C}$.

In consequence, K has a unique fixed point $\sigma^* \in \mathcal{C}$ and $K^n c \to \sigma^*$ as $n \to \infty$ for any $\sigma \in \mathcal{C}$.

By the definition of K, the fixed points of K in $\mathcal{C}$ coincide with the solutions to (30.7) in $\mathcal{C}$.

As a consequence, the path $\{c_t\}$ generated from $(a_0, z_0) \in S$ using policy function σ^* is the unique optimal path from $(a_0, z_0) \in S$.

30.4 Implementation

We use the CRRA utility specification

$$u(c) = \frac{c^{1-\gamma}}{1 - \gamma}$$

The exogeneous state process $\{Z_t\}$ defaults to a two-state Markov chain with state space $\{0, 1\}$ and transition matrix P.

Here we build a class called `IFP` that stores the model primitives.

```
ifp_data = [
    ('R', float64),              # Interest rate 1 + r
    ('β', float64),              # Discount factor
    ('γ', float64),              # Preference parameter
    ('P', float64[:, :]),        # Markov matrix for binary Z_t
    ('y', float64[:]),           # Income is Y_t = y[Z_t]
    ('asset_grid', float64[:])   # Grid (array)
]

@jitclass(ifp_data)
class IFP:
```

(continues on next page)

(continued from previous page)

```
    def __init__(self,
                 r=0.01,
                 β=0.96,
                 γ=1.5,
                 P=((0.6, 0.4),
                    (0.05, 0.95)),
                 y=(0.0, 2.0),
                 grid_max=16,
                 grid_size=50):

        self.R = 1 + r
        self.β, self.γ = β, γ
        self.P, self.y = np.array(P), np.array(y)
        self.asset_grid = np.linspace(0, grid_max, grid_size)

        # Recall that we need R β < 1 for convergence.
        assert self.R * self.β < 1, "Stability condition violated."

    def u_prime(self, c):
        return c**(-self.γ)
```

Next we provide a function to compute the difference

$$u'(c) - \max\left\{\beta R\, \mathbb{E}_z (u' \circ \sigma)\left[R(a-c) + \hat{Y},\, \hat{Z}\right],\; u'(a)\right\} \tag{30.10}$$

```
@njit
def euler_diff(c, a, z, σ_vals, ifp):
    """
    The difference between the left- and right-hand side
    of the Euler Equation, given current policy σ.

        * c is the consumption choice
        * (a, z) is the state, with z in {0, 1}
        * σ_vals is a policy represented as a matrix.
        * ifp is an instance of IFP

    """

    # Simplify names
    R, P, y, β, γ  = ifp.R, ifp.P, ifp.y, ifp.β, ifp.γ
    asset_grid, u_prime = ifp.asset_grid, ifp.u_prime
    n = len(P)

    # Convert policy into a function by linear interpolation
    def σ(a, z):
        return interp(asset_grid, σ_vals[:, z], a)

    # Calculate the expectation conditional on current z
    expect = 0.0
    for z_hat in range(n):
        expect += u_prime(σ(R * (a - c) + y[z_hat], z_hat)) * P[z, z_hat]

    return u_prime(c) - max(β * R * expect, u_prime(a))
```

Note that we use linear interpolation along the asset grid to approximate the policy function.

The next step is to obtain the root of the Euler difference.

```
@njit
def K(σ, ifp):
    """
    The operator K.

    """
    σ_new = np.empty_like(σ)
    for i, a in enumerate(ifp.asset_grid):
        for z in (0, 1):
            result = brentq(euler_diff, 1e-8, a, args=(a, z, σ, ifp))
            σ_new[i, z] = result.root

    return σ_new
```

With the operator K in hand, we can choose an initial condition and start to iterate.

The following function iterates to convergence and returns the approximate optimal policy.

```
def solve_model_time_iter(model,    # Class with model information
                          σ,        # Initial condition
                          tol=1e-4,
                          max_iter=1000,
                          verbose=True,
                          print_skip=25):

    # Set up loop
    i = 0
    error = tol + 1

    while i < max_iter and error > tol:
        σ_new = K(σ, model)
        error = np.max(np.abs(σ - σ_new))
        i += 1
        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")
        σ = σ_new

    if error > tol:
        print("Failed to converge!")
    elif verbose:
        print(f"\nConverged in {i} iterations.")

    return σ_new
```

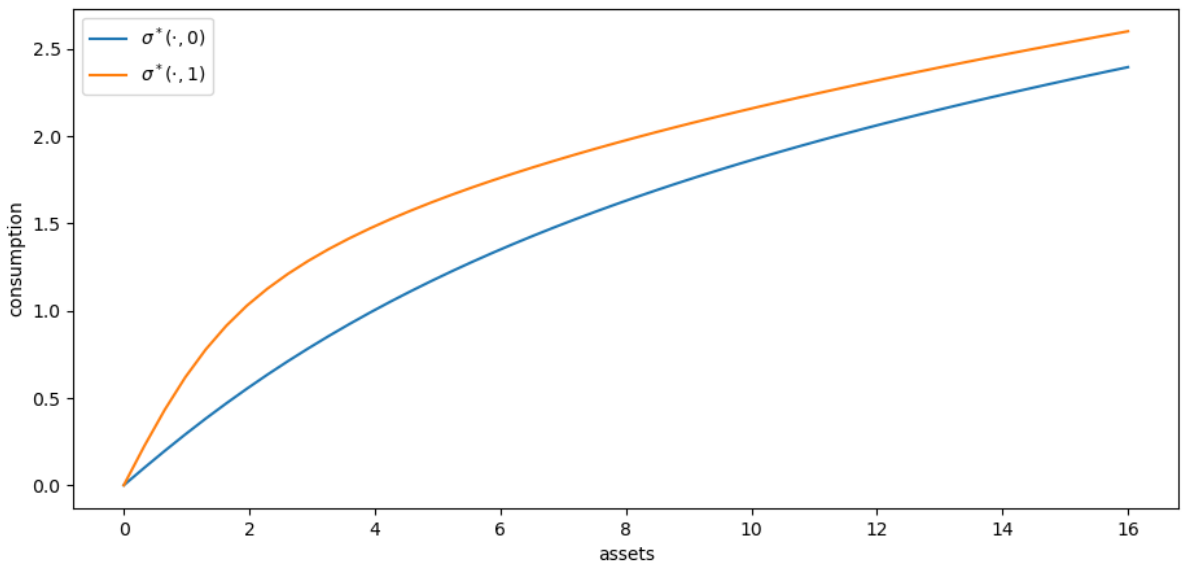
Let's carry this out using the default parameters of the `IFP` class:

```
ifp = IFP()

# Set up initial consumption policy of consuming all assets at all z
z_size = len(ifp.P)
a_grid = ifp.asset_grid
a_size = len(a_grid)
σ_init = np.repeat(a_grid.reshape(a_size, 1), z_size, axis=1)

σ_star = solve_model_time_iter(ifp, σ_init)
```

```
Error at iteration 25 is 0.011629589188247191.
Error at iteration 50 is 0.0003857183099467143.

Converged in 60 iterations.
```

Here's a plot of the resulting policy for each exogeneous state z.

```
fig, ax = plt.subplots()
for z in range(z_size):
    label = rf'$\sigma^*(\cdot, {z})$'
    ax.plot(a_grid, σ_star[:, z], label=label)
ax.set(xlabel='assets', ylabel='consumption')
ax.legend()
plt.show()
```



The following exercises walk you through several applications where policy functions are computed.

30.4.1 A Sanity Check

One way to check our results is to

- set labor income to zero in each state and
- set the gross interest rate R to unity.

In this case, our income fluctuation problem is just a cake eating problem.

We know that, in this case, the value function and optimal consumption policy are given by

```
def c_star(x, β, γ):

    return (1 - β ** (1/γ)) * x


def v_star(x, β, γ):
```

(continues on next page)

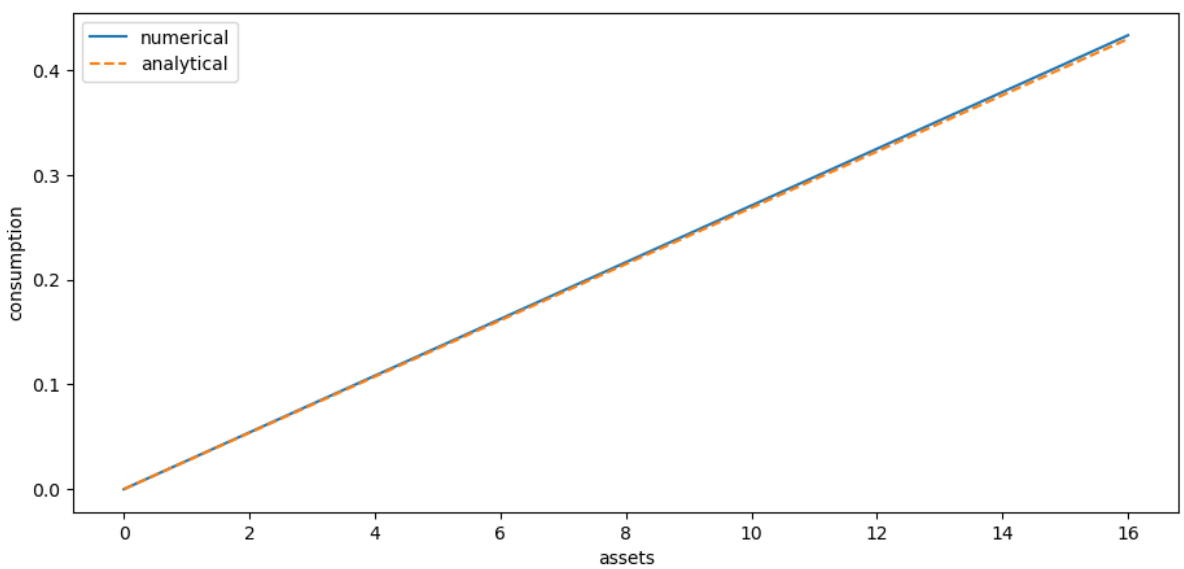
(continued from previous page)

```
    return (1 - β**(1 / γ))**(-γ) * (x**(1-γ) / (1-γ))
```

Let's see if we match up:

```
ifp_cake_eating = IFP(r=0.0, γ=(0.0, 0.0))

σ_star = solve_model_time_iter(ifp_cake_eating, σ_init)

fig, ax = plt.subplots()
ax.plot(a_grid, σ_star[:, 0], label='numerical')
ax.plot(a_grid, c_star(a_grid, ifp.β, ifp.γ), '--', label='analytical')

ax.set(xlabel='assets', ylabel='consumption')
ax.legend()

plt.show()
```

```
Error at iteration 25 is 0.023332272630545603.
Error at iteration 50 is 0.005301238424249788.
Error at iteration 75 is 0.0019706324625650695.
Error at iteration 100 is 0.0008675521337955794.
Error at iteration 125 is 0.00041073542212249903.
Error at iteration 150 is 0.00020120334010509389.
Error at iteration 175 is 0.00010021430795081887.

Converged in 176 iterations.
```



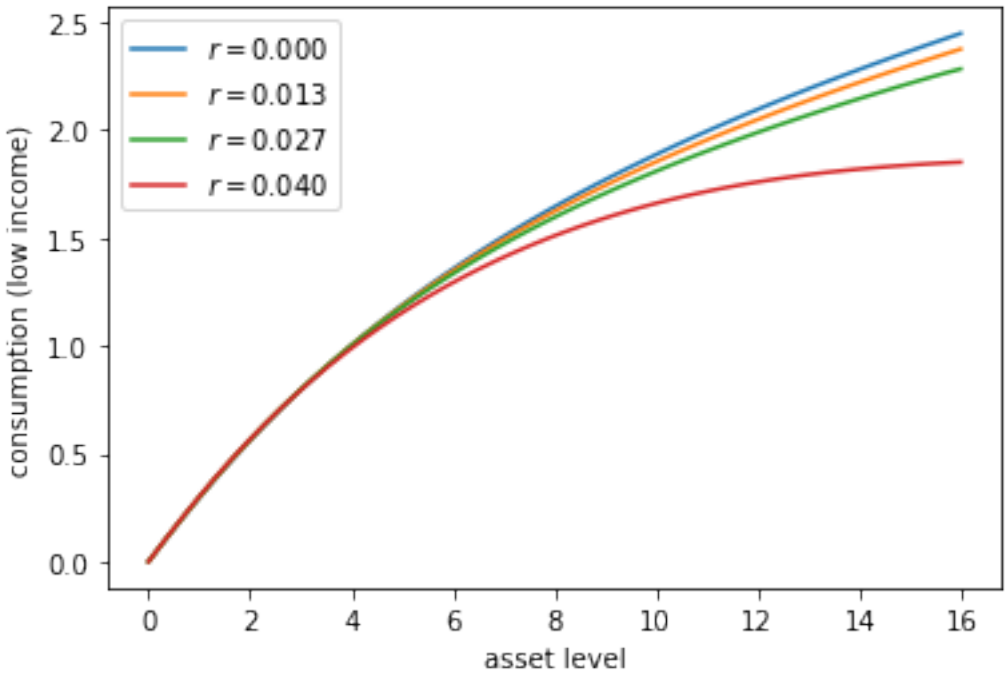
Success!

30.5 Exercises

Exercise 30.5.1

Let's consider how the interest rate affects consumption.

Reproduce the following figure, which shows (approximately) optimal consumption policies for different interest rates



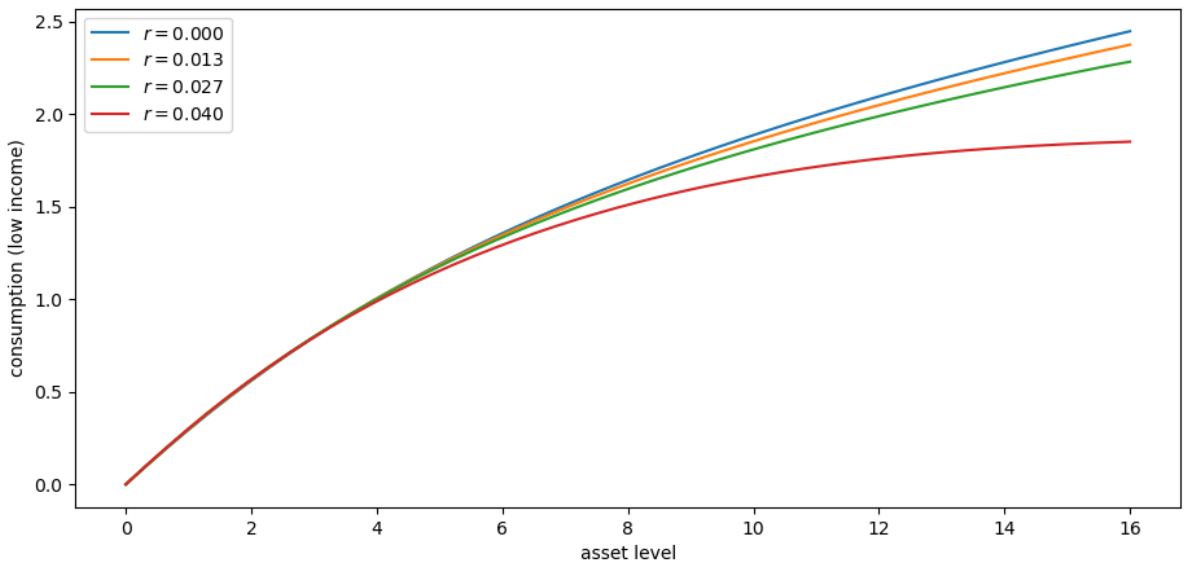
- Other than `r`, all parameters are at their default values.
- `r` steps through `np.linspace(0, 0.04, 4)`.
- Consumption is plotted against assets for income shock fixed at the smallest value.

The figure shows that higher interest rates boost savings and hence suppress consumption.

Solution to Exercise 30.5.1

Here's one solution:

```
r_vals = np.linspace(0, 0.04, 4)

fig, ax = plt.subplots()
for r_val in r_vals:
    ifp = IFP(r=r_val)
    σ_star = solve_model_time_iter(ifp, σ_init, verbose=False)
    ax.plot(ifp.asset_grid, σ_star[:, 0], label=f'$r = {r_val:.3f}$')

ax.set(xlabel='asset level', ylabel='consumption (low income)')
ax.legend()
plt.show()
```

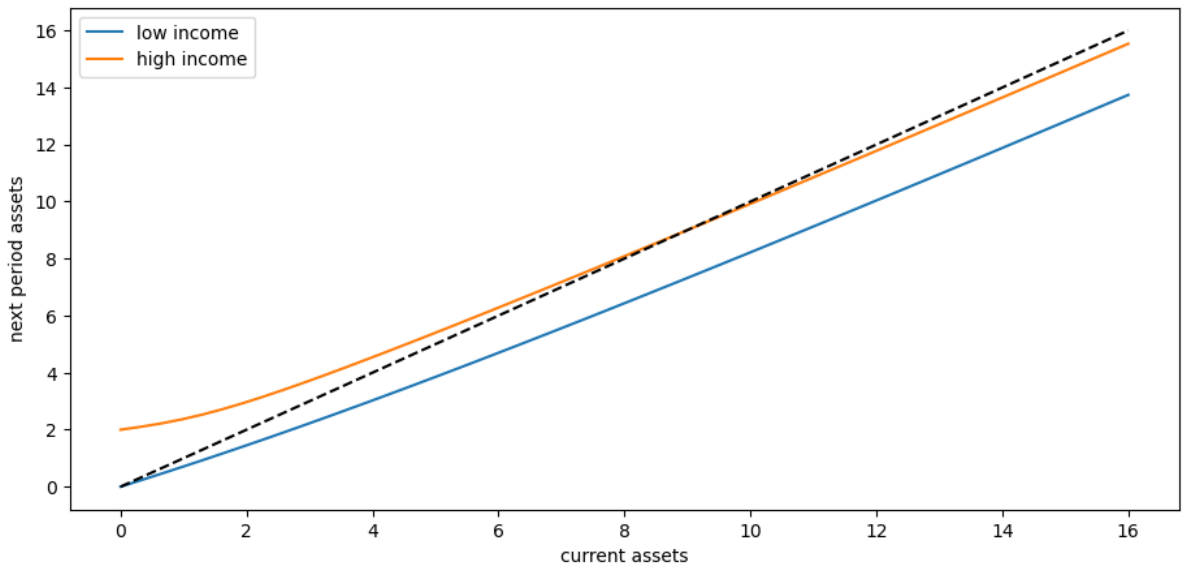


Exercise 30.5.2

Now let's consider the long run asset levels held by households under the default parameters.

The following figure is a 45 degree diagram showing the law of motion for assets when consumption is optimal

```
ifp = IFP()

σ_star = solve_model_time_iter(ifp, σ_init, verbose=False)
a = ifp.asset_grid
R, y = ifp.R, ifp.y

fig, ax = plt.subplots()
for z, lb in zip((0, 1), ('low income', 'high income')):
    ax.plot(a, R * (a - σ_star[:, z]) + y[z] , label=lb)

ax.plot(a, a, 'k--')
ax.set(xlabel='current assets', ylabel='next period assets')

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



The unbroken lines show the update function for assets at each z, which is

$$a \mapsto R(a - \sigma^*(a, z)) + y(z)$$

The dashed line is the 45 degree line.

We can see from the figure that the dynamics will be stable — assets do not diverge even in the highest state.

In fact there is a unique stationary distribution of assets that we can calculate by simulation

- Can be proved via theorem 2 of [HP92].
- It represents the long run dispersion of assets across households when households have idiosyncratic shocks.

Ergodicity is valid here, so stationary probabilities can be calculated by averaging over a single long time series.

Hence to approximate the stationary distribution we can simulate a long time series for assets and histogram it.

Your task is to generate such a histogram.

- Use a single time series $\{a_t\}$ of length 500,000.
- Given the length of this time series, the initial condition (a_0, z_0) will not matter.
- You might find it helpful to use the `MarkovChain` class from `quantecon`.

Solution to Exercise 30.5.2

First we write a function to compute a long asset series.

```
def compute_asset_series(ifp, T=500_000, seed=1234):
    """
    Simulates a time series of length T for assets, given optimal
    savings behavior.

    ifp is an instance of IFP
    """
    P, y, R = ifp.P, ifp.y, ifp.R  # Simplify names
```

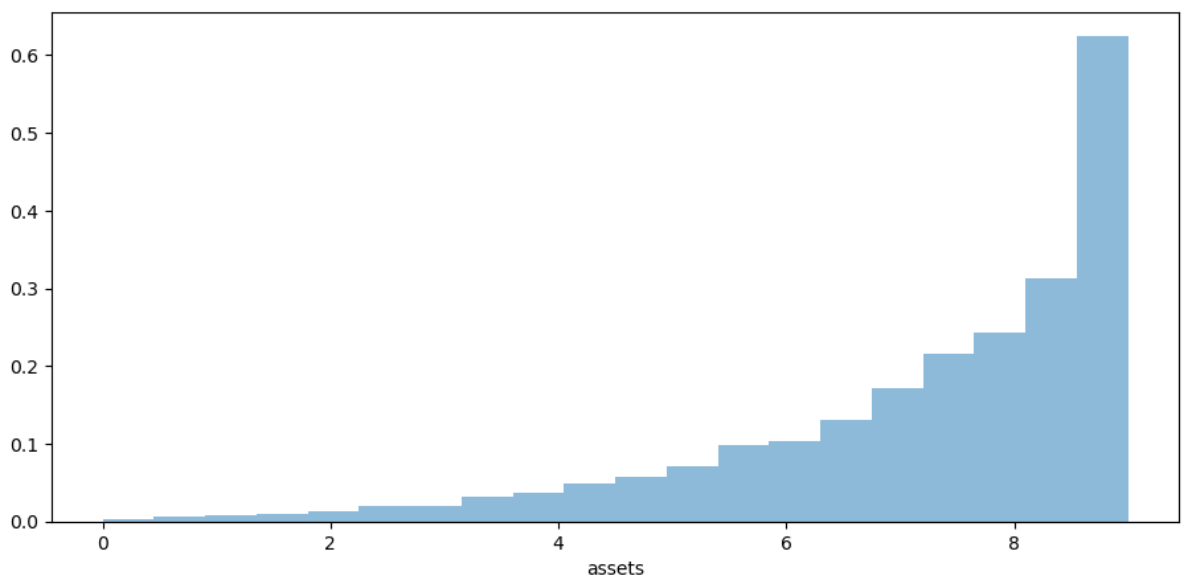
(continues on next page)

(continued from previous page)

```
    # Solve for the optimal policy
    σ_star = solve_model_time_iter(ifp, σ_init, verbose=False)
    σ = lambda a, z: interp(ifp.asset_grid, σ_star[:, z], a)

    # Simulate the exogeneous state process
    mc = MarkovChain(P)
    z_seq = mc.simulate(T, random_state=seed)

    # Simulate the asset path
    a = np.zeros(T+1)
    for t in range(T):
        z = z_seq[t]
        a[t+1] = R * (a[t] - σ(a[t], z)) + y[z]
    return a
```

Now we call the function, generate the series and then histogram it:

```
ifp = IFP()
a = compute_asset_series(ifp)

fig, ax = plt.subplots()
ax.hist(a, bins=20, alpha=0.5, density=True)
ax.set(xlabel='assets')
plt.show()
```



The shape of the asset distribution is unrealistic.

Here it is left skewed when in reality it has a long right tail.

In a *subsequent lecture* we will rectify this by adding more realistic features to the model.

Exercise 30.5.3

Following on from exercises 1 and 2, let's look at how savings and aggregate asset holdings vary with the interest rate

Note: [LS18] section 18.6 can be consulted for more background on the topic treated in this exercise.

For a given parameterization of the model, the mean of the stationary distribution of assets can be interpreted as aggregate capital in an economy with a unit mass of *ex-ante* identical households facing idiosyncratic shocks.

Your task is to investigate how this measure of aggregate capital varies with the interest rate.

Following tradition, put the price (i.e., interest rate) on the vertical axis.

On the horizontal axis put aggregate capital, computed as the mean of the stationary distribution given the interest rate.

Solution to Exercise 30.5.3

Here's one solution

```
M = 25
r_vals = np.linspace(0, 0.02, M)
fig, ax = plt.subplots()

asset_mean = []
for r in r_vals:
    print(f'Solving model at r = {r}')
    ifp = IFP(r=r)
    mean = np.mean(compute_asset_series(ifp, T=250_000))
    asset_mean.append(mean)
ax.plot(asset_mean, r_vals)

ax.set(xlabel='capital', ylabel='interest rate')

plt.show()
```

```
Solving model at r = 0.0
```

```
Solving model at r = 0.0008333333333333334
```

```
Solving model at r = 0.0016666666666666668
```

```
Solving model at r = 0.0025
```

```
Solving model at r = 0.0033333333333333335
```

```
Solving model at r = 0.004166666666666667
```

```
Solving model at r = 0.005
```

```
Solving model at r = 0.005833333333333334
```

```
Solving model at r = 0.006666666666666667
```

```
Solving model at r = 0.007500000000000001
```

```
Solving model at r = 0.008333333333333333
```

```
Solving model at r = 0.009166666666666667
```

```
Solving model at r = 0.01
```

```
Solving model at r = 0.010833333333333334
```

```
Solving model at r = 0.011666666666666667
```

```
Solving model at r = 0.0125
```

```
Solving model at r = 0.013333333333333334
```

```
Solving model at r = 0.014166666666666668
```

```
Solving model at r = 0.015000000000000001
```

```
Solving model at r = 0.015833333333333335
```

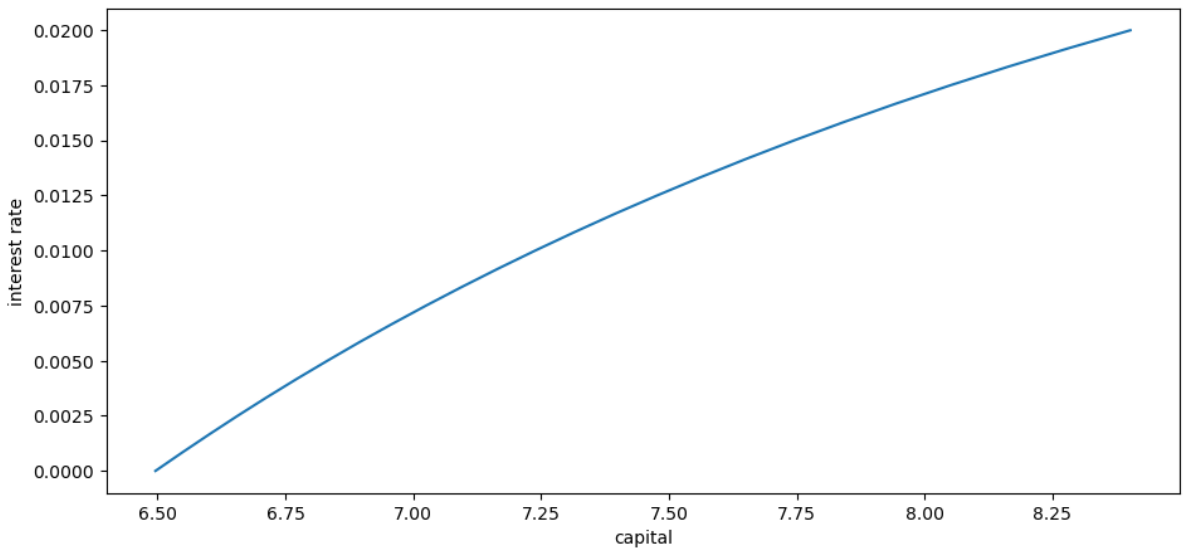
```
Solving model at r = 0.016666666666666666
```

```
Solving model at r = 0.0175
```

```
Solving model at r = 0.018333333333333333
```

```
Solving model at r = 0.01916666666666667
```

```
Solving model at r = 0.02
```



As expected, aggregate savings increases with the interest rate.

CHAPTER

THIRTYONE

THE INCOME FLUCTUATION PROBLEM II: STOCHASTIC RETURNS ON ASSETS

Contents

- *The Income Fluctuation Problem II: Stochastic Returns on Assets*
  - *Overview*
  - *The Savings Problem*
  - *Solution Algorithm*
  - *Implementation*
  - *Exercises*

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

```
!pip install quantecon
!pip install interpolation
```

31.1 Overview

In this lecture, we continue our study of the *income fluctuation problem*.

While the interest rate was previously taken to be fixed, we now allow returns on assets to be state-dependent.

This matches the fact that most households with a positive level of assets face some capital income risk.

It has been argued that modeling capital income risk is essential for understanding the joint distribution of income and wealth (see, e.g., [BBZ15] or [ST19]).

Theoretical properties of the household savings model presented here are analyzed in detail in [MST20].

In terms of computation, we use a combination of time iteration and the endogenous grid method to solve the model quickly and accurately.

We require the following imports:

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

(continues on next page)

(continued from previous page)

```
from numba import njit, float64
from numba.experimental import jitclass
from quantecon import MarkovChain
```

```
/home/runner/miniconda3/envs/quantecon/lib/python3.11/site-packages/numba/core/
↪decorators.py:262: NumbaDeprecationWarning: numba.generated_jit is deprecated.␣
↪Please see the documentation at: https://numba.readthedocs.io/en/stable/
↪reference/deprecation.html#deprecation-of-generated-jit for more information and␣
↪advice on a suitable replacement.
  warnings.warn(msg, NumbaDeprecationWarning)
```

31.2 The Savings Problem

In this section we review the household problem and optimality results.

31.2.1 Set Up

A household chooses a consumption-asset path $\{(c_t, a_t)\}$ to maximize

$$\mathbb{E}\left\{\sum_{t=0}^{\infty} \beta^t u(c_t)\right\} \tag{31.1}$$

subject to

$$a_{t+1} = R_{t+1}(a_t - c_t) + Y_{t+1} \text{ and } 0 \leq c_t \leq a_t, \tag{31.2}$$

with initial condition $(a_0, Z_0) = (a, z)$ treated as given.

Note that $\{R_t\}_{t\geq 1}$, the gross rate of return on wealth, is allowed to be stochastic.

The sequence $\{Y_t\}_{t\geq 1}$ is non-financial income.

The stochastic components of the problem obey

$$R_t = R(Z_t, \zeta_t) \quad \text{and} \quad Y_t = Y(Z_t, \eta_t), \tag{31.3}$$

where

- the maps R and Y are time-invariant nonnegative functions,
- the innovation processes $\{\zeta_t\}$ and $\{\eta_t\}$ are IID and independent of each other, and
- $\{Z_t\}_{t\geq 0}$ is an irreducible time-homogeneous Markov chain on a finite set Z

Let P represent the Markov matrix for the chain $\{Z_t\}_{t\geq 0}$.

Our assumptions on preferences are the same as our *previous lecture* on the income fluctuation problem.

As before, $\mathbb{E}_z \hat{X}$ means expectation of next period value $\hat{X}$ given current value $Z = z$.

31.2.2 Assumptions

We need restrictions to ensure that the objective (31.1) is finite and the solution methods described below converge.

We also need to ensure that the present discounted value of wealth does not grow too quickly.

When $\{R_t\}$ was constant we required that $\beta R < 1$.

Now it is stochastic, we require that

$$\beta G_R < 1, \quad \text{where} \quad G_R := \lim_{n\to\infty} \left(\mathbb{E} \prod_{t=1}^{n} R_t \right)^{1/n} \tag{31.4}$$

Notice that, when $\{R_t\}$ takes some constant value R, this reduces to the previous restriction $\beta R < 1$

The value G_R can be thought of as the long run (geometric) average gross rate of return.

More intuition behind (31.4) is provided in [MST20].

Discussion on how to check it is given below.

Finally, we impose some routine technical restrictions on non-financial income.

$$\mathbb{E}\, Y_t < \infty \text{ and } \mathbb{E}\, u'(Y_t) < \infty$$

One relatively simple setting where all these restrictions are satisfied is the IID and CRRA environment of [BBZ15].

31.2.3 Optimality

Let the class of candidate consumption policies $\mathcal{C}$ be defined *as before*.

In [MST20] it is shown that, under the stated assumptions,

- any $\sigma \in \mathcal{C}$ satisfying the Euler equation is an optimal policy and
- exactly one such policy exists in $\mathcal{C}$.

In the present setting, the Euler equation takes the form

$$(u' \circ \sigma)(a, z) = \max \left\{ \beta\, \mathbb{E}_z\, \hat{R}\, (u' \circ \sigma)[\hat{R}(a - \sigma(a, z)) + \hat{Y},\, \hat{Z}],\, u'(a) \right\} \tag{31.5}$$

(Intuition and derivation are similar to our *earlier lecture* on the income fluctuation problem.)

We again solve the Euler equation using time iteration, iterating with a Coleman–Reffett operator K defined to match the Euler equation (31.5).

31.3 Solution Algorithm

31.3.1 A Time Iteration Operator

Our definition of the candidate class $\sigma \in \mathcal{C}$ of consumption policies is the same as in our *earlier lecture* on the income fluctuation problem.

For fixed $\sigma \in \mathcal{C}$ and $(a, z) \in \mathbf{S}$, the value $K\sigma(a, z)$ of the function $K\sigma$ at (a, z) is defined as the $\xi \in (0, a]$ that solves

$$u'(\xi) = \max \left\{ \beta\, \mathbb{E}_z\, \hat{R}\, (u' \circ \sigma)[\hat{R}(a - \xi) + \hat{Y},\, \hat{Z}],\, u'(a) \right\} \tag{31.6}$$

The idea behind K is that, as can be seen from the definitions, $\sigma \in \mathcal{C}$ satisfies the Euler equation if and only if $K\sigma(a, z) = \sigma(a, z)$ for all $(a, z) \in \mathbf{S}$.

This means that fixed points of K in $\mathcal{C}$ and optimal consumption policies exactly coincide (see [MST20] for more details).

31.3.2 Convergence Properties

As before, we pair $\mathcal{C}$ with the distance

$$\rho(c, d) := \sup_{(a,z)\in\mathbf{S}} \left|(u' \circ c)(a, z) - (u' \circ d)(a, z)\right|,$$

It can be shown that

1. $(\mathcal{C}, \rho)$ is a complete metric space,
2. there exists an integer n such that K^n is a contraction mapping on $(\mathcal{C}, \rho)$, and
3. The unique fixed point of K in $\mathcal{C}$ is the unique optimal policy in $\mathcal{C}$.

We now have a clear path to successfully approximating the optimal policy: choose some $\sigma \in \mathcal{C}$ and then iterate with K until convergence (as measured by the distance ρ).

31.3.3 Using an Endogenous Grid

In the study of that model we found that it was possible to further accelerate time iteration via the *endogenous grid method*.

We will use the same method here.

The methodology is the same as it was for the optimal growth model, with the minor exception that we need to remember that consumption is not always interior.

In particular, optimal consumption can be equal to assets when the level of assets is low.

Finding Optimal Consumption

The endogenous grid method (EGM) calls for us to take a grid of *savings* values s_i, where each such s is interpreted as $s = a - c$.

For the lowest grid point we take $s_0 = 0$.

For the corresponding a_0, c_0 pair we have $a_0 = c_0$.

This happens close to the origin, where assets are low and the household consumes all that it can.

Although there are many solutions, the one we take is $a_0 = c_0 = 0$, which pins down the policy at the origin, aiding interpolation.

For $s > 0$, we have, by definition, $c < a$, and hence consumption is interior.

Hence the max component of (31.5) drops out, and we solve for

$$c_i = (u')^{-1} \left\{ \beta \, \mathbb{E}_z \hat{R} (u' \circ \sigma) \left[\hat{R} s_i + \hat{Y}, \, \hat{Z} \right] \right\} \tag{31.7}$$

at each s_i.

Iterating

Once we have the pairs $\{s_i, c_i\}$, the endogenous asset grid is obtained by $a_i = c_i + s_i$.

Also, we held $z \in \mathsf{Z}$ in the discussion above so we can pair it with a_i.

An approximation of the policy $(a, z) \mapsto \sigma(a, z)$ can be obtained by interpolating $\{a_i, c_i\}$ at each z.

In what follows, we use linear interpolation.

31.3.4 Testing the Assumptions

Convergence of time iteration is dependent on the condition $\beta G_R < 1$ being satisfied.

One can check this using the fact that G_R is equal to the spectral radius of the matrix L defined by

$$L(z, \hat{z}) := P(z, \hat{z}) \int R(\hat{z}, x)\phi(x)dx$$

This identity is proved in [MST20], where ϕ is the density of the innovation ζ_t to returns on assets.

(Remember that Z is a finite set, so this expression defines a matrix.)

Checking the condition is even easier when $\{R_t\}$ is IID.

In that case, it is clear from the definition of G_R that G_R is just $\mathbb{E}R_t$.

We test the condition $\beta \mathbb{E}R_t < 1$ in the code below.

31.4 Implementation

We will assume that $R_t = \exp(a_r \zeta_t + b_r)$ where a_r, b_r are constants and $\{\zeta_t\}$ is IID standard normal.

We allow labor income to be correlated, with

$$Y_t = \exp(a_y \eta_t + Z_t b_y)$$

where $\{\eta_t\}$ is also IID standard normal and $\{Z_t\}$ is a Markov chain taking values in $\{0, 1\}$.

```
ifp_data = [
    ('γ', float64),              # utility parameter
    ('β', float64),              # discount factor
    ('P', float64[:, :]),        # transition probs for z_t
    ('a_r', float64),            # scale parameter for R_t
    ('b_r', float64),            # additive parameter for R_t
    ('a_y', float64),            # scale parameter for Y_t
    ('b_y', float64),            # additive parameter for Y_t
    ('s_grid', float64[:]),      # Grid over savings
    ('η_draws', float64[:]),     # Draws of innovation η for MC
    ('ζ_draws', float64[:])      # Draws of innovation ζ for MC
]
```

```
@jitclass(ifp_data)
class IFP:
    """
    A class that stores primitives for the income fluctuation
```

(continues on next page)

(continued from previous page)

```
    problem.
    """

    def __init__(self,
                 γ=1.5,
                 β=0.96,
                 P=np.array([(0.9, 0.1),
                             (0.1, 0.9)]),
                 a_r=0.1,
                 b_r=0.0,
                 a_y=0.2,
                 b_y=0.5,
                 shock_draw_size=50,
                 grid_max=10,
                 grid_size=100,
                 seed=1234):

        np.random.seed(seed)  # arbitrary seed

        self.P, self.γ, self.β = P, γ, β
        self.a_r, self.b_r, self.a_y, self.b_y = a_r, b_r, a_y, b_y
        self.η_draws = np.random.randn(shock_draw_size)
        self.ζ_draws = np.random.randn(shock_draw_size)
        self.s_grid = np.linspace(0, grid_max, grid_size)

        # Test stability assuming {R_t} is IID and adopts the lognormal
        # specification given below.  The test is then β E R_t < 1.
        ER = np.exp(b_r + a_r**2 / 2)
        assert β * ER < 1, "Stability condition failed."

    # Marginal utility
    def u_prime(self, c):
        return c**(-self.γ)

    # Inverse of marginal utility
    def u_prime_inv(self, c):
        return c**(-1/self.γ)

    def R(self, z, ζ):
        return np.exp(self.a_r * ζ + self.b_r)

    def Y(self, z, η):
        return np.exp(self.a_y * η + (z * self.b_y))
```

Here's the Coleman-Reffett operator based on EGM:

```
@njit
def K(a_in, σ_in, ifp):
    """
    The Coleman--Reffett operator for the income fluctuation problem,
    using the endogenous grid method.

        * ifp is an instance of IFP
        * a_in[i, z] is an asset grid
        * σ_in[i, z] is consumption at a_in[i, z]
    """
```

(continues on next page)

(continued from previous page)

```
    # Simplify names
    u_prime, u_prime_inv = ifp.u_prime, ifp.u_prime_inv
    R, Y, P, β = ifp.R, ifp.Y, ifp.P, ifp.β
    s_grid, η_draws, ζ_draws = ifp.s_grid, ifp.η_draws, ifp.ζ_draws
    n = len(P)

    # Create consumption function by linear interpolation
    σ = lambda a, z: interp(a_in[:, z], σ_in[:, z], a)

    # Allocate memory
    σ_out = np.empty_like(σ_in)

    # Obtain c_i at each s_i, z, store in σ_out[i, z], computing
    # the expectation term by Monte Carlo
    for i, s in enumerate(s_grid):
        for z in range(n):
            # Compute expectation
            Ez = 0.0
            for z_hat in range(n):
                for η in ifp.η_draws:
                    for ζ in ifp.ζ_draws:
                        R_hat = R(z_hat, ζ)
                        Y_hat = Y(z_hat, η)
                        U = u_prime(σ(R_hat * s + Y_hat, z_hat))
                        Ez += R_hat * U * P[z, z_hat]
            Ez = Ez / (len(η_draws) * len(ζ_draws))
            σ_out[i, z] =  u_prime_inv(β * Ez)

    # Calculate endogenous asset grid
    a_out = np.empty_like(σ_out)
    for z in range(n):
        a_out[:, z] = s_grid + σ_out[:, z]

    # Fixing a consumption-asset pair at (0, 0) improves interpolation
    σ_out[0, :] = 0
    a_out[0, :] = 0

    return a_out, σ_out
```

The next function solves for an approximation of the optimal consumption policy via time iteration.

```
def solve_model_time_iter(model,        # Class with model information
                          a_vec,        # Initial condition for assets
                          σ_vec,        # Initial condition for consumption
                          tol=1e-4,
                          max_iter=1000,
                          verbose=True,
                          print_skip=25):

    # Set up loop
    i = 0
    error = tol + 1

    while i < max_iter and error > tol:
        a_new, σ_new = K(a_vec, σ_vec, model)
```

(continues on next page)

(continued from previous page)

```
        error = np.max(np.abs(σ_vec - σ_new))
        i += 1
        if verbose and i % print_skip == 0:
            print(f"Error at iteration {i} is {error}.")
        a_vec, σ_vec = np.copy(a_new), np.copy(σ_new)

    if error > tol:
        print("Failed to converge!")
    elif verbose:
        print(f"\nConverged in {i} iterations.")

    return a_new, σ_new
```

Now we are ready to create an instance at the default parameters.

```
ifp = IFP()
```

Next we set up an initial condition, which corresponds to consuming all assets.

```
# Initial guess of σ = consume all assets
k = len(ifp.s_grid)
n = len(ifp.P)
σ_init = np.empty((k, n))
for z in range(n):
    σ_init[:, z] = ifp.s_grid
a_init = np.copy(σ_init)
```

Let's generate an approximation solution.

```
a_star, σ_star = solve_model_time_iter(ifp, a_init, σ_init, print_skip=5)
```

```
Error at iteration 5 is 0.5081944529506557.
```

```
Error at iteration 10 is 0.1057246950930697.
```

```
Error at iteration 15 is 0.03658262202883744.
```

```
Error at iteration 20 is 0.013936729965906114.
```

```
Error at iteration 25 is 0.005292165269711546.
```

```
Error at iteration 30 is 0.0019748126990770665.
```

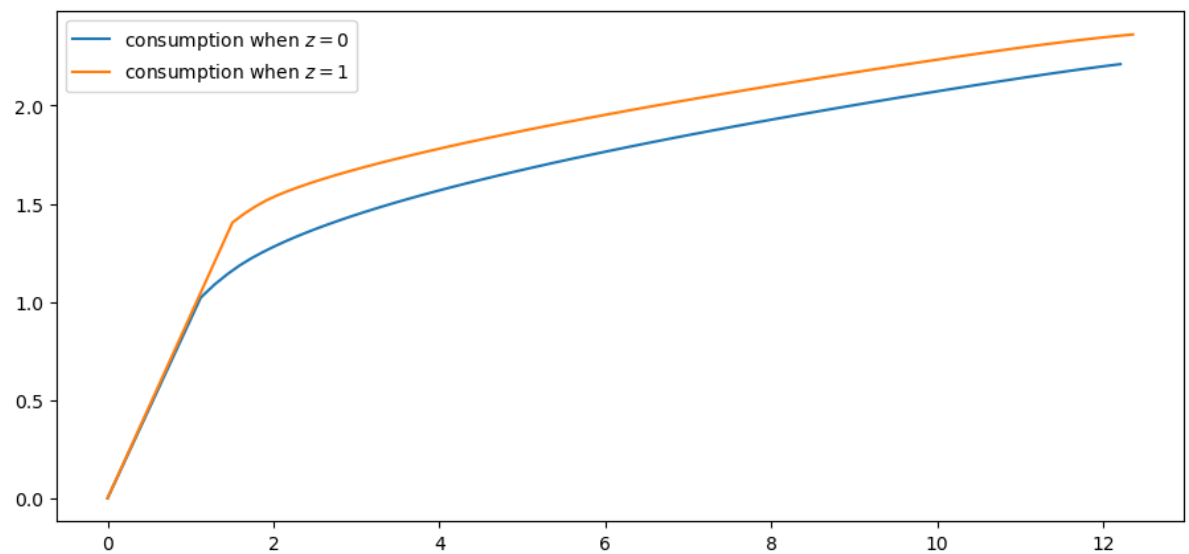
```
Error at iteration 35 is 0.0007219210463285108.
```

```
Error at iteration 40 is 0.0002590544496094971.
```

```
Error at iteration 45 is 9.163966595426842e-05.

Converged in 45 iterations.
```

Here's a plot of the resulting consumption policy.

```
fig, ax = plt.subplots()
for z in range(len(ifp.P)):
    ax.plot(a_star[:, z], σ_star[:, z], label=f"consumption when $z={z}$")

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



Notice that we consume all assets in the lower range of the asset space.

This is because we anticipate income Y_{t+1} tomorrow, which makes the need to save less urgent.

Can you explain why consuming all assets ends earlier (for lower values of assets) when $z = 0$?

31.4.1 Law of Motion

Let's try to get some idea of what will happen to assets over the long run under this consumption policy.

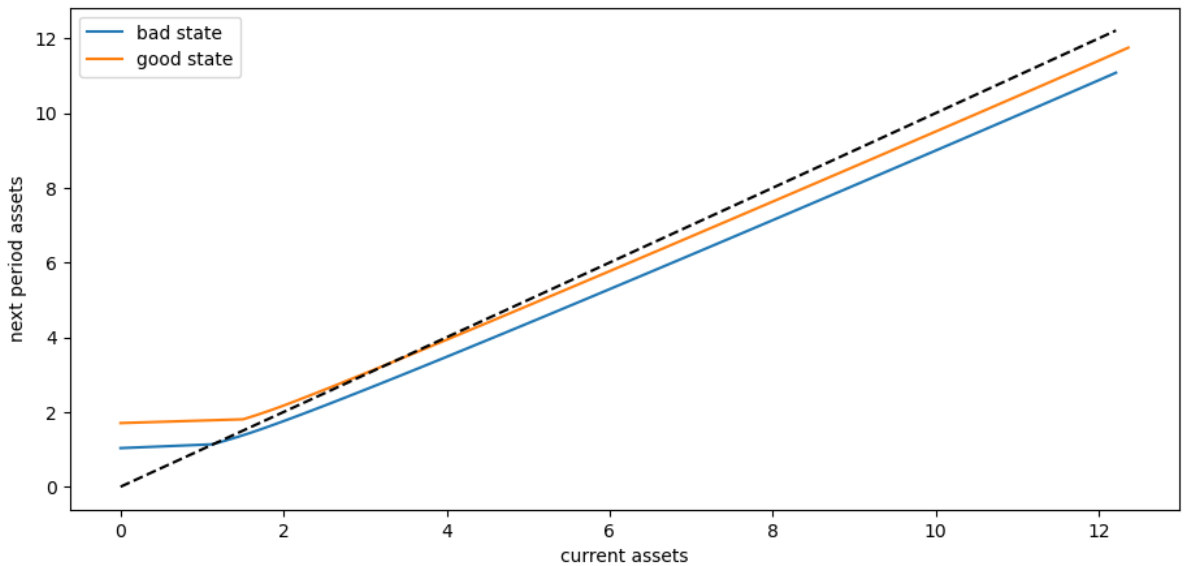
As with our *earlier lecture* on the income fluctuation problem, we begin by producing a 45 degree diagram showing the law of motion for assets

```
# Good and bad state mean labor income
Y_mean = [np.mean(ifp.Y(z, ifp.η_draws)) for z in (0, 1)]
# Mean returns
R_mean = np.mean(ifp.R(z, ifp.ζ_draws))

a = a_star
fig, ax = plt.subplots()
for z, lb in zip((0, 1), ('bad state', 'good state')):
    ax.plot(a[:, z], R_mean * (a[:, z] - σ_star[:, z]) + Y_mean[z] , label=lb)
```

(continues on next page)

(continued from previous page)

```
ax.plot(a[:, 0], a[:, 0], 'k--')
ax.set(xlabel='current assets', ylabel='next period assets')

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



The unbroken lines represent, for each z, an average update function for assets, given by

$$a \mapsto \bar{R}(a - \sigma^*(a, z)) + \bar{Y}(z)$$

Here

- $\bar{R} = \mathbb{E}R_t$, which is mean returns and
- $\bar{Y}(z) = \mathbb{E}_z Y(z, \eta_t)$, which is mean labor income in state z.

The dashed line is the 45 degree line.

We can see from the figure that the dynamics will be stable — assets do not diverge even in the highest state.

31.5 Exercises

Exercise 31.5.1

Let's repeat our *earlier exercise* on the long-run cross sectional distribution of assets.

In that exercise, we used a relatively simple income fluctuation model.

In the solution, we found the shape of the asset distribution to be unrealistic.

In particular, we failed to match the long right tail of the wealth distribution.

Your task is to try again, repeating the exercise, but now with our more sophisticated model.

Use the default parameters.

Solution to Exercise 31.5.1

First we write a function to compute a long asset series.

Because we want to JIT-compile the function, we code the solution in a way that breaks some rules on good programming style.

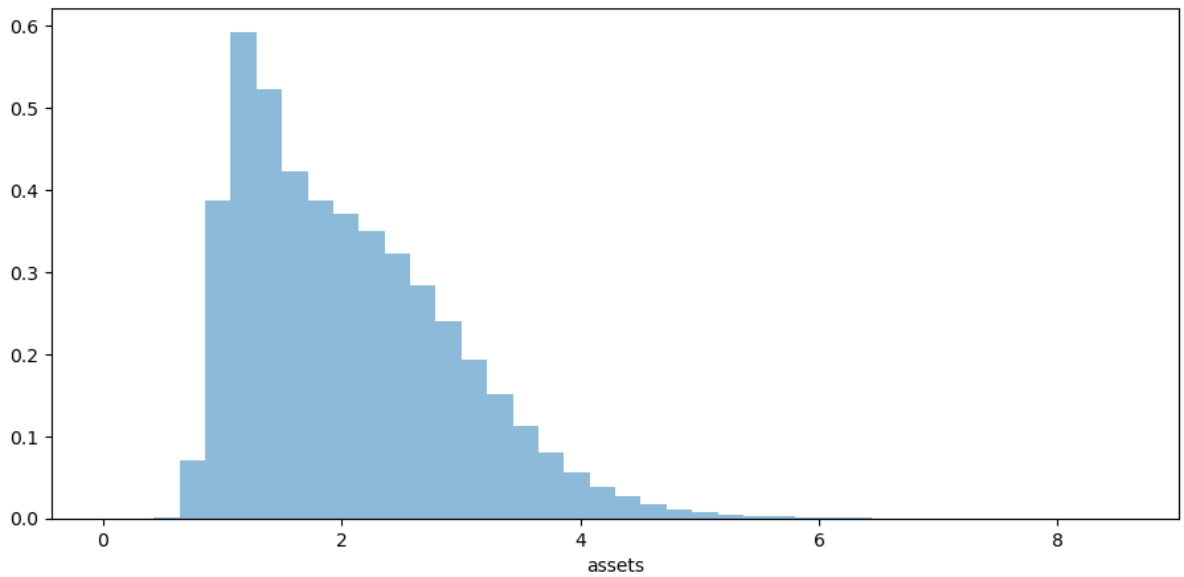
For example, we will pass in the solutions `a_star, σ_star` along with `ifp`, even though it would be more natural to just pass in `ifp` and then solve inside the function.

The reason we do this is that `solve_model_time_iter` is not JIT-compiled.

```
@njit
def compute_asset_series(ifp, a_star, σ_star, z_seq, T=500_000):
    """
    Simulates a time series of length T for assets, given optimal
    savings behavior.

        * ifp is an instance of IFP
        * a_star is the endogenous grid solution
        * σ_star is optimal consumption on the grid
        * z_seq is a time path for {Z_t}

    """

    # Create consumption function by linear interpolation
    σ = lambda a, z: interp(a_star[:, z], σ_star[:, z], a)

    # Simulate the asset path
    a = np.zeros(T+1)
    for t in range(T):
        z = z_seq[t]
        ζ, η = np.random.randn(), np.random.randn()
        R = ifp.R(z, ζ)
        Y = ifp.Y(z, η)
        a[t+1] = R * (a[t] - σ(a[t], z)) + Y
    return a
```

Now we call the function, generate the series and then histogram it, using the solutions computed above.

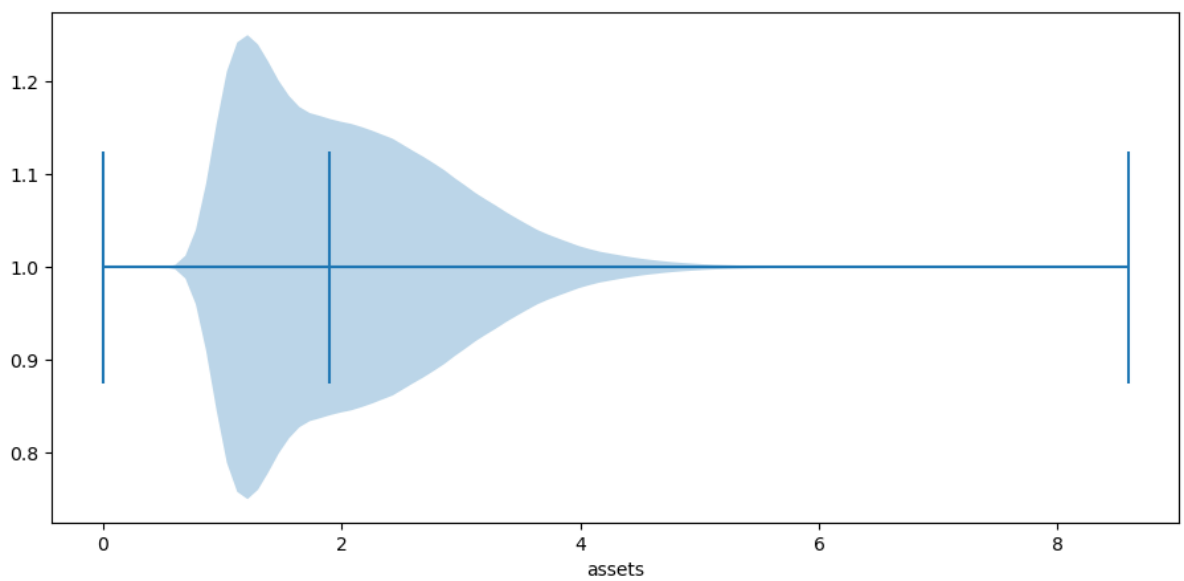
```
T = 1_000_000
mc = MarkovChain(ifp.P)
z_seq = mc.simulate(T, random_state=1234)

a = compute_asset_series(ifp, a_star, σ_star, z_seq, T=T)

fig, ax = plt.subplots()
ax.hist(a, bins=40, alpha=0.5, density=True)
ax.set(xlabel='assets')
plt.show()
```



Now we have managed to successfully replicate the long right tail of the wealth distribution.

Here's another view of this using a horizontal violin plot.

```
fig, ax = plt.subplots()
ax.violinplot(a, vert=False, showmedians=True)
ax.set(xlabel='assets')
plt.show()
```



Part V

Other

CHAPTER

THIRTYTWO

TROUBLESHOOTING

Contents

- *Troubleshooting*
  - *Fixing Your Local Environment*
  - *Reporting an Issue*

This page is for readers experiencing errors when running the code from the lectures.

32.1 Fixing Your Local Environment

The basic assumption of the lectures is that code in a lecture should execute whenever

1. it is executed in a Jupyter notebook and
2. the notebook is running on a machine with the latest version of Anaconda Python.

You have installed Anaconda, haven't you, following the instructions in this lecture?

Assuming that you have, the most common source of problems for our readers is that their Anaconda distribution is not up to date.

Here's a useful article on how to update Anaconda.

Another option is to simply remove Anaconda and reinstall.

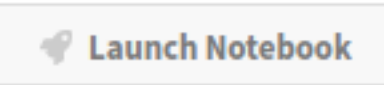
You also need to keep the external code libraries, such as QuantEcon.py up to date.

For this task you can either

- use conda install -y quantecon on the command line, or
- execute !conda install -y quantecon within a Jupyter notebook.

If your local environment is still not working you can do two things.

First, you can use a remote machine instead, by clicking on the Launch Notebook icon available for each lecture



Second, you can report an issue, so we can try to fix your local set up.

We like getting feedback on the lectures so please don't hesitate to get in touch.

32.2 Reporting an Issue

One way to give feedback is to raise an issue through our issue tracker.

Please be as specific as possible. Tell us where the problem is and as much detail about your local set up as you can provide.

Another feedback option is to use our discourse forum.

Finally, you can provide direct feedback to contact@quantecon.org

CHAPTER

THIRTYTHREE

REFERENCES

CHAPTER

THIRTYFOUR

EXECUTION STATISTICS

This table contains the latest execution statistics.

Document	Modified	Method	Run Time (s)	Status
BCG_complete_mkts	2024-01-30 05:09	cache	49.2	✅
BCG_incomplete_mkts	2024-01-30 05:10	cache	80.51	✅
asset_pricing_lph	2024-01-30 05:10	cache	2.24	✅
black_litterman	2024-01-30 05:11	cache	32.55	✅
cake_eating_numerical	2024-01-30 05:11	cache	19.27	✅
cake_eating_problem	2024-01-30 05:11	cache	1.56	✅
career	2024-01-30 05:11	cache	13.59	✅
coleman_policy_iter	2024-01-30 05:12	cache	15.79	✅
egm_policy_iter	2024-01-30 05:12	cache	7.88	✅
ge_arrow	2024-01-30 05:12	cache	1.62	✅
harrison_kreps	2024-01-30 05:12	cache	5.66	✅
ifp	2024-01-30 05:12	cache	25.24	✅
ifp_advanced	2024-01-30 05:13	cache	18.86	✅
intro	2024-01-30 05:13	cache	0.82	✅
inventory_dynamics	2024-01-30 05:13	cache	9.52	✅
jv	2024-01-30 05:13	cache	13.48	✅
kesten_processes	2024-01-30 05:14	cache	57.81	✅
lake_model	2024-01-30 05:14	cache	12.49	✅
lucas_model	2024-01-30 05:14	cache	9.25	✅
markov_asset	2024-01-30 05:14	cache	7.05	✅
mccall_correlated	2024-01-30 05:16	cache	72.03	✅
mccall_fitted_vfi	2024-01-30 05:16	cache	13.37	✅
mccall_model	2024-01-30 05:16	cache	14.28	✅
mccall_model_with_separation	2024-01-30 05:16	cache	8.84	✅
mccall_q	2024-01-30 05:16	cache	16.05	✅
odu	2024-01-30 05:17	cache	50.45	✅
optgrowth	2024-01-30 05:18	cache	46.43	✅
optgrowth_fast	2024-01-30 05:19	cache	26.87	✅
orth_proj	2024-01-30 05:19	cache	1.0	✅
samuelson	2024-01-30 05:19	cache	11.78	✅
sir_model	2024-01-30 05:19	cache	2.78	✅
status	2024-01-30 05:13	cache	0.82	✅
troubleshooting	2024-01-30 05:13	cache	0.82	✅
wealth_dynamics	2024-01-30 05:20	cache	45.58	✅
zreferences	2024-01-30 05:13	cache	0.82	✅

These lectures are built on `linux` instances through `github actions` so are executed using the following hardware specifications

BIBLIOGRAPHY

[Aiy94] S Rao Aiyagari. Uninsured Idiosyncratic Risk and Aggregate Saving. *The Quarterly Journal of Economics*, 109(3):659–684, 1994.

[Axt01] Robert L Axtell. Zipf distribution of us firm sizes. *science*, 293(5536):1818–1820, 2001.

[BB18] Jess Benhabib and Alberto Bisin. Skewed wealth distributions: theory and empirics. *Journal of Economic Literature*, 56(4):1261–91, 2018.

[BBZ15] Jess Benhabib, Alberto Bisin, and Shenghao Zhu. The wealth distribution in bewley economies with capital income risk. *Journal of Economic Theory*, 159:489–515, 2015.

[BCG18] Alberto Bisin, Gian Luca Clementi, and Piero Gottardi. Capital and hedging demand with incomplete markets. Technical Report, NYU and EUI, 2018.

[BL92] Fischer Black and Robert Litterman. Global portfolio optimization. *Financial analysts journal*, 48(5):28–43, 1992.

[BDM+16] Dariusz Buraczewski, Ewa Damek, Thomas Mikosch, and others. *Stochastic models with power-law tails*. Springer, 2016.

[Cap85] Andrew S Caplin. The variability of aggregate demand with (s, s) inventory policies. *Econometrica*, pages 1395–1409, 1985.

[Car06] Christopher D Carroll. The method of endogenous gridpoints for solving dynamic stochastic optimization problems. *Economics Letters*, 91(3):312–320, 2006.

[CR83] Gary Chamberlain and Michael Rothschild. Arbitrage, Factor Structure, and Mean-Variance Analysis on Large Asset Markets. *Econometrica*, 51(5):1281–1304, September 1983. URL: https://ideas.repec.org/a/ecm/emetrp/v51y1983i5p1281-304.html, doi:.

[Coc05] John H. Cochrane. *Asset Pricing: revised edition*. Princeton University Press, Princeton, New Jersey, 2005.

[Col90] Wilbur John Coleman. Solving the Stochastic Growth Model by Policy-Function Iteration. *Journal of Business \\& Economic Statistics*, 8(1):27–29, 1990.

[DFH06] Steven J Davis, R Jason Faberman, and John Haltiwanger. The flow approach to labor markets: new data sources, micro-macro links and the recent downturn. *Journal of Economic Perspectives*, 2006.

[Dea91] Angus Deaton. Saving and Liquidity Constraints. *Econometrica*, 59(5):1221–1248, 1991.

[DH10] Wouter J Den Haan. Comparison of solutions to the incomplete markets model with aggregate uncertainty. *Journal of Economic Dynamics and Control*, 34(1):4–27, 2010.

[Dic75] J Dickey. Bayesian alternatives to the f-test and least-squares estimate in the normal linear model. In S.E. Fienberg and A. Zellner, editors, *Studies in Bayesian econometrics and statistics*, pages 515–554. North-Holland, Amsterdam, 1975.

[DRS89] Timothy Dunne, Mark J Roberts, and Larry Samuelson. The growth and failure of us manufacturing plants. *The Quarterly Journal of Economics*, 104(4):671–698, 1989.

[Eva87] David S Evans. The relationship between firm growth, size, and age: estimates for 100 manufacturing industries. *The Journal of Industrial Economics*, pages 567–581, 1987.

[Gab16] Xavier Gabaix. Power laws in economics: an introduction. *Journal of Economic Perspectives*, 30(1):185–206, 2016.

[Gib31] Robert Gibrat. *Les inégalités économiques: Applications d'une loi nouvelle, la loi de l'effet proportionnel*. PhD thesis, Recueil Sirey, 1931.

[Gor95] Geoffrey J Gordon. Stable function approximation in dynamic programming. In *Machine Learning Proceedings 1995*, pages 261–268. Elsevier, 1995.

[Hal87] Bronwyn H Hall. The relationship between firm size and firm growth in the us manufacturing sector. *The Journal of Industrial Economics*, pages 583–606, 1987.

[HS08] L P Hansen and T J Sargent. *Robustness*. Princeton University Press, 2008.

[HJ91] Lars Peter Hansen and Ravi Jagannathan. Implications of Security Market Data for Models of Dynamic Economies. *Journal of Political Economy*, 99(2):225–262, April 1991. URL: https://ideas.repec.org/a/ucp/jpolec/v99y1991i2p225-62.html, doi:10.1086/261749.

[HR87] Lars Peter Hansen and Scott F Richard. The Role of Conditioning Information in Deducing Testable. *Econometrica*, 55(3):587–613, May 1987.

[HS01] Lars Peter Hansen and Thomas J. Sargent. Robust control and model uncertainty. *American Economic Review*, 91(2):60–66, 2001.

[HK78] J. Michael Harrison and David M. Kreps. Speculative investor behavior in a stock market with heterogeneous expectations. *The Quarterly Journal of Economics*, 92(2):323–336, 1978.

[HK79a] J. Michael Harrison and David M. Kreps. Martingales and arbitrage in multiperiod securities markets. *Journal of Economic Theory*, 20(3):381–408, June 1979.

[HK79b] J. Michael Harrison and David M. Kreps. Martingales and arbitrage in multiperiod securities markets. *Journal of Economic Theory*, 20(3):381–408, June 1979. URL: https://ideas.repec.org/a/eee/jetheo/v20y1979i3p381-408.html, doi:.

[Hop92] Hugo A Hopenhayn. Entry, exit, and firm dynamics in long run equilibrium. *Econometrica: Journal of the Econometric Society*, pages 1127–1150, 1992.

[HP92] Hugo A Hopenhayn and Edward C Prescott. Stochastic Monotonicity and Stationary Distributions for Dynamic Economies. *Econometrica*, 60(6):1387–1406, 1992.

[Hug93] Mark Huggett. The risk-free rate in heterogeneous-agent incomplete-insurance economies. *Journal of Economic Dynamics and Control*, 17(5-6):953–969, 1993.

[Jov79] Boyan Jovanovic. Firm-specific capital and turnover. *Journal of Political Economy*, 87(6):1246–1260, 1979.

[Kam12] Takashi Kamihigashi. Elementary results on solutions to the bellman equation of dynamic programming: existence, uniqueness, and convergence. Technical Report, Kobe University, 2012.

[Kre81] David M. Kreps. Arbitrage and equilibrium in economies with infinitely many commodities. *Journal of Mathematical Economics*, 8(1):15–35, March 1981. URL: https://ideas.repec.org/a/eee/mateco/v8y1981i1p15-35.html, doi:.

[Kuh13] Moritz Kuhn. Recursive Equilibria In An Aiyagari-Style Economy With Permanent Income Shocks. *International Economic Review*, 54:807–835, 2013.

[Lea78] Edward E Leamer. *Specification searches: Ad hoc inference with nonexperimental data*. Volume 53. John Wiley \\& Sons Incorporated, 1978.

[LS18] L Ljungqvist and T J Sargent. *Recursive Macroeconomic Theory*. MIT Press, 4 edition, 2018.

[Luc78] Robert E Lucas, Jr. Asset prices in an exchange economy. *Econometrica: Journal of the Econometric Society*, 46(6):1429–1445, 1978.

[MST20] Qingyin Ma, John Stachurski, and Alexis Akira Toda. The income fluctuation problem and the evolution of wealth. *Journal of Economic Theory*, 187:105003, 2020.

[MdRV10] V Filipe Martins-da-Rocha and Yiannis Vailakis. Existence and Uniqueness of a Fixed Point for Local Contractions. *Econometrica*, 78(3):1127–1141, 2010.

[McC70] J J McCall. Economics of Information and Job Search. *The Quarterly Journal of Economics*, 84(1):113–126, 1970.

[MM58] Franco Modigliani and Merton H. Miller. Corporation finance and the theory of investment. *American Economic Review*, XLVIII(3):261–297, 1958.

[Nea99] Derek Neal. The Complexity of Job Mobility among Young Men. *Journal of Labor Economics*, 17(2):237–261, 1999.

[PalS13] Jenő Pál and John Stachurski. Fitted value function iteration with probability one contractions. *Journal of Economic Dynamics and Control*, 37(1):251–264, 2013.

[Rab02] Guillaume Rabault. When do borrowing constraints bind? Some new results on the income fluctuation problem. *Journal of Economic Dynamics and Control*, 26(2):217–245, 2002.

[Ref96] Kevin L Reffett. Production-based asset pricing in monetary economies with transactions costs. *Economica*, pages 427–443, 1996.

[Rei09] Michael Reiter. Solving heterogeneous-agent models by projection and perturbation. *Journal of Economic Dynamics and Control*, 33(3):649–665, 2009.

[Rom05] Steven Roman. *Advanced linear algebra*. Volume 3. Springer, 2005.

[Ros78] Stephen A Ross. A Simple Approach to the Valuation of Risky Streams. *The Journal of Business*, 51(3):453–475, July 1978. URL: https://ideas.repec.org/a/ucp/jnlbus/v51y1978i3p453-75.html.

[Ros76] Stephen A. Ross. The arbitrage theory of capital asset pricing. *Journal of Economic Theory*, 13(3):341–360, December 1976. URL: https://ideas.repec.org/a/eee/jetheo/v13y1976i3p341-360.html, doi:.

[Sam39] Paul A. Samuelson. Interactions between the multiplier analysis and the principle of acceleration. *Review of Economic Studies*, 21(2):75–78, 1939.

[Sar87] Thomas J Sargent. *Macroeconomic Theory*. Academic Press, New York, 2nd edition, 1987.

[SE77] Jack Schechtman and Vera L S Escudero. Some results on an income fluctuation problem. *Journal of Economic Theory*, 16(2):151–166, 1977.

[Sch14] Jose A. Scheinkman. *Speculation, Trading, and Bubbles*. Columbia University Press, New York, 2014.

[Sta08] John Stachurski. Continuous state dynamic programming via nonexpansive approximation. *Computational Economics*, 31(2):141–160, 2008.

[ST19] John Stachurski and Alexis Akira Toda. An impossibility theorem for wealth in heterogeneous-agent models with limited heterogeneity. *Journal of Economic Theory*, 182:1–24, 2019.

[SLP89] N L Stokey, R E Lucas, and E C Prescott. *Recursive Methods in Economic Dynamics*. Harvard University Press, 1989.

[Sun96] R K Sundaram. *A First Course in Optimization Theory*. Cambridge University Press, 1996.

[SB18] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

INDEX

A

An Introduction to Job Search, 291
Asset Pricing: Finite State Models, 91

D

Dynamic Programming
    Computation, 458, 471
    Theory, 458
    Unbounded Utility, 458

E

Elementary Asset Pricing, 179

F

Finite Markov Asset Pricing
    Lucas Tree, 98
Fixed Point Theory, 172

J

Job Search VI: On-the-Job Search, 335
Job Search VI: On-the-Job Search, 335

K

Kesten processes
    heavy tails, 64

L

Lake Model, 397
Linear State Space Models, 63
Lucas Model, 168
    Assets, 168
    Computation, 172
    Consumers, 169
    Dynamic Program, 169
    Equilibrium Constraints, 170
    Equilibrium Price Function, 170
    Pricing, 169
    Solving, 171

M

Markov Asset Pricing
    Overview, 91
Markov process, inventory, 15
Modeling
    Career Choice, 321
Modeling COVID 19, 5
Models
    Harrison Kreps, 137
    Lucas Asset Pricing, 167
    Markov Asset Pricing, 91
    McCall, 276
    On-the-Job Search, 335
    Pricing, 92

O

On-the-Job Search
    Model, 336
    Model Features, 336
    Parameterization, 336
    Programming Implementation, 337
    Solving for Policies, 340
Optimal Growth
    Model, 454, 470
    Policy Function, 465
    Policy Function Approach, 455
Optimal Growth I: The Stochastic Optimal Growth Model, 453
Optimal Growth II: Accelerating the Code with Numba, 469
Optimal Growth III: Time Iteration, 481
Optimal Growth IV: The Endogenous Grid Method, 493
Optimal Savings
    Computation, 504, 519
    Problem, 502
    Programming Implementation, 505
Orthogonal Projection, 149

P

Pricing Models, 91, 92
    Risk Aversion, 92
    Risk-Neutral, 92
python, 192

T

The Income Fluctuation Problem I: Basic Model, 501
The Income Fluctuation Problem II: Stochastic Returns on Assets, 517

U

Unbounded Utility, 458